

**ELEC371 CLASS NOTES-
INTRODUCTION TO REAL-TIME
OPERATING SYSTEMS
(2001)**

by

Robert Betz
(Copyright 2000, 2001)

**Department of Electrical and Computer
Engineering
University of Newcastle, Australia**

Acknowledgements

The following notes have been compiled from a number of sources and from personal experience. The first versions of the notes were largely based on Associate-Professor Peter Moylan's notes titled "The Design of Real-time Software, Class Notes for ELEC370".

Contact details

The author of these notes can be contacted as follows:

Associate Professor R.E. Betz,
Department of Electrical and Computer Engineering
University of Newcastle, Callaghan, NSW, 2308,
Australia.

Phone: +61-2- 4921-6091

FAX: +61-2-4921-6993

email: reb@ecemail.newcastle.edu.au

<http://www.ee.newcastle.edu.au/users/staff/reb/>

Table of Contents

Chapter 1	<i>INTRODUCTION TO OPERATING SYSTEMS</i>	1-1
References		1-1
What is an Operating system?		1-1
What is Multi-programming?		1-2
Why were Multi-programming Systems Developed?		1-2
Definition of an Operating System		1-2
Types of Operating Systems		1-3
Some more definitions – Processors, Programs and Tasks		1-4
Hierarchical Structure of Operating Systems.		1-5
A digression – the role of interrupts in Operating Systems.		1-6
What is an interrupt?		1-6
Why are interrupts crucial to Operating Systems?		1-6
Example: Producer-Consumer Problem		1-8
Chapter 2	<i>PRINCIPLES OF SOFTWARE DESIGN</i>	2-1
Problems with Real Time Design		2-2
Some Suggestions on Program Structure		2-3
The efficiency trap		2-4
Chapter 3	<i>OVERVIEW OF MULTI-TASKING</i>	3-1
Coroutines		3-1
Tasks		3-3
Anatomy of an Operating System Kernel		3-4
The Dispatcher		3-4
Synchronization Primitives		3-5
Protected and Non-protected Operating Systems		3-5
Brief Discussion of Memory Management		3-6
Brief Discussion of Supervisor Mode		3-6
Chapter 4	<i>SEMAPHORES</i>	4-1
Basic Concepts		4-1
Wait and Signal		4-2
Semaphores as Resource Counters		4-5
Task Synchronization		4-6
Consumer-Producer Problem Revisited		4-7
More Complex Synchronization Problems		4-10
Multiple unit resource allocation		4-10
Dining philosophers problem		4-11
Readers and Writers problem		4-12
Variations on the semaphore		4-15
Logic semaphores		4-15
Timed semaphores, sleeping tasks and simulated task suspension		4-16

Semaphores and priorities	4-17
Other synchronization primitives	4-18
Event primitives	4-18
Monitors	4-18
A practical semaphore structure	4-20
Deadlock and Starvation	4-21
Resource types	4-21
Reusable resources	4-21
Consumable resources	4-23
The conditions for deadlock	4-23
Techniques to handle deadlock	4-24
Deadlock prevention	4-24
Deadlock detection	4-25
Deadlock avoidance	4-26
Summarizing	4-28

Chapter 5 *TASK MANAGEMENT* 5-1

Task descriptors	5-1
Kernel Queues	5-1
Description of UNOS-V1.5	5-2
Unos-V1.5 Task Control Block (TCB) and Semaphore Structures	5-3
Queue Operation in UNOS-V1.5	5-6
UNOS-V1.5 task creation	5-8
The Dispatcher	5-10
The UNOS-V1.5 Dispatcher	5-10
The task switch operation	5-12
The UNOS-V1.5 task switch	5-13
Subtasks and Interrupt Tasks	5-14
Time-slicing	5-15
Unos time slicing	5-16
Task suspension	5-16
Dynamic Priorities	5-18
Definitions Useful for Priority Inheritance	5-20
A General Priority Inheritance Algorithm	5-24
An Implementation of Priority Inheritance	5-25
Some Explicit Implementations	5-29
The Inherit1 Algorithm	5-29
The Inherit2 Algorithm	5-29
The Inherit3 Algorithm	5-29
Interesting Benefit	5-33
Higher Order Scheduling	5-33
Some Definitions and Background	5-33
Basic Rate-Monotonic Analysis	5-35
Fundamental Results	5-36
Rate-Monotonic Scheduling	5-36
Utilisation Bound	5-36
Completion Time Test	5-37
Deadline-Monotonic Scheduling	5-39
Relaxing the Assumptions – Generalised Rate-Monotonic Analysis	5-40

Task Synchronisation	5-40
Release Jitters	5-44
Arbitrary Deadlines	5-44
Chapter 6 <i>TASK COMMUNICATION AND TIMING</i>	6-1
Intertask Communications	6-1
Common Data Buffer Technique	6-1
Circular Buffer Technique	6-2
Mail Box Technique	6-3
Pointer Transfer Technique	6-3
Examples of Intertask Communication in Some Real Operating Systems	6-4
Unix	6-4
UNOS	6-4
Timing in Real-time Systems	6-6
Time Delays	6-7
UNOS timers	6-8
UNOS timer creation	6-9
UNOS timer queues	6-9
UNOS timer maintenance routines	6-9
UNOS user interface routines	6-10
UNOS timed semaphore wait	6-11
Applications of UNOS timers	6-11
UNOS Heap Management	6-12
Chapter 7 <i>INTERRUPTS</i>	7-1
Basic interrupt mechanism	7-1
Finer details of the interrupt mechanism	7-2
Hardware structure of interrupts	7-3
Other types of interrupts	7-4
Interrupt priorities	7-6
Interrupt tasks	7-6
Interrupt handlers and the dispatcher	7-7
Relaxing the “no task switch” condition	7-9
Chapter 8 <i>DEVICE DRIVERS</i>	8-1
What is a device driver?	8-1
Structure of a device driver	8-1
Design objectives and implications	8-2
Anatomy of an I/O request	8-3
The device handler	8-4
Buffering	8-6
Issues related to efficiency	8-7
Chapter 9 <i>AN INTRODUCTION TO FILE SYSTEMS</i>	9-1
Logical Structure of File Systems	9-1

Basic Implementation	9-2
Sequential Files	9-3
Indexed Sequential Access Method (ISAM)	9-4
Indexed file	9-5
Directories	9-6
Symbolic Naming	9-7
Access Rights	9-7
Physical Structure of File Systems	9-9
Physical Structure of Disks	9-9
Disk allocation table	9-10
Record blocking	9-11
File allocation table	9-11
Some examples	9-12
MS-DOS file allocation	9-12
FAT32 File System	9-15
UNIX file allocation	9-16
Miscellaneous File System Issues	9-17
Links	9-17
File System Consistency	9-18
Real-time issues	9-19

Chapter 10 *VIRTUAL MEMORY AND PROTECTION* 10-1

Virtual addressing - basic concepts.	10-2
Virtual memory models	10-3
Segmentation - basic principles	10-3
Paging - basic principles	10-4
Some implementation issues	10-6
The best of both worlds - combined paging and segmentation	10-9

Chapter 11 *CASE STUDY - THE 80386-DX PROCESSOR* . 11-1

Application Programmer's Register Model	11-1
System Programmer's Register Model	11-2
System Flags	11-3
Memory-Management Registers	11-4
Control Registers	11-5
Segmented Address Models in the 80386	11-6
Flat Model	11-6
Protected Flat Model	11-6
Multi-Segment Model	11-6
Segment Translation in the 80386	11-6
Segment Selectors	11-7
Segment Descriptors	11-8
Paging in the 80386	11-10
Translation Lookaside Buffer.	11-12
Combined Segment and Page Translation	11-12
The Flat Paging Model – Getting rid of Segmentation	11-12
Segmentation and Paging Together	11-12

Protection in the 80386 Processor	11-14
Segment-level Protection	11-15
Segment Descriptors and Protection	11-15
Type Checking	11-15
Limit Checking	11-17
Privilege Levels	11-17
Restricting Access	11-19
Restricting Access to Data	11-19
Restricting Control Transfers	11-20
Gate Descriptors	11-21
Stack Switching	11-24
Returning from a Procedure	11-26
Pointer Validation	11-27
Descriptor Validation	11-27
Pointer Integrity and RPL	11-27
Page Level Protection	11-28
Restriction of Addressable Domain	11-29
Multitasking Support in the 80386	11-29
Task State Segment	11-31
TSS Descriptor	11-32
Task Register	11-33
Task Gate Descriptor	11-33
Task Switching	11-33
Task Linking	11-35
Busy Bit	11-35
Task Address Space	11-35
Protected Input/Output	11-36
I/O Addressing in the 80386	11-36
Protection and I/O	11-37
I/O Privilege Level	11-37
I/O Permission Bit Map	11-37
Conclusion	11-37

Appendix A *UNOS-V2.0 TUTORIAL* A-1

Introduction	A-1
General Layout of UNOS Based Software	A-2
Detailed Example of the Use of UNOS	A-3
The Main Module	A-3
The Initunos Module	A-6
Ustask Module	A-14
Task Modules	A-18
Task0 Module	A-18
Task1 Module	A-29
Task 2 Module	A-31
Header Files	A-34
unos.h	A-34
general.h	A-41
Common Header Files for User Tasks	A-42
comdec.h	A-42

pcscrn.h	A-42
tasks.h	A-42
time.h	A-42
Other important header files	A-42
hwpc.h	A-42
config_os.h	A-42

Appendix B *A SIMPLE WINDOW SYSTEM* B-1

Introduction	B-1
User Functions	B-2
How to Use the Windowing System	B-9
A Note on Setting Character Attributes	B-11
Reference Section	B-14
The Point Class	B-14
POINT::POINT DESCRIPTION	B-14
POINT::OPERATOR= DESCRIPTION	B-15
POINT::MOVETO DESCRIPTION	B-16
POINT::SET_ATTRIBUTES DESCRIPTION	B-17
POINT::WRITE_TO_PT DESCRIPTION	B-18
POINT::RETURN_ATTRIBUTES	B-19
The Screen Class	B-20
SCREEN::SCREEN DESCRIPTION	B-21
SCREEN::PRINT DESCRIPTION	B-22
SCREEN::CLR_LINE DESCRIPTION	B-23
SCREEN::CLR_EOL DESCRIPTION	B-24
SCREEN::CLR_BOL DESCRIPTION	B-25
SCREEN::CLR_SCR DESCRIPTION	B-26
SCREEN::SET_SCRN_BACKGRD DESCRIPTION	B-27
SCREEN::DRAW_BOX DESCRIPTION	B-28
SCREEN::DRAW_HORIZ_LINE DESCRIPTION	B-29
SCREEN::DRAW_VERT_LINE DESCRIPTION	B-30
SCREEN::SCROLL DESCRIPTION	B-31
Window Class	B-32
WINDOW::WINDOW DESCRIPTION	B-33
WINDOW::CLEAR_WINDOW DESCRIPTION	B-34
WINDOW::SET_WINDOW_BACKGRD_COLOUR DESCRIPTION	B-35
WINDOW::PAINT_WINDOW_BACKGRD DESCRIPTION	B-36
WINDOW::GO_WINDOW_XY DESCRIPTION	B-37
WINDOW::PRINT DESCRIPTION	B-38
WINDOW::PRINT DESCRIPTION	B-39
WINDOW::SET_STRING_COLOUR DESCRIPTION	B-40
WINDOW::SET_STRING_BACKGRD_COLOUR DESCRIPTION	B-41
WINDOW::SCROLL_WINDOW DESCRIPTION	B-42
WINDOW::WINDOW_CLEAR_LINE DESCRIPTION	B-43
WINDOW::GET_TOP/BOT_LEFT/RIGHT/XY DESCRIPTIONS	B-44
WINDOW::DRAW_HORIZ_LINE DESCRIPTION	B-45
WINDOW::DRAW_VERT_LINE DESCRIPTION	B-46

Appendix C	<i>UNOS-V2.0 REFERENCE MANUAL</i>	C-1
INTRODUCTION		C-1
KEY FEATURES		C-2
REAL-TIME SERVICES		C-2
HARDWARE REQUIREMENTS		C-4
INTRODUCTION TO UNOS-V2.0 KERNEL PRIMITIVES		C-4
Kernel Initialisation		C-4
setup_os_data_structures		C-4
Task Management		C-5
create_task		C-5
change_task_priority		C-5
rtn_task_priority		C-5
start_tasks		C-5
rtn_current_task_num		C-5
rtn_current_task_name_ptr		C-5
chg_task_tick_delta		C-5
Task scheduling management		C-5
preemptive_schedule		C-5
reschedule		C-5
start_time_slice		C-5
stop_time_slice		C-5
chg_base_ticks_per_time_slice		C-5
Time management		C-5
create_timer		C-5
start_timer		C-6
reset_timer		C-6
stop_timer		C-6
Intertask communication and synchronization		C-6
create_semaphore		C-6
init_semaphore		C-6
wait		C-6
timed_wait		C-6
usignal		C-6
return_semaphore_value		C-6
create_lock		C-6
lock		C-6
unlock		C-6
send_mess		C-6
send_qik_mess		C-6
rcv_mess		C-7
size_mbx		C-7
size_mbx_mess		C-7
free_mbx		C-7
used_mbx		C-7
flush_mbx		C-7
Memory Management		C-7
umalloc		C-7
ucalloc		C-7
ufree		C-7

ret_free_mem	C-7
Miscellaneous	C-7
return_interrupt_status	C-7
DETAILED DESCRIPTION OF USER INTERFACE	C-7
Kernel Initialisation Functions	C-8
SETUP_OS_DATA_STRUCTURES DESCRIPTION	C-8
Task Management Functions	C-9
CREATE_TASK DESCRIPTION	C-9
CHANGE_TASK_PRIORITY DESCRIPTION	C-11
RTN_TASK_PRIORITY DESCRIPTION	C-12
START_TASKS DESCRIPTION	C-13
RTN_CURRENT_TASK_NUM DESCRIPTION	C-14
RTN_CURRENT_TASK_NAME_PTR DESCRIPTION	C-15
CHG_TASK_TICK_DELTA DESCRIPTION	C-16
Task Scheduling Management Functions	C-17
PREEMPTIVE_SCHEDULE DESCRIPTION	C-17
RESCHEDULE DESCRIPTION	C-18
START_TIME_SLICE DESCRIPTION	C-19
STOP_TIME_SLICE DESCRIPTION	C-20
CHG_BASE_TICKS_PER_TIME_SLICE DESCRIPTION	C-21
Time Management Functions	C-22
CREATE_TIMER DESCRIPTION	C-22
START_TIMER DESCRIPTION	C-23
RESET_TIMER DESCRIPTION	C-24
STOP_TIMER DESCRIPTION	C-25
Intertask Communication and Synchronization Functions	C-26
CREATE_SEMAPHORE DESCRIPTION	C-26
INIT_SEMAPHORE DESCRIPTION	C-27
WAIT DESCRIPTION	C-28
TIMED_WAIT DESCRIPTION	C-29
USIGNAL DESCRIPTION	C-30
RETURN_SEMAPHORE_VALUE DESCRIPTION	C-31
CREATE_LOCK DESCRIPTION	C-32
LOCK DESCRIPTION	C-33
UNLOCK DESCRIPTION	C-34
DESTROY_LOCK DESCRIPTION	C-35
SEND_MESS DESCRIPTION	C-36
SEND_MESS_MACRO DESCRIPTION	C-37
SEND_MESS_NB_MACRO DESCRIPTION	C-38
SEND_QIK_MESS DESCRIPTION	C-39
RCV_MESS DESCRIPTION	C-40
SIZE_MBX DESCRIPTION	C-41
SIZE_MBX_MESS DESCRIPTION	C-42
FREE_MBX DESCRIPTION	C-43
USED_MBX DESCRIPTION	C-44
FLUSH_MBX DESCRIPTION	C-45
Memory Management Functions	C-46
UMALLOC DESCRIPTION	C-46
UCALLOC DESCRIPTION	C-47
UFREE DESCRIPTION	C-48

RET_FREE_MEM DESCRIPTION	C-49
Miscellaneous Functions	C-50
RETURN_INTERRUPT_STATUS DESCRIPTION	C-50
Appendix D THE KEYBOARD SYSTEM	D-1
Introduction	D-1
Organisation and Use	D-1
Appendix E UNOS SERIAL HANDLERS	E-1
INTRODUCTION	E-1
USER INTERFACE	E-1
SETTING UP AND USING THE SERIAL CHANNELS	E-2
The header files	E-2
USER INTERFACE DESCRIPTION	E-4
CREATE_SERIAL DESCRIPTION	E-4
Appendix F COMMAND DECODER	F-1
Introduction	F-1
Basic Data Structures	F-1
Decoder Tables - How to use them	F-3
Example	F-6
Appendix G UNOS ALARM HANDLING	G-1
Features	G-1
Data structures used in the Alarm System	G-1
The Alarm Number Array	G-2
The Alarm Structure.	G-2
Ancillary Data Structures	G-3
Low Level User Interface to the Alarm System	G-5
finCreateAlarm Description	G-5
fulgAlarm Description	G-6
fulgRetNumUnresetAlarms Description	G-7
fulgResetAlarm Description	G-8
fvdStopConseqSuppr Description	G-9
fulgResetAllAlarmsClearOnce Description	G-10
fulgRetCumulAlarms Description	G-11
finRetCumulStartTimeDate Description	G-12
finRetFirstUnresetAlarmTimeDate Description	G-13
finRetLastUnrestAlarmTimeDate Description	G-14
finClearCumulAlarms Description	G-15
Some Notes About Usage	G-16
Appendix H UNOS-V2.0 LISTING	H-1
CONDITIONS OF USE OF THE UNOS KERNEL	H-1

MISCELLANEOUS INFORMATION	H-1
FULL KERNEL LISTING	H-3
<i>BIBLIOGRAPHY</i>	b-i

List of Figures

Figure 1-1: The “Onion skin” structure of a generic layered operating system	1-5
Figure 1-2: Block diagram of the structure of the Windows NT Operating System.	1-7
Figure 1-3: Producer-Consumer Format	1-9
Figure 1-4: Block diagram of a circular buffer structure.	1-9
Figure 4-1: Wait and signal interaction in a producer consumer example.	4-6
Figure 4-2: The dining philosophers problem	4-11
Figure 4-3: Dining philosophers solution with only five semaphores	4-13
Figure 4-4: Progress of two tasks that eventually deadlock on two shared resources	4-22
Figure 4-5: Reusable resource deadlock scenario.	4-22
Figure 4-6: Memory allocation deadlock	4-23
Figure 4-7: Consumable resource induced deadlock	4-23
Figure 4-8: Circular wait on resources	4-24
Figure 4-9: The banker’s algorithm with multiple resources.	4-27
Figure 5-1: Central Table structure of UNOS.	5-3
Figure 5-2: Organization of Task Control Block Structures.	5-4
Figure 5-3: Relationship between the central table and the priority queues.	5-7
Figure 5-4: Possible state transitions if the suspended task is allowed.	5-17
Figure 5-5: Example of Priority Inheritance	5-20
Figure 5-6: Graphical representation of Definition 1	5-21
Figure 5-7: Graphical representation of Definition 2	5-21
Figure 5-8: Blockage trees for Policy 2	5-23
Figure 5-9: Blockage trees for Policy 3.	5-24
Figure 5-10: Change of blocking tress when a lock is released – Policy 1 case.	5-28
Figure 5-11: Example of the tracking through lock structures and TCBs to find the ultimate blocker of a task.	5-32
Figure 5-12: Periodic task timing.	5-35
Figure 5-13: Scheduling the task set of Table 5-1 with a Rate-Monotonic algorithm.	5-37
Figure 5-14: Example showing that schedulability of low priority tasks does not guarantee that of higher priority tasks.	5-40
Figure 5-15: Interaction of three jobs with semaphores using a priority ceiling protocol. ..	5-42
Figure 6-1: Mail Box Address translation	6-5
Figure 6-2: Mail box structure used in UNOS	6-6
Figure 7-1: Interrupt controller based system structure	7-4
Figure 7-2: Daisy chained interrupt structure	7-5
Figure 8-1: General structure of a device driver.	8-1
Figure 8-2: Device and stream information for a task	8-3
Figure 8-3: Device Control block (DCB) and device request queue.	8-5
Figure 8-4: Sketch of the control flow in an I/O system	8-7
Figure 9-1: Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree	9-3
Figure 9-2: Indexed sequential file access	9-4
Figure 9-3: Fully indexed file system.	9-5
Figure 9-4: Simple single level directory structure	9-6
Figure 9-5: Hierarchical directory structure	9-7
Figure 9-6: Format of a typical disk surface.	9-9

Figure 9-7: Disk allocation tables: (a) Block number free list. (b) Bit map free list.	9-11
Figure 9-8: Contiguous file allocation scheme.	9-12
Figure 9-9: Chained file allocation scheme.	9-13
Figure 9-10: Indexed file allocation scheme.	9-13
Figure 9-11: A MS-DOS directory entry	9-15
Figure 9-12: Structure of the UNIX i-node	9-16
Figure 9-13: Structure of a UNIX directory entry.	9-17
Figure 9-14: Locating a file in the UNIX file system.	9-18
Figure 9-15: File system consistency check results: (a) consistent file system; (b) inconsistent file system	9-19
Figure 10-1: Virtual to Physical Address Mapping	10-2
Figure 10-2: Base-limit relocation scheme	10-4
Figure 10-3: Format of segmented virtual address	10-4
Figure 10-4: Addressing for full segmentation	10-5
Figure 10-5: Address mapping for paging	10-6
Figure 10-6: Page mapping via associative store	10-7
Figure 10-7: Page algorithm with associative store.	10-8
Figure 10-8: Address mapping for paging within segments	10-10
Figure 11-1: Application Register Set for the Intel 80386 Microprocessor.	11-2
Figure 11-2: System Flags for the 80386 Processor	11-3
Figure 11-3: Memory Management Registers	11-4
Figure 11-4: Control Registers in the 80386 Microprocessor.	11-5
Figure 11-5: Location of the segment descriptors via the GDTR and the LDTR.	11-7
Figure 11-6: Block diagram of the segment selector.	11-8
Figure 11-7: Block diagram of a segment descriptor.	11-9
Figure 11-8: Page translation mechanism in the 80386 microprocessor.	11-11
Figure 11-9: Format of a page table entry	11-11
Figure 11-10: Combined segmentation and paging address translation	11-13
Figure 11-11: Segmentation and paging with each segment having its own page table. ...	11-14
Figure 11-12: Descriptor fields used for protection.	11-16
Figure 11-13: Protection ring structure	11-18
Figure 11-14: Privilege Checks required for access to data.	11-20
Figure 11-15: Privilege Checks required for control transfers without a Gate.	11-22
Figure 11-16: Call Gate Descriptor	11-23
Figure 11-17: Call Gate Mechanism	11-24
Figure 11-18: Intralevel Control Transfers	11-25
Figure 11-19: Gate Interlevel Call and Return	11-26
Figure 11-20: GDT set up for a paged, flat memory model.	11-30
Figure 11-21: Diagram of a Task State Segment.	11-31
Figure 11-22: TSS Descriptor Layout	11-32
Figure 11-23: Location of the TSS using the Task Register.	11-34
Figure 11-24: Nested TSS's and the NT bit status.	11-36
Figure A-1: Arrangement of the initialisation modules.	A-2
Figure D-1: Block diagram of the structure of the keyboard decoding system	D-1
Figure F-1: Schematic of decoding table relationships for ascii character decoding.	F-4

References

Some references for the course are listing in the Bibliography. It should be noted that this list is by no means complete. Almost any introductory book on operating systems will contain most of the basics of operating systems. The real-time aspects of operating systems are not nearly as well covered. Some useful references are [1–5]

The best references on the details of the protected mode operation of the 80386 series of micro-processors are the reference manuals supplied by the chip manufacturer Intel [6–8].

What is an Operating system?

The answer can have a variety of different definitions depending on the view point taken. I shall canvas a few possibilities and then end up with the definition that will be assumed for this course.

Misconceptions:–

- Most people experience with computers by using a time shared system such as VMS, UNIX or RSTS. The use of the editors, compilers, and the command interpreter to the casual user appear to be the “system”.
- many popular personal computers come with a Disk Operating System (DOS) (e.g. MSDOS, CP/M, Mac OS etc.) and the functionality of these “operating systems” is then assumed to be what an operating system is.

The above misconceptions are not silly or stupid since the name operating system is often mis-used by computer vendors. Let us consider each of the above and highlight where the problem is.

When someone is using a time shared system such as VMS or UNIX the editors, compilers and command interpreters are really only a user interface to the operating system services and are no more part of the operating system than a program that a user may write.

The use of the term disk operating system to describe say MSDOS is only a very limited use of the term operating system. What MSDOS provides is a set of routines which enable the user to issue a command to, for example, save an area of memory to a disk without having to worry about all the details of how the disk hardware works and how to format the information on the disk. A similar situation also occurs in relation to writing information to the screen. This software interface to the hardware is used by both the command interpreter (which is simply a program which runs after the machine boots up) and other applications. Your immediate reaction to this might be “*Well that's what an operating system does, doesn't it?*”. Well the short answer to that response is – yes and no. The abstraction of the hardware so that the user is presented with a *virtual machine* which is much nicer to use is one of the features of all operating sys-

tems. However the other feature of operating systems called *multi-programming* is only present in a very rudimentary form in a DOS such as MSDOS.

What is Multi-programming?

Multi-programming is a term which means that several unrelated programs can reside in a computers memory and take turns using the main processing unit. Anyone that has used VAX VMS or UNIX has experienced a multi-programming environment because these systems can execute a large number of user programs seemingly simultaneously on the one central processing unit. In fact to the user it should appear that he or she is the sole user of the system.

Why were Multi-programming Systems Developed?

The multi-programming concept was born from the observation that computers spent much of their time waiting for slow peripheral devices to either store or retrieve data. In the early days of computing the peripheral devices were very slow – e.g. paper tape, magnetic tape and Tele-types (10 characters/sec) were very common. Although peripheral devices have increased considerably in speed so have computers, therefore even today the relative speed difference between the peripherals (e.g. the disk system, vdu's) is probably about the same. Therefore the same issue is still relevant.

What's wrong with waiting for an I/O job to be done? – Answer: one is simply not getting the best out of the computing hardware. For a single user situation it doesn't matter that much (although multi-programming systems are now very popular in this situation because of other benefits). In the late 1960's and early 70's computers of only modest power by today's standards were very expensive. Therefore it was important to get the maximum utility out of these machines. Waiting for I/O was unproductive time. Hence systems began to appear which could carry out work on someone else's job whilst another job was waiting on I/O or some other resource which was not available when it requested use of it.

The software which enabled the orderly scheduling of jobs in such an environment is known as a multi-programming operating system. Such software also provided hardware abstraction and access control. Collectively the provision of these two aspects is what I shall call an “operating system”. Notice that MSDOS above does not provide any proper support for multi-programming.

Given the above discussion let us now try to formalise our definition of an operating system.

Definition of an Operating System

An operating system is a piece of software which implements the following functions:–

- Permits multi-programming. This section of the operating system is called the *kernel*. This software consists of a number of procedures which perform such functions as creating new tasks, deciding which task should run at any particular moment, providing a mechanism to switch the processor from one task to another task, provide primitives to allow controlled access to shared resources.
- Provides a virtual machine interface to the Input/Output (I/O) devices connected to the computer system. This means that the operating system must contain software to interface user applications to the underlying hardware without the application having to know any of the detail of how the hardware works. Furthermore, the operating system must control access to this hardware since it is possible that several jobs in a multi-programming system may want to do I/O at the same time. It is this factor which provides the main reason for placing this software in the kernel.

- The operating system provides memory management facilities. These facilities allow the allocation and deallocation of memory to user tasks.
- Finally the operating system provides file control software which allows applications to manipulate files on devices such as disks and tapes without having to worry about the allocation of physical blocks on the storage medium to the files.

To summarise the above points what we are really saying is that the operating system provides a mechanism for sharing of resources and controls access to the resources.

The term *resources* is being used in an abstract sense since a resource can be a physical resource such as a printer or disk drive, or it can be a software entity such as a data structure in the memory of the machine. Even the central processing unit itself which is executing the kernel is a resource which the operating system is controlling.

Types of Operating Systems

Operating systems can be broken into two broad categories based on the applications they are suitable for:–

- real time operating systems
- non-real time operating systems.

There is some debate as to what constitutes a real time operating system. For example, is an airline booking system a real time operating system? Obviously when a terminal operator types in a booking they expect a reply from the computer system within, say a couple of seconds. Therefore there is a time constraint on what is acceptable performance. Many Computer Scientists consider this to be an example of a real time operating system. Engineers on the other hand would not consider this to be a real time operating system. A real time operating system to an engineer has *hard* time constraints as opposed to *soft* time constraints. A hard time constraint is defined as a time constraint if violated that will result in the system not fulfilling its function. The difference is best demonstrated by example.

The FA-18 uses a computer system to control its terrain following system. The algorithm which controls the flight surfaces must execute at precisely times 40 times per second. If the precision in the timing is not maintained the aircraft may become unstable and crash. If the software carrying out this function was operating under the control of an operating system it would be classified as a real time operating system.

Now returning to the airline booking system example. The time constraint on such a system is not hard – if the operator has to wait an extra second for a response when use is particularly heavy then it will not make any difference. The airline bookings will still occur.

Another categorisation of operating systems is as follows:–

- static operating system.
- dynamic operating system

A static operating system is one where the number of tasks can be accurately defined before run time. Therefore when the system starts up a number of tasks are created and this number does not change.

A dynamic system on the other is one in which the number of tasks is not known a-priori. Tasks are created and destroyed whilst the system is operating. An example of this situation that is familiar is a time shared computing system, where the number of tasks changes in an indeterminate way as users log in and out of the system.

The static/dynamic classification often corresponds to the real time/non-real time as described above. Real time applications of operating systems are often *embedded* – i.e. they are stored in read only memory (ROM) in a computer system which forms a part of some larger system (e.g. a laser printer, most VDUs, some dishwashers etc.). Because the operating system and the code for all its jobs are in ROM then it is physically impossible to create and destroy tasks. Therefore by nature many real time systems are static. There are of course always exceptions to this generalization. Similarly it is easy to see that a operating system underlying such an application as the airline booking system would have to be very much a general purpose multiuser operating system.

Some more definitions – Processors, Programs and Tasks

Program:—a sequence of instructions which when executed carry out some activity. This definition is more or less what most people who have been involved in programming would accept.

Task:— a task is an action which is carried out by executing a sequence of instructions (i.e. by executing a program or programs). This action may provide a system function or an application function. The other requirement of a task is that it is an entity that the operating system can run concurrently. At first sight it appears that a task and a program are the same thing. However a task is a superset of a program. This is best demonstrated by an example. Consider a program that implements a FIFO queue. This program could be shared by a number of other tasks which as part of their actions require the use of a fifo queue. This then makes a task appear as a program module which is using another program module, however remember activations of program modules in a non-operating system environment cannot execute concurrently. Therefore the concept of a task is intimately tied to the operating system environment. In fact when a task is created the operating system creates an execution environment which is separate from all other task execution environments (i.e. separate data areas, separate stack for procedure calls). Several tasks may share the same code but the code is executed in a different execution environment. As far as each task is concerned it is executing on its own processor.

The difference between a task and a program can be seen in the following example. Consider a computer system which allows users to edit ascii files. Generally in computer systems which allow this the editor code is designed to be reentrant – that is multiple execution threads can execute the code at the same time. Reentrant code often has the same structure as recursive code. Recursive code allows a procedure in languages such as Pascal and C to call themselves. This implies that each reincarnation of the procedure has its own set of local variables (any global variables will be manipulated by each reincarnation). This property is required by code which is being used by multiple tasks, otherwise one task will corrupt the data being manipulated by the other task. Back to the example at hand. In the case of the editor, each user will be running a copy of the editor. The task or action required by the user is the editing of the file. Therefore each user is running a task. However each task is actually executing the same instructions, in fact it is executing physically the same instructions (or program) – i.e. there is physically only one copy of the editor in memory. However the data manipulated by these instructions is different for each task, and the instructions are effectively being executed on different processors.

Processors:— something which executes instructions. There can be one or more processors in a system. If there are as many processors as tasks then there can be real concurrency in the execution of the tasks – i.e. the tasks are actually being executed simultaneously. At the

other extreme if there is only one processor in the system then all the tasks have to share it. At any point in time only one task is actually being executed. Since to the user such a system still appears as if there are the same number of processors as tasks (executing more slowly than in the multiprocessor case) then this is often called logical concurrency.

Hierarchical Structure of Operating Systems.

Most software systems are structured so that their complexity can be more easily handled, and operating systems are no exception. Modern operating systems are based on a layered structure, with each layer having a well defined role and interface to the layers below and above it. The first operating system to be totally designed using these principles was the ICL George III OS of the very early 1970's. The layered structure means that each layer can be changed considerable with little or no effect on the other layers in the system. In addition, it enforces a logical structure which will make the total system more easily understood, and consequently more easily maintained.

The typical structure of a generic layered operating system is shown in Figure 1-1. The central

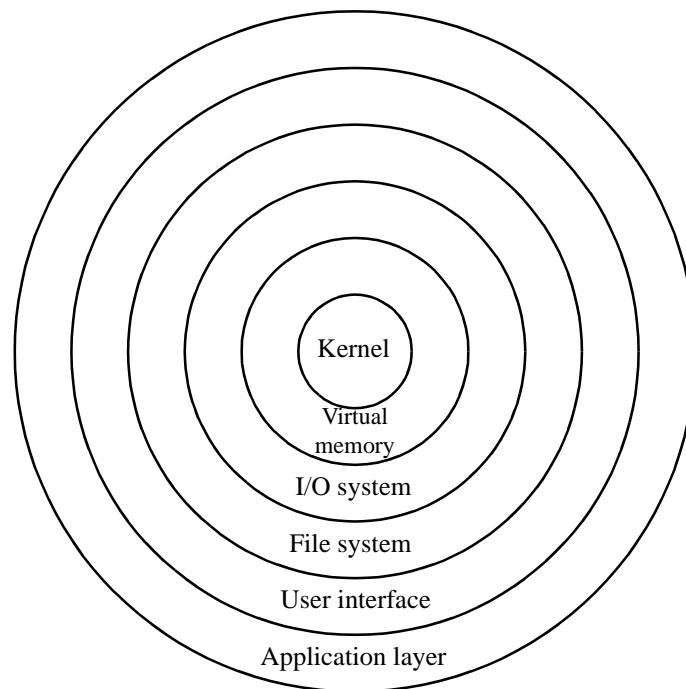


Figure 1-1: The "Onion skin" structure of a generic layered operating system

part of the operating system is known as the kernel (because of the similarity with a nut). The purpose of the kernel is to provide the multi-tasking support for the system. In other words this is the part of the system that can switch from one task in the system to another. It is usually a relatively small amount of software, and is often written in assembly language, since it requires access to very low level functionality in the processor. The modern trend in operating systems is to have a "micro-kernel" - that is a very small kernel (although it must be said that small is a relative term; many of these micro-kernels are of the order of 100 kilobytes in size). The reason for this is portability - the kernel is primarily assembly language and has to be rewritten for each machine that the operating system is ported to.

The layers of the operating system structure above the kernel are usually written in some high

level language. The next layer above the kernel is also very privileged and handles all the memory allocation and protection in the system. The outer layers usually consist of large amounts of code and provide such facilities as I/O (i.e. terminal input/output, printing facilities, disk I/O etc.), a file system on devices such as disks, a user interface, and so on. The depth of the layer in a protected mode operating system also often indicates the degree of trust one has in the reliability of the software, and consequently on the privilege level the software has. Note that the user layer of the system has the least privilege in the system.

As a specific example of a modern layered operating system kernel, Figure 1-2 shows the structure of the Microsoft Windows NT operating system. This system has two privilege levels – a user level (least privileged) and a supervisor level (most privileged). The reason for choosing this type of an OS model (as compared to the complete hierarchical model shown in Figure 1-1) was to implement a flat memory model environment for the user. If a layered model had been used then the internal parts designated as operating in kernel mode would also be layered. This would have implied the use of a segmented memory architecture.

Another interesting aspect of the Windows NT kernel structure is the Hardware Abstraction Layer (HAL). The purpose of this layer in the kernel is to make the kernel itself as independent as possible from the particular hardware that the system is running on. Therefore, in theory the only code which needs to be re-written is the HAL code and the kernel should be able to carry out task switching.

The Protected Subsystem section of the Operating System runs in user mode even though the subsystems are really part of the OS. This is implemented in this way to make the OS more reliable – the subsystems are not crucial for the system to run, therefore if one of them crashes, for whatever reason, the rest of the system should be unaffected. By running these subsystems in user mode they do not have the privilege to damage vital parts of the operating system.

A digression – the role of interrupts in Operating Systems.

What is an interrupt?

An interrupt is a hardware generated subroutine call. The processor hardware generates one of these subroutine calls (or interrupt) usually in response to some external event. For example the closure of a switch may cause a line connected to the interrupt mechanism of the central processing unit to go high. The processor will generally finish the execution of the current machine language instruction and then it will generate a call to a subroutine. The subroutine is found usually by the CPU looking in a particular memory location for its address. After the subroutine has completed execution control returns to the instruction after the one where the interrupt occurred. It should be noted that interrupts can also be caused via software initiated events such as arithmetic exceptions, memory protection violations etc.

There are several differences between an interrupt executed subroutine and a software initiated subroutine. Firstly, an interrupt routine can be called at any point in the code execution. Therefore it is indeterminate when an interrupt subroutine will be executed. Secondly, upon entry to an interrupt subroutine the CPU status at the time of the interrupt is saved. Typically an interrupt subroutine will automatically save the program counter and the flag register (whereas a normal subroutine only saves the program counter). The CPU registers are then usually explicitly saved. This state is restored upon exit from the interrupt routine and enables the interrupted routine to continue execution as if there had not been an interrupt.

Why are interrupts crucial to Operating Systems?

Consider a multi-programming environment without interrupts. The situation is that a particu-

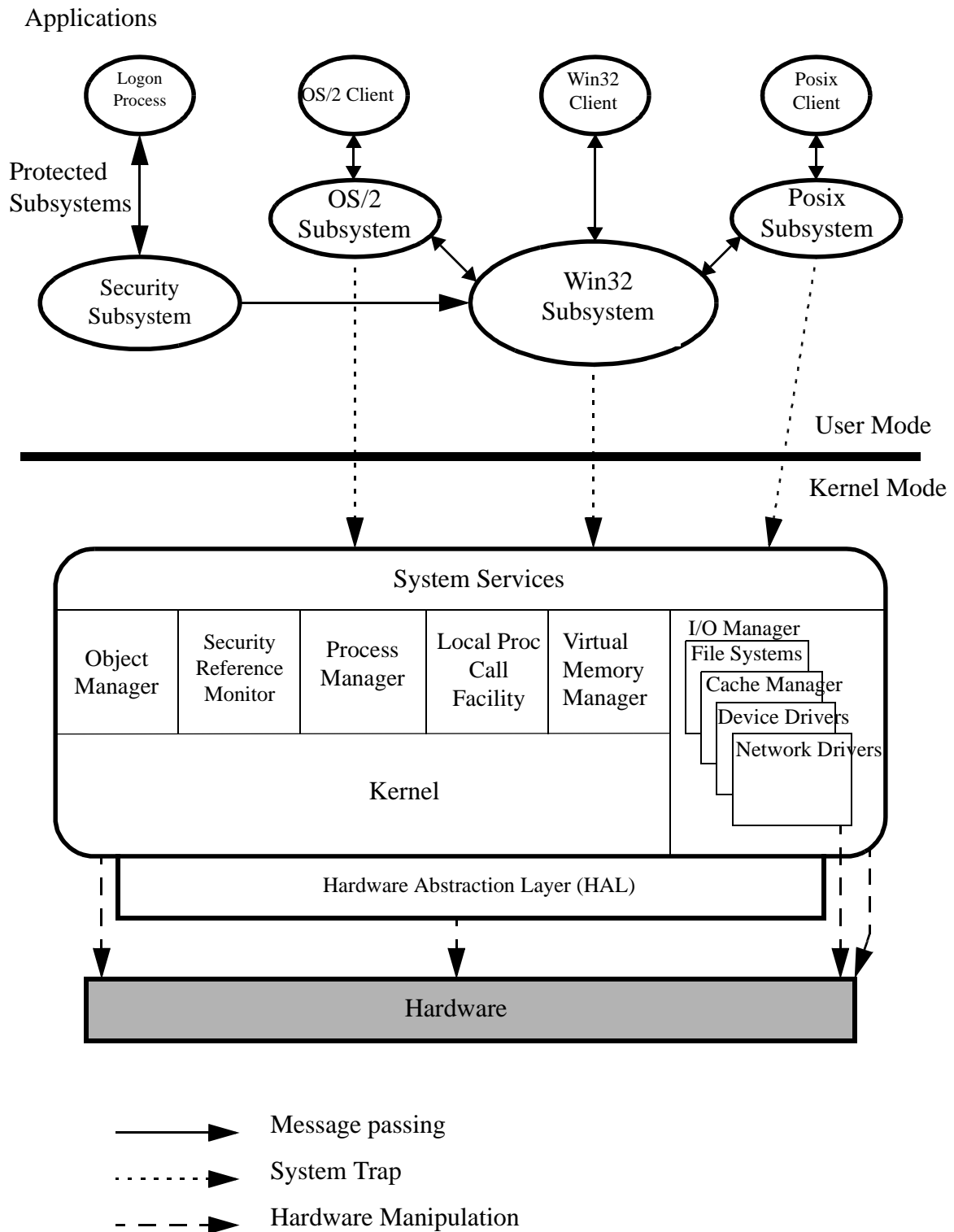


Figure 1-2: Block diagram of the structure of the Windows NT Operating System.

lar task is sending characters out of a slow serial device. After one character is sent the task attempts to send another character but the serial device will not be ready for some time to transmit another. The sending task then calls the operating system to request that another task

run while it is waiting to do more I/O. The operating system will then suspend the sending task and run another. In this situation the question which arises is – “how does the operating system know when the serial hardware is ready to output another character?”

There was really no satisfactory answer to this problem until the invention of the hardware interrupt. One possible solution was that the sending task was periodically run to recheck to see if the serial hardware was ready – i.e. a polling solution. This is clumsy and difficult to implement without a timer interrupt, and in any case time is being wasted executing a routine which while polling is achieving nothing.

If the system supports interrupts then the hardware is designed so that when the serial hardware is ready to send another character it initiates an interrupt which runs a subroutine which in turn informs the kernel that the hardware is ready to accept another character. The kernel will then schedule the relevant task to run. The task will then output another character and then again stop executing until another interrupt occurs.

If interrupts were not present then it would be virtually impossible to implement transparent time slice scheduling. Time slicing is designed to ensure that compute bound jobs cannot hog the central processor in a multi-programming environment. A hardware interrupt occurs periodically (driven by some clock hardware) which causes the kernel to be entered. The kernel then runs a procedure within itself called the scheduler which decides what the central processor should run for the next period of the clock. Time shared system users have experienced an environment which uses time slicing. If interrupts were not present to allow this feature then explicit calls to the kernel would have to appear in application code to allow other tasks to get a share of the CPU. It should be noted that it is not always necessary for a time slice interrupt to be present for a system to function. In many systems (e.g. real time systems) a task will carry out a specific action and then call the kernel to request that another task be allowed to run.

To show the benefits of interrupts in general consider the following example of a producer-consumer problem. This example also highlights one of the main problems that can occur in concurrency – critical sections.

Example: Producer-Consumer Problem

In order to demonstrate the benefits of the interrupt approach in general programming let us consider a problem that commonly occurs in programming – the producer-consumer problem.

The basic producer consumer problem is shown in Figure 1-3. The consumer task may in turn be a producer to another data storage buffer in the chain. Therefore the notation of producer/consumer is a relative term.

A more concrete example of such a problem is the circular buffer problem. In this particular case the producer is the serial input hardware and its associated input software routines. The data produced is stored in a circular buffer. The consumer is the software which reads the data stored in the circular buffer and does some further processing.

A circular buffer is a simple data structure which is commonly used to store serial data. It can be thought of as a circular array of memory locations with two pointers into the array. One pointer is known as **put_ptr** points to the location in the memory array where the next character is to be put. The other pointer is **get_ptr** and this points to the location in the memory array which contains the next character to be read out. In addition to the pointers a counter for the number of free locations and the size of the buffer are kept. A block diagram of a circular buffer is shown in Figure 1-4. Even though the buffer can be conceptually thought of as a circular memory array, it physically is a linear section of memory, which with appropriate checks

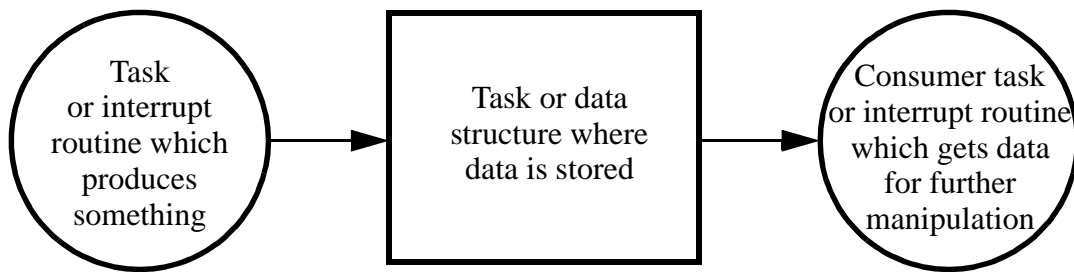


Figure 1-3: Producer-Consumer Format

on the buffer pointers functions logically as though it is circular. The put routine is the pro-

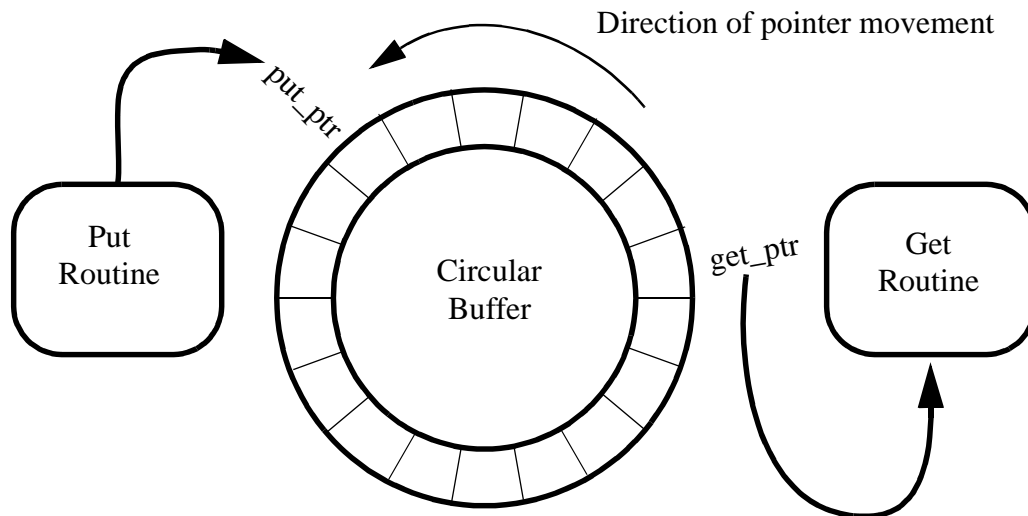


Figure 1-4: Block diagram of a circular buffer structure.

ducer in the block diagram, and the get routine is the consumer. Each of these routines manipulates the **put_ptr** and **get_ptr** to place and retrieve characters from the buffer. Checks must be made to see if the buffer is full or empty before placing or retrieving characters from it.

The bones of the software to implement putting characters into and getting characters out of this buffer consists of two main procedures:– **get** and **put**. These procedures should be designed to work with an arbitrary circular buffer. An outline of both of these appears below:–

```
type
  circular_buffer = record
    buffer_size : integer;
    used : integer;
    free : integer;
    put_ptr : integer;
    get_ptr : integer;
    buf : array [ 0..buf_size-1 ] of char;
  end;
```

```
var buffer : circular_buffer;

procedure put ( ch : char ; var cir_buf : circular_buffer);
begin
    if cir_buf.free > 0 then
        begin
            (* put the character into the buffer at
               the put_ptr *)
            cir_buf.buf [ cir_buf.put_ptr ] := ch;
            (*update the put_ptr and the used and free locations*)
            cir_buf.free := cir_buf.free - 1;
            cir_buf.used := cir_buf.used + 1;
            cir_buf.put_ptr := (cir_buf.put_ptr + 1 ) mod
                               cir_buf.buffer_size;
        end;
    else
        carry out error handling if necessary;
    end;
end;

procedure get ( var ch : char; var cir_buf : circular_buffer );
begin
    (*check to see if there is a character to be read*)
    if cir_buf.used > 0 then
        begin
            (*read the character out of the buffer *)
            ch := cir_buf.buf [ cir_buf.get_ptr ];
            (*update the pointer and the number of free locations*)
            cir_buf.get_ptr := (cir_buf.get_ptr + 1 ) mod
                               cir_buf.buffer_size;
            cir_buf.free := cir_buf.free + 1;
            cir_buf.used := cir_buf.used - 1;
        end;
    end;
end;
```

There are two basic approaches to getting data to be put into this buffer. The most obvious one is the polling approach. This approach uses the CPU to emulate a multiplexer. The CPU executes a program which very quickly looks at all the devices in the system which must fill buffers of this type.

There are a number of problems with this approach:—

- CPU spends time just carrying out the polling. This can lead to low performance.
- Polling forces the software to be of a certain design – for example there must be a loop which can execute fast enough so that no characters will be missed. In the case of a multi-tasking operating system there would need to be a task which was running fast enough to ensure that characters are not missed. This time of execution is difficult to estimate especially if the rate at which characters can arrive varies greatly (e.g. have keyboard input as well as computer to computer input).

The other approach to the obtain the data for the buffer is the interrupt approach. This approach overcomes the two main difficulties mentioned above since the CPU only interacts with a hardware device when there is some data to manipulate. Consequently there is no waste of time needlessly polling. Furthermore, since the CPU response is event driven then there are no problems with vastly differing data transfer rates. However it should be pointed out that in some software situations the overhead of the interrupt routine may be intolerable and a polling

approach (even in an operating system environment) can be better. An example of this can be found in many time shared computer systems.

A time shared system is characterized by a large number of serial input lines. If an interrupt occurs on every key stroke the overhead of the interrupts and the associated operating system kernel activity can cause the system to grind to a halt. Consequently many of these systems use a hardware buffer which can store a number of incoming characters without any CPU intervention. The operating system then runs a task every so often which polls these buffers and if they contain characters flushes them.

Let us now see how the above **put** and **get** routines would be used in a piece of polling software and in an interrupt driven piece of software

Polled solution (using an executive):-

```
program polled_example;
  begin
    (*initialise the buffer variables*)
    cir_buf.put_ptr := 0;
    cir_buf.get_ptr := 0;
    cir_buf.buffer_size := buf_size;
    cir_buf.free := buf_size;
    cir_buf.used := 0;
    kingdom_come := false;
    (*now start the main processing loop*)
    repeat
      (*check to see whether the buffer is full. If so then bypass
      the following section else see if there is a character waiting
      in the hardware to be input.*)
      if cir_buf.free > 0 then
        begin
          (*now check to see if there is any input from the serial
          hardware device*)
          if input_rdy then
            begin
              ch := getchar(serial_device);
              put ( ch, cir_buf );
            end; (*if*)
          end; (*if*)
          (*now check to see if the buffer is empty. If empty then go and
          do the other jobs in the system else invoke the consumer code*)
          if cir_buf.used > 0 then
            begin
              (*must be characters in the buffer so get them and do
              further processing on them - e.g. say echo them to the
              screen*)
              get ( ch, cir_buf );
              further_processing_of_ch ( ch );
            end; (*if*)
          (*enter at this point to carry out the other tasks that the
          system should do. Note that these must be carried out in time to
          allow a repoll of the input device*)
          other_jobs;
        until kingdom_come;
      end; (*program polled_example*)
    end;
```

Notice that in the above example that having a buffer may not make any sense. If the characters are read from the hardware one at a time then a reasonable question to ask is why bother buff-

ering it. It would probably make more sense to read the character from the hardware and then call the processing routine straight away. However if the hardware is set up so that there is some hardware buffering as described above then having the circular buffer can make sense, assuming that the processing routine only processes one character at a time.

The performance of the above polled solution is dominated by how fast the **repeat-until** loop can be executed. Notice also that it has the undesirable property that the other jobs that have to be done have to be written in such a way that the main loop can still be executed at some minimum rate. If there is a long processing job that must be performed then it has to be broken down into a number of subtasks each one being executed in turn on each loop through the executive. This obviously can be messy.

Now let's consider the interrupt driven approach to this problem. The software for this consists of three separate pieces of code. Firstly there is the initialisation code.

```
program interrupt_example;
begin
    (*initialise the buffer variables*)
    cir_buf.put_ptr := 0;
    cir_buf.get_ptr := 0;
    cir_buf.buffer_size := buf_size;
    cir_buf.free := buf_size;
    cir_buf.used := 0;
    terminated := false;
    (*now enable the system interrupts*)
    enable_interrupts;
    (*can now enter an executive to carry out other functions*)
    repeat
        other_jobs;
    until terminated;
end; (*program interrupt_example*)
```

The next two sections of code which can run concurrently with the other jobs in the system are the interrupt routines. These are the equivalents to the producer and consumer tasks in an operating system.

```
interrupt procedure producer;
    enable__interrupts;
    (*now check to see if there is room in the circular buffer*)
    do while cir_buf.free = 0
    end; (*while*)
    (*must be free room in the buffer so get character and put it into the
    buffer*)
    ch := getchar ( serial_device );
    put ( ch, cir_buf );
end; (*interrupt procedure producer*)
```

```
interrupt procedure consumer;
    enable_interrupts;
    (*now check to see if there is data in the buffer*)
    do while cir_buf.free = buf_size;
    end; (*while*)
    (*must be data in the buffer so get it*)
    get ( ch, cir_buf );
    further_processing_of_ch;
end; (*interrupt procedure consumer*)
```

In the above solution it is crucial to realise that the producer consumer and main program rou-

tines can run in logical concurrency. The fact that the execution of the producer code is event driven means that regardless of whether there is hardware buffering or not the circular buffer performs a filtering function so that fast bursts of incoming data will be stored. Data will only be lost if the circular buffer becomes full.

The particular solution above is a poor solution because there are possible time dependent failure modes. Consider the following scenario:–

Both devices busy and **cir_buf.free = buf_size**. Both the producer and consumer devices become busy.

input interrupts

output interrupts before the **cir_buf.free** location has been decremented.

deadlock – stuck in the output routine waiting for data to appear in the buffer, but will never return to the input routine to decrement **cir_buf.free**.

A second problem is a reentrancy problem. Because the interrupts are enabled in the interrupt routine an interrupt can occur whilst the interrupt routine is being executed. This can cause two problems. the first is that interrupt routines are generally manipulating global data structures (since one can not pass parameters to an interrupt routine) and the second incarnation of the interrupt routine could cause corruption of data structures which are already being manipulated by the first incarnation of the routine.

A third problem is that it is not a good idea to poll within an interrupt routine. In fact it is for this reason that the interrupts are enabled in the routine.

The fourth problem with this interrupt implementation is related to the others – the increment and decrement operations on the **cir_buf.free** location may not be indivisible. This causes a problem if an interrupt occurs in the middle of the increment or decrement. This can be a particular problem when high level languages are used.

Consider the following time sequence execution of the interrupt routines:–

Input Routine

```
;SAVE THE REGISTERS
PUSH AX
PUSH BX
PUSH BP
;ENABLE INTERRUPTS
EI
;OTHER PROCESSING
MOV AX, WORD PTR FREE(old)
```

——Get an output interrupt——→

Output Routine

```
PUSH AX
PUSH BX
.
.
PUSH BP
EI
;OTHER PROCESSING
.
;UPDATE FREE
MOV AX, WORD PTR FREE(old)
;INCREMENT FREE
INC AX
MOV WORD PTR FREE(new), AX
;RESTORE REGISTERS
POP BP
```

```

      .
      .
      POP BX
      POP AX
      RETI
◀—Return from output int. —
;DECREMENT FREE
DEC AX
MOV WORD PTR FREE(new), AX
;RESTORE REGISTERS
POP BP
.
.
POP BX
POP AX
RETI

```

The `cir_buf.free` location after this sequence should contain the same value as it did before the sequence – we have put a character into the buffer and then taken a character out of the buffer. However because the decrement and increment operations are not indivisible the actual result is that the *free location is now one less than it should be*. The section of code where the increment/decrement is done in this routine is called a **critical section** because of the problems which occur if there is concurrent execution of the code. For the code to work correctly only one interrupt routine should be allowed to enter these sections. In the case of an interrupt routine this is handled by disabling the CPU interrupts. As we shall see later an operating system has more sophisticated primitives which are used for critical section protection. In this case critical section is generalized to resource protection. In this case the `cir_buf.free` location is a shared resource between two interrupt routines.

A better interrupt solution which overcomes all the above problems would have the general form of that below:–

The following shows the general form for both the **producer** and **consumer** routines. The particular example is for the **producer**.

```

interrupt procedure producer;
begin
  (*check to see if free space for data*)if cir_buf.free > 0 then
    begin
      ch = getchar ( serial_device );
      put ( ch, cir_buf );
    end; (*if*)
  else
    begin
      (*no room in the buffer so issue an error message and if
      a software control serial channel issue a control S*)
      indicate_error ( serial_buf_full_err );
      issue_ctrl_S;
    end; (*else*)
  end; (*interrupt procedure producer*)

```

When an interrupt occurs the interrupts are automatically disabled. Since they are not reenabled within the routine then the interrupts remain disabled preventing reentrancy problems and making the increment and decrement operations indivisible. Note also that the routine does not do any internal polling. Since interrupts are disabled whilst the interrupt routine is executing it

is a essential that interrupt routines are very short

Although the above example has concentrated on a normal interrupt routine solution to a serial channel handler many of the issues involved are common to operating systems. Interrupt routines have the same sort of concurrency problems that individual tasks have in an operating system. In fact often a task switch in an operating system is initiated by an interrupt. The classic producer-consumer problem in operating systems is very similar to that just described except that the interrupt routines are replaced by tasks. The data need not be produced by some hardware device but may be the output of some computation by the producer. The circular buffer is then an intertask communications buffer.

The design of a piece of software goes through a number of stages:

- Requirements definition
- Specification
- Conceptual design and data structure design
- Coding
- Verification

A problem which often occurs is that the first three of the above are bypassed and programmer dives straight into the last – the coding (this is especially a problem with engineers).

Question: Why is this approach a problem?

Answer: This approach generally results in poor code from every point of view – probably doesn't work, poor documentation (if any), not maintainable due to the rats nest structure of the code. Even if it does work the performance may be poor.

The only case where the above approach is acceptable is if the code is very short and is to be run once or twice and then thrown away. On any programming project of moderate or greater size time spent on understanding what the problem is and how the software should be constructed before coding can save many weeks (or months) further down the track.

Let's have a brief look at each of the stages mentioned above:

Requirements specification:– this stage means that one must understand the requirements of the completed piece of code. In a work for client situation this means that the clients requirements of the software should be understood clearly. In this situation a requirements document which is read and agreed upon by the client.

Software specification:– this is a more formal specification of the software in greater detail. The higher level requirements specification is broken down into details which are more implementation oriented. This phase usually results in a document which is intended for programmers.

Software design:– this means the detailed design of the code and the data structures to allow the formal software specification to be met.

Coding:– writing the code.

Verification:– testing to see if the code fulfills the requirements specification. This stage could uncover design faults or coding errors. If a design fault is found at this stage and the project is reasonably large the costs of fixing it can be very high.

The two most crucial stages in the software development are the first two. Generally if these

two stages are correct all the others will fall into place. This stage of design can be very time consuming. For example it is very easy to say that you need a program to control machine X. However it is much more difficult to say exactly what the controller should do in all possible situations. For example such questions as what sequences of operations must be performed, what outputs must be measured, what inputs must be controlled, what error conditions can arise and what actions must be carried out if they do, what are the timing constraints? In any reasonable size problem the answer to such questions in total is quite complex. These questions should be addressed at the requirements and software specification stages.

Probably the most common software design methodology is *top down design*. This methodology proposes that the overall problem to be solved is specified. This problem is then broken down into a smaller number of sub-problems which are hopefully independent of one another. These sub-problems are then further broken down into smaller sub-problems and so on until the sub-problems remaining are reasonably trivial. Therefore the philosophical basis is divide and conquer. The top down approach can not only be applied to the design but also to the coding. One can start with only a few procedures which correspond to the highest level sub-problems. Each of these procedures can then call several other procedures which correspond to the next level of sub-problem. If as the new procedures are added they are tested then the debugging of the program can be much less painful and the programmer can have a high degree of confidence that the code is correct.

The opposite design methodology is *bottom-up* design. As the name implies this implies that the low level functions are designed first and gradually the design works towards the higher level functions. There are several problems which arise if this design philosophy is followed:

- the programmer often never has a complete understanding of the total functionality of the program that he or she is trying to create.
- one can begin by implementing the wrong primitive level functions, this fact only becoming evident after the higher level functions have been implemented.

Obviously the above two problems are related and feedback on one another. The second is really a micro example of the problem caused by the first point. The first point whilst leading to the second also can have the consequence that the whole overall design of the system proves to be incorrect. Usually the end result of a bottom-up design approach is a lot of recoding and a less than satisfactory piece of software even if one does end up getting it to work.

In practice software design and coding is a mixture of the above two approaches. Sometimes a bottom-up approach is required to get a feel for the problem, but this should be regarded as prototype software to be thrown away after an idea has been tested. Regardless of the approach when a piece of software has been completed it should have the appearance of a top-down design.

Problems with Real Time Design

Special problems occur with the design of real time software because the operation of the system is critically dependent on low level software components such as interrupt routines. In fact it is very difficult to do any testing of the system without these low level components being present in some form, since by definition the system is driven by the timing of external asynchronous events.

The design approach is usually a mixture of top-down design and bottom-up design as mentioned in the previous section. The preliminary design is to start with a top-down approach in order to understand the problem in its totality. This step is essential in order to reveal the critical real time sections of the system and also to suggest the low level interrupt handlers that may

be required. In some cases many of them are obvious, however some may not be until the problem is thought about in detail.

Real time software system design cannot be considered in isolation from the hardware of the system. It is the combination of the hardware and software which will allow the timing requirements to be satisfied. Hardware issues can even impinge on the testing of the software of a real time system. Often in order to test the software adequately a simulated plant must be built. This allows the software to be tested under controlled conditions with simulated real timeness. The logical correctness of the software is tested. In addition many real time systems are controlling critical plants and erroneous software could result in catastrophic damage to the process being controlled. Sophisticated hardware debugging aids have been developed in order to allow real time debugging of software. These devices are known as In-Circuit Emulators (ICE). They allow complex breakpoint conditions to be set up (e.g. stop program execution if there is a write to a particular memory location by a procedure belonging to a particular module) using hardware. This means that the software runs at normal speed (therefore meaning that the real time behaviour of the software is not effected). Software debuggers cannot carry out these types of operations (at least not with real timeness maintained). Most ICE's also offer symbolic debugging allowing the user to see the source language statements when breakpoints are encountered.

Normally in the initial stages of real time software design hardware simulators are not used. Instead many of the external realtime hardware interactions can be simulated by tasks operating under the operating system being used or even under another operating system which is part of the software development system. The former of these two approaches can often capture much of the real time behaviour of the true hardware (depending on the nature of the external events that one is attempting to simulate). In both cases the logical correctness of the software can be tested.

Some problems that occur in real-time software operation are unique to real time systems. For example the software can be logically correct and can function correctly most of the time. However occasionally the system seems to misbehave – this can be exhibited by say the controlled plant output suddenly undergoing an excursion. The programmer then has to ask the question “Is it the hardware, the software, the control algorithm or some real time problem?”. In a situation like this the problem is often a real time problem – that is a combination of events is occurring which is causing a critical task to be delayed longer than it should. These problems are very difficult to diagnose under a conventional operating system. A good real time operating system should have low level support to help detect when timing constraints are not being satisfied and capture the state of the system under this condition to aid in diagnosis.

Some Suggestions on Program Structure

Most reasonable size to large computer programs are written by a team of people. Therefore each member of the team has to understand how their section of the program will interact with the sections written by other people. In order to do this the program must be well written (since the person that wrote a particular section may have left the team). Even if the program has been written by one person the same is true.

The important attributes of the program are structure and readability. It is usually in sections of code which are difficult to read and unstructured (the two often but not always go together) that most errors occur since these features usually show that the programmer had not thought through the structure for the piece of code before writing it (see principles of software design above).

Some rules which should be obeyed to generate a “good” program:

- use modular design principles – program divided into collections of related procedures called modules. If the language allows it the procedures should be divided into two types – private and global. The private procedures can not be accessed by procedures outside the module, whereas the global ones can. The module should also contain private data which can only be manipulated by global procedures within the module. These modular techniques are really essential in multiple programmer situations, but should also be practiced by a single programmer. The use of modular programming results in more maintainable code.
- do not use global variables. The satisfaction of this requirement is related to modules in that if something has to be declared global then it is done in the private part of a module. This means that only procedures related to that module can modify it, thereby reducing the chance of side effects. The use of local variables also adds to readability on a micro scale because at the point where a procedure is called the variables passed to the procedure indicate what is being used and what is being modified by the procedure without having to read the details of the procedure.
- procedures should be short and the flow of control should where possible be linear. It should not resemble a “rats nest”. One should where possible avoid the use of “goto” statements. There are some circumstances where the “goto” statement is justified (in fact in assembly language it is impossible to avoid) but in high level language programs these are generally rare.
- the physical code should be written so that it is clearly laid out. The code is essentially a literary work which must be understood by others. Variable names should contain enough characters to convey what the variable does. Similarly for procedure and module names. All modules should contain a substantial commented section at the beginning describing what the module does and its overall structure. Similarly procedures should be commented at the beginning so that its function can be understood without having to read the procedure body. The internals of a procedure need only be commented if something unusual or non-obvious has been done.

The efficiency trap

Programming style has often been sacrificed in the name of program efficiency. However this is generally a false concept. In sacrificing the program style and structure one may find that an incorrect but very fast program has been generated, as opposed to a usually only slightly slower but correct program.

There are two main ways of generating better efficiency:

- *micro-efficiency* – i.e. tweeking individual procedures or sections thereof in order to achieve better speed.
- *macro-efficiency* – i.e. changing the algorithms or data structures to achieve better speed.

The macro-efficiency approach is the best way of improving a programs speed. Careful selection of data structures and algorithms in the design stage is the best time to implement this, as macro changes well down the program development path are very costly in both time and money. Micro-efficiency on the other hand usually only gives a small improvement in performance after a considerable amount of effort. In fact the best people to carry out micro-efficiency improvements are the people who write the compilers being used.

Coroutines

Coroutines are an entity half between a subroutine call and a task. It is for this reason that we shall examine them here.

Let us first examine in detail subroutines or procedures. A computer program usually consists of a number of subroutines which are called from some stream of execution. If **PROC** is the name of a subroutine this is usually done by a **CALL PROC** assembly language instruction. When the subroutine is entered the return address – that is the address of the next machine language instruction after the **CALL** is pushed onto the stack of the machine. After the subroutine has done what it is supposed to do a **RET** instruction is executed. This instruction causes the data which was pushed onto the stack to be put into the program counter of the processor resulting in execution returning to the machine language instruction after the original **CALL**. Notice that two control transfer primitives are required in a subroutine call – the **CALL** primitive to enter the subroutine and the **RETURN** primitive to get back to the calling code. In the case of a coroutine only the **CALL** like primitive is present, there is no **RETURN**. The operative differences are best demonstrated by the example below.

We shall assume that the processor has a coroutine instruction which is written as **RESUME**. The processor is running **PROC1** and in the process executes a **RESUME** instruction. This causes the processor to transfer control to **PROC2** which we shall assume has been running at sometime previously. This at this point sounds very much like a procedure call, however there is an important difference – **PROC2** does not start executing at the start point of the procedure but at the point just after where it had been previously stopped by a **RESUME** in its own code stream. The diagram below demonstrates the relationship between the two procedures which are executing as coroutines.

	PROC2
	INSTRUC 1
	INSTRUC 2
	RESUME PROC1
PROC1	
INSTRUC 1	
INSTRUC 2	
INSTRUC 3	
INSTRUC 4	
RESUME PROC2	
	INSTRUC 4
	INSTRUC 5
	INSTRUC 6
	INSTRUC 7
	INSTRUC 8
	RESUME PROC1
INSTRUC 6	
INSTRUC 7	
INSTRUC 8	

Chapter 3 — OVERVIEW OF MULTI-TASKING

INSTRUC 9
INSTRUC 10
RESUME PROC2

INSTRUC 10
INSTRUC 11

and etc.

The code execution starts off with **PROC2** which carries out several instructions before executing a **RESUME PROC1**. This then causes **PROC1** to begin execution and this routine will continue carrying out its instructions until it executes a **RESUME PROC2** in its code stream. This causes the processor to resume execution of **PROC2** at the instruction after the **RESUME PROC1**. This pattern then continues with the processor effectively being shared between the two procedures. This same pattern can be defined for three or more procedures.

What is the low level mechanisms to allow such an entity as a coroutine? Somehow the resumption address must be stored somewhere for each of the procedures which are executing. One simple technique would be as follows:

- (i) pop the resumption address of the other routine from the stack
- (ii) push the resumption address of the currently executing routine onto the stack
- (iii) jump to the address obtained from step (i).

This approach has several practical problems which could cause it to fail even in the simple case of the previous example. Firstly if there are more than two coroutines then one would end up with the situation that the resumption address of the next coroutine may not be on the top of the stack. Secondly the implication of the above example is that the coroutines do not call conventional subroutines – this is almost always not true. If a subroutine is called then the resumption address may again not be on the top of the stack. Even if this problem is overcome there is still a problem because the stack contains subroutine return address information for various coroutines at various depths within the stack. An example of a stack format under this type of situation is shown below:

COROUTINE 1 STACK	RETURN ADDRESS SUB. 1.1
	RETURN ADDRESS SUB. 1.2
	RETURN ADDRESS SUB. 1.3
	RESUMPTION ADDR 1
COROUTINE 2 STACK	RETURN ADDRESS SUB. 2.1
	RETURN ADDRESS SUB. 2.2
	RETURN ADDRESS SUB. 2.3
	SUBROUTINE PARAMETER 1
	SUBROUTINE PARAMETER 2
	RETURN ADDRESS SUB. 2.4
	RESUMPTION ADDR 2
COROUTINE 3 STACK	RETURN ADDRESS SUB. 3.1

etc., etc.

Notice that as control is transferred between the coroutines (which can occur in any order) then the position one goes to in the stack is changed. If for example **COROUTINE 1** is resumed then the stack pointer would have to point to **RESUMPTION ADDR 1**. If another two calls were made to a subroutines in the coroutine then the return addresses for **COROUTINE 2** would be over written.

All of the above problems can be overcome by keeping a separate stack for each of the coroutines. By doing this the resumption address is always on the top of the stack and the stacks can grow arbitrarily within the area allocated to them, thereby overcoming the problem of subroutine calls overwriting information for other tasks. The stack pointers for each task then must be

stored somewhere where they can be found when the resume operation occurs. Usually they are stored in a special location associated with the task. When a **RESUME** is executed the resumption address of the current task is stored on its stack and the stack pointer value is stored in the special location. The new stack pointer is then retrieved for the task to be resumed and the resumption address popped off the stack. It is this feature that makes coroutine implementation very much like a task. It essentially operates in its own environment. One difference between a coroutine and a task is that reentrant coroutines normally do not make sense. A coroutine cannot be reentered from another thread of execution so that it is being simultaneously executed by two threads (although one may be able to think of an implementation where reentrancy is possible). This is due to the implementation of the **RESUME** switching mechanism. Because the address to resume at is stored on the stack of the coroutine to be resumed, all resume operation will start at the same address. In a system that allows reentrancy the return address to a task is stored on the current task stack. This allows each task in the system to have a different address to return to in some common reentrant task. Another deficiency of the coroutine concept is that scheduling of the various coroutines is “hardwired” into the coroutine code. This makes changing the scheduling of execution difficult. If a coroutine executes a **RESUME** to itself then it simply continues to execute.

The **RESUME** instruction mentioned in the above examples is usually not present in most processors. A language which supports coroutines usually simulates the **RESUME** instruction mentioned in the above examples. One high level language which does support coroutines is Modula 2. The **RESUME** statement in this language has the form:

TRANSFER (P1, P2)

where **P2** is the coroutine to be resumed. The **TRANSFER** command carries out the appropriate storage of resumption addresses and switches to the appropriate stack for the coroutine.

The fact that coroutines are implemented in so few languages is a measure of how useful they are considered to be. In most cases a conventional implementation of tasks within an operating system environment is more versatile than coroutines.

Some operating system environments essentially operate very similarly to a coroutine based system. One example is the very popular Windows operating system by Microsoft. In this system all “tasks” within the system “cooperatively” multi-task. This means that the running task must release the processor to another task. It differs from a pure coroutine approach in that the transfer is not directly from one task to another, but is carried out via a call to the Windows kernel. The flaws of this approach are evident when Windows “locks up”. This is usually due to an error in an application which results in it not releasing the processor, consequently the kernel cannot regain control of the system and for all practical purposes the system has crashed.

Tasks

As mentioned in the section “Some more definitions – Processors, Programs and Tasks” on page 1-4, a task is an action carried out by executing a series of instructions. The instructions themselves constitute the program which itself can be broken down into modules and procedures. An analogy that most people can relate to is that when making a cake; the recipe is the program and the execution of the recipe by mixing the ingredients is the task. Obviously one can have two people using the same recipe (or program) but making two different cakes. This corresponds to a piece of reentrant code being executed by two different tasks.

In an operating system context tasks execute in logical concurrency. That is usually only one

task is being executed at any particular point in time in a single processor system. In a multiple processor it is possible to have true concurrent execution, however in most of the following discussion it will be assumed that one has a single processor.

In order to create the illusion of concurrent execution on a single processor there must be a mechanism for rapidly switching between tasks. The mechanism which achieves this is known as a *task switch*. The similarity between task switching and coroutines can be seen. Indeed multitasking can be simulated by writing a coroutine for each task. This type of multitasking has several limitations which can be summarised as follows:

- a coroutine is executed after an explicit call from another coroutine. Therefore a multitasking solution based on using coroutines relies on cooperation between tasks. Given that in many cases tasks within an operating system are logically independent then this is not a satisfactory solution.
- the problem of organizing the calls between coroutines so that all coroutines have a fair share of the processor would be very complex and prone to error. Indeed in many cases tasks may not get a share of the processor at all.
- coroutines do not contain any scheduler. This makes it difficult to determine which tasks out of those available is actually ready to run.
- basic memory management.

It is for these reasons that a more general method of controlling task switches is required.

Anatomy of an Operating System Kernel

The operating system kernel is a collection of procedures which provide the environment for multitasking. The functionality provided by the kernel in relation to task switching is in broad terms as follows:

- the ability to switch from one task to another based on interrupt or software driven events
- usually provides some way of determining which tasks should be running based on priority.
- provides synchronization primitives which are used for intertask communication, resource counting and task critical section protection.

In the following subsections I will provide a brief description of the main components of a kernel. In all cases a more detailed look at these kernel sections will be carried out in future sections of these notes.

The Dispatcher

The dispatcher (also known sometimes as the low level scheduler) is a piece of software in the kernel which decides what is the next task in the system to run and then carries out the task switch to that task. Generally the selection of the task is done on a priority basis.

Physically the dispatcher consists of the following:

- a procedure or collection of procedures which execute an implementation specific algorithm to decide what task will be the next to run when system control exits the kernel. This set of procedures are collectively often known as the scheduler.
- a collection of queue maintenance routines which allow the manipulation of data structures which contain all the information that the kernel needs to know about a task (these structures are known as the task control blocks or the task descriptors).

The order of actions in the dispatcher are as follows:

- save the state of the task running when the kernel was entered.
- place the saved state of the task on the appropriate kernel queue.
- execute the scheduler procedure or procedures and select out of the tasks ready to run the highest priority one.
- restore the state of the selected highest priority task.
- leave the kernel and enter the execution stream of the new task.

Typically there are three conditions which will cause the dispatcher to be invoked:

- (i) A task has released a resource which results in a previously blocked task becoming ready. If this task is higher priority than the current task then the dispatcher will be called.
- (ii) A task becomes blocked on a resource. The blocked task is placed on a queue of blocked tasks and a new task is chosen to run.
- (iii) An interrupt occurs which results in a task switch becoming appropriate.

Synchronization Primitives

The kernel generally provides a set of primitives which simultaneously allow intertask communications, resource counting and critical section protection. These primitives are often called **WAIT** and **SIGNAL**. These will be discussed in considerable detail in a following section.

Sometimes the kernel will provide higher level primitives such as message passing. Often a message passing system will exist in a system but may not be in a kernel. This is done to keep the kernel small and allows the potential user to not use message passing and therefore not have any code overhead

Protected and Non-protected Operating Systems

Operating Systems can be written to support two different types of hardware platforms – protected mode hardware and non-protected mode hardware. A protected mode system has hardware built into the processor so that tasks can be prevented from corrupting each other. This is usually achieved by only allowing tasks access to certain parts of the physical memory map. On the other hand a non-protected processor allows any address in the physical memory map to be accessed from anywhere. Therefore an errant task can corrupt other tasks within the system. The primary motivation to use protected mode processors is to detect errors in the task software and deal with them in a graceful way (which does not include a system crash).

Given that a more robust system can be built using protected mode processors why would someone use a non-protected mode processor? The answer to this question is partly historic and partly economic. Protected mode processors have been around for a long time, however they traditionally involved a great deal of extra hardware and were therefore much more expensive to purchase. Non-protected mode processors were aimed at the low cost end of the market, such as embedded applications. The advent of the microprocessor to some extent has relieved the economic constraint (since it does not cost much more to produce a complex processor as compared to a simple one – although the processor development costs will be much larger). The first microprocessors did not have the integration density to allow on board protection hardware, hence most of the earlier designs do not have protection features. However most of the processor designs developed from the mid 1980s onwards employ protection (although

much of the software that runs on them does not use this feature).

The two basic attributes of protection are *memory mapping* and *privilege checking*. The memory mapping serves to protect the memory used by one task from corruption by another task. It usually provides a mapping function between virtual addresses and physical addresses. The privilege checking provides two fundamental modes that the processor can operate in – *user mode* and *supervisor mode*. User mode is the mode that the user tasks operate in and the supervisor mode is the mode that the kernel operates in.

Brief Discussion of Memory Management

Memory management will be considered in more detail in a later section of the course. However I will attempt to outline the basic principles briefly here.

In the section above I indicated that memory management can be divided into two sections. This was a logical division since in most circumstances the memory mapping and the memory protection usually go hand in hand. Therefore the following discussion will assume that both are present.

Memory mapping translates *virtual* addresses into *physical* addresses. The virtual address is the address that a task uses in its code. The physical address is a physical memory location in the memory map of the system. The mapping between the virtual to physical memory is carried out in real time by the memory map hardware on the processor. For each task within the system there is a different virtual to physical address mapping. This mapping is part of the task's context which has to be restored by the kernel when a task switch occurs. This means that two different tasks can be operating in the same virtual address space but in a different physical address space. The virtual to physical address mapping for a particular task is set up at task creation time by kernel routines. Usually all tasks are mapped to have their own piece of physical memory. As will be explained in the next paragraph the user task cannot change this mapping therefore it is impossible for a task to access the memory belonging to another task. If a task has to communicate with another task then this must be done via operating system routines which provide a regulated gateway between tasks.

Brief Discussion of Supervisor Mode

The above memory mapping/protection scheme could fail if the user tasks can modify the virtual to physical mapping. To protect against this occurring the instructions to do this are privileged and cannot be executed by a user privilege task. Only the kernel routines are allowed to execute privileged instructions. Some processors have more than just two type of instructions, this allowing finer control. Using this technique only a very restricted set of procedures in the operating system would be allowed to operate at the most privileged level thereby giving more robustness against corruption at the kernel software level. The mode which allows privileged instructions to be executed is often called supervisor mode.

If the supervisor mode instructions are privileged and cannot be executed by the user then the question arises as to how one can enter supervisor mode. There are three common techniques:–

- there is an automatic change to supervisor mode upon any interrupt (this includes software interrupts). This implies that application specific user interrupt routines will be considered as operating system device drivers. They have to be carefully written and rigorously tested because they can completely crash the system.
- There is an automatic change of mode to supervisor mode when a particular software interrupt occurs. This then allows the user interrupt routines to run in user mode.
- there is a special “call” instruction which is used to enter the kernel and at the same time

switch modes.

A more complete discussion of the above protection and memory management issues can be found in Chapter 10 and Chapter 11.

Basic Concepts

In the previous discussion on interrupt routines, particularly on the producer/consumer example we introduced the concept of a critical section. At the interrupt level critical section protection was handled by disabling interrupts so that only one thread of code execution could manipulate the critical variables. In an operating system similar situations can occur when two or more tasks are the separate threads of execution which manipulate some common data structure or other system resource. The disabling of interrupts in the tasks will also protect the shared resource from corruption but it leads to undesirable side effects, namely that interrupts can be disabled for considerable lengths of time depending on the nature of the manipulations on the shared data structure. In a real time system this could lead to performance problems due to the disruption of the precision of the timing. For these reasons a different technique is used for critical section protection based on a construct known as a semaphore (a word which means signal or alternatively a device which is used to send signals).¹ The system construct which we develop is used for a variety of other tasks in addition to critical section protection, but for the moment we shall concentrate on critical section protection.

Let us go back to the critical section protection implementation which was used with the interrupt driven consumer/producer example used in a previous section. The conceptual form of this protection can be generalized to the following form:

```
enter_critical_section ( flag );
perform the critical section operations
leave_the_critical_section ( flag );
```

where the flag variable is used to indicate to the other execution threads that the critical section has been entered.

Let us now have a look in some more detail at the **enter_critical_section** procedure itself.

```
procedure enter_critical_section (var flag : flag_type );
begin
    if state of the flag indicates that another task has entered
    the critical section then wait until the other task changes the
    flag

    enter here only if no other task is in a critical section - set
    the flag to indicate that this task has entered the critical
    section
end;
```

1. The concept of semaphores in computing was only developed in the mid 1960's. However, flags or signals being used to control access of a resource has been used on railways to control trains on tracks for a long time prior to this.

As can be seen from the above code the **enter_critical_section** routine itself is a critical section. If an interrupt occurs just after the flag has been tested, and another task sets the flag, then the first task and the second task will both enter the critical section at the same time. This can be simply solved in the same way that interrupt routine critical sections were handled – disable interrupts during the critical testing of the flag. At a later stage we shall look at the implementation of this routine in more detail.

In the first part of the above routine the entering task has to wait if the flag is set. The inference is that a busy wait is performed. If the interrupts are disabled just before the testing of the flag then a busy wait would result in interrupts being permanently disabled and the whole system would appear to lock up completely since a task switch can never occur in this condition. To get around this problem we need to introduce the concept of a *blocked* task.

A task in most operating systems can be in three basic states:

- (i) running – i.e. the task which is currently executing on the processor.
- (ii) ready but not running – that is the task can run when processor time is allocated to it by the scheduler.
- (iii) blocked – the task is not running and will not be scheduled even if there is free processor time.

A task enters the blocked state when it is waiting for some condition to be satisfied. For example in the case of the critical section protection routine if a task or tasks could not get past the flag then they would become blocked on that flag. They do not consume processor time by busy waiting but are placed on a list of tasks which are blocked on the same flag. The operating system scheduler is then invoked and a new ready task to run is chosen and dispatched.

If a task becomes blocked then there must be some way of making it ready to run again when the condition which blocked it is released.

The flag variable approach to be taken in the rest of the course is the known as the *semaphore* approach. This approach was first defined by Dijkstra in 1965.

A semaphore consists of two main components (although actual implementations may contain more) arranged in a record or structure. A Pascal implementation could therefore be:

```
type
    semaphore = record
        count : integer;
        blocked_list : queue
    end; (* record *)
```

The variable **blocked_list** is a queue of blocked tasks on the semaphore. Usually this queue is implemented as a linked list of tasks. There is a separate queue for each semaphore.

Wait and Signal

The procedure names usually used to carry out the **enter_critical_section** and **leave_critical_section** functions are called **wait** and **signal**.

The implementation of a **wait** procedure is relatively straight forward:

```
procedure wait ( var sem : semaphore );
begin
    enter_kernel;
    sem.count = sem.count - 1;
    if sem.count < 0 then
        begin
```

```

        place the current task on the sem.blocked_list;
        call dispatcher - choose another task to run;
    end (* if *)
    leave_kernel;
end; (* wait *)

```

The procedures **enter_kernel** and **leave_kernel** are there to ensure the indivisibility of the **wait** procedure. The actual implementation of this can vary depending on the hardware structure of the system. However for the moment one can consider it to be a **disable** instruction.

For use in critical section protection the initial value of the semaphore count is '1'. When a task executes a **wait** the value is decremented to '0' and a return from the procedure occurs. The task can then proceed into the critical section. If a task switch occurs whilst this task is in its critical section then it is possible that another task can require access to the critical section. This task would then call the **wait** procedure to request access permission. The semaphore count would again be decremented but since the semaphore count will be less than 0 the dispatcher will be called.

The count variable of the semaphore is interpreted as follows:

semaphore=1 \Rightarrow no task currently inside its critical section

task is inside its critical section. This task has effectively claimed the semaphore.

task is inside its critical section and the absolute value of the count value is the number of tasks that are blocked on semaphore.

The **signal** procedure carries out the inverse operation to the **wait** procedure. The form of this procedure is as follows:

```

procedure signal ( var sem : semaphore );
begin
    enter_kernel;
    sem.count = sem.count + 1;
    if sem.blocked_list is non-empty then
        begin
            remove a task from the top of the blocked queue and place
            it on the top of the priority queue - i.e. mark it ready
            call dispatcher ( if a pre-emptive task switch is to
            occur);
        end; (* if *)
    leave_kernel;
end; (* signal *)

```

Each call to the **signal** routine removes only one task from the blocked queue. This is consistent with the fact that a call to **wait** blocks at most one task and the **signal** reverses the action of the **wait**.

The other point to note is that the **sem.count** variable ends up with its original value of one. This can be seen by realising that each **wait** is paired with a **signal**. If this pairing is not adhered to then it is possible that a critical section can become permanently locked resulting in a deadlock situation and consequent system failure. When the **wait** and **signal** procedures are being called from within the same task then this is relatively easy to see, however often the **wait** is called from one task and the **signal** is called in another. This makes it a little harder to see if the calls to **wait** are balanced by calls to **signal**.

Chapter 4 — *SEMAPHORES*

An example of how **wait** and **signal** procedures are used for critical section protection is shown below. The data structure to be shared is a record of the format shown:

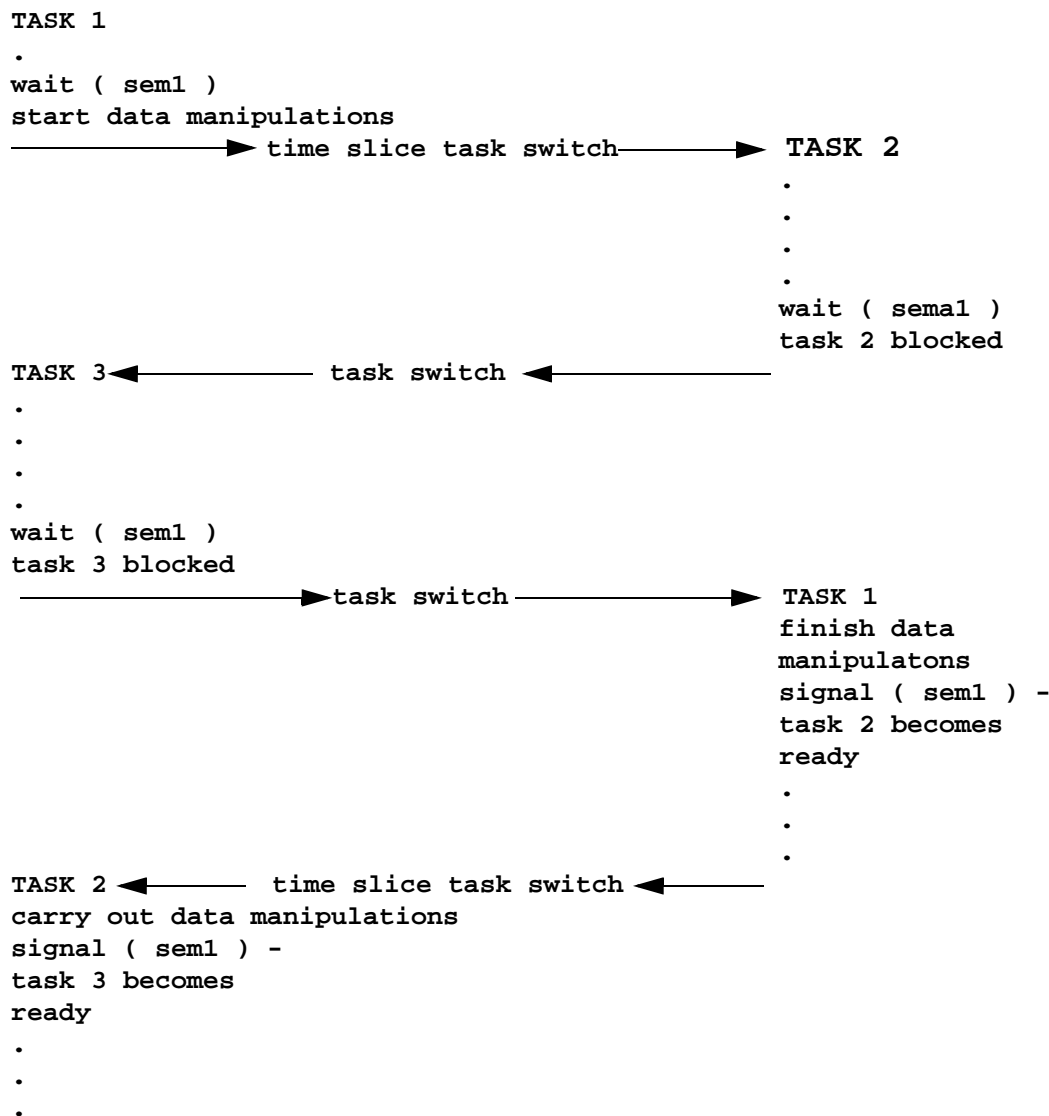
```
var shared_data :  
    record  
        cs_protect : semaphore;  
        otherfields : whatever  
    end; (* record *)
```

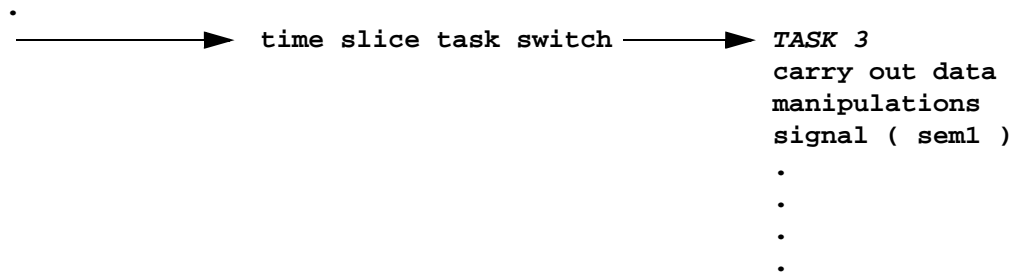
This record is to be read and written to by two or more tasks. At the points of access in each of the tasks we require the following:

```
wait ( shared_data.cs_protect );  
carry out operations on the shared data  
signal ( shared_data.cs_protect );
```

The initial value of the semaphore count needs to be one. This is usually set up at the time the semaphore is created.

The diagram shown below graphically shows the thread of execution through three tasks when all of them are attempting to manipulate the same data structure. In the following diagram it is assumed that the three tasks have the same priority. The semaphore **sem1** is assumed to have an initial value of '1'.





etc., etc.

Each shared data structure in a multi-tasking system needs a semaphore. One way of ensuring that a semaphore is associated with the data structure that it is protecting is to include the semaphore record inside a larger record containing the data structure. This leads to better programmer readability. However it can make an operating system monitor more difficult to write because there is no central point from where the monitor procedures can interrogate all the semaphore data structures and queues in the system. The kernel does not really know what a user has created inside his or her data structures. Therefore other housekeeping information would have to be kept by the **wait** and **signal** routines so that the kernel knows about them.

Situations do occur where semaphore protection may seem unnecessary. For example, operations such as reading or writing a single byte or incrementing a single byte where the processor has an instruction which can increment data directly in memory. If the code is written in high level language it is wise to use semaphore protection because one is never sure what the code being produced really is. If the system contains multiple CPU's then some of these seemingly indivisible instructions can be stopped in the middle due to the intervention of another processor. More elaborate forms of protection are needed in these circumstances.

Semaphores as Resource Counters

The use of semaphores as resource counters is best demonstrated by the circular buffer example (see Figure 1-4 for the form of a circular buffer). The circular buffer contains a number of units of a resource which are produced by task1 and consumed by task2 (assuming that the data is the resource) or is consumed by task1 and produced by task2 (assuming that the buffer locations are the resource). Therefore the definition of the resource depends on what one is doing. If data is to be placed in the buffer then obviously the number of free locations would be the resource. If data is read from the buffer then data available is the resource.

With the resource (whatever it may be) we associate a semaphore **res** and set **res.count** equal to the number of units of the resource that exist at any particular moment. During the initialisation the **res.count** value is set to the initial value of the resource – e.g. the size of the buffer for the “space available semaphore” and zero for the “data available semaphore”.

The basic structure of the tasks in the circular buffer example, where the resource is the data in the buffer is shown in Figure 4-1.

The semaphore value is being used as a counter of the resource. However, more than this is occurring because if there is no resource available then the task requesting it will become blocked, and therefore release the processor to do other jobs.

From a semaphore point of view the resource counter example differs from critical section protection in that the semaphore value can in general be greater than one and the **wait** and **signal** occur in different routines.

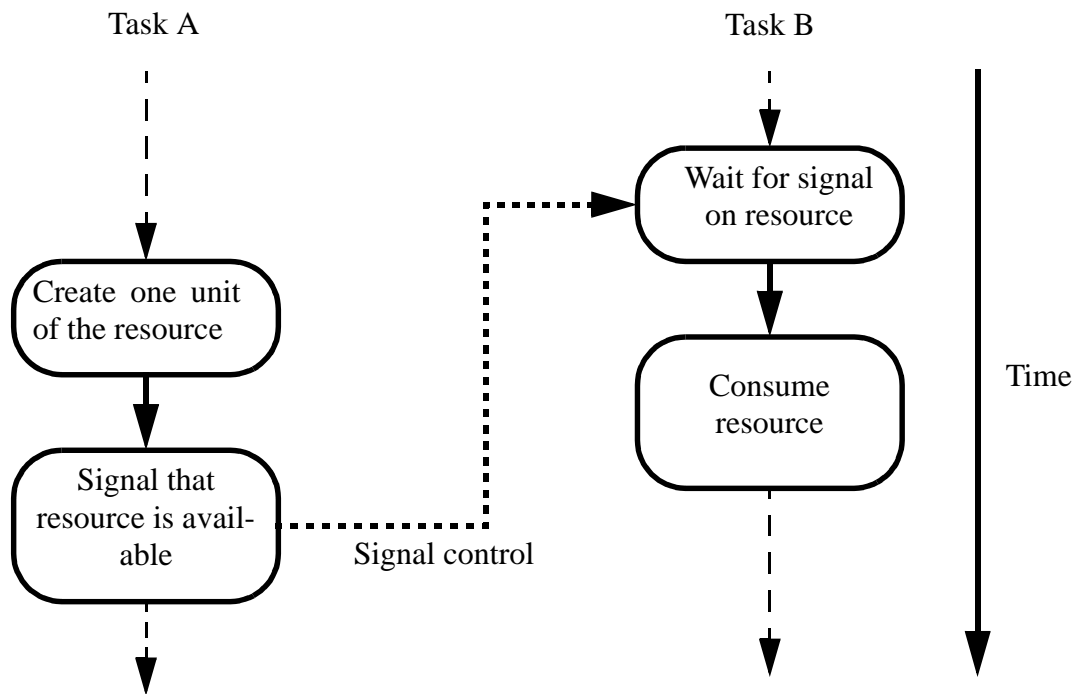


Figure 4-1: Wait and signal interaction in a producer consumer example.

Task Synchronization

The use of semaphores as task synchronization primitives differs from the resource counting and critical section protection in that there is no physical resource involved. There is an abstract resource which one may call “permission to proceed”.

One common example of the use of a synchronization semaphore is for initialization. When a real time operating system starts certain critical tasks may have to initialise variables within the system before other tasks can be allowed to access them. Because of the multitasking one can not be sure which tasks may reach the parts in their code where they are going to use these variables first. Therefore one must have a way of ensuring that the initialisation task completes initialisation before the others access the variables. There are several ways of doing this and one of them is to use semaphores to force synchronization between the tasks.

For example consider that there are three tasks which must wait on data in another task to be initialised. The following scheme could be used to organise this.

Task 1		
initialise data;		
signal (synch);		
signal (synch);		
signal (synch);		
.		
.		
Task 2	Task 3	Task 4
wait (synch);	wait (synch);	wait (synch)
.	.	.
.	.	.
.	.	.

The initial value of the **sync** semaphore should be zero.

Consumer-Producer Problem Revisited

We have already looked at the producer-consumer problem in relation to interrupt routines. We noted that the interrupt routines contained critical sections which had to be protected by disabling interrupts. We shall now look at the same problem but with tasks as the producer and consumers and the critical section protection being handled by semaphores. The communications between the two tasks is being handled by a circular buffer. The resource counting semaphores are known as **space_available** and **data_available**. In addition there are critical section protection semaphores for the critical sections within the circular buffer routines.

The basic functionality of the tasks are as follows:

Producer

```
repeat
    get data
    put data into buffer
until kingdom_come;
```

Consumer

```
repeat
    get data from buffer
    process
until kingdom_come;
```

In the above producer routine it could be considered as both a producer and a consumer in a practical situation. For example the producer routine could be waiting on data to be produced from a serial port on the machine. This data would be made available by an interrupt which would typically store the data in a buffer of its own. Once the data was placed in the buffer the interrupt routine could signal that the data is available. The producer (which in this case is acting as a consumer) which to this point was blocked on a semaphore would get the data and then pass it onto the second buffer. Note that in the buffers described below we have included two locations **used** and **free** to keep track of the number of used and free locations in the buffer. If there were an operating system command to interrogate the values of semaphore variables then these would be redundant, since the **data_available** and **space_available** semaphores do the same job.

Let us look at the code for this in more detail:

```
type
    data_type = char;
    buffer_pointer : ^buffer;
    buffer = record
        data_available : semaphore;
        space_available : semaphore;
        used_free_update : semaphore;
        buf_size : integer;
        used : integer;
        free : integer;
        get_ptr : integer;
        put_ptr : integer;
        buf : array [0..( buffer_size - 1 ) ]
    end (* record *)
```

```
(*****)
```

```
(*  
put_buffer  
This procedure allows the calling routine to put a character into the  
circular buffer. All the required semaphore operations are carried out  
transparently to the calling routine. If there is no room in the buffer  
then the calling task becomes blocked.  
NB. This procedure is reentrant and can be used for all circular buffers  
in the system.  
Parameters : - character to be placed in the buffer  
              - pointer to the buffer structure.  
)
```

```
procedure put_buffer ( data : data_type; buf_ptr : buffer_pointer );  
begin  
    with buf_ptr^ do  
        begin  
            wait ( space_available );  
            (* now put the data in buffer *)  
            buf [ put_ptr ] := data;  
            put_ptr := ( put_ptr + 1 ) mod buf_size;  
            wait ( used_free_update );  
            used := used + 1;  
            free := free - 1;  
            signal ( used_free_update );  
            signal ( data_available );  
        end;  
    end; (* put_buffer *)
```

```
(*****)
```

```
(*  
get_buffer  
This function retrieves data from the circular buffer. As with the put  
routine all the semaphore handling is done within the routine and is  
transparent to the caller. If there is no data available in the buffer  
then the calling task becomes blocked. The function is written so that  
it can be used for all circular buffers in the system.  
Parameters : - pointer to the buffer  
Returns : - data from the buffer  
)
```

```
function get_buffer ( buf_ptr : buffer_pointer ) : data_type;  
var temp_data : data_type;  
begin  
    with buf_ptr^ do  
        begin  
            wait ( data_available );  
            (* now get data from buffer *)  
            temp_data := buf [ get_ptr ];  
            get_ptr := ( get_ptr + 1 ) mod buf_size;  
            wait ( used_free_update );  
            used := used - 1;  
            free := free + 1;  
            signal ( used_free_update );  
            signal ( space_available );  
        end;  
    end;
```

```
        get_buf := temp_data;
    end; (* get_buf *)

(*****)

(*
buf_used
This function returns the number of used locations in the circular buffer
whose pointer is passed in.
Parameters : - pointer to the circular buffer
*)

function buf_used ( buf_ptr : buffer_pointer ) : integer;
    var temp_used integer;
    begin
        with buf_ptr^ do
            begin
                wait ( used_free_update );
                temp_used := used;
                signal ( used_free_update );
            end;
            buf_used := temp_used;
        end; (* buf_used *)
    (*****)

(*
buf_free
This function returns the number of free locations in the circular buffer
whose pointer is passed in.
Parameters : - pointer to the circular buffer
*)

function buf_free ( buf_ptr : buffer_pointer ) : integer;
    var temp_free integer;
    begin
        with buf_ptr^ do
            begin
                wait ( used_free_update );
                temp_free := free;
                signal ( used_free_update );
            end;
            buf_free := temp_free;
        end; (* buf_free *)
```

The above procedures are a set of general purpose circular buffer access routines. They can be used by any number of producer-consumer tasks in an operating system environment. Notice that the routines use three semaphores. In each routine one semaphore is used for resource counting and the other for critical section protection. Note that the **wait** and **signal** procedures for a particular semaphore are called from a different task.

The actual procedures would therefore be:

PRODUCER

```
const kingdom_come = false;
new ( buf_ptr );
repeat
    data := get_data;
    put_buffer ( data, buf_ptr );
```

```
until kingdom_come;
```

CONSUMER

```
const kingdom_come = false;
repeat
    data = get_buffer ( buf_ptr );
    do something with data;
until kingdom_come;
```

Note that there must be some way of letting the consumer task know what the buffer pointer value is. This could be done by calling a function in the task that creates the buffer which will return a pointer to it.

Note that in the above example the **used** and **free** locations in the buffer structure are protected as a critical section by a semaphore. For efficiency reasons a **disable** and an **enable** would probably be used because the operations carried out in the critical section are very quick. The overhead of a **wait** or **signal** is considerable compared to this.

More Complex Synchronization Problems

Multiple unit resource allocation

The indiscriminate use of semaphores can lead to problems. Consider the case of allocating resources in blocks greater than one. For example we may wish to claim a number of locations within a buffer structure. The code for this may be written as follows:

```
procedure allocate ( n : integer );
    (* allocates n units of a resource to the caller *)
    var j integer;
    begin
        for j = 1 to n do
            begin
                wait ( sema );
                allocate one unit of resource
            end; (* for *)
        end; (* allocate *)
```

The above procedure will not work in a multitasking setting because it allows *deadlock* to occur. Consider the case where the buffer has four free locations and we have two tasks wishing to get 3 locations. If the following code sequence occurs then both tasks will be blocked on getting the last resource but each will not be able to release a resource.

TASK 1		TASK 2
allocate one unit		
allocate one unit	→ TASK SWITCH →	allocate one unit
		allocate one unit
← SEMAPHORE BLOCK TASK SWITCH ←		task 2 blocked
task 1 blocked		

Note that all of the resource has been consumed but neither task has satisfied its request.

The above deadlock solution can be fixed in several ways. A **wait** primitive can be defined which allows multiple units to be allocated as an indivisible operation. If not enough units are available then the calling task will become blocked until they do. This solves the problem because the deadly embrace of two tasks claiming the resource at the same time is broken. This solution can also be simulated using conventional semaphores by making the claiming of the resource a critical section and protecting it with another semaphore. The code below shows

this:

```
procedure allocate ( n : integer );  
  var j integer;  
  begin  
    wait ( mutual_exclusion );  
    for j := 1 to n do  
      begin  
        wait ( sema );  
        allocate one unit of resource  
      end; (* for *)  
    signal ( mutual_exclusion );  
  end; (* allocate *)
```

Dining philosophers problem

The dining philosophers problem is a classic problem given in most books on operating systems to demonstrate how obvious solutions using semaphores can lead to deadlock situations.

The problem is usually stated as follows. Five philosophers are sitting around table alternately thinking and eating at random times. They do not necessarily eat at the same time of day. Therefore the philosophers behave the same as a number of tasks running concurrently. There are five forks on the table, one between each two philosophers. There is a single bowl of spaghetti in the middle of the table and a philosopher has to have two forks to eat it (see Figure 4-2). The forks can only come from either side of a philosopher. Therefore to get two forks a philosopher may have to wait for a neighbour to finish with one or both of them.

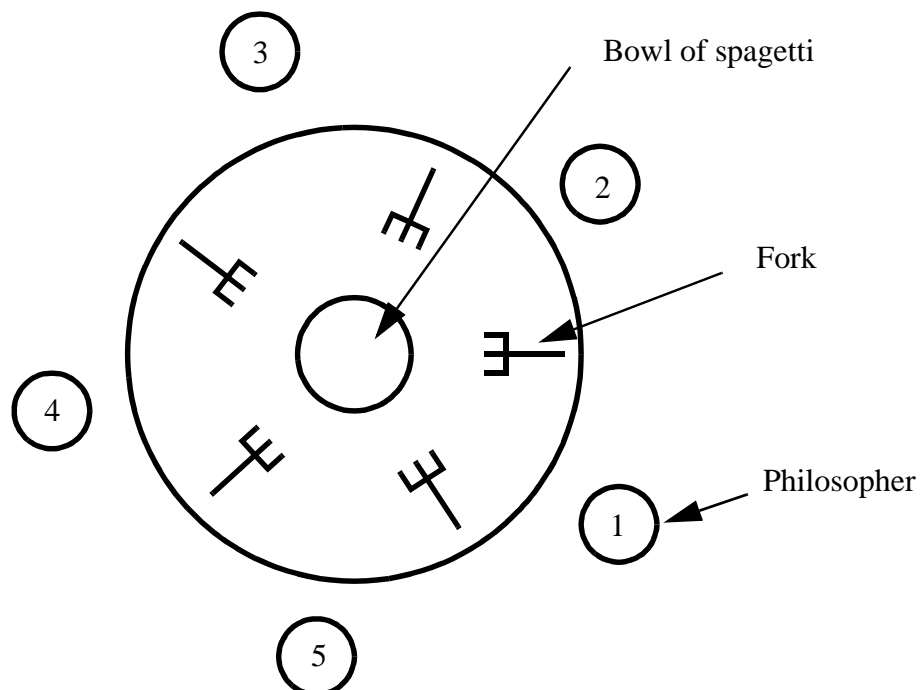


Figure 4-2: The dining philosophers problem

The obvious solution to this problem is programmed as follows using semaphores.

```
procedure pick_up_two_forks ( j: philosopher_number );
begin
    wait ( fork [ j ] );
    pick up left fork;
    wait ( fork [ ( j + 1 ) mod 5 ] );
    pick up right fork;
end (* pick up two forks *)
```

The problem with this solution is that if all the philosophers simultaneously pick up the left fork then none of the philosophers can pick up the right fork and therefore a deadlock occurs.

In a similar way to the last problem one solution is to make the picking up of the two forks indivisible.

```
procedure pick_up_two_forks ( j : philosopher_number );
begin
    wait ( control );
    wait ( fork [ j ] );
    pick up left fork;
    wait ( fork [ ( j + 1 ) mod 5 ] );
    pick up right fork;
    signal ( control );
end; (* pick_up_two_forks *)
```

The above procedure ensures that a philosopher has picked up two forks (and can therefore eat) before another philosopher can pick up forks. If a philosopher becomes blocked in the middle of this process then no other philosophers can pick up forks until the first philosopher becomes unblocked. Multiple philosophers can be eating at once but only one can be picking up forks. A better solution is to make the initial value of the **control** semaphore equal to 4. This will allow up to 4 philosophers to execute the procedure simultaneously but there will always be one philosopher not competing for a fork. This means that a deadlock will not occur because at least one of the four competing philosophers will be able to get both forks.

The above algorithm for the dining philosophers can be solved with only 5 semaphores if one of the philosophers picks up a fork on the opposite side from all the others (i.e. one is left handed whilst all the others are right handed). This then means that at least one can pick up two forks. Figure 4-3 below shows one scenario under this condition. In this diagram philosopher #1 picks up the fork to his right at time T1. There is then a task switch to philosopher #2. Because there is no longer a control semaphore, then philosopher #2 is not blocked and he is able to pick up a fork to his right at time T2. This pattern continues with philosopher #3 and #4. Philosopher #5 is left handed, so he attempts to pick up the fork to his left at time T5. He becomes blocked because fork #4 has been picked up by philosopher #4. Since philosopher #5 is blocked then he is unable to pick up fork #5, and consequently philosopher #1 is able to get this fork and is able to eat.

Readers and Writers problem

The readers and writers problem is far more complicated as compared to the dining philosophers, in addition to being of more practical use. The problem arises whenever there is a shared database which is being interrogated and written to by several tasks. The obvious solution to the problem is to block all writes whilst there is any read in progress and to block both reads and writes when a write is in progress.

The above protection can be obtained by using critical section protection based on semaphores

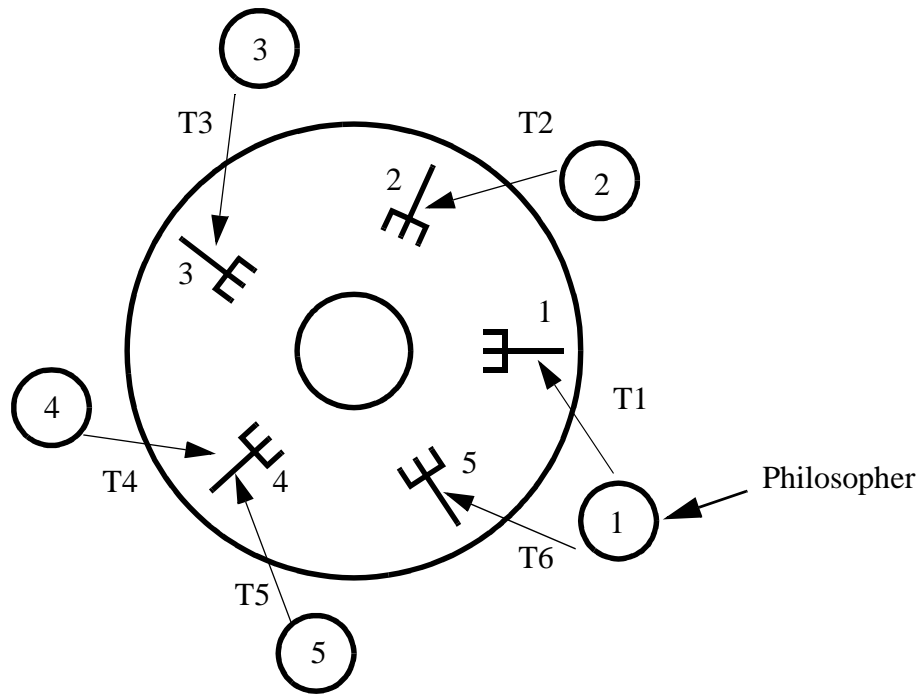


Figure 4-3: Dining philosophers solution with only five semaphores

which would ensure that only one task could access the data base at any point in time. This is a poor solution because tasks could remain blocked for long periods of time and any reasonable size system with many people accessing the data base would give very poor performance under these conditions. As mentioned in the previous paragraph the best solution will allow simultaneous readers but will allow exclusive access to a writer.

The solution below is just one possible solution to this problem which uses three semaphores. There are a variety of solutions possible, the main differences being how fair it is to both readers and writers.

```

var
  no_writers, io_permission : semaphore;
  readers : record
    in_progress : integer;
    update_permission : semaphore;
  end;

(* All semaphores have an initial count = 1 and the semaphore queue is
initially empty. The initial value of readers.inprogress is 0*)

procedure write;
begin
  wait ( no_writers );
  wait ( io_permission )
  carry out the actual write operation
  signal ( io_permission );
  signal ( no_writers );
end; (* write *)

procedure read;
begin

```

```
(* first the funny bit to make access fair for writers - i.e. not
to lock them out for too long.*)
wait ( no_writers );
signal ( no_writers );
(* now the protection to update the number of readers accessing
the data base *)
wait ( reader.update_permission );

if readers.in_progress = 0 then
    wait ( io_permission );
readers.in_progress = readers.in_progress + 1;
signal ( readers.update_permission );

now carry out the actual read operations;

(* now decrement the number of readers in progress and check
to see if there are any still left. If not then signal that
all io is finished *)

wait ( readers.update_permission );
readers.in_progress = readers.in_progress - 1;
if readers.in_progress = 0 then
    signal ( io_permission );
signal ( readers.update_permission );
end; (* read *)
```

Now let us analyse the operation of these routines. To simplify the analysis let us firstly neglect the **no_writers** semaphore. If this were the solution proposed then it would not allow any corruption of the data base. The provision of the **no_writers** semaphore is for fairness. We shall have a look at this aspect soon.

The operation of the **write** routine is fairly self evident. If there are any I/O operations in progress (i.e. can be due to readers or writers) then the write routine is blocked until they are finished. Similarly if there is a **write** in progress then any read which comes along will be blocked on the **io_permission** semaphore.

The problem with this implementation is that it is unfair to the writers. Consider the following execution sequence. Time is vertical down the page.

```
reader 1
reader 2
writer 1 – becomes blocked on io_permission
reader 3
reader 4
more readers arrive
.
.
the readers all have to finish
signal (io_permission)
writer 1 now becomes active
```

The problem is that the writer is delayed until all readers have finished. It is not serviced in the order that a request is made to access the data based. This is not fair to the writer and could cause undue delays in updates to the data base.

Now let us consider the influence of introducing the **no_writers** semaphore. There are two important scenarios to consider:

- (i) there are no writers
- (ii) there are writers and readers

If there are no writers then on each call to the **read** routine a **wait** and then a **signal** occurs, obviously with no net result. In fact it appears that it is pointless in having this code. However let us now consider the situation where there is a mixture of readers and writers.

```
reader 1
reader 2
writer 1 – causes no_writers = 0; blocked on io_permission
reader 3 – blocked on no_writers
reader 4 – blocked on no_writers
readers 1 & 2 unblock writer 1
write 1 unblock reader 3
reader 3 unblocks reader 4
```

The first two readers simply fall through the **no_writers** semaphore and then claim the **io_permission** semaphore. Then writer 1 comes along and is blocked on the **io_permission** semaphore. Next readers 3 and 4 come along and they are blocked on the **no_writers** semaphore which was claimed by the writer. It is this which prevents the readers from having precedence over the writer – the writer must finish before the extra two readers can continue (when the writer signals the **no_writers** semaphore).

Notice the function of the **wait** and **signal** pairs in the read routine. If the first task on the top of the blocked queue for **no_writers** is a reader then when it becomes unblocked it immediately signals the next reader. Therefore a succession of readers can begin executing. If one of the waiting tasks is again a writer then when it runs it will immediately become blocked again (on **io_permission**) and no more readers will be able to run concurrently until the current lot have finished and the blocked writer runs. *Therefore the wait and signal pairs effectively operate like a broadcasting system.* This mechanism can be used to activate multiple tasks from a single event without having to know *a-priori* how many tasks are blocked. This is an example of how semaphores can be used to communicate between tasks.

The discussion so far has concentrated on conventional implementations of semaphores. There are a number of enhancements which can be made to the basic semaphore to make it more useful in real applications.

Variations on the semaphore

Logic semaphores

It is often desired to have a task continue on a logical combination of other occurrences. For example one may wish that a task execute if (A and B) or C occurs. The semaphores presented so far only allow a task to continue if some condition is true. It would probably be possible to build semaphores which carried out logical operations, however in practice this is virtually never done. The reasons are that the semaphore routines would become too complex, and it fairly easy to simulate the common logic conditions with conventional semaphores.

For example the 'AND' function is very easy to simulate as follows:

```
wait ( A );
wait ( B );
```

The above simulates that the A AND B condition has to occur before the task executing the code could proceed.

The A OR B case is not quite so easy to implement. If the A and B conditions are mutually exclusive (that is an XOR function) then the solution degrades to a simple solution:

```
wait ( AorB );
```

where **AorB** is the name of the semaphore, and the following would be executed in any of the tasks which could cause A or B to become true:

```
signal ( AorB );
```

One of the problems with this solution is that one would have to be sure that the A and B conditions are truly mutually exclusive or else the semaphore would end up being incremented too often resulting in an incorrect value.

The more general case which allows the normal OR case of the A and B conditions is more of a problem. One solution involves checking flags to see if the other condition in the OR function has occurred upon a condition becoming true. If so then no signal is performed, because the other condition will have already done it. This type of checking requires a second semaphore for critical section protection of the flag variable (i.e. **A** and **B** in the example below).

```
wait ( guard );
A := true;
if not B then
    signal ( AorB );
signal ( guard );
```

Notice that the **guard** semaphore is required because if a task switch occurs after A becomes true, which results in B becoming true, then neither of the tasks will carry out the signal.

Timed semaphores, sleeping tasks and simulated task suspension

The semaphore primitives discussed so far have not included any concept of timing in them. A consequence of this is that deadlock situations are fatal in the sense that the blocked task can never continue and there may be no notification of the deadlock condition. In fact deadlock conditions often reflect themselves in an apparent complete system crash. One way of rectifying this problem is to include the concept of maximum time that a task can be blocked on a semaphore. If a **signal** occurs before this time has elapsed then the semaphore behaves as discussed in the previous section. However if the time out period elapses before a **signal** occurs then a return from the **wait** occurs. The task which called the semaphore can distinguish between these two situations because the **wait** procedure has been modified to become a function which returns a code to indicate the reason for the return. The return code can then be used to decide on whether an error handler needs to be called.

Timed waits have another use in an operating system which is largely unrelated to their original use as a traditional semaphore. They can be used as a way that a task can put itself to sleep for a specified number of clock ticks. In order to do this a semaphore needs to be created with an initial value of zero. This semaphore would not be used by any other task – i.e. there will be no signals on this semaphore. Therefore any call to a **timed_wait** will result in the task becoming blocked for the specified time period. This feature can be used to replace the **task_to_sleep** primitive offered in many real time operating systems.

The function call to a timed wait primitive would typically be:

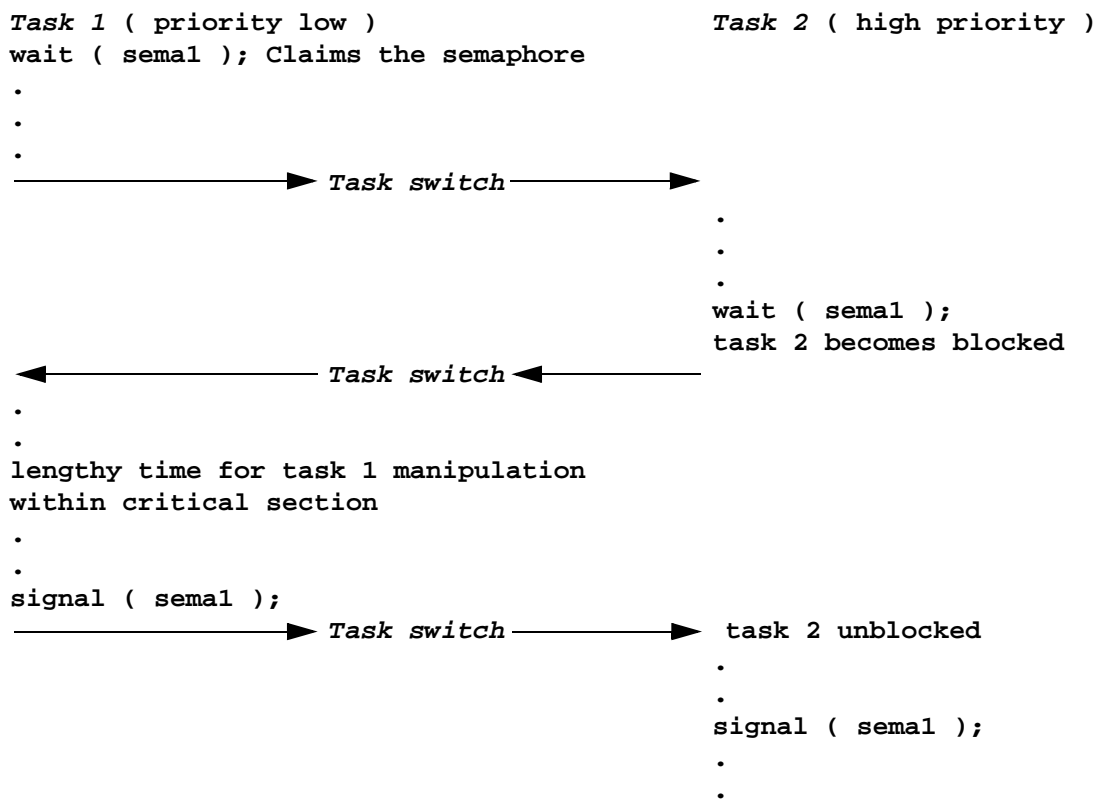
```
function timed_wait ( sema : semaphore; timeout_value :
                    integer ) : return_code;
```

Semaphores can be used for another use besides those cited so far. A suspended task is a task

which will not be run regardless of what other state it is in. That is may be on a ready queue when suspended or it may be on a semaphore queue, but it will not run again whilst suspended. On way of achieving this is to include a special bit in the task control block for tasks which is used to indicate whether or not a task is suspended. This bit is checked by the dispatcher and if the task is suspended then the task is ignored and another task is looked for to run. Instead of having a special provision for this feature a semaphore can be used to simulate it. It is dangerous for a task to be asynchronously suspended by another task since the suspending task does not know what resources the suspended task may have claimed at the suspension time. Therefore a suspend is most safely done by executing the suspend operation within the task to be suspended – i.e. it suspends itself. Hence it would be just as easy for the task to block itself on a semaphore. The unsuspend operation would then be a signal by another task.

Semaphores and priorities

The normal operation of semaphore blocked queues is FIFO based (first in first out). For tasks of equal priority this is the equitable way of organising the queue. However this approach can cause timing problems when tasks of vastly different priorities are using a common resource. To illustrate the problem consider the situation described below.



etc., etc.

Notice that in the above execution sequence because the low priority task claimed the semaphore the high priority task was blocked waiting for a low priority task to complete manipulation of the critical section. The length of time that the low priority task will take to do this and release the resource depends on what other tasks are runnable in the system at any point of time. The effect of this is to lower the priority of the high priority task 2 to that of the low priority task 1. The priority chosen for task 2 has effectively been temporarily lost.

The case illustrated above is for the case of two tasks. If there are a number of low priority tasks which have become blocked on a semaphore before the high priority task then the FIFO mechanism will mean that the high priority task will have to wait until all the low priority tasks

have released the semaphore.

These difficulties can be overcome by implementing the queuing on priority order instead of FIFO order and creating the concept of a dynamic and static priority. The priority ordering of the blocked queues means that the highest priority task is on the top of the queue and will be the next to be unblocked upon a signal. This overcomes the problem mentioned above when multiple low priority tasks may be ahead of a high priority task in the queue. The original problem of the low priority task claiming the semaphore and stopping a high priority task from running is solved by temporarily increasing the priority of the low priority claiming task to that of the high priority task waiting on the semaphore – i.e. there is a dynamic priority change. The priority of the low priority task is returned to its normal priority, or static priority when it releases the semaphore.

Other synchronization primitives

The discussion throughout these lecture notes have concentrated on semaphore primitives. However these are not the only primitives which can be used.

Event primitives

An event flag is a boolean variable which is associated with a queue of tasks. When the boolean variable becomes true then all the tasks waiting on the ‘event’ become unblocked and become runnable. The low level procedures in an event flag driven system are typically:– **wait_for_event**, **set_event_flag**, **clear_event_flag** and **test_event_flag**. It is possible to simulate event primitives with semaphores and vice-versa. In fact if one includes a semaphore multiplier in the semaphore structure then it is possible for a semaphore to unblock a fixed number of tasks with one signal. As each task is added to the queue then the multiplier is incremented so that when a signal arrives they are all unblocked.

The event flag is an example of another low level procedure. However there are higher level primitives which can also be used for task synchronization. An example of this is the message passing synchronization method.

The basic operations in a message passing system are obviously **send_message** and **receive_message**. The **send_message** is analogous to **signal** and the **receive_message** to **wait**. However the difference is that information of a more complex kind can be sent in the messages as well. In addition this scheme can also be used across a loosely coupled multiprocessor network to send messages and give synchronization.

Monitors

The undisciplined use of semaphores is rather susceptible to error – a programmer can easily place **wait** and **signal** operations in the wrong place, or even omit them altogether. The result of such errors is that data will most probably be corrupted. To overcome such problems there have been a number of proposals for programming language constructs which force the programmer to declare shared data and resources explicitly and then enforce mutually exclusive access to such shared objects. One of the most influential and widely used constructs is the *monitor*.

A monitor consists of:

- (i) the data comprising the shared object.
- (ii) a set of procedures which can be called to access the object.
- (iii) a piece of program which initializes the object (this program is executed only once,

when the object is created).

Even though the monitor concept was first proposed in 1974 (by Hoare) it has concepts of information hiding and object orientation similar to modern object oriented languages such as C++.

Let us consider again the consumer-producer problem. The buffer for passing data items between the two tasks might be represented by a monitor consisting of:

- (i) the buffer space and pointers.
- (ii) two procedures **deposit** and **extract** which can be called by tasks to place an item in the buffer or to remove an item from it.
- (iii) a piece of program to create and initialize the buffer and its pointers.

The compiler for a language incorporating monitors must ensure that access to a shared object can be made only by calling a procedure of the corresponding monitor. This is usually guaranteed by the normal scoping rules that are present in most modern block structured languages. The compiler must also ensure that the procedures within the monitor are implemented as mutually exclusive critical sections. For example in the buffer example the **deposit** and **extract** procedures are mutually exclusive. It should be emphasized that the mutual exclusion is now a responsibility of the compiler and not the programmer. Responsibility for other forms of synchronization remain with the programmer. For example, in the buffer example other synchronization primitives are required in order to prevent tasks from attempting to deposit data into a full buffer.

The synchronisation primitives suggested by Hoare, and adopted in most monitor implementations, are not **wait** and **signal** operations on semaphores, but similar operations on objects called *conditions*. Like a semaphore, a condition has two operations which can be applied to it – one is to delay a process and the other to resume it. These operations will be referred to as **delay** and **resume**. The definitions of **delay** and **resume** are:

delay: suspend execution of the calling process.

resume: resume execution of some task suspended after a **delay** on the same condition. If there are several such tasks, choose one of them; if there is no such task then do nothing.

The **delay** operation releases exclusion on the monitor. If this were not done then no other task would be able to enter the monitor to perform a **resume**. Similarly **resume** releases exclusion, handing it to the task which is being resumed (assuming there is a task to be resumed). Although conditions and semaphores are being used for similar purposes, the following differences are significant:

- (i) A condition, unlike a semaphore, does not have a value. When a **delay** is executed the result is that the calling task is placed on a queue. Similarly the **resume** removes a task blocked on the queue for a particular condition.
- (ii) The execution of a **delay** always has the effect of blocking the calling task, unlike a semaphore which only blocks the calling task when the semaphore value goes to zero. The **resume** only has an effect when there is a task blocked on the condition which is the subject of the **resume**, else it does nothing. The **signal** always has an effect – it removes a task from the semaphore queue or increments the value of the semaphore.

As an example of the use of conditions for synchronisation, here is a more detailed look at the buffer monitor:

```
var buffer : monitor
```

```
begin
    var B : array[0..N-1] of item;
    nextin, nextout: 0..N-1;          (* buffer pointers *)
    nonfull, nonempty : condition;    (* synchronisation primitives *)
    count: 0..N                       (* number of item in the buffer *)
procedure deposit (x :item);
begin
    if count = N then
        delay ( nonfull );           (* avoid overflow *)
    B[nextin] := x;
    nextin := ( nextin + 1 ) mod N;
    count := count + 1;
    resume ( nonempty );              (* resume any waiting consumer *)
end;

procedure extract ( var x : item );
begin
    if count = 0 then
        delay ( nonempty );          (* avoid underflow *)
    x := B[nextout];
    nextout := ( nextout + 1 ) mod N;
    count := count - 1;
    resume ( nonfull );              (* resume any waiting producer *)
end;

(* initialisation section *)
nextin := 0;
nextout := 0;
count := 0;
end;
```

A practical semaphore structure

Up to this stage the semaphore structures shown have been very simple in nature, consisting of a semaphore value location and a pointer to a queue of blocked tasks. In practice semaphore structures can be much more complex than this. As an example the semaphore structure for the current version of the University of Newcastle Operating System (UNOS) is as below (in 'C' type notation):

```
unsigned int semaphore_value;
unsigned int semaphore_multiplier;
struct tsk_ctrl_blk* tcb_ptr;
unsigned int semaph_bgn_q_index;
void ( *semaph_queue_handler ) ();
```

'C' notation was used here because UNOS is written entirely in 'C'. However for those uninitiated in 'C' the components of the structure can still be readily understood as follows:

- (i) **unsigned int semaphore_value** – as the name implies this is the normal semaphore count as in the semaphores used to date.
- (ii) **unsigned int semaphore_multiplier** – this is a new term which enables a signal to cause multiple signals. For example if the multiplier value is set to 4 then a signal will cause 4 blocked tasks (if there are 4 blocked tasks) to become unblocked. Therefore the semaphore can be simply made to simulate an event. If there is only a single task blocked then the value of 4 would cause that task to run 4 times.
- (iii) **struct tsk_ctrl_blk* tcb_ptr** – this is a pointer to the task control block of

the task which has claimed the semaphore. This is required because of the use of dynamic priorities in UNOS. If a `wait` occurs on a semaphore then the task which has claimed the semaphore may need to have its dynamic priority adjusted.

- (iv) **unsigned int semaph_bgn_q_index** – this contains an index into a table of pointers. This index is the pointer to the beginning of the queue of blocked tasks waiting on this semaphore.
- (v) **void (*semaph_queue_handler)()** – this is a pointer to the handling routine which carries out the queuing and dequeuing operations on blocked queues. This option allows both priority queue ordering or fifo queue ordering depending on the handler pointed to by this variable. When a semaphore is created the user can stipulate which type of queuing mechanism will be used.

This is obviously one example of a semaphore structure and is totally implementation dependent. This example was given to indicate what other information may be included in the structure.

Deadlock and Starvation

The reader has already been exposed to the concept of deadlock in some examples already presented. In this section we shall look at this problem and the related problem of starvation in a more general context.

Starvation occurs when a task is runnable but it does not get the processor. This can occur if there are high priority tasks in the system that completely hog the processor, not allowing lower priority tasks to get a share of the processor. Therefore starvation differs from deadlock in that it does not involve semaphores, but is related to task priorities.

Deadlock is defined as a permanent blockage of a set of tasks that are competing for system resources or communicating with each other. If one considers two tasks running, and both these tasks obtain one resource, and then request another resource. Both resources are being requested by both tasks. This situation has the potential for deadlock. Consider the diagram in Figure 4-4 [4]. This diagram shows two tasks as they evolve over time. The horizontal axis shows the time of task 1, and the vertical axis task 2. Notice the order of the requests for the resources. Firstly T_1 requests and obtains R_1 . Then T_2 requests and obtains R_2 . Next T_1 attempts to obtain R_2 but cannot get it because T_2 has it. Therefore T_1 becomes blocked and T_2 runs again. It then requests R_1 , but cannot get it as T_1 has this resource. Therefore both tasks are now blocked waiting on a resource that the other task already has. The tasks will remain in this state unless the system has some way of getting out of deadlocks.

Resource types

Reusable resources

Reusable resources are resources that can be used by only one process at a time, but are not depleted by that use. The resources acquired by a process are later released and can be reused by another process. Examples are: processors, I/O channels, semaphores, data structures, files etc.

Consider the following example of deadlock involving two tasks and two reusable resources:

Clearly under certain task switch scenarios the above sequence of requests can result in a deadlock situation. If the task switch occurs at the time indicated in Figure 4-5 then Task 1 and Task 2 cannot proceed.

Another example of this type of problem is a memory allocation problem. This problem is vir-

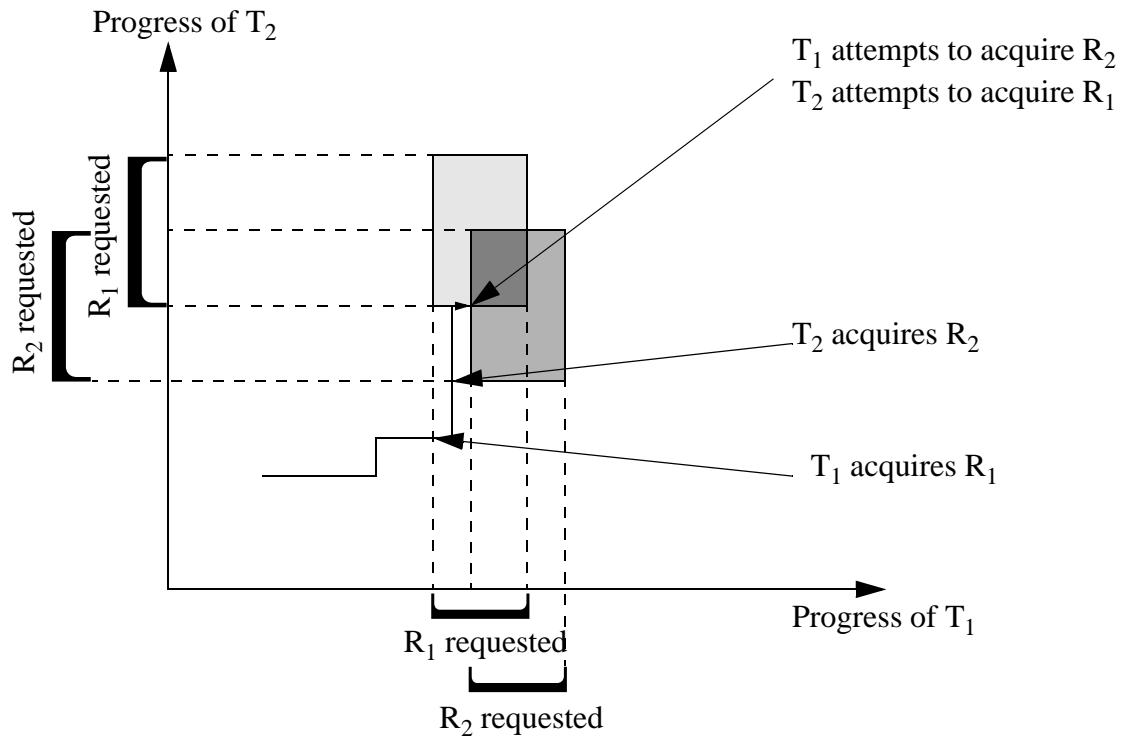


Figure 4-4: Progress of two tasks that eventually deadlock on two shared resources

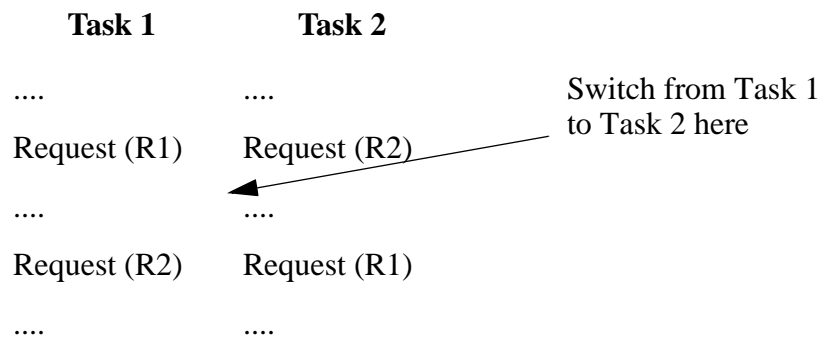


Figure 4-5: Reusable resource deadlock scenario.

tually the same as the multiple resource allocation problem that we considered in the section titled “Multiple unit resource allocation”. In the example that was presented in that section the solution was to make the resource allocation indivisible. Consider the situation shown in Figure 4-6, where we are allocating memory from a pool of 200K bytes of memory. If there is a task switch at the point indicated then there will be a deadlock situation since both processes have claimed memory but cannot get all the memory they require. The solution of the section titled “Multiple unit resource allocation” can be applied to this problem, but it is not an elegant solution. It is implemented by forcing the tasks to block allocate a section of memory that is large enough to supply all the memory requests that it will make. The smaller requests are then

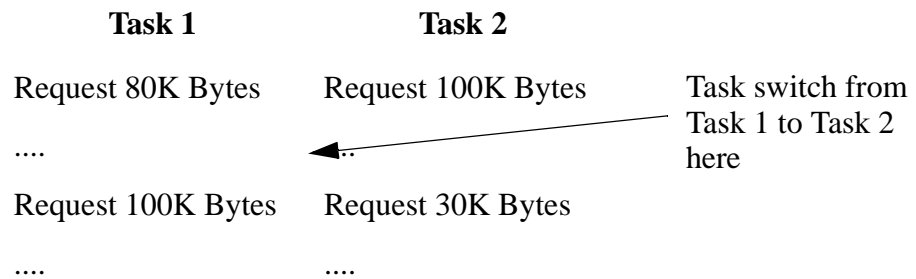


Figure 4-6: Memory allocation deadlock

sub-allocated from this local memory pool. This approach then ensures that at least one of the tasks will have its memory demands satisfied.

An alternative approach (and one that is often used in practice) is to use virtual memory to implement the memory system. This ensures that the pool of memory is larger than the sum total of requests that will be made in any practical situation, and consequently there is not possibility of the situation in Figure 4-6 occurring.

Consumable resources

Consumable resources are resources that are produced by a task, and then destroyed by a task. Often there is no limit on the number of consumable resources that can be produced. Examples of consumable resources are: characters in a buffer, interrupts, signals, information in a buffer etc. As can be seen from these examples the information produced in a producer/consumer application is typical of consumable resources. Consider the situation in Figure 4-7. This shows two tasks sending and receiving messages via a system communication mechanism. Assuming that the receive is a blocking primitive (which it usually is), then system would deadlock waiting on messages to be sent.

One might be tempted to say that the situation in Figure 4-7 is a programming error. However, in complex systems with many interacting tasks it is very easy for this situation to inadvertently occur. Therefore strategies need to be developed so that such situations can be identified.

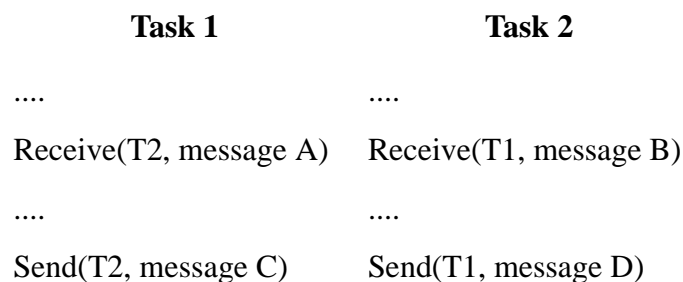


Figure 4-7: Consumable resource induced deadlock

The conditions for deadlock

We shall develop necessary and sufficient conditions for deadlock. If the following policy con-

ditions exist then deadlock may occur:

- (i) *Mutual exclusion*: Only one task may use a resource at a time.
- (ii) *Hold-and-wait*: A task may hold allocated resources while awaiting the assignment of other resources.
- (iii) *No forced resource release or no preemption*: No resource can be forcibly removed (or preemptively removed) from a task holding it.

The conditions above are generally the conditions that exist in most operating systems, not because the desire is too produce deadlock situations, but because they are required to allow the system to function correctly from other points of view. It should be noted that the above conditions are only necessary conditions for deadlock, but they are not sufficient conditions. The fourth condition is actually a consequence of the first three conditions.

- (iv) *Circular wait*: This is also known as a deadly embrace (see Figure 4-8). A *closed chain* of tasks exists, such that each task in the chain holds at least one resource needed by the next task in the chain.

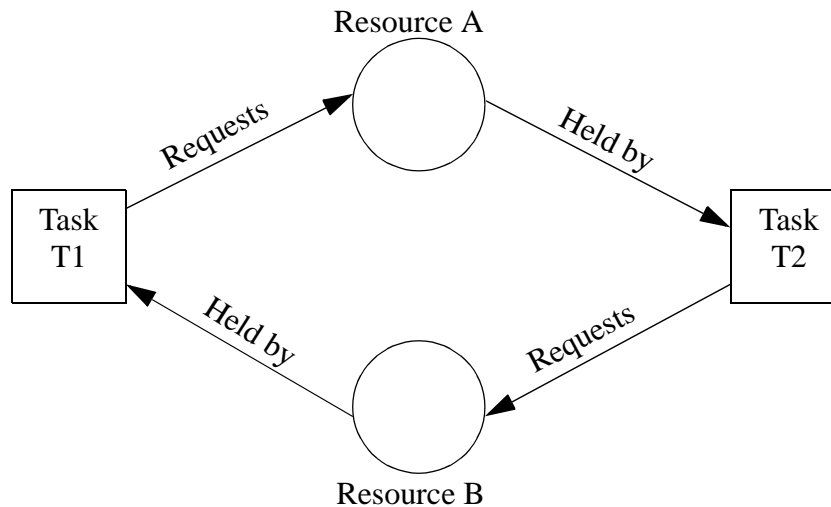


Figure 4-8: Circular wait on resources

Techniques to handle deadlock

There are several different strategies to handle deadlock. It should be noted that all these techniques have their advantages and disadvantages. There are also differences between the philosophies of the various approaches.

Deadlock prevention²

The objective of this approach is to prevent the conditions for deadlock from occurring *a priori*. The indirect methods prevent one or more of the necessary conditions for deadlock, whilst the direct approach actually prevents the circular wait condition. Let us examine each of these:

- ***Mutual exclusion***: Usually this condition cannot be relaxed. Mutual exclusion generally has to be supplied in an operating system environment to allow resources to be shared.

2. Note that there critical section protection techniques using different synchronization primitives which make deadlock impossible due to deadly embrace. See Section “Task Synchronisation” on page 5-40.

- **Hold-and-wait:** This can be prevented by requiring that a task request all the resources that it requires at one time. Clearly this can result in a task being held up for considerable periods of time, until all the requested resources become available. In addition, once the resources have been granted they may be denied to other tasks for long periods of time, even though the holding task is not using the resources.
- **No forced resource release (no preemption):** If this condition is relaxed then the resource allocation strategy must be changed. For example if a task is denied a requested resource, then it must release all the resources that it holds and then re-request all the original resources together with the additional resource. An alternative is that if a task requests a resource held by another task, then the operating systems requests that the holding task release the resource. Note that this strategy can only be applied to a resource whose state is easily preserved and restored later (e.g. the processor).
- **Circular wait:** There are two main techniques for preventing circular waits from developing. One obvious technique is to make sure that a task can only request one resource at a time. If the task already has a resource when it makes a second request it must release the first resource. One can see a number of potential problems with this approach. For example how would one handle sending a file to a printer, where one would have to request a file resource and a printer resource simultaneously.

A second approach is to order all the resources in the system with a numerical ordering. Therefore R_i precedes R_j if $i < j$. A task can only obtain a resource if its number is greater than any resources that it currently holds. To see why this works consider the two task based circular wait. If task 1 requests and gets R_i and then requests R_j and task 2 acquires R_j and requests R_i . This situation would imply that R_i precedes R_j and R_j precedes R_i , which clearly cannot be true. This is implemented in practice by ensuring that the resource requests in a task follow this ordering principle. For example, task 1 requests resources R2, R4, R6, in this order. Task 2 requests R1, R4, R6, in this order. One can see that either task 1 or task 2 will obtain R4 first, and therefore the other task will be blocked, until the first releases the R4.

Note that circular wait prevention may be inefficient in that it slows down the allocation of resources to tasks.

Deadlock detection

This involves the detection of deadlock conditions *after* the deadlock has occurred and requires an algorithm that can detect cycles in directed graphs. Checks for deadlocks can be made as frequently as each allocation of a resource, or less frequently based on some system configurable time. The former has the advantage that detection occurs quickly and the algorithm is simple as it is based on incremental changes to the systems state, but on the down side it can consume considerable processor time.

Once deadlock has been detected the remaining question is what to do about it. Possible approaches are:

- (i) Abort all deadlocked tasks. This is the most common strategy.
- (ii) Rollback deadlocked tasks to some predefined checkpoint and then restart all tasks. The odds are that the deadlock will not occur again because of the asynchronous nature of task execution.
- (iii) Successively abort deadlocked tasks until deadlock disappears.
- (iv) Successively preempt (release) resources until the deadlock disappears.

Deadlock avoidance

Deadlock avoidance is subtly different from deadlock prevention. Deadlock prevention was based on preventing one of the four conditions for deadlock from occurring. This process usually involved placing restrictions on how resources were allocated, which in turn impacted on the system performance. Deadlock avoidance does not place any restrictions on resource allocation, but instead tries to determine if the resource allocation is going to potentially produce deadlock at some future time. The approach taken is:

- (i) Do not start a task if its demands might lead to deadlock.
- (ii) Do not grant an incremental resource request to a task if this allocation might lead to deadlock.

Task initiation denial:

Basically this policy states that a new task can only start if the number of each of the resources in the system is greater than or equal to the worst case claim on them from the existing tasks plus the claims from the task to be created. If this is not the case then the task cannot be created as there will be the potential for deadlock.

This is hardly optimal as it is taking the worst case requirement for the resources for each task and then says that these are all requested at the same time. For more information on this topic refer to Stallings.

Resource Allocation Denial:

This strategy is referred to as the banker's algorithm because of its similarity to the behaviour of a banker with a limited amount of money to loan and a number of customers. In the banking analogy it is assumed that each borrower (which is analogous to a task) has a line of credit to a certain value. Furthermore, it is assumed that a borrower may not pay back the loan until the full amount allowed in the line of credit has been borrowed. The borrower may borrow incremental amounts at different times up to their total credit. Therefore, the banker may not allow an incremental increase in the amount loaned to a certain borrower (still within their line of credit) if that means that there is not enough money to allow the complete borrowing requirement of other customers. This strategy will ensure that there are borrowers that can get their complete borrowing requirement, and thereby assure that they will repay their loan, and make these repayed funds available for other borrowers.

In order to understand this algorithm we need to define the concepts of states and safe states. The **state** of the system is defined as the current allocation of resources to the tasks in the system. A **safe state** is state in which there is at least one ordering of task execution that will allow all tasks to run to completion without resulting in a deadlock.

To understand how this algorithm works consider the following example taken from Tanenbaum. The following discussion is with reference to Figure 4-9. This diagram shows two matrices. The one on the left shows how many of each resource is assigned to each of the five tasks. The matrix on the right shows how many resources each task still needs in order to complete. Tasks must state their total resource needs before execution begins so that the system can compute the right hand matrix at each instant of time of execution.

The three vectors at the right show the following:

- (a) E \equiv the existing resources, i.e. the total number of each resource available.
- (b) P resources already possessed by tasks in the system.
- (c) A resources still available (i.e. unallocated).

Therefore we can see that the total resources are six tape drives, three plotters, four printers and two punches. Of these five tape drives, three plotters, two printers and two punches have been allocated to tasks. The available resource is simply the difference between the resources available and the resources allocated.

An algorithm for checking to see if a state is safe can now be stated.

- (i) Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system may deadlock since it may be possible for no process to be able to run to completion.
- (ii) Assume the task of the row chosen requests all the resources it needs (which is guaranteed to be possible from (i)) and finishes. Mark that task as terminated and add all its resources to the A vector.
- (iii) After (ii) the system is now in a new state, therefore (i) and (ii) are repeated until all the tasks are terminated. The initial state is then said to be safe.

If there are several tasks that can be chosen in step 1 it does not matter which one is chosen.

Remark: It should be noted that the Banker's algorithm is a very conservative algorithm. If a state is not safe with this algorithm it does not mean that deadlock will occur; it simply means that deadlock is a possibility. The conservatism arises from the assumption that a task will simultaneously hold all its required resources. In many situations this does not occur – a task will claim a resource and then return it for use by other tasks prior to requesting some other resource. However, these circumstances are implementation specific. In order to have an algorithm that will always work, then the conservative approach of assuming that it will simultaneously hold all its resources is taken.

In the example of Figure 4-9 the state is safe. If task B requests the printer it is able to get it. Even after granting this task D can also get its printer resource, and it can clearly run to completion as it does not require any other resources. When it returns its resources to the available pool then task A or E can run and so on until all the tasks complete. However if E was granted the last printer then the available vector would be 1000, and this leads to deadlock.

Processes	Tape drives	Plotters	Printers	Punches	
A	3	0	1	1	$E = (6,3,4,2)$ $P = (5,3,2,2)$ $A = (1,0,2,0)$
B	0	1	0	0	
C	1	1	1	0	
D	1	1	0	1	
E	0	0	0	0	
Resource assigned					
Processes	Tape drives	Plotters	Printers	Punches	
A	1	1	0	0	
B	0	1	1	2	
C	3	1	0	0	
D	0	0	1	0	
E	2	1	1	0	
Resources still needed					

Figure 4-9: The banker's algorithm with multiple resources.

The basic principle therefore is that there has to be at least one row (corresponding to one task) in the resources needed matrix that can have all its resource requests satisfied at each step

in the algorithm. If a resource request is made that results in no rows (and hence no tasks) for which the resource requests can be satisfied, then this resource request will not be granted, and the requesting task will be blocked.

Summarizing

As can be seen from the discussion of deadlock it is a difficult problem to solve in a general manner. All of the solutions proposed have their limitations and restrictions. There are other strategies that have been developed for specialised applications that give satisfactory performance, but they still do not solve the general problem. Integrated schemes have been devised that combine a number of these approaches. Refer to Stallings for more detail.

Task descriptors

The fundamental structure in an operating system is the task control block (tcb) or the task descriptor. This structure contains all the information that the kernel needs to know about a task. In fact it is the means by which the kernel recognises and manipulates tasks.

A typical tcb may look like the following:

```

type
  task_pointer = ^task_descriptor;
  task_descriptor = record
    next : task_pointer;
    saved_state : task_state;
    .
    .
    .
    other fields as needed
  end (*record *)

```

There are usually a number of fields in the tcb but at this stage I have purposely missed out all but the essential ones. All implementations of a kernel require that there at least be a **next** field and a **saved_state** field. The **next** field is used to form linked lists of tcbs. This is how queues of tcbs are formed in the operating system. The **saved_state** field is required in order to save the current state of a task upon a task switch. The other information which would appear in a practical tcb is generally speaking implementation dependent. Examples of the other types of information are the priority of the task, the task number, housekeeping information which makes it easier for operating system monitoring routines to function etc.

The kernel typically keeps an array of task control blocks, the maximum number being the maximum number of tasks that the system can support. In a dynamic operating system, such as VMS or UNIX the number of tcbs can be altered dynamically as the number of tasks in the system are altered. However, in real time operating system applications the total number of tasks is known *a priori* and therefore the required number of tcbs is known. In this case memory for the tcbs can be allocated as an array of records.

The tcbs contain sensitive information which is only of use to the kernel. Therefore the tcbs should not be exported outside the kernel.

Kernel Queues

Inside a kernel there are a number of queues. Generally these queues are queues of tcbs. For example there is a queue of tcbs representing tasks blocked on semaphores. If there are a number of priority levels there are queues associated with each priority level.

The current state of a task is represented implicitly by the queue that the task's tcb is on. At any given time all tasks will be in one of the queues in the kernel, with the following exceptions:

- (i) The currently running task is usually not on any queue.
- (ii) The special task known as the null task sometimes is not on any queue (this is an implementation decision). My preferred technique is to treat the null task the same as any other task and put it on a queue.
- (iii) In some systems there are sub-tasks and interrupt tasks which are outside the control of the dispatcher and are therefore not on any queues.

One of the most important queues in the kernel is the *ready* queue. This queue contains the tcbs of the tasks which can run if processor time is available. When the dispatcher is called it gets the first task on the queue and places the current task on the end of the queue. This type of FIFO queuing mechanism, or round robin scheduling is fair to tasks and provides a simple queue implementation.

Most operating systems do not have a single queue for ready tasks but instead have a number of queues, one for each priority level in the system. As a specific example we shall look at the queue structure of UNOS (University of Newcastle Operating System).

Description of UNOS-V1.5

The University of Newcastle Operating System (UNOS) is a small operating system kernel which was written to satisfy two main objectives:

- to be small, fast and provide the necessary support for real time applications
- to be portable – i.e. one should be able to port it to different hardware platforms very easily and rapidly.

The first of these objectives can only be achieved by having the experience to guide the design of the system so that the required features are included. The second objective was achieved by writing almost the entire operating system in ‘C’. It should be noted that the first and second objectives require a trade-off. Obviously the size and speed of the operating system could be improved by writing it entirely in assembly language for the target processor. However, portability would be very poor, requiring a line by line translation to move it from one processor to another.

UNOS is based on the operating system model proposed by Lister (see “Fundamentals of Operating Systems” reference). The central structure to his operating system is a data structure known as the central table. This table in UNOS is implemented as an array of pointers to the various kernel queues and the current task. The UNOS central table is structured as follows (see Figure 5-1). Note that the “Index” in the following denotes the index into the central table array.

- Index 0 – Pointer to the current tasks tcb. The current task is not on any queue in the system.
- Index 1 to $2 * (\text{number of priorities})$ inclusive – Sets of pointers to the beginning and end of the priority queues. These pointers are in pairs, the first pointer pointing to the beginning or top of the priority queue and the second pointer pointing to the end of the priority queue. Due to the nature of the queues this means that these pointers are pointing to tcbs.
- Index $[1 + (2 * \text{number of priorities})]$ – A set of pointers to the beginning and the end of the active timer queue. These pointers are pointing to timer structures involved in system timing.
- Index $[3 + (2 * \text{number of priorities})]$ – A set of pointers to the beginning and the end of the pool of inactive timers.

Table of Pointers

0	Pointer to the current tcb
1	Begin of High Priority Q
2	End of High Priority Q
3	Begin Low Priority Q
4	End of Low Priority Q
5	Begin of NULL Priority Q
6	End of NULL Priority Q
7	Begin Active Timer Q
8	End Active Timer Q
9	Begin Inactive Timer Q
10	End Inactive Timer Q
.	Begin sema 0 Blocked Q
.	End sema 0 Blocked Q
	⋮
	⋮
	⋮
	⋮
	Begin sema N Blocked Q
	End sema N Blocked Q

Figure 5-1: Central Table structure of UNOS.

- Index $[5 + (2 * \text{number of priorities})]$ to Index $[5 + (2 * \text{number of priorities}) + (2 * \text{number of semaphores})]$ inclusive – Sets of pointers to the beginning and the end of the semaphore queues. These pointers point to the tcbs representing the tasks which are blocked on the semaphores in the system.

It should be noted that all the queues pointed to by the entries in the central table are linked lists of structures. Therefore the storage for the queues has effectively been allocated when the storage for the structures which are to be placed on the queues is allocated. This means that the queue manipulation in UNOS is very fast – it simply involves manipulating pointers to data structures, and does not involve any movement of the data structures themselves. More detail is present on the form of some of these data structures in section “Queue Operation in UNOS-V1.5” on page 5-6.

The central table approach has the advantage that there is a single data structure within the kernel through which virtually all the kernels data structures and information can be found. This centralization of information makes it easier to write operating system monitoring routines which allow a user to monitor operating system performance and operation. A good operating system monitor is invaluable as a debugging aid during real time system development.

Unos-V1.5 Task Control Block (TCB) and Semaphore Structures

The two other main data structures in UNOS are the task control block (tcb) structures and the semaphore structures. Both the tcb and semaphore structures are reference by double indirec-

tion. The central structure associated with the tcb's is an array called tcb (see Figure 5-2). This array is an array of pointers which point to the task control blocks. The array has a user defined size. The advantage of this approach is that the tcb structures themselves are dynamically allocated during task creation, and hence there is not a large amount of space consumed by statically defined structures. The maximum number of tasks that can be created is defined by the number of pointer elements in the tcb pointer table.

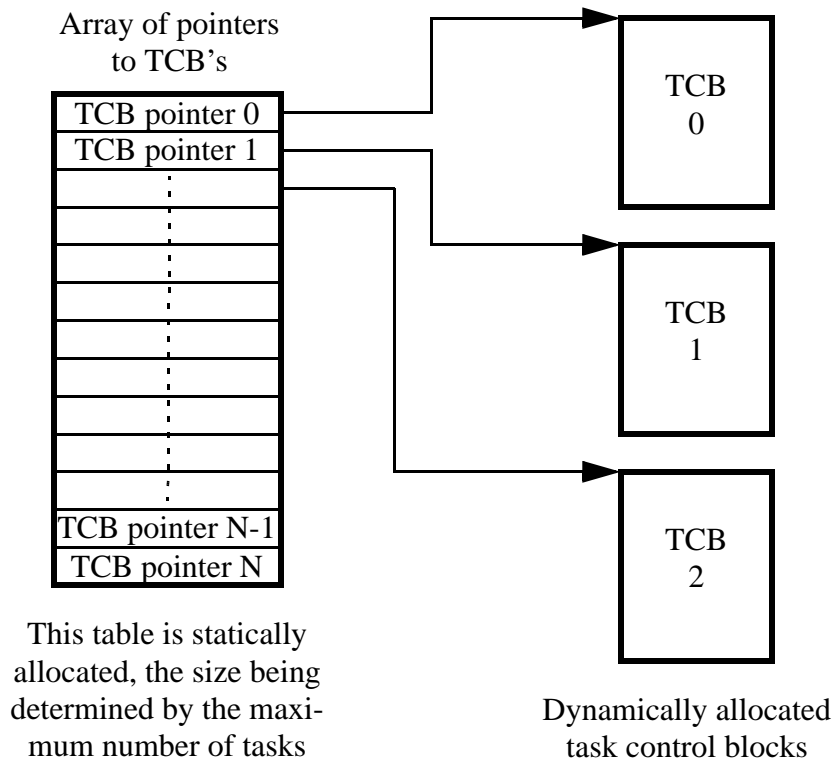


Figure 5-2: Organization of Task Control Block Structures.

The data structure general layout for the semaphores is very similar to that for the tcb's. An array of pointers is again used to store pointers to the dynamically allocated semaphore structures. As with the tcb's the size of this array is defined at the kernel compile time, and therefore the number of semaphores that can be defined in the system are limited.

The reason for designing the tcb and semaphore structures to operate in this way is the same as the logic behind using the central table approach. By having centralised structures which contain arrays of pointers it is easy to locate various operating system components when debugging the system and when writing system monitor functions. For time shared operating system this type of implementation would not be as satisfactory, since the runtime environment is usually less defined than that of a real-time operating system. In time shared systems virtually all system structures can be created and destroyed dynamically at run time. In the UNOS case, tcb's and semaphores are created but never destroyed, since destruction of these structures does not make sense in a ROM based environment.

The task control block structure for UNOS-V1.5 has the following form:

```
typedef
    struct tsk_ctrl_blk {
```

```
unsigned int task_num;
char *task_name_ptr;
unsigned char static_priority;
unsigned char dynamic_priority;
unsigned int ticks_before_kernel_entry;
unsigned int ticks_to_go;
unsigned char task_status;
struct tsk_ctrl_blk* prev_task_ptr;
struct tsk_ctrl_blk* next_task_ptr;
struct time_struct* timer_ptr;
char timeout_flag;
char q_type;
unsigned int cent_tab_index;
unsigned int sema_blocked_on;
unsigned int num_semas_claimed;
unsigned int task_stkbase;
unsigned int task_stkptr;
unsigned int task_baseptr;
unsigned char fp_save_area [150];
} task_control_block;
```

As can be seen from the above the structure is quite complex. Many of the components can be deduced from their names, however some are worth further discussion.

When a task is created it is given a task number and task name (stored in **task_num** and **task_name_ptr** respectively). The number of the task is the index into the tcb pointer table, and can be used to find the tcb for a task. The task number is defined by the operating system, its value depending on the creation order of the tasks. The task name is used for addressing the mail box of a task. It is a user defined name that does not change if the number of tasks in the system changes, or if the task creation order is varied.

Notice that there are two priorities stored in the structure – a static priority and a dynamic priority (this priority structure has been replaced in Version 2.0 of UNOS, known as UNOS-V2). See “Some Explicit Implementations” on page 5-29 for a better description of this approach. The static priority is what is normally considered as being the priority of a task. Under most conditions the static and dynamic priorities are the same. Under conditions where a low priority task has claimed a semaphore (i.e. has carried out a **wait** causing the semaphore value to go to zero) and then a high priority task has awoken and wanted the same resource protected by the semaphore, then the dynamic priority comes into play. Essentially the dynamic priority of the task claiming the resource is raised to that of the high priority task until the resource is released. All scheduling is carried out using the dynamic priority.

The **ticks_before_kernel_entry** and **ticks_to_go** entries are related to the time slice performance of the system. The first parameter sets the number of clock ticks the task will run before the kernel is called and the dispatcher runs (assuming that the task does not get blocked for some reason). Its value is defined at task creation time, although the user can change its value whilst the system is running. The **ticks_to_go** value is loaded with the value of **ticks_before_kernel_entry** and then decremented on every clock tick. When its value reaches zero the kernel is entered.

The **timer_ptr** and **timeout_flag** are used when a task is blocked on a **timed_wait**. The **timer_ptr** points to the timer structure being used for the time operation. The flag is used to indicate that a **timed_wait** has timed out, and that the task has not become runnable because of a **signal**.

The semaphore related fields of the tcb are connected with the dynamic priority mechanism.

The majority of the status of a task is saved on the stack upon kernel entry. Consequently if the task stack pointer is saved before switching to a new task stack then this state can be preserved. The task stack pointers are saved in the tcb in the locations **task_stkptr** and **task_stkbase**. If a floating point coprocessor is used then the internal registers of this must also be preserved, as they are also part of the context of a task. The **fp_save_area [150]** is provided for these.

Queue Operation in UNOS-V1.5

The queue operation in UNOS is driven from the central table as noted above. If the beginning and the end entry in the central table contains a NULL pointer then the queue is empty, any other value being a pointer to an object which is on the queue.

The ready queue in UNOS is arranged as a number of priority queues, each priority queue itself being in FIFO order. The number of priorities in the system is user definable. In practice it is difficult to figure one what to do with more than about eight priorities. Assuming that a task switch is too occur, the dispatcher begins looking in the priorities queues starting with the high priority queue. It finds the beginning of the high priority queue by looking at the appropriate index point into the central table. If the begin queue pointer contains a NULL then it moves to the next priority queue entry in the central table. This continues until it finds a non-NULL pointer. This pointer points to a tcb, which contains the state of the next task to run. The tcb is removed from the queue and its pointer is passed to the dispatcher. The dispatcher then retrieves the stored information in the tcb and starts the task running. As noted above, the priority queues are implemented as a linked list. In the UNOS case this is doubly linked list. For example, the connection between the central table, a priority queue and the tcbs is shown Figure 5-3.

The queues in UNOS have been implemented as a doubly linked list so that tcbs can be placed and removed in the middle of a queue. It should be noted that having a doubly linked list adds complication to the kernel queue handling routines and therefore overhead upon each kernel operation. Therefore careful consideration should be given to whether it is necessary to have them. In the case of UNOS, the system of dynamic priorities and the timer mechanism necessitated the requirement to be able to remove and insert tcbs from middle of queues. If it is a requirement to be able to remove and insert into the middle of queues then another consideration is how the queues should be implemented. If the queues are on average going to contain a lot of tcbs and the position to insert has to be searched for then an alternative queue structure may be more efficient.

The structure of all the priority queues in UNOS is the same. The number of priority queues is a user selectable variable (up to 256). However, as mentioned earlier, it must be remembered that the more priorities that are included the slower the kernel will become (since the kernel has to search through more queues). Moreover, the programmer is faced with the problem of how to manage a large number of priorities in a sensible fashion. For these reasons a value between 8 and 16 priorities is probably a reasonable compromise between kernel speed and versatility.

As mentioned previously all operating systems require at least one task to be runnable at any point in time. This is generally done by creating a task known as the null task. In some systems the null task is not kept on any queue in the system, and if all ready queues have been searched and no task has been found to run then the null task is run. In UNOS the null task appears on a queue, as does any other task (see Figure 5-1). The advantage of this approach is that the null task is not a special case as far as the kernel is concerned. It is effectively the lowest priority task in the system. To ensure that it is always runnable it should not consume any of the

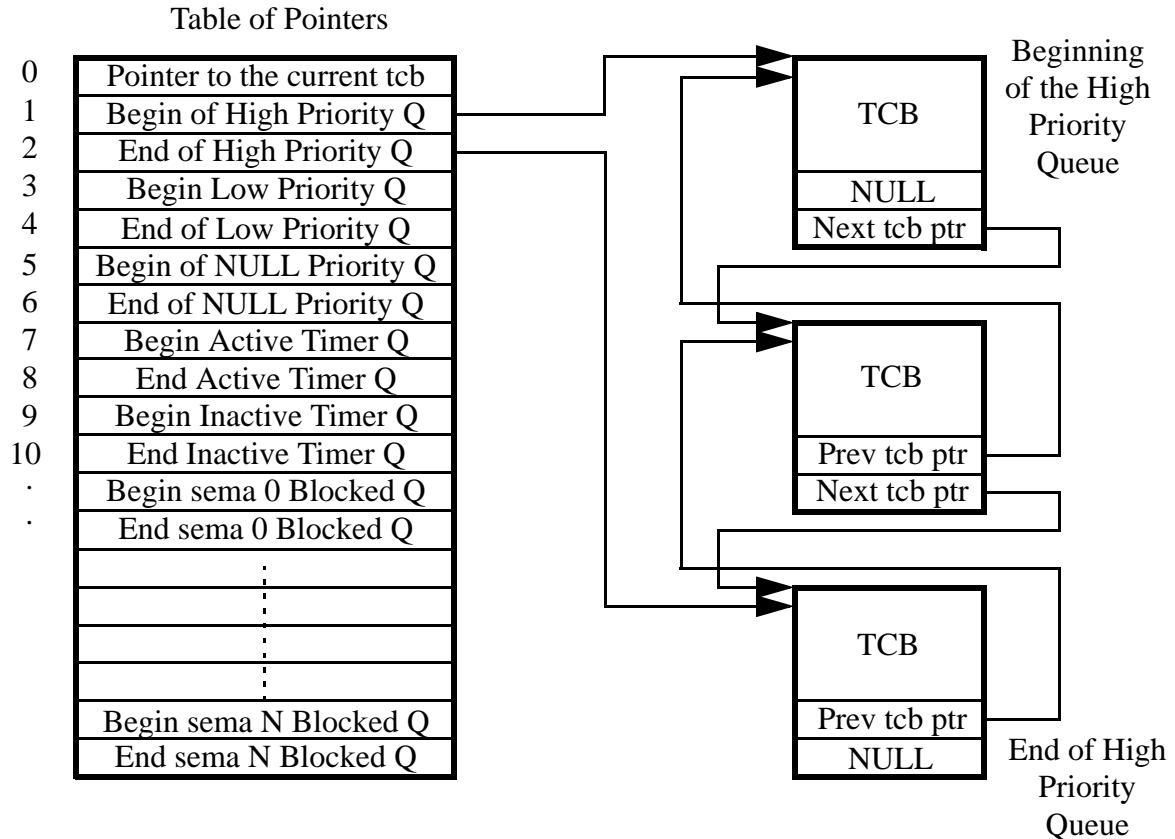


Figure 5-3: Relationship between the central table and the priority queues.

resources in the system (i.e. it should not have a **wait** in it). Indeed in some processors which have a **HALT** instruction the null task consists of the following:

```

procedure null_task
  begin
    repeat
      halt;
    until forever;
  end; (* null_task *)

```

When the processor executes the **HALT** instruction it stops. However, it is essential that in this condition that the processor can still receive interrupts, else it will remain locked in this state.

Some people are confused by the fact that the null operation is implemented as a task. The question they ask themselves is “Why can't the null task be implemented as a loop in the kernel?”. The answer is, if this were done then the system would lock up because interrupts are generally disabled in the kernel. Therefore no other task could become runnable, since the current state of the system can not be changed. Therefore the system would be effectively dead-locked, and would probably appear to have crashed to a user.

Because the null task is very much like a normal task in UNOS it is possible to run multiple tasks at the null priority. This can be done provided one of the tasks corresponds to the normal null task, the other tasks at the null priority can use system resources without any problem.

UNOS-V1.5 task creation

One of the most difficult aspects of an operating system kernel to initially understand is the mechanism for bootstrapping the system so that tasks are created and the kernel can begin to run. Rather than try and approach this problem in an abstract fashion we will use UNOS's task creation process as an example.

From the users perspective the task creation process is very simple. One just calls a kernel function called **create_task** passing in the appropriate parameters (see the reference manual in the Appendix for a description of the parameters). However this simplicity belies the complexity within the **create_task** procedure.

At the point that the **create_task** function is called it is assumed that most of the core data structures for the kernel have been created (e.g. the central table, the tcb array table and the semaphore array table etc.). This is carried out by the software data structure initialisation procedures. Amongst other things they set up the memory heap to be used by UNOS, and then allocated the main data structures to be used by the operating system from it.

Upon the entering the **create_task** function the following actions are carried out:

- (i) A check is made to see if too many tasks are being created. The maximum number of tasks is limited by the size of the tcb pointer array, which is effectively a static structure since it is created at the system data structure initialisation.
- (ii) The tcb data structure is then allocated on the heap, the structure is initialized with values passed into the **create_task** procedure, and then placed on the appropriate kernel queue.
- (iii) Two semaphores are dynamically created for the task's mail box. These semaphores are used for resource counting the number of free slots in the mail boxes.
- (iv) The mail box structure is allocated on the heap and then the mail box address is registered with the mail exchange agent within the kernel. The mail exchange agent resolves all the addresses when messages are sent between tasks.
- (v) A stack for a particular task is allocated on the heap. The stack size is determined from one of the parameters passed into the **create_task** function. The pointers to this stack are then copied into the stack pointer registers, making this new stack the current stack. Normally one must not attempt to use local variables once this stack change has occurred, since the variables will be looked for on the new stack, but they actually exist on the old stack. The task address (which is the address of the procedure that is the start of the task) is stored in a local variable. This has to be the case because the **create_task** procedure is used for creating all tasks, and consequently a global variable for the task name would be overwritten on each call to the procedure. Therefore, it is necessary to transfer the task address local variable from the old stack to the same relative position on the new stack, thereby making the address available upon the first switch to the task.
- (vi) The kernel control block (a small data structure which the kernel uses to determine the reason for a kernel entry) is set up to indicate the kernel entry is for task creation purposes. A pointer to the new tcb is placed in the current task position of the central table (this is an unusual situation because the tcb is also at this stage on its priority queue). This will create the illusion to the kernel that the new tcb is the current task. A software interrupt is then carried out to enter the kernel. The kernel carries out the operations that it would do if the new task were the current task in an already executing system. This results in the registers being pushed onto the new stack, and the new stack pointer

being saved into the tcb for the task (remember that the tcb for the task has already been placed on the appropriate system queue). After this has occurred then control immediately returns to the **create_task** routine. The dispatcher is not called, The net result of this action is that data has been stored on the stack in a form that corresponds to that created upon a normal kernel entry, and this stack configuration has been preserved in the tcb for the task. This means that when this task becomes the current task then this information is popped off the stack. Consequently the return under the first task switch will be back to the **create_task** routine.

- (vii) Upon return from the kernel entry back to the **create_task** function there is a check to see if the task function should be called. As mentioned previously, the task function address was stored on the newly created stack. Therefore, when the first task switch to the newly created task occurs the execution of the task will start by returning back to the **create_task** routine (since the stack format effectively stored in the task's tcb is identical to the one that exists upon the return at the task creation time). There is a global variable that is checked to determine whether the current execution of the **create_task** is in a task creation mode or whether it is after a task switch. This global variable is clear during task creation, and is set just before the null task is set running and the system interrupts are enabled. If the global variable is not set then execution continues to the end of the **create_task** function. If the variable is set then the task entry function is called, the address being present on the stack. A return is never made back to this point, because the tasks are effectively an infinite loop.
- (viii) If the task function is not called, the stack that was current upon entry to the **create_task** routine is restored.
- (ix) The task initialization procedure is called if there is one and the **create_task** function is exited.

One of the interesting features of the above task creation process is that the task stacks are being set up by an interrupt entry to the kernel. This means that the stack organization will be exactly as is required by the particular compiler being used. Many kernels require the stack to be manually set up – a process that is prone to error, and has to be modified if different compilers are being used.

The above task creation process continues until all the tasks have been created in the system. During this process the system interrupts are disabled and the operating system is not running. At this stage the processor is simply executing a 'C' program. The final task to be created in the system is the null task. This task is not created using the process above, but instead the task entry point is simply called from the bottom of the main program. The stack used by this task is the stack created by the compiler to allow the main program to run. Before entering the main loop of the null task, the global variable, mentioned above, which allows the other tasks to start is set. Then the interrupts are enabled. The main loop of the null tasks is then entered. Execution will then continue in this loop until an interrupt occurs. The occurrence of an interrupt will usually cause the kernel to be invoked and the highest priority task to be scheduled. The process described above then occurs which causes each task, as it is scheduled, to begin running for the first time.

It should be noted that UNOS task creation is a process that is carried out before the kernel begins operating – i.e. it is essentially a static process. In the context of the intended applications for the UNOS kernel this is not a limitation. UNOS was designed for real-time embedded system applications, which are nearly always ROM based. For a system requiring dynamic runtime creation and destruction of tasks the process becomes more complex.

The Dispatcher

The dispatcher is the part of the kernel that chooses the next task to run and then starts or dispatches that task. There is some contention as to whether the dispatcher should be called the scheduler. There certainly is a scheduling aspect to the dispatch process, but the term scheduler in many operating systems is reserved for a high level scheduling algorithm that changes the priorities of tasks based on real-time conditions within the system. The dispatcher is sometimes also called the low level scheduler.

There are several ways that the dispatcher is entered:

- (i) from a call to **wait** if the calling task becomes blocked.
- (ii) from a call to **signal** to make a task runnable and that task has a higher priority than the current task.
- (iii) a time slice interrupt resulting in a time slice entry to the kernel.

The basic operation of the dispatcher is quite simple:

if (the currently running task is not suitable to continue, because it is blocked, or because its time quota has expired or because a higher priority task has become runnable) then
 place the current task on the appropriate priority queue and then choose another task from the priority queues;
 perform a task switch.

The above outline is a little more complicated than a practical system would be since upon entry to the dispatcher it is already known that the current task has become blocked. The dispatcher is usually broken up into a number of procedures to handle the various entry conditions.

The UNOS-V1.5 Dispatcher

In UNOS the dispatcher consists of a number of call specific handlers (e.g. a handler for the wait operation, a software initiated pre-emptive task switch, a time slice entry to the kernel, etc.) and then some generic queue handling routines which are used by these handlers. For example a **wait** operation carries out the following basic operations:

```
procedure wait ( semaphore_num )
begin
    disable;
    if semaphore [ semaphore_num ].semaphore_value > 0 then
        (* decrement the semaphore *)
        semaphore [ semaphore_num ].semaphore_value =
            semaphore [ semaphore_num ].semaphore_value - 1;
    else
        begin
            (* Semaphore already zero - setup kcb *)
            kcb.semaphore_num = semaphore_num;
            kcb.entry_type = wait_entry;
            (* Now enter the kernel using a software interrupt *)
            geninterrupt ( kernel );
        end;
    enable;
end;
```

In the kernel the following actions are carried out:

```
interrupt procedure kernel
```

```
begin
    save the stack pointer in the tcb of the current task;
    do case kcb.entry_type
        .
        .
        .
        wait_entry : wait_handler;
        .
        .
    end;
    copy the stack pointer from the current task tcb to
    the processors SP register;
    (* task switch is carried out by the normal      *)
    (* return from an interrupt routine              *)
end; (* kernel *)
```

The **wait_handler** then carries out the operations appropriate for the wait operation:

```
procedure wait_handler
begin
    (* place the current task on the semaphore *)
    (* queue                                   *)
    semaphore_queue_handler ( queue );
    (* now call the scheduler to find another task *)
    (* run                                       *)
    scheduler;
end;
```

It should be noted that in the above that the **kernel** procedure decides what handler to call via the contents of the **kcb** (the kernel control block). This structure is used by the calling tasks to tell the **kernel** procedure what is required. It is necessary because the kernel is entered via an interrupt and consequently cannot accept parameters. The **kcb** has the following format:

```
unsigned char entry_type;
unsigned int semaphore_index;
unsigned int task_number;
unsigned char new_priority;
```

Note that all the components of the **kcb** may not be used on any particular call to the kernel. Most of its components are self evident. One that isn't is the **semaphore_index**. The semaphores in UNOS are created in a similar way to the tcb's – i.e. there is an array of pointers which point to the semaphore structures. The index referred to in this component is the index into this array, which is also the semaphore number. The pointer contained at this location is then used to refer to the semaphore structure itself.

The operation of the **semaphore_queue_handler** depends on how the blocked queues are implemented. If they are simple first in first out (FIFO) queues then the queue handler would put the task to be blocked onto the end of the queue for the particular semaphore. If dynamic priorities have been implemented then the semaphore queues are ordered and the **semaphore_queue_handler** would put the tcb of the task into the correct spot on the semaphore queue, based on the blocked task's priority, and carry out other housekeeping operations associated with dynamic priority operations.

The **scheduler** is the procedure which searches through the priority queues looking for a ready task. It begins with the high priority queue and searches through to the null queue where it will always find a task ready to run. When a ready task is found, its tcb is removed from the

top of the queue and a pointer to the removed tcb is placed in the first location of the central table (i.e. the task has now become known as the current task but the task switch is yet to occur).

In order to facilitate the queue manipulation within the dispatcher there are a number of queue maintenance routines, namely:

- **remove_queue** – this routine allows a tcb to be removed from anywhere within a queue of tcbs (i.e. from priority queues or semaphore queues). A pointer to the tcb which is to be removed is passed to the routine along with the begin queue index (for the particular semaphore) into the central table.
- **remove_top_queue** – this routine is similar to the **remove_queue** routine except that it only removes tcbs from the top of the queue. This routine was included so that in cases where tasks will only be removed from the top of the queue the simple routine can be used and thereby decrease the time spent in the kernel.
- **add_queue** – this routine adds the tcb to the end of the queue whose central table begin queue index is passed. This routine is used to place tcbs into the priority queues.

The semaphore queue addition process is a little different from the priority queue addition process. This is due to the implementation of the dynamic priority changing. It requires that the tcbs on the blocked queues be in priority order. Therefore the tcb placement in queue involves searching then the blocked queue and then inserting the tcb at the correct position.

The task switch operation

After the dispatcher has decided which task is the next task to run it then performs the actual task switch operation which makes the selected task the next to actually execute on the processor. This operation is generally very hardware dependent, especially with processors which provide hardware support for task switching (e.g. the Intel 80286, 80386, 80486 and Pentium).

Let us consider a processor which does not provide hardware support for task switching. While a task is running it uses the processors registers. When a switch occurs from one task to another the total operating environment of the current task must be saved and the state of a previously running task must be restored. All the information relevant to the task switch is usually stored in the tcb of the task in question. As we shall see in a moment, the saving and restoring of the task's state can simply be a matter of saving and restoring the tasks stack pointer. When this is the case the procedure for task switching is very simple.

As discussed in previous sections each task in an operating system has a separate stack. Therefore upon a task switch it is essential that the stack pointer for a task be saved and restored so that the task will be operating with the correct stack. As noted in the paragraph above the registers must also be saved and restored. Sometimes the compiler for the language in which the kernel is written will ensure that registers are saved when a procedure is entered, however this can not be counted on if the operating system is to be portable. One way of ensuring that the registers are saved in totality is to always enter the kernel via an *interrupt*. Many compilers have the facility to define interrupt procedures which automatically put into an interrupt procedure a preamble to save the registers, and at the end of the procedure code to restore the registers. If the kernel is written in assembly language there is no advantage in this approach. However, when the operating system has been written in a high level language (as UNOS is), the interrupt approach is an elegant way of saving the registers, including the flags. If the machine architecture supports multiple memory models, as does the Intel 8086 series of machines, then additional benefit is gained because the stack is automatically in the correct form if the memory model is changed, without any code changes. On the negative side such

mechanisms are also non-portable. The only way to ensure portability to other machines and compilers is to write small assembly language routines which are called at the beginning and the end of the kernel entry procedure to save and restore registers respectively. The differences due to memory models have to be explicitly taken into account by the programmer.

In a protected mode system using an interrupt to enter the kernel is sometimes necessary because it is often the way a change is made from user to supervisor mode. However, in a non-protected mode operating system, such as UNOS, the kernel could be entered via an ordinary call provided that the interrupts are immediately disabled and the appropriate register save operations are performed.

Given that the registers are saved as discussed above, then in many cases the only register which must be saved in the tcb is the stack pointer.

Under these circumstances the task switch procedure is very simple:

```
procedure task_switch ( T : task_ptr );
(* This procedure performs a task switch from the *)
(* current task to the task pointed to by T. It is *)
(* assumed that the current task has already been *)
(* put on the appropriate kernel queue. *)
begin
  (* initialise time slicing quota for task *)
  if T <> current_task_pointer then
    begin
      save any registers which have not been
      saved;
      current_task_pointer^.SP := stack_pointer;
      current_task_pointer := T;
      stack_pointer := T^.SP;
      restore the registers for task T;
    end; (* if *)
  end; (* task_switch *)
```

Notice that the procedure behaves differently from a normal procedure in that when the procedure returns it is popping the return address from a different stack than that when the procedure was entered. The return will occur to the procedure that the new task was executing when the procedure **task_switch** was called.

Some processors have built in task switch operations. Examples of such processors are the Intel 80286, 80386, 80486 and Pentium. In the 80286 there is a special fixed size segment which is used to store the tasks state (i.e. registers). A special **jump** instruction with this segment as the operand is used to cause the necessary saving and restoring of the task states. Therefore the task switch becomes a single instruction. The actual jump operation causes the state of the registers to be saved into the current task state segment, and the state to be retrieved from the new task state segment. This retrieved state includes the program counter, therefore execution continues from the last instruction of the new task. It should be noted that having this facility does not eliminate the need for a tcb. Information such as task priority, time quotas etc. are still required. Instead of a stack pointer in the tcb to refer to the task state there is now a pointer to the *task state segment*. Refer to “Multitasking Support in the 80386” on page 11-29 for more information.

The UNOS-V1.5 task switch

The task switch operation for UNOS has already been discussed briefly in the previous section on the UNOS dispatcher. UNOS uses an interrupt technique to enter the kernel which means

that different memory models which are present on most Intel 8086 targeted compilers can be catered for. Kernel routines which are called from tasks are entered as conventional procedures. However, if the requested action requires a task switch then the kernel proper must be entered. This is done by a software interrupt which saves the program counter and the processor flags automatically. The preamble code generated by most 8086 based compilers saves the registers on the stack and a stack frame is then generated for any local variables in the interrupt procedure. The first piece of user code upon entry to the kernel is to save the stack pointer and the base pointer. In the case of the 8086 processors the stack pointer actually consists of two 16 bit pointers known respectively as the stack base pointer and the stack pointer. The base pointer is saved because local variables local variables in the interrupt routine are referenced in the stack via the base pointer. It must be restored to make sure that the local variables being referred after the task switch are those in the stack of the new task.

The addition of a coprocessor in an 8086 based system also has implications on the task switch operation. A maths coprocessor is a resource that is being shared between all the tasks in the system. If a task is carrying out mathematical operations then the coprocessor registers are in a particular state. If then, for example, a time slice interrupt occurs and the kernel is entered, the state of these registers have to be preserved. If one of the tasks switched to uses the coprocessor then the registers would be corrupted if they are not saved. The registers are saved in an area of the tcb. Because of the number of registers that have to be saved in large, efforts are made to minimise the number of times this occurs. In UNOS the coprocessor registers are not saved at the initial entry point to the kernel, but are saved at the end depending on whether the task being switched to is the same task that was running upon kernel entry. If the tasks are the same then there is no need to save the registers since the resumption of the task will simply be picking up the execution where it left off. On the other hand if the task is different then the registers are saved.

One could go one step further with this minimisation approach, and have a flag with each task that indicates whether it uses the coprocessor or not. If it doesn't then the coprocessor registers are not saved. This has not been implemented in the UNOS task switch because it relies on the programmer to specify at task creation time whether a task uses the coprocessor or not. Even if the correct decision is made at this stage, as the software system evolves it is difficult to keep track of whether procedures called by a task may be using the coprocessor.

Subtasks and Interrupt Tasks

The existence of the hardware task switch mentioned above allows the possibility in some systems of the task switch instruction being executed from a task. In the 80286, 386, 486 and Pentium it is possible for a user task to execute a **CALL** with a task state segment as the operand. This effectively causes a task switch without the dispatcher knowing about it. Tasks which are called from other tasks in this manner are known as *sub-tasks*. A sub-task has some but not all the properties of a coroutine. The common property is that the sub-task is called from the main task as one coroutine is called from another. The difference is that the return to the originating task is not via a **call** to the originating task but by a *return from interrupt*. From the dispatchers point of view the originating task and the sub-task appear as one task to it. If an entry to the kernel were to occur whilst the sub-task was operative then the kernel would have the task state segment of the sub-task as the active task state segment. The nesting of the tasks is stored within the task state segments, so that when the sub-task terminates the originating task is reactivated. Because the originating task and the sub-task cannot be distinguished by the kernel there is only one tcb for the both. It should be noted that the sub-task and the originating task do not run concurrently, but the sub-task functions as a task in the sense that the next call to it from the originating task will result in execution continuing from the point where it left off last

time. Refer to “Protection in the 80386 Processor” on page 11-14 for more details in these mechanisms.

Interrupt routines are in some way similar to sub-tasks. On machines such as the 80286 etc. there are explicit interrupt tasks. When an interrupt occurs a task switch occurs to the interrupt routine (i.e., it has a separate stack). These task switches are again unknown to the dispatcher, and the interrupt task effectively becomes a sub-task of the task which was running at the time of the interrupt. The advantage of running the interrupt routine as a task is that the task that was running at the time of the interrupt is protected from problems in the interrupt routine. In the case of normal interrupt routines, they run in the context of the task that was executing at the time of the interrupt. Therefore they have access to that task's data and can corrupt it if they have errors in them. This can lead to perplexing problems, because the interrupt routine that executes rarely has anything to do with the task that has been interrupted.

In a system which does not have interrupt tasks the situation is not that much different. When an interrupt occurs the stack of the currently executing task is used by the interrupt routine. Therefore the interrupt routine is not a sub-task in the sense that it uses a separate stack, but in all other respects it behaves as a sub-task.

In one does not want interrupt initiated task switches to be unknown to the dispatcher then the only option is to make all interrupts cause a “normal” kernel based task switch. The interrupts would then go through the normal task switch mechanism, and therefore be subject to the priority scheduling. The main problem with this approach is the efficiency of the interrupts. In real time systems where there can be large numbers of interrupts occurring at high rates, the overhead of a task switch on each interrupt is intolerable.

Another aspect of interrupts that is particularly relevant in real-time systems is interrupt latency. The kernel is by nature a non-reentrant piece of code, therefore interrupts are usually disabled whilst the kernel is executing. This could result in unacceptable interrupt latency in very demanding real-time applications. One possible solution to this is to enable interrupts in the kernel. The problem that can occur if this is allowed is that an interrupt routine may result in a call to the kernel. A common situation, for example, is the interrupt routine carrying out some processing and signalling the kernel using the kernel **signal** call. Such an operation offers the possibility of a task switch, and therefore reentrancy into the kernel. One way of handling this reentrancy problem, and at the same time allowing the interrupt routine to carry out its processing, is to set a flag when the kernel is entered (prior to re-enabling the system interrupts). If an interrupt occurs during kernel execution, and that interrupt routine calls a kernel routine (e.g. a **signal**), then the flag is checked in the kernel routine. If set, to indicate that there is reentrancy to the kernel, then the requested kernel operation is delayed and a return is made to the kernel. When the kernel has virtually completed execution, but before control returns to an executing task, the delayed call is carried out

The above operation is fairly messy. If, on the kernel call, it is realised that the kernel is being called reentrantly, then the operation has to be remembered and control has to return back to the kernel. This process is made more feasible if the kernel call is at the end of the interrupt routine (which it should be anyway, for other reasons). The delayed calls have to be queued and then carried out when control reaches the end of the kernel. The results from these kernel calls may change the task to be switched to when the kernel is finally left.

Time-slicing

Time-slicing is a software mechanism which ensures that, on a periodic basis, the kernel has the processor. Therefore, the dispatcher is guaranteed control of the processor and can ensure that the processor is being shared amongst the various tasks. If time-slicing is not present then

the dispatcher only gets the processor when a task becomes blocked, or when an interrupt occurs which may force a reschedule. Therefore, under this type of set up it is possible for a compute bound task to effectively deadlock all the other tasks in the system.

The basic idea of time-slicing is that a regular clock interrupt occurs which in one way or another causes the dispatcher to be called. The current task which is running at the time of the interrupt is placed on the end of its ready queue and the next ready task is run. Therefore even if the tasks which are ready to run are all the same priority then they will share the processor.

The actual implementation of the time-slice may vary in detail from one operating system to another. One of the common implementations is to include a **time_to_go** field in the tcb. This field contains a counter which is decremented upon each entry to the clock interrupt routine. When it reaches zero then the kernel is entered and the dispatcher is called. The question of how to pick the time quotas for the various tasks is very application specific and must be decided together with the priority of the tasks.

The other question to answer is what the time slice frequency should be. Obviously the higher the frequency of the time slice then the greater the overhead inside the clock interrupt routine. Therefore the time slice frequency is a compromise between responsiveness of the kernel and the kernel overhead. The clock frequency of the system depends on the time resolution required in the particular application. Typically the high priority tasks will be given a certain number of these clock ticks which is a smaller value than the lower priority tasks. This helps ensure that the no high priority task will monopolise the CPU, preventing another high priority task having reasonable access to the CPU.

Unos time slicing

The time slicing model used in UNOS is fairly conventional. UNOS allows one to select whether time slicing is enabled or not via a system call.

The time slice frequency in UNOS is decoupled from basic tick timing resolution. This allows the tick frequency to be raised to achieve the timing resolution required for a particular application, without affecting the time slice frequency. The user can select a default number of ticks before a time slice instigated kernel call will occur. Each individual task when created can have specified a tick delta around this default value. This facility allows tasks of the same priority to be given an “effective secondary priority”, as they can now be made to execute for different amounts of time when they are being round robin scheduled.

Task suspension

A suspended task is a task which is present in the system, but which is temporarily removed from participation in dispatcher operations. Therefore in this sense it is similar to a blocked task. However it differs from a blocked task in that a task can be suspended asynchronously by another task and does not get into this blocked state by requesting a resource by executing a **wait** operation.

It would appear that another kernel queue is needed to hold suspended tasks. However there is a complication because it is possible that a task can be blocked on a semaphore queue when a request is made by another task to suspend it. One way of handling this situation is to make a suspension illegal in this case, however this is very restrictive. One approach which is commonly taken is to allow a task to be in a “blocked and suspended state” as well as a “suspended state”. When a “blocked and suspended” task is activated via a **signal** it then becomes only a suspended task. If the same task is activated then it would revert back to a ready task again. The various states and their transitions are shown in Figure 5-4.

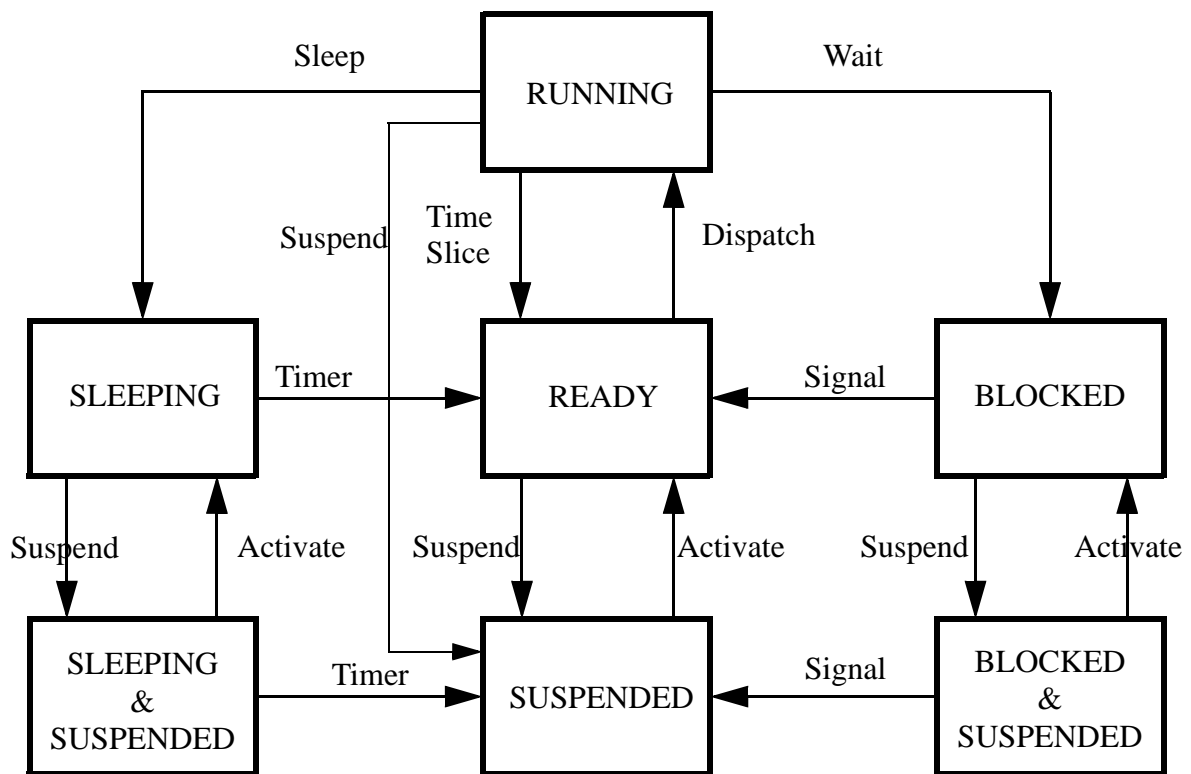


Figure 5-4: Possible state transitions if the suspended task is allowed.

One of the simplest ways to implement suspended tasks is to have a bit in the tcb which is clear when a task is active (i.e. not suspended) and set when suspended. This bit is tested by the scheduler when a dispatching operation is being carried out, and if set then the task is simply ignored and the scheduler looks at the next task on the queue. The suspended tasks remain on the ready list. This implementation is slightly inefficient because the dispatcher has to keep looking through the queues and ignoring tasks which are suspended on every dispatching operation. One way of eliminating this is to remove a suspended task from the ready queue when it is found. The task could be put onto another special queue of ready suspended tasks or could be simply be put into limbo – i.e., it is not on any queue. If such tasks were to be reactivated then the activating task would need to have some way of identifying the tcb of the suspended task so that it could be placed back on the appropriate priority queue. One common way of locating the tcb is via the task number (although this is very implementation specific).

If a task is asynchronously suspended there is always the risk that the task will be suspended after claiming a resource. If this is the case then any other task wishing to use that resource will be deadlocked until the suspended task is reactivated. Therefore the only safe way of suspending a task is for it to suspend itself. If this is the case then the question immediately arises as to whether this is any different than blocking the task on a semaphore– the answer is no. Therefore, in my judgement, the suspend state is of doubtful use and probably should not be included in a real time kernel.

One can think of solutions which overcome the suspension of a resource claiming task prob-

lem, but in the light of the dubious merits of the suspend operation the additional complexity is not balanced by the added functionality.

Dynamic Priorities

In a real-time system the precise timing of execution of high priority tasks is essential. Depending on the application that the real-time kernel is being applied to, the consequences of a timing failure may be catastrophic.

An operating system that provides priorities for the tasks in the system can provide, under certain circumstances, precise execution of critical tasks. This can be achieved by very careful organisation of task priorities and interaction between tasks. However, situations often occur where such organisation is very difficult, or sometimes cannot be achieved.

As an example consider the following situation. A system has a number of tasks running, with at least one of these being higher priority than the others. The high priority task (T_1) shares a critical section with one of the low priority tasks (T_2). Assuming that T_1 is blocked on a synchronization semaphore, then it is possible for T_2 to execute and enter this critical section. If an interrupt arrives, resulting in T_1 becoming active, and if it tries to obtain the same critical section lock, it will become blocked. This blockage will remain until T_2 releases the lock. In a complex system with many tasks this blockage period could be unbounded, since T_2 could itself be prevented from running because of intermediate priority level tasks running. In effect, the priority of T_1 has been lowered to that of T_2 , the task with the semaphore.

It should be noted that the following discussion is with reference to semaphores that are being used to protect critical sections; these will be known as “locks” in order to distinguish them from semaphores being used for synchronization and resource counting.

The obvious solution to the above problem is the use *priority inheritance* or *dynamic priorities*, where the priority of a task can be temporarily raised to reflect the priority of any higher priority task which it is blocking. For example, in the scenario cited above, T_2 would have its priority temporarily raised to that of T_1 . Clearly T_2 could not be blocked by intermediate priority tasks and would run until it released the lock. At this point the priority of T_2 would revert back to its normal value and T_1 would run.

Although the above solution to the priority inheritance problem appears simple, attractive, and easy to implement, there are some hidden traps. The most difficult implementation problem is what we will call the *disinheritance* problem:

given that a task has had its priority temporarily increased, when do we lower the priority again; and to what level do we lower it?

Possible strategies to solve the disinheritance problem are:

Strategy 1: Ensure that a task’s priority is always equal to the highest priority of any task which it is directly, or indirectly, blocking.

Strategy 2: Demote a task, at the time it leaves a critical section, to the priority it had when it entered that critical section. This is relatively easy to implement, since the task’s old priority can be stored in the lock data structure, but it has a tendency to drop a task’s priority too soon.

Strategy 3: Restore a task to its normal priority as it leaves the outermost of nested critical sections. This strategy is obviously too conservative, in that it sometimes leaves the task’s priority high for too long.

Of the above three strategies, Strategy 1 is the most accurate interpretation of pure priority

inheritance. On the surface it also appears to be the most difficult to implement since it has to take into account transitive blocking – i.e. the task that has obtained a lock has itself become blocked on a lock obtained by another task, and so on. The ultimate blocker of the first task in this queue is the last task in the queue. Strategies 1 and 2 are not equivalent; in fact strategy 2 is an unsatisfactory rule in that it can lead to high priority tasks being blocked for significant periods of time.

In order to demonstrate some of the difficulties that can arise with disinheritance consider a typical situation from the report “The Priority Disinheritance Problem”, by P.J. Moylan, R.E. Betz and R.H. Middleton.

Assume a system of four tasks; T_1 , T_2 , T_3 and T_4 and two locks or semaphores denoted L_{13} and L_{14} , the subscripts denoting the tasks that share the locks. The subscripts on the tasks show the tasks priority, with 1 being the lowest. Assume that task T_1 executes the following sequence of operations:

```

Obtain ( $L_{13}$ )
.
.
Obtain ( $L_{14}$ )
.
.
Release ( $L_{14}$ )
.
.
Release ( $L_{13}$ )

```

Consider the following sequence of events:

- (a) T_1 is initially the only task able to run, and it successfully enters both its critical sections;
- (b) T_3 arrives, runs, and is blocked on an **Obtain** (L_{13}). As a result, the priority of T_1 is increased to 3, and T_1 continues to run;
- (c) T_2 arrives, but does not run because T_1 has a higher priority (i.e. priority 3);
- (d) T_4 arrives, runs, and is blocked on an **Obtain** (L_{14}). As a result, the priority of T_1 is increased to 4, and T_1 continues to run;
- (e) T_1 executes its **Release** (L_{14}), thereby unblocking T_4 . The priority of T_1 also drops at this point;
- (f) T_4 runs, and quickly completes its execution.

Figure 5-5 shows graphically the above sequence of operations. The height of the graphs represent the current priority. The shaded areas indicate when a task cannot run, either because it is blocked or because there is a higher priority task in the system.

The interesting question here is what happens after (f). At this stage T_4 is no longer wanting to run, T_3 is blocked by T_1 , and T_2 and T_1 are able to run. If pure priority inheritance is adopted (i.e. Strategy 1) then T_1 should run at this stage, since it is blocking the highest priority task. If, however, we adopt the strategy of dropping the priority of a task, as it leaves a critical section (Strategy 2), to the priority it had on entry to that section, then the priority of T_1 will drop to 1 at that point. This is indicated by the dotted line in Figure 5-5. Note that the assumption that T_1 obtained L_{13} and L_{14} during interval (a) is crucial for the drop to the priority 1 to occur. The task T_2 has been prevented from running all this time, since other tasks had a higher priority. Therefore when the priority of T_1 drops then T_2 begins to run. Meanwhile T_3 is still blocked

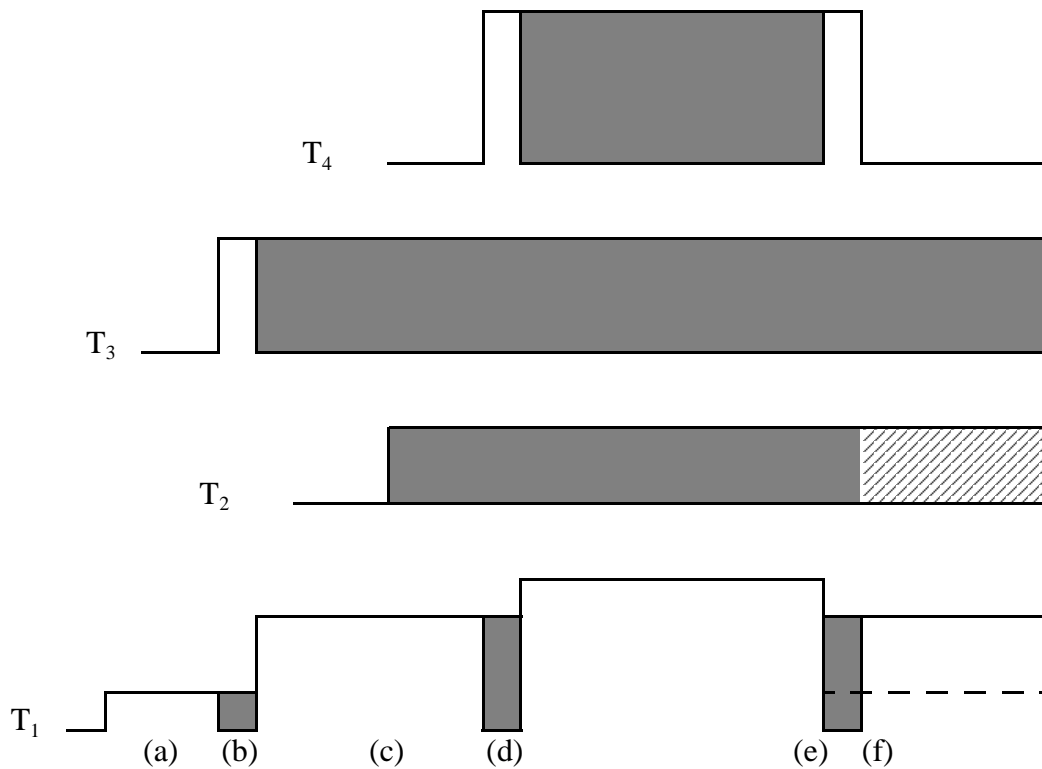


Figure 5-5: Example of Priority Inheritance

because T_1 still has lock L_{13} . Therefore T_3 may be blocked for an indeterminate time.

If strategy 3 is followed then the priority of T_1 will not drop at point (e) but will remain at priority 4 until L_{13} is released. Therefore T_4 may experience an unacceptable delay in its execution.

Definitions Useful for Priority Inheritance

As can be seen from the above example, obvious adhoc solutions can lead to delay problems with the execution of the tasks. The situation in the example is simple compared to those that occur in any reasonable size real-time system, therefore the probability of priority related delay problems would increase considerably.

In order to develop solution approaches that will work in every situation, a systematic approach to the problem must be taken. In order to do this we need some definitions. We will be assuming pre-emptive scheduling throughout this discussion. This term has the conventional meaning:

if task T has a higher (dynamic) priority than the currently running task, then there is an immediate task switch to T as soon as T enters the system, or becomes unblocked, or obtains this higher priority, as appropriate. Equivalently, a task can only run if all higher priority tasks are blocked or inactive.

The blocking of tasks in a system can be represented by a directed graph. Each node in the graph is a task, and there is a link from T_1 to T_2 iff T_2 is directly blocked by T_1 . A cycle in such

a graph would indicate deadlock. If we assume that there is no deadlock, then the graph is a “forest” of “trees”. These graphs will be used in the following examples and definitions.

Definition 1: The set $B(T, t)$ is the set of all tasks directly blocked by task T at time t. Figure 5-6 shows graphically, using blocking trees, the meaning of this definition.

Definition 2: The set $B^*(T, t)$ is the set of all tasks directly or indirectly blocked by task T at time t. That is, $B^*(T, t)$ if $B(T, t)$ or if $B^*(T, t)$ for some $t < t$. Figure 5-7 shows graphical the meaning of this definition.

Definition 3: The set $B^*(T, t)$ is the union of _____ and the singleton set $\{T\}$.

The inclusion of T in the set _____ is not intended to mean that T can block itself. The definition is to simplify notation when it is convenient to consider that T is blocking itself (even though it actually isn't).

Notation: A task T has two priorities; a base priority _____, which is assigned by the programmer, and a dynamic priority _____ which is the priority used for all scheduling decisions.

Definition 4: A pre-emptive scheduling method is a Priority Inheritance method if scheduling is based on the dynamic priority of a task given by:

A blocked task is blocked **on** a lock and blocked **by** another task. In traditional implementations of binary semaphores, the *blocked-on* information is used (explicitly or implicitly) and the *blocked-by* information is discarded. In other words, in traditional semaphore implementations the kernel needs to know only the semaphore that a task is blocked on. When the semaphore or lock becomes available the kernel uses this information to activate the blocked task. Information as to which task *has or had* the lock is usually not needed by the kernel.

When priority inheritance is used, both sets of information are potentially needed. The blocked-on information, as in the conventional case, tells the system which task to unblock when a lock becomes available, and the blocked-by information allows the system to propagate inherited priorities.

The meaning of blocked-by needs more thought. What do we mean by saying that task T_1 is blocked by task T_2 ? We will take it to mean the following:

- (a) T_1 is trying to obtain lock L_1 , but is prevented from doing so because T_2 is holding a lock L_2 , where either $L_2=L_1$ or there is some policy in place (e.g. a grouping of locks into equivalence classes) which prevents T_1 from obtaining L_1 as long as T_2 holds L_2 ;
- (b) T_1 will be granted lock L_1 when T_1 is no longer blocked by any task, and this unblocking can happen only when the blocking task releases a lock.

The above meaning for blocked-by indicates a class of algorithms rather than one explicit algorithm. Consider the following four policies that utilise the above meaning of blocked-by.

Policy 1: T_1 is blocked by T_2 if T_2 holds the lock which T_1 is trying to obtain. This is the “obvious” definition of the blocked-by relation, but is not necessarily the best. The particular policy does acknowledge the situation where the task that has the lock is itself blocked on another lock, thereby preventing it from running and releasing the lock.

Policy 2: T_1 is blocked by T_2 if:

- (a) T_2 holds the lock L which T_1 is trying to obtain, and T_1 is at the head of the list of tasks blocked on L ; or
- (b) T_2 is the predecessor of T_1 on the list of tasks blocked on L .

Figure 5-8 shows graphically the situations for (a) and (b).

Policy 3: T_1 is blocked by T_2 if:

- (a) T_2 holds the lock which T_1 is trying to obtain, and T_2 is a ready or running task; or
- (b) the lock L is held by a blocked task T_3 , and T_3 is blocked by T_2 .

These situations are depicted in Figure 5-9. In situation (b) T_1 is blocked by T_2 even though T_2 does not have the lock L .

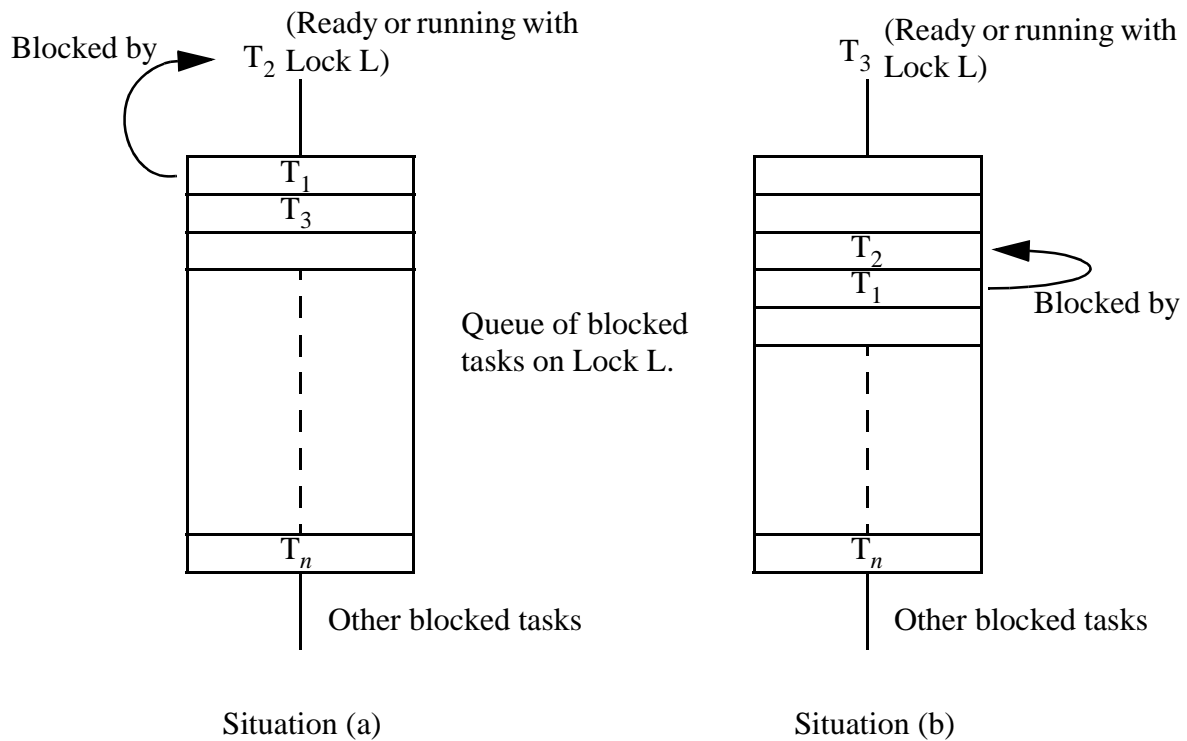


Figure 5-8: Blockage trees for Policy 2

Policy 4: Suppose that we assign priorities to locks as well as to tasks; let $LP(L)$ denote the (static) priority of lock L , as assigned by the programmer. We define T_1 to be blocked by T_2 if:

- (a) of all locks which are currently held by any task, T_2 holds the lock L of highest priority $LP(L)$;
- (b) T_1 is attempting to obtain a lock (not necessarily the same one); and
- (c) $T_1 \neq T_2$, and the dynamic priority of T_1 is less than or equal to $LP(L)$.

Policies 1–3 are equivalent from the viewpoint of the external caller, but are different in terms of internal implementation. Policy 2 minimises the width of the blocking trees, and thereby identifies the task to be unblocked when a lock is released. Policy 3, at the other extreme, eliminates transitive blocking by minimising the height of the blocking trees. As can be seen from Figure 5-9, Policy 3 flattens the blocking tree to one level, since a task is said to be blocked by the ultimate blocker, which is found by traversing the transitive blocking. For example, in situation (b) in Figure 5-9, T_1 is blocked by T_2 , even though T_2 does not have the lock that T_1 requires. However, it is ultimately T_2 that is preventing T_1 from obtaining the lock, as it holds a lock that is preventing T from running, which in turn is blocking T_3 from running.

Policy 4 is different both internally and externally from the others. Its distinctive feature is that at any time there is a *unique* “blocker”; that is, all blocked tasks are blocked by the same task. It is conjectured that this scheme is the same as a published algorithm known as the “Priority Ceiling Protocol”, however this is difficult to prove.

The following results will be useful in the next section.

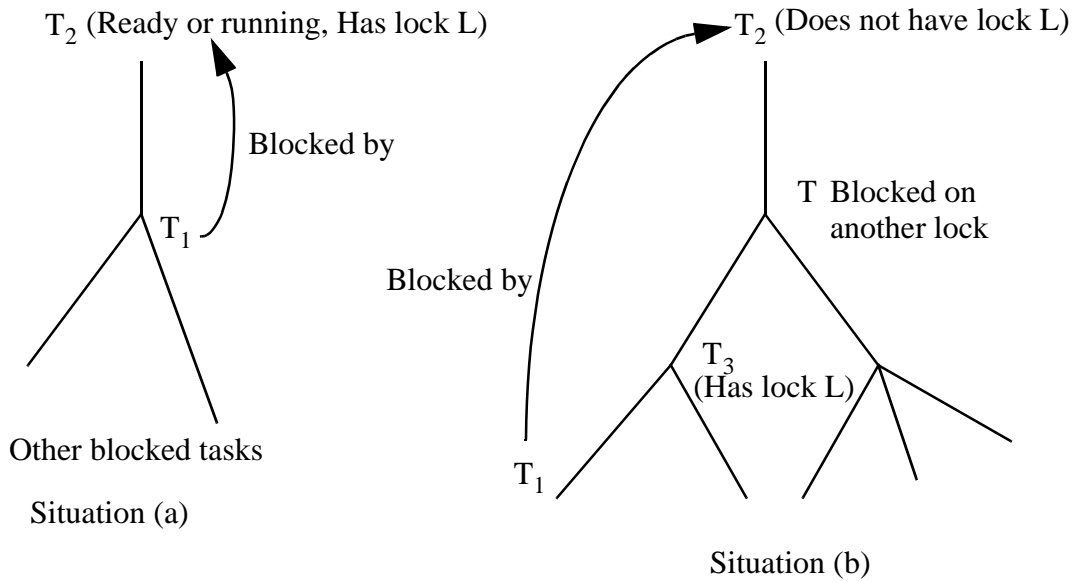


Figure 5-9: Blockage trees for Policy 3.

Lemma 1: For any task T and time t ,

$$DP(T, t) \geq BP(T_i) \text{ for all } T_i \in B^*(T, t)$$

This lemma simply says that the dynamic priority of a task T is always greater than or equal to the base priority of any of the tasks that it directly or indirectly blocks, or its own base priority.

Lemma 2: For any task T and time t there is a task

$$T_1 \in B^*(T, t) \text{ with } BP(T_1) = DP(T, t).$$

This lemma simply says that there is always a task in a blocking tree whose base priority is determining the dynamic priority of the task at the root of the tree. Note that this includes the task at the root of the tree itself.

Theorem 1: The dynamic priority of the running task is greater than or equal to that of any other active task.

Proof: See Moylan, Betz and Middleton. Note that the proof explicitly relies on the assumption that there is no deadlock in the system.

A General Priority Inheritance Algorithm

This section presents a general solution to the problem of keeping track of the priorities of tasks. The basic idea is to keep, for each task, a count of the number of tasks it is directly blocking at each priority level, and to remember, for each lock, the priority of the task which has to be unblocked when the lock becomes available.

The solution presented makes the assumption that the list of ready tasks is priority ordered, but does not make any assumptions about the ordering of tasks of equal priority. No assumptions are made about the ordering of tasks on a blocked list.

To compute the dynamic priority of a task, we do not need to keep complete information about the blocking trees; it suffices to count the number of tasks in a tree at each priority level. With each task we can associate a set of counts:

$T.count1[j] =$ the number of tasks in $B^*(T, t)$ whose base priority is j .

From definition 4, $T.count1[j]$ is obviously the largest j for which $T.count1[j]$ is non-zero. Unfortunately, $T.count1[j]$ can be expensive to compute. For example, if a task becomes blocked, it might itself be blocking a number of tasks. Therefore, the addition of this tasks blocking tree would mean that the counts for all the priority levels would have to be re-evaluated for all the tasks above that node in the blocking tree.

Some of the above overhead can be saved by working with the following counts:

where $T.count2[j] =$ the number of tasks in $B^*(T, t)$ whose base priority is j and T is blocked otherwise. Notice that this scheme requires that only the tasks directly blocked by a task T have to be considered. For example, if a new task becomes added to the bottom of a blocking tree, and this task may itself have its own blocking tree, then only the count value for the j value equal to its base priority needs to be propagated further up the tree, as compared to the whole count array for the previous count1 case. The use of the dynamic priority of the directly blocked tasks effectively captures the relevant information for the priorities further down in the blocking tree. Formally it can be shown, from Lemma 2, that $T.count2[j] = T.count1[j]$ iff T is blocked, so the count2 array still encodes the information we need but is easier to compute.

An Implementation of Priority Inheritance

Pseudo code for the **Obtain(Lock)**, **Release(Lock)** and **Promote** operations, based on the ideas above follow.

```
PROCEDURE Promote (T: Task; p: PriorityLevel);
  (* Called when T is (directly or indirectly) blocking a task*)
  (* of priority p; thus the dynamic priority of T must be *)
  (* increased if necessary to match this. If T is itself *)
  (* blocked, the priority increase ripples up the blocking *)
  (* tree. *)
  VAR oldp: PriorityLevel;

  BEGIN
    INC (T.count[p]);
    (* Get the current priority and save *)
    oldp := T.DP;
    IF oldp < p THEN
      (* If the new priority is higher then change T's priority *)
      T.DP := p;
      IF T.Blocked THEN

        (* Enter here is T itself is blocked. Have to update T *)
        (* on the blocked queue to reflect its new priority, *)
        (* and then update the count value for the task that is *)
        (* blocking T. *)

        update the position of T in T.BlockedOn.BlockedList;
        DEC (T.BlockedBy.count[oldp]);

        (* Now move up the blocking tree to the next task *)
```

Chapter 5 — *TASK MANAGEMENT*

```
        Promote (T.BlockedBy, p);
    ELSE

        (* Enter here if T is not blocked.                                *)
        update the position of T in the ready list;
    ENDIF;
ENDIF;
END Promote;

(*-----*)

PROCEDURE Obtain (L: LOCK);

    (* This procedure obtains a lock, waiting if necessary                *)

    VAR Blocker: Task;

    BEGIN
        IF the current task should be blocked THEN
            let Blocker be the task which caused the blocking;
            CurrentTask.Blocked := TRUE;

            (* Note that the BlockedBy variable is set to Blocker          *)
            (* and not L.Holder. This is to account for the                *)
            (* different policies for BlockedBy.                            *)

            CurrentTask.BlockedBy := Blocker;
            CurrentTask.BlockedOn := L;
            add CurrentTask to L.BlockedList;

            (* Now alter the priority of the blocking tasks higher        *)
            (* up the blocking tree as necessary.                          *)

            Promote (Blocker, CurrentTask.DP);
            SelectAnotherTask;
        ELSE
            L.Locked := TRUE;
            L.Holder := CurrentTask;
        ENDIF;
    END Obtain;

(*-----*)

PROCEDURE Release (L: LOCK);

    (* Releases lock L. This can lead to a change in dynamic              *)
    (* priorities, and/or a task switch.                                    *)

    VAR j, k: PriorityLevel;
        T, U: Task;

    BEGIN
        IF L.BlockedList is empty THEN
            L.Locked := FALSE;
```

```

ELSE
    choose a task T to unblock;
    remove T from L.BlockedList;
    L.Holder := T;
    T.Blocked := FALSE;

    (* Some of the tasks which were blocked by          *)
    (* CurrentTask are now blocked by T. This can occur *)
    (* because CurrentTask may have been the holder of  *)
    (* of more than one lock. Therefore some of the     *)
    (* tasks blocked by CurrentTask could have blocked *)
    (* on this second lock.                             *)

    j := T.DP;
    DEC (CurrentTask.count[j]);

    (* Now fix up the count arrays in the two blocking *)
    (* trees and the BlockedBy field in the tasks      *)
    (* blocked in lock L.                             *)

    FOR each task U in L.BlockedList DO
        k := U.DP;
        U.BlockedBy := T;
        DEC (CurrentTask.count[k]);
        INC (T.count[k]);
    ENDFOR

    (* Remark: if we consistently choose the T of highest *)
    (* DP (i.e. we choose the task from the top of a      *)
    (* ordered priority blocked list), the above operations *)
    (* will have no effect on T.DP. With some strategies - *)
    (* for example, unblocking in FIFO order, it is possible *)
    (* that T must be promoted at this stage. This would   *)
    (* occur since there may be a higher priority task     *)
    (* buried in the blocked queue.                         *)

    IF T was not the highest priority task in L.BlockedList THEN
        check the T.count array and increase T.DP if necessary;
    ENDIF

    (* Does the CurrentTask need to be demoted? This would *)
    (* occur if the dynamic priority of the CurrentTask was *)
    (* raised due to the task that has become unblocked.   *)

    IF (CurrentTask.DP = j) AND (CurrentTask.count[j] = 0) THEN
        (* Find the new lower priority for the CurrentTask *)
        decrement j until CurrentTask.count[j] > 0;
        CurrentTask.DP := j;
        insert CurrentTask into the ready list, and perform
            a task switch to T;
    ELSE
        (* CurrentTask priority should remain unaltered *)
        insert T into the ready list;
    ENDIF;
ENDIF;
END Release;

```

The algorithm above actually describes a class of algorithms, with some of the details left unspecified. For example, the `T.BlockedBy` variable would indicate different things depending on which of the four “blocked-by” policies one adopted – e.g. the `T.BlockedBy` variable would only equal `T.BlockedOn.Holder` if Policy 1 were adopted. Another point to note is that the **Release** code only covers Policies 1-3 for simplicity reasons.

A non-obvious point in the **Release** operation is the way the “blocked-by” status of some tasks changes when another task becomes unblocked. Suppose that T_1 holds L , and it unblocks T_2 by releasing L . All tasks blocked on L now become blocked by T_2 instead of T_1 (see Figure 5-10). The tasks in set S_1 are blocked on another lock held by T_1 . (Note Figure 5-10 is for the Policy 1 case, similar diagrams can be constructed for other “blocked-by” Policies).

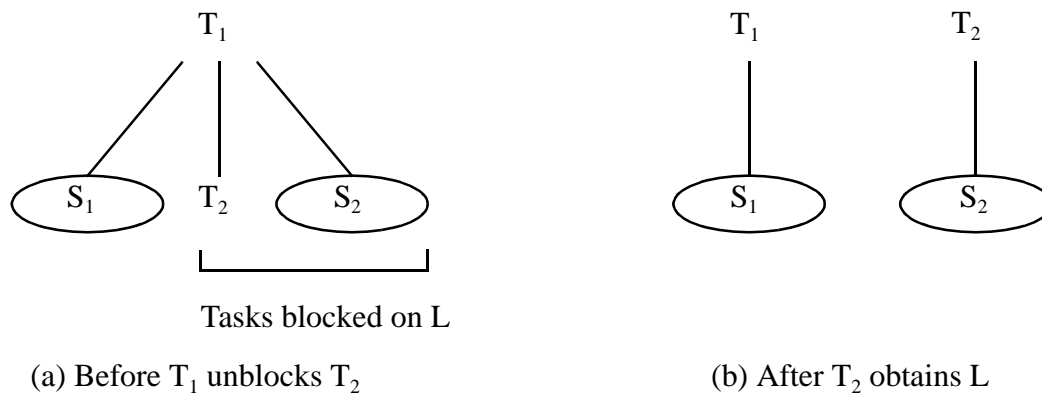


Figure 5-10: Change of blocking tree when a lock is released – Policy 1 case.

The final part of the release code relies on the following result:

Theorem 2: Suppose that the running task T_1 releases a lock at time t , and define $p = DP(T_1, t^-)$. Then, immediately after time t ,

- (a) there does not exist any ready task with a dynamic priority greater than p ; and
- (b) there exists a ready task T_2 (with the possibility that $T_1 = T_2$) such that

Proof: See Moylan, Betz and Middleton. Based on Theorem 1, Lemma 1 and Definition 4.

This implies that one of two things must happen: either the running task (whose priority must be as high as any other task in the system) keeps its old priority, and therefore continues to be eligible to be the running task; or the running task drops its priority, in which case a task switch must occur to another task whose *dynamic priority is the same as the old priority of the original task*. These two possibilities are distinguished by looking at the “count” fields.

An interesting observation is that the dynamic priority of the running task remains constant, although the identity of the running task may change, except if the running task becomes inactive (due to semaphore blockage), or a new task, of higher priority than the running task comes into the system (due to unblocking on a semaphore for example). This is because when the running task becomes blocked it is always replaced by a task of the same dynamic priority, either because it already had that priority or because it gained it through inheritance. When the running task drops its priority, there is always another task of the same priority to replace it, from Theorem 2.

This observation has an important consequence for practical implementations. As will be seen in the next section, we can sometimes avoid having to keep an explicit record of the dynamic priority of each task. Instead, we simply record the dynamic priority of the running task. From Theorem 2, this changes only when tasks enter or leave the system.

Some Explicit Implementations

This section briefly discusses actual implementations of the Lock and Release mechanisms, based on the different definitions of “blocked-by”. All the implementations were coded and tested.

The Inherit1 Algorithm

This is based on “blocked-by” Policy1. The implementation is virtually identical to that of the previous section. A minor difference is that there is no need to maintain any “BlockedBy” information, since this information is available in the “Holder” part of the Lock record.

The code is compact and reasonably efficient, and has the advantage that it is fairly easy to understand. However, it is not the most efficient implementation of priority inheritance.

The Inherit2 Algorithm

One gets a different implementation although the same external behaviour, when the “blocked-by” definition of Policy 2 is used. The obvious way to keep a task list is to use a doubly linked list, where the “previous” pointer does double duty as a “blocked-by” pointer. Superficially, this approach is attractive because tasks are kept on this list in the order in which they are going to be unblocked. In practice, problems arise because the priority inheritance implies that from time to time we have to reposition tasks on this list. It turns out that this causes major problems in updating the “count” information when the priority of a task changes. For example, if a blocked task inherits a higher priority then it may move up higher in a blocked list and the “counts” for other tasks in the blocked list will have to change to reflect the change of the priority structure of the tasks blocked by them.

We conclude that this is not an attractive algorithm for practical implementation.

The Inherit3 Algorithm

This algorithm, which is based on Policy 3, is built around the notion that the blocker of a task is its ultimate blocker. This effectively defines transitive blocking out of existence. For example, if T_1 holds a lock which T_2 is trying to obtain, and T_2 is holding a lock which T_3 is trying to obtain, then we say that both T_2 and T_3 are blocked by T_1 . Therefore the blocking trees only have unit height, and this has the following interesting consequences

- (a) task promotion is very simple, since it does not have to ripple up the tree;
- (b) the dynamic priority of a blocked task is always equal to its base priority.

The only real trouble spot in implementing this is in the redistribution, as illustrated in Figure 5-10, which must be done when a lock is released. In principle we have to re-compute the blocker of every task which is blocked by the task holding the lock. Although this is not tremendously complex, it is an overhead that we would prefer to avoid if possible.

The solution is to avoid maintaining an explicit representation of the “blocked-by” relation. Instead, a task finds its blocker by checking the holder of the lock on which it is blocked, by checking further if that task is also blocked, and so on until an unblocked task is found. This does require looping, but the loop is tight and usually only a few executions would be required.

Furthermore, this removes the need to have any explicit record of a task’s dynamic priority.

Chapter 5 — *TASK MANAGEMENT*

Theorem 2 tells us the dynamic priority of the next task to run, and it is a simple matter to find that task, again by tracing through the “blocked-on” information. Given this, it turns out that we can also do without the “count” arrays.

This is the most efficient algorithm developed. For this reason this algorithm is the one that is used to implement priority inheritance in the PMOS (Peter Moylan’s Operating System) and UNOS-V2.0 kernels. In order to implement it a new primitive is introduced in addition to the semaphore. This is known as the “lock” primitive. It is only used for critical section protection, the semaphore being used for resource counting and synchronization applications.

Pseudo code for the routines required to implement these primitives appears below:

```
PROCEDURE RunNextTask;
  (* Performs a task switch to the first task on the active          *)
  (* list whose dynamic priority is CurrentPriority, if that task    *)
  (* is not blocked. If it is blocked, we switch to its blocker    *)
  (* instead.                                                       *)

  VAR T: Task;
      L: Lock;

  BEGIN
    T := ActiveList[CurrentPriority].head;
    LOOP
      (* Now determine whether the task at the top of the queue    *)
      (* is waiting for a semaphore. If it is not waiting          *)
      (* or the lock it is waiting on is available then exit      *)
      (* this loop and switch to the task T. Else switch to the    *)
      (* the holder of the lock (which must be of lower or        *)
      (* equal priority to the current task, else it would        *)
      (* already be running.                                       *)

      L := T^.WaitingFor;
      IF (L = NIL) OR NOT L^.Locked THEN
        EXIT (* LOOP *)
      END (* IF *)
      T := L^.Holder;
    END (* LOOP *)
    TaskSwitch (T);
  END RunNextTask;
```

```
(*-----*)
```

```
PROCEDURE Obtain (L: Lock);

  (* Obtains lock L, waiting if necessary *)

  VAR Blocker: Task;
      M: Lock;

  BEGIN
    IF L^.Locked THEN

      (* Work out the blocker of the current task. The following *)
      (* loop continues looking through the tree of tasks         *)
      (* transitively blocked until it finds a task that is not    *)
```

```

        (* waiting for a lock, or the lock that it is competing for *)
        (* has become unlocked. *)

M := L;
REPEAT
    Blocker := M^.Holder;
    M := Blocker^.WaitingFor;
UNTIL (M = NIL) OR NOT M^.Locked;

CurrentTask^.WaitingFor := L;
TaskSwitch (Blocker);
END (* IF *)

(* When execution resumes at this point (via RunNextTask) *)
(* L^.Locked will be FALSE. *)

CurrentTask^.WaitingFor := NIL;
L^.Locked := TRUE;
L^.Holder := CurrentTask;
END Obtain;

(*-----*)

PROCEDURE Release (L: Lock);

    (* Releases lock L. This implicitly changes the "blocked by" *)
    (* status of all tasks blocked by the current task, but it turns *)
    (* out to be faster to let procedure RunNextTask to recheck the *)
    (* blocking status than to keep track of "blocked by" information. *)
    (* All we need to know here is whether we've inherited a priority *)
    (* greater than our base priority. If so, we must have been *)
    (* blocking another task which should run now - we can trust *)
    (* RunNextTask to work out the identity of that task. *)

BEGIN
    L^.Locked := FALSE;
    L^.Holder := NIL;
    IF CurrentTask^.BP < CurrentPriority THEN
        RunNextTask;
    END (* IF *)
END Release;

```

The tests on the **Locked** section of the lock structure in the **RunNextTask** and **Obtain** functions are required because there is not a blocked list for the locks in the system. This means that the **WaitingFor** section of the task control block is not updated when a lock that a task is blocked on is released. Consequently, it is possible to have the **WaitingFor** parameter pointing to a lock, but the lock itself has been released since the **Obtain** request was made.

As an example of where this test is important consider the following scenario. We have two tasks, of equal priority, ready to run in a system which supports time slicing. The tasks are competing for a lock, L_1 . Task T_1 obtains L_1 , and then there is a time slice causing task T_2 to run. Task T_2 already has a lock L_2 and attempts to obtain L_1 and becomes blocked. Conse-

quently, the `WaitingFor` field in T_2 's task control block indicates that it is waiting for L_1 . At this point L_1 .**Locked** is true. When T_2 becomes blocked a task switch occurs back to T_1 , which continues to execute and leaves the L_1 protected critical section. Upon leaving the critical section, there is *not* a task switch, since the base priority of T_1 is equal to the current priority.

Prior to the next time slice, another higher priority task, T_3 , enters the system. This task wants L_2 , and consequently has to trace through the blocked list. The holder of L_2 is T_2 , and T_2 is waiting on L_1 . However, L_1 is now unlocked, therefore T_2 can now run at the priority of T_3 .

As can be seen from this example, the check on L_1 is crucial, since T_2 would not run otherwise, and therefore would not unlock L_2 .

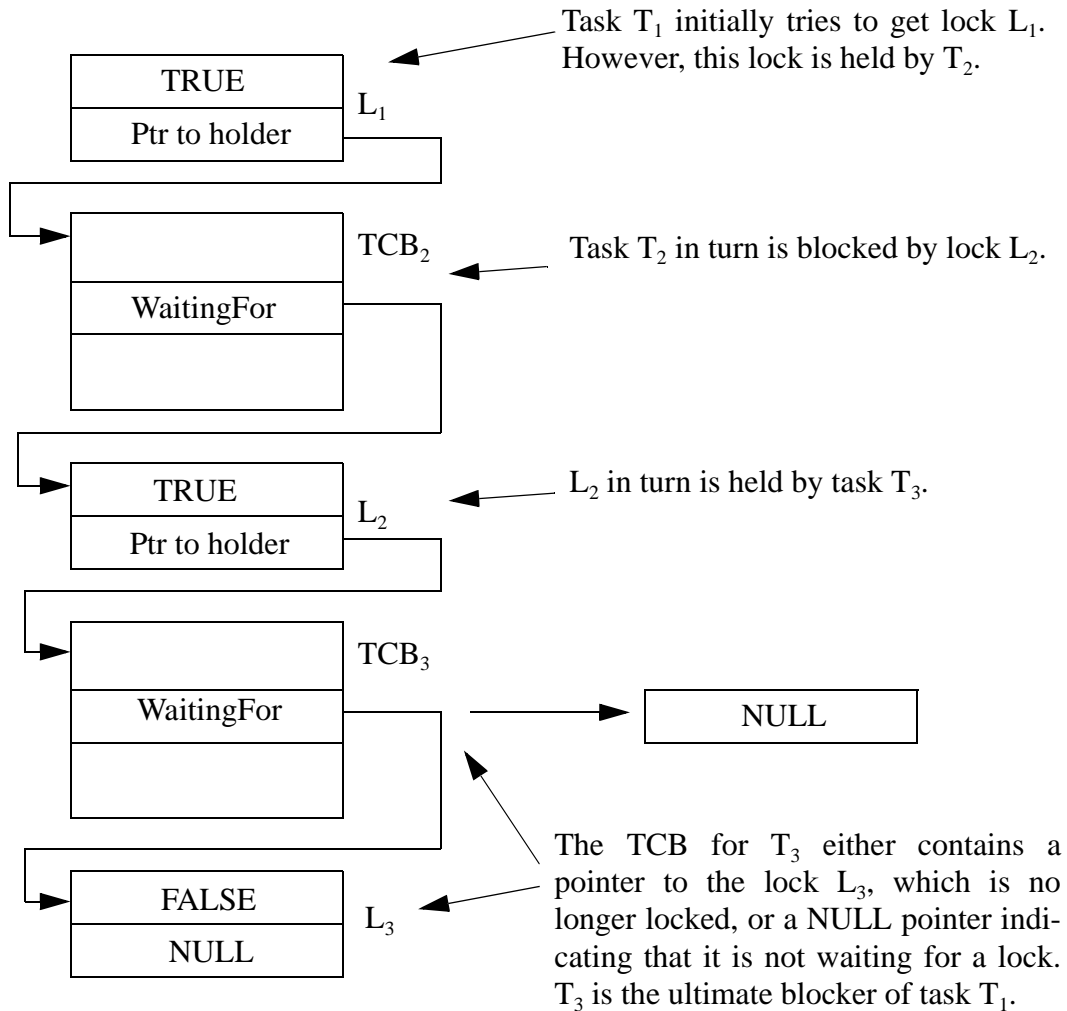


Figure 5-11: Example of the tracking through lock structures and TCBs to find the ultimate blocker of a task.

The process of searching through the linked list of locks is shown diagrammatically in Figure 5-11. In this scenario, the running task T_1 (which by implication is the highest priority runnable task in the system) has requested L_1 , but L_1 has already been locked by another task, task T_2 . The T_2 is waiting for another lock, L_2 , which in turn is held by T_3 . Therefore, the effective ultimate holder of the L_1 is T_3 . As can be seen from Figure 5-11 the ultimate blocker is found by traversing the linked list created by the Lock structure and the **WaitingFor** field of

the TCB of the tasks. Notice, as mentioned previously, a check has to be made of the **Locked** boolean variable of Lock structure to make sure that it has not been cleared when unlocked (this is required since there is no backward connection from the Lock structure to the TCB that is waiting for it).

Interesting Benefit

An interesting observation can be made about this system for handling priority inversion:

Deadlock can be detected using the lock approach.

The easiest way to see this is by an example. Consider the situation shown in Figure 4-8. This is a typical deadlock scenario – the so-called deadly embrace. As discussed in Chapter 4, this situation can result when conventional critical section protection using semaphores is used. Now consider the same situation as depicted in Figure 4-8, but using lock primitives. In the lock primitives, the lock variable stores the blocker in the holder location of the **LOCK** primitive. Therefore, the circularity of the deadly embrace situation can be detected, since as one transverses through the holder locations to find the ultimate blocker, one would return to the task at the top of the blocked queue. Once this situation has been detected then some other action would have to be taken to relieve the situation.

It should be noted that another well known algorithm for handling priority inversion (the priority ceiling protocol [9]) prevents deadlocking from occurring all together. This is effectively achieved by the algorithm implicitly implementing a restricted resource allocation policy.

Higher Order Scheduling

Thus far we have considered how to handle tasks which have priorities assigned *apriori* so that the scheduling of the tasks is carried out in a manner which does not unduly usurp the priority of the task, even when tasks of differing priorities are sharing resources. However, there has been no consideration as to how to choose the priorities of the tasks so that the system will perform in a satisfactory manner.

For a hard real-time system the criteria for “satisfactory performance” is very stringent – each critical task has a deadline which must always be met. If the deadline is not met then the consequences are catastrophic. Therefore, a technique for determining whether each task in a set of tasks will always satisfy their respective deadlines is required.

The following discussion is largely based on two semi-tutorial papers [10] and [11] which discuss Rate-Monotonic Scheduling. This scheduling technique has become a de facto standard for scheduling hard real-time systems. For example, this technique is used for scheduling the real-time control tasks in the space shuttle computers. The presentation here is not designed to be comprehensive, but it should give the reader a working knowledge of the fundamental ideas behind rate-monotonic scheduling, and how to set up this strategy for a real system.

Some Definitions and Background

Before looking at the details of the algorithm it is prudent to define some of the terms used in the real-time operating systems literature.

The measures of merit used to evaluate real-time systems may significantly differ from those typically used for other systems. In particular, such measures include:

schedulability — the degree of system loading below which the task timing requirements can be ensured with at least one scheduling policy.

responsiveness — the latency of the system in responding to external events.

stability — the capability of the system to guarantee the time requirements of the most critical tasks in the cases where it is impossible to guarantee the time requirements of all tasks (a typical case is the so-called *transient overload* condition).

There are three kinds of real-time tasks, depending upon their arrival pattern:

periodic — the task has a regular inter-arrival time, call *period*.

sporadic — there is a lower bound on the task inter-arrival time.

irregular — the task can arrive at any time.

Whilst hard real-time scheduling has its roots in the operations research field, there are some important differences – namely that operations research results are usually based on queuing theory or stochastic analysis which is not satisfactory when deterministic results are required, and secondly most operations research analysis is for a single occurrence of the tasks and not repeated invocations of the same task.

The following variables are defined to describe the key timing parameters of tasks:

- C_i is the (worst case) computation time of task τ_i .
- T_i is the invocation (or arrival) period of task τ_i .
- D_i is the deadline of task τ_i .
- R_i is the first invocation (or arrival) time of task τ_i .

The T_i period time usually corresponds to the periodic repetition rate of a control task. The deadline time is usually related to a time *from* the R_i time.

Now for a few more definitions:

job — each invocation of a task in an infinite series is called a *job*.

release time — is the time when a task is ready to be executed (usually corresponds with the beginning of a new period). This would correspond to the time when a task becomes ready or runnable in UNOS jargon.

completion time — is the time when a tasks execution is completed.

response time — is the difference between its completion time and its release time.

A simple example of the use of these variables is illustrated in Figure 5-12. One can see that the j th job of task τ_i is ready for execution at time $R_i + (j-1)T_i$. It required C_i time units in order to execute, therefore it must be completed no later than $R_i + (j-1)T_i + C_i$. Therefore a periodic task τ_i is characterised by a quadruple (τ_i, T_i, D_i, R_i) .

In the analysis following it shall be assumed that the system is pre-emptive. This means that a task's execution can be suspended at any time and another higher priority task can run. The original task can then run once the high priority task has completed execution.

The algorithms used in practice for scheduling in hard real-time systems are *priority-driven* pre-emptive algorithms. These algorithms assign priorities to tasks according to a policy. After this the system chooses the highest priority task to run at any particular point in time, even if this involves pre-empting a lower priority task.

Assuming one has a task set τ , and a priority assignment is assigned to each of the tasks in this set. There are several definitions that can be applied to this situation:

feasible priority assignment — a priority assignment to a task set is said to be *feasible* if all deadlines of all tasks for all jobs are met with the priority assignment. Therefore feasibility

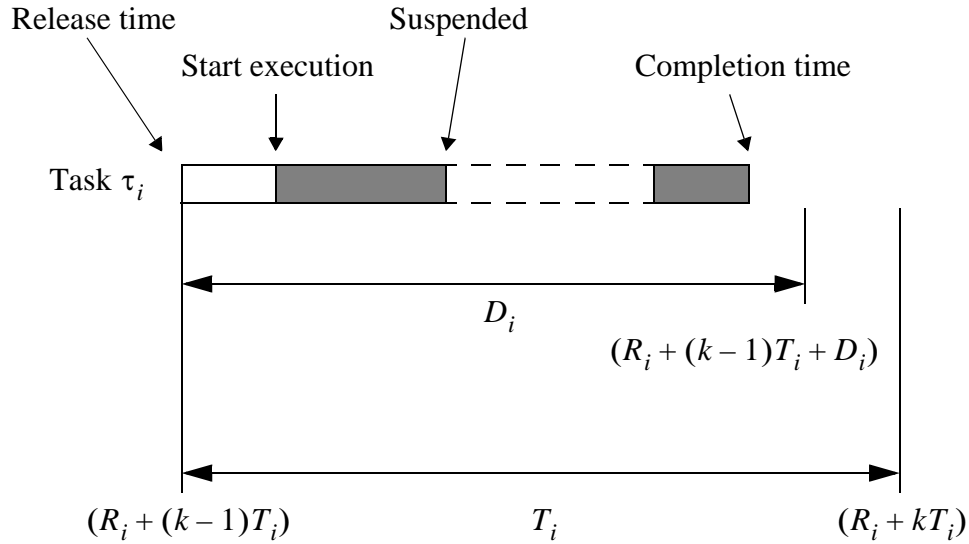


Figure 5-12: Periodic task timing.

ity is related to the assigned priorities.

schedulable task set — a task set is said to be *schedulable* by the assignment algorithm that produced the feasible priority assignment. In other words a task set is schedulable by an algorithm if all the task deadlines are met by the schedule produce by the algorithm.

optimal scheduling algorithm — a scheduling algorithm is *optimal* if *all* the task sets for which a feasible priority assignment exists are schedulable by that algorithm. In other words if there is some arbitrary algorithm that operates on some arbitrary task set to produce a feasible set of priority assignments (i.e. the task set can be scheduled by this algorithm), then an optimal algorithm can also produce a feasible priority assignment for the same task set (i.e. the optimal algorithm can also schedule the task set). Therefore if the task set is schedulable by any arbitrary algorithm, then it can also be scheduled by the optimal algorithm as well. The word “optimal” is a bit of a misnomer since there is nothing to optimise – there is no objective function to minimise.

The remainder of this section will concentrate on *static* scheduling algorithms. A static algorithm is one in which the priorities for tasks are assigned initially and then they remain the same from then on. An example of such an algorithm is the *Rate-Monotonic* scheduling algorithm. On the other hand a *dynamic* algorithm is one in which the priorities of tasks may change on each invocation. An example is the *Earliest Deadline First* algorithm, in which the task with the highest priority is the one with the nearest deadline. It has been proven [12] that this algorithm is an optimal priority-driven scheduling algorithm, and necessary and sufficient conditions for testing schedulability of the tasks was derived.

Although the Rate-Monotonic algorithm is not optimal (it is only optimal among all static scheduling algorithms only), its predictability and stability under transient overload is so appreciated that it is becoming an industry standard.

Basic Rate-Monotonic Analysis

This section shall consider the most basic results for a fixed priority pre-emptive algorithm operating on a task set $\{\tau_1, \dots, \tau_n\}$. The following assumptions are made:

- all the tasks are periodic.
- $(C_i \leq D_i \leq T_i)$ for all i .
- tasks are independent, i.e. no inter-task communications or synchronisation is permitted.
- there is a single processor.

These assumptions are rather restrictive for real systems. For example virtually all real systems will involve inter-task communications. Many systems also involve non-periodic tasks. Some of these will be relaxed in analysis presented later.

For simplicity in the following analysis we shall assume that the indexes of the tasks also reflect their priority – for example τ_i has a higher priority than τ_j if $i < j$.

Fundamental Results

The following results were proved in [12]:

Theorem 5-1: *The longest response time for any job of a task τ_i occurs when it is invoked simultaneously with all higher priority tasks (i.e. when $R_1 = R_2 = \dots = R_i$)*

If all the tasks of higher priority than a task τ_i are invoked at the same time then this time is known as a *critical instant*.

Theorem 5-2: *A fixed priority assignment is feasible provided the deadline of the first job of each task starting from a critical instant is met.*

The net result of the above two theorems is that one can assume that the arrival times of the tasks are zero – i.e. $R_1 = R_2 = R_3 = \dots = R_n = 0$, since this assumption takes care of the worst possible case. Therefore only the first deadline of each task must be met when the task is scheduled with all higher priority tasks in order for a fixed priority assignment to be feasible.

Rate-Monotonic Scheduling

The Rate-Monotonic scheduling algorithm was proposed in [12] assuming that $D_i = T_i$ for all i . The algorithm was very simple – assign priorities inversely to the task periods. Therefore a task τ_i receives a higher priority than task τ_j if $T_i < T_j$. This algorithm was proved to be optimal among all static scheduling algorithms (assuming $D_i = T_i$ for all i).

Consider the task set shown Table 5-1 and the resultant scheduling by the Rate-Monotonic algorithm shown in Figure 5-13.

Table 5-1: Task set with deadlines equal to periods

Task	T_i	D_i	C_i
τ_1	100	100	40
τ_2	150	150	40
τ_3	350	350	100

Utilisation Bound

A sufficient (but not necessary) condition for schedulability of the Rate-Monotonic algorithm was derived in [12]:

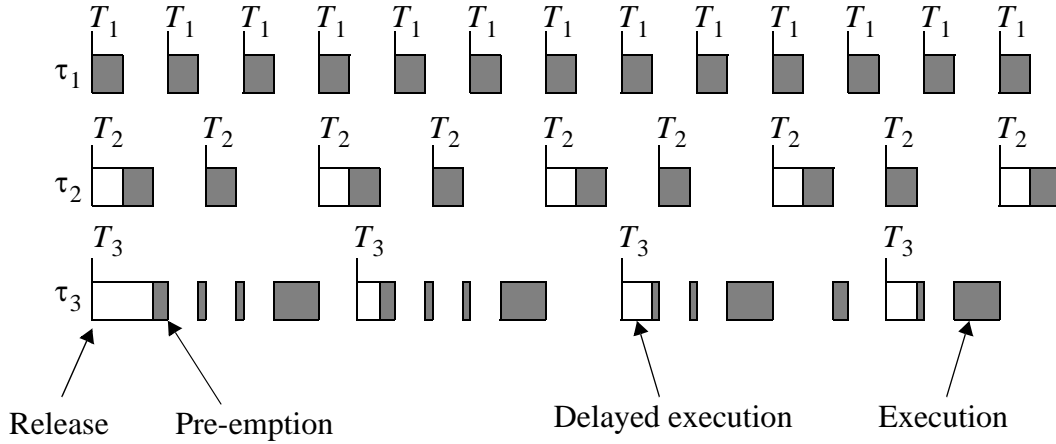


Figure 5-13: Scheduling the task set of Table 5-1 with a Rate-Monotonic algorithm.

Theorem 5-3: Given a periodic task set with $\tau_i \leq \tau_j$ for all i , the Rate-Monotonic algorithm yields a feasible priority assignment if:

(1)

The term $\frac{\tau_i}{T_i}$ represents the *utilisation factor* of task T_i . Therefore the theorem states that there is an upper bound on the total utilisation of the task set, above which the Rate-Monotonic algorithm cannot be guaranteed to give a feasible priority assignment. It should be noted that the upper bound on task utilisation is 0.693, this being derived as follows:

(2)

The fact that this is only a sufficient condition can be seen from a counter example. Consider the situation in Table 5-1. The processor utilisation in this case is:

(3)

which is considerably above the previous sufficient condition limit.

Completion Time Test

Reference [13] derived an exact analysis to find the worst case response time for a given task, assuming fixed priority, independent tasks and deadlines less than periods (i.e. $\tau_i \leq \tau_j$ for all i). The analysis considers a critical instant and makes use of Theorem 5-2. The following equation gives the worst case response time R_i of a task T_i :

(4)

where $hp(i)$ is the set of all tasks with higher priority than τ_i .

The right hand side of (4) is essentially the *cumulative processor time demand* for tasks in $hp(i)$ – i.e. the task τ_i and all the tasks of higher priority than τ_i . The symbol $\lceil x \rceil$ means the ceiling or next highest integer to x , except if x is an integer in which case it is x . Therefore:

$n(t)$ is the overall number of jobs τ_i at time t , since it gives the number of τ_i s in the time t . Given this then it is possible to deduce that:

$p_i(t)$ is the processor time *demanded* by task τ_i by time t . In other words and hence:

$P_i(t)$ is the total processor time by task τ_i and all the higher priority tasks to some time t . Due to the integer nature of the $n(t)$ term in the above expression, then it is possible to increase t until we have:

(5)

This time is denoted as t_i in (4) – this is the smallest time where (5) is satisfied. Clearly this is the total time required to run all the higher priority tasks as well as task τ_i .

The t_i is easily computed by performing an iteration, starting with $t_i = 0$ to give:

(6)

and is terminated when $t_i = t_{i-1}$. The convergence is guaranteed iff [14]:

(7)

Example:

Consider the situation of Table 5-1. Let us consider task τ_1 . the question is “does it satisfy its deadline?” Using the above algorithm we can write:

$$W_3(5) = 100 + (3)(40) + (2)(40) = 300$$

Therefore τ_3 , and therefore the worst case response time is 300 time units. Since the deadline for τ_3 is 350 time units, it satisfies its deadline.

The same result can also be worked out using a brute force approach. If one moves from left to right in Figure 5-13 using the values in Table 5-1 one can write:

Theorem 5-4: *A fixed priority assignment for a task set τ such that for each i , τ_i is feasible iff*

(8)

This is referred to as the *Completion Time Test*.

Remark: *The condition must be verified for each i . It is a common mistake to think that if the lowest priority task meets its deadline then all the tasks will meet their deadlines. This is false, since the schedulability of a task does not guarantee the schedulability of higher priority tasks.*

As an example of the situation in the above remark consider Table 5-2. The worst case comple-

Table 5-2: Task set with deadlines less than or equal to periods

Task			
	100	100	10
	200	180	170
	250	250	10

tion time of task τ_1 is trivial and is 10. Computing the worst case completion time $W_1(10)$ yields: $W_1(10) = 100$, and $100 < 100$. Thus the worst case completion time for τ_1 is 100 time units, whereas the deadline is 100. Therefore it does not meet its deadline. However, if we consider the lowest priority task τ_3 the analysis is: $W_3(10) = 100 + (3)(10) + (2)(10) = 170$. Therefore the worst case completion time for τ_3 is 170 time units, which is less than the 250 time unit deadline. A diagram of the scheduling in this case is shown in Figure 5-14.

Deadline-Monotonic Scheduling

In the Rate-Monotonic algorithm the deadlines are assumed to coincide with the end of the task

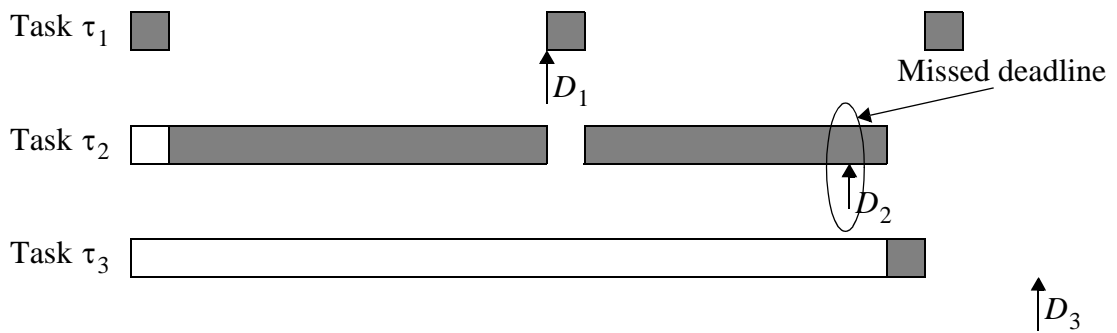


Figure 5-14: Example showing that schedulability of low priority tasks does not guarantee that of higher priority tasks.

periods. However, in the situation where $D_i \leq T_i$ the task has to complete in general within the time period, and under this condition it was proved [12] that giving *higher priorities to tasks with narrower windows is optimal among fixed priority algorithms*.

It was subsequently proved in [15] that when $D_i \leq T_i$ (i.e. deadlines are less than or equal to the task periods), the Rate-Monotonic priority assignment is no longer optimal. However, assigning higher priorities to tasks with narrower windows is still optimal among fixed-priority algorithms. This algorithm was called the *Deadline-Monotonic Scheduling Algorithm*. With this algorithm, the task having the smallest deadline is assigned the highest priority. In other words, τ_i has a higher priority than τ_j whenever D_i is smaller than D_j .

Relaxing the Assumptions – Generalised Rate-Monotonic Analysis

The previous results were derived under a restrictive set of assumptions that helped in the development of basic theoretical results. The most restrictive of these assumptions were that there was not inter-task communication or synchronisation in the system, and that all the tasks were periodic.

Task Synchronisation

As noted in “Dynamic Priorities” on page 5-18 if one has tasks with different priorities interacting with a common resource then there is the possibility of *priority inversion* occurring – i.e. the high priority task is blocked on a resource held by a low priority task, effectively meaning that the high priority task has the priority of the low priority task until the resource is released. It was also noted that the high priority task can be prevented from running for an unbounded length of time.

A solution to this problem was the *priority ceiling protocol* [9]. This algorithm uses the concept of priority inheritance similar to that described in “Dynamic Priorities” on page 5-18. The priority ceiling of a semaphore S is defined as the priority of the highest priority task that may lock S . For example, if a task τ_k uses a semaphore S_1 , and τ_k is the highest priority task using this semaphore, then the priority ceiling of S_1 is $P(k)$ (the priority of task τ_k).

Consider $S^*(i)$ to be a semaphore with the highest priority ceiling among all the semaphores currently locked by tasks *different from* τ_i . Task τ_i can lock another semaphore only if its priority is strictly higher than the priority ceiling of $S^*(i)$ (noting that $S^*(i)$ has to be locked, if it

isn't then τ_i can obtain the other semaphore), otherwise it gets blocked on S_2 (it will be awakened when the above condition becomes true). If τ_i cannot obtain the semaphore then it essentially becomes suspended until the task that has a semaphore at the ceiling priority releases the semaphore.

The above generic explanation of the ceiling protocol is a bit confusing. It is easier to get a grasp of how the protocol works from a specific example. The following example essentially comes from [9]. This example also demonstrates a key property of the priority ceiling protocol – the most that a high priority task can be delayed is the longest critical section shared with any other task. One cannot obtain chained priority inversion (also known as chained blocking) using this system¹. Therefore the maximum delay has a predictable maximum boundary that is lower than other techniques.

Consider three tasks (or jobs) denoted as τ_1 , τ_2 and τ_3 . τ_1 is the highest priority of these tasks. τ_1 needs access to semaphores S_1 and S_2 *sequentially* for critical section purposes, while τ_2 accesses S_1 and τ_3 accesses S_2 for critical section purposes. The following discussion is shown diagrammatically in Figure 5-15. The highest priority task that may access a semaphore sets the priority ceiling of the semaphore, therefore the priority ceiling of S_1 and S_2 is p_1 (the priority of τ_1). Let τ_1 start at t_0 and then obtain the lock S_1 . At time t_1 job τ_2 is initiated and preempts τ_1 . However, at time t_2 when τ_2 attempts to lock S_2 the run-time system would find that the priority of τ_2 is not higher than the priority ceiling of S_2 of the *locked* semaphore S_2 , and hence τ_2 would become blocked on trying to obtain the semaphore. Because τ_2 has become blocked on a semaphore that τ_1 has, then τ_1 inherits the priority of τ_2 and resumes execution. Before τ_1 exits the S_1 critical section τ_3 is initiated. It of course preempts τ_1 (which is currently running at p_2 priority) and then attempts to obtain the lock S_2 , which it cannot obtain because the priority of τ_3 is not strictly higher than the priority ceiling of S_2 (which is p_1). Therefore τ_3 inherits the priority of τ_2 and then executes until it releases S_2 , and at this point τ_2 can now run. The τ_1 priority drops back to p_1 since it no longer blocks any task of higher priority. τ_2 can now run unhindered and release lock S_2 . It can then obtain and release lock S_1 . When it completes (i.e. terminates) τ_3 can then run and obtain the S_2 lock.

The important bit about the last section is that when τ_2 became blocked there was only one lock current in the system – τ_1 had not been able to obtain S_2 , and consequently when τ_1 executed after τ_3 had released S_2 it could obtain S_2 without any blockage. Therefore τ_2 was only delayed for one critical section (and not two as would have been the case if τ_3 had have obtained the S_1 lock).

1. Chained blocking of a high priority task can occur when the high priority task is serially entering critical sections. If these critical sections have already been obtained by other tasks then the high priority task can be repeatedly delayed until the low priority tasks release the shared resources.

J_1	J_2	J_3	Time	Comment
		Lock(S_1)	t_0	
	.		t_1	J_2 becomes active and preempts J_1 .
	.			
	Attempt at Lock(S_2) – suspended	.	t_2	J_2 cannot get S_2 because the task priority is not higher than the priority ceiling P_1 obtained from S_1 . J_3 resumes with priority 2.
		.		
		.		
		.		
.			t_3	J_1 becomes active and J_3 is preempted (remember J_3 now has priority of 2).
.				
.				
.				
Lock(S_1) – suspended		.	t_4	J_1 is suspended because its priority is not higher than the current semaphore ceiling priority of P_1 . J_3 inherits priority 1.
		.		
		.		
		.		
		.		
		.		
Obtain S_1		Unlock(S_1)	t_5	$J_3 \rightarrow 3$ and control reverts to .
.				
.				
Unlock()				serially obtains
Lock()				and . It then finishes
.				allowing to run. The
.				priority ceiling is still
Unlock()				since is used by
.				.
Terminate	Obtain			
	.			releases and
	.			continues execution.
	Unlock()			There is no priority
	.			ceiling in the system
	.			now.

An easier way to view the operation of this is to realise that the pool of semaphores used by the high priority task have a priority ceiling equal to the priority of that task once a semaphore out of the group is used by any other task. Once this happens no other task can get access to any of semaphore pool until the task that has the one semaphore has finished with it. Therefore the high priority task cannot be serially blocked (or chain blocked) because only one of the semaphores in the pool it uses can be locked. The task that has the lock will inherit the priority of the highest priority task trying to access the pool of semaphores.

The basic theorem proved in [9] is

Theorem 5-5: (i) A task τ_i can be blocked (by lower priority tasks) for at most C_i time units, where C_i is the duration of the longest critical section executed by a task of lower priority than τ_i guarded by a semaphore whose priority ceiling is greater than or equal to the priority of τ_i .

(ii) The priority ceiling protocol is deadlock free.

It should be noted that the above example of the action of the priority ceiling protocol implies that the interaction between the tasks could be implemented by having a single semaphore for critical sections at each priority level. If a high priority task and low priority task are interacting then the semaphore would be at the priority level of the highest priority task. Therefore instead of τ_i interacting with S_i and τ_j it would only interact with S_j , a single semaphore for priority j . Therefore τ_i would sequentially access S_j instead of S_i and S_j . The execution scenario is similar to the previous example and would proceed as follows: τ_i starts execution at t_0 and obtains S_j . Then τ_j starts and preempts τ_i , but it also attempts to access S_j and is blocked, since S_j has already been claimed. Therefore τ_i continues execution, having inherited the priority of τ_j . Whilst still in its critical section τ_j starts and attempts to obtain S_j , but it becomes blocked as well. τ_i then inherits the priority of τ_j and continues to execute until it finishes its critical section. τ_j can then obtain the semaphore. This assumes that the semaphore queue is priority ordered. When it finishes the first of its two critical sections it continues to execute since it has a higher priority than τ_j (which is blocked on S_j). Therefore it is able to execute the second of the critical sections. When τ_j releases the processor then τ_i can run and obtain the semaphore. Note that using this semaphore arrangement we obtain the same effect as the previous example (from [9]). This possibility was not seen in [9].

One obvious question one can ask from the previous scenario is; “what happens if the two critical sections used by τ_i are nested, as opposed to be serially arranged?”. The answer is that it does not make sense to have nested semaphores when there is only one semaphore per priority level – the single semaphore is the total critical section, and no nesting is required to achieve the protection.

The above observation means that a priority ceiling protocol can be implemented using the UNOS lock system by having a lock associated with each priority level. Any low priority tasks interacting with a high priority task would do so via the single lock associated with the high priority task priority level. Since there is only a single lock for each level, which is used for interactions with tasks of a lower priority, then it is not possible for more than one lock to be

obtained by lower priority tasks. This technique is effective at eliminating chained blocking, but it breaks down in relation to *transitive* blocking. If transitive blocking occurs then the blockage time of a high priority task could no longer be just one critical section. The priority ceiling protocol prevents transitive blocking from occurring. In order to prevent transitive blocking with the single semaphore per priority arrangement the discipline of the priority ceiling protocol must be applied.

It should be noted that having a blockage length for only the maximum critical section length is being obtained at the cost of blocking other tasks when they could run under a pure priority inheritance scheme. For example, in the above scenario task J_2 is blocked when it attempts to obtain S_{p_1} , whereas in a pure priority inheritance scheme J_2 would be able to run at this time (instead of J_3 as in the first scenario).

The completion time expression in equation (6) can be modified to account for the maximum blockage time as follows:

$$W_i(k+1) = C_i + B_i + \sum_{j \in hp(i)} C_j \left\lceil \frac{W_i(k)}{T_j} \right\rceil \quad (9)$$

where B_i is the longest duration that task τ_i can be blocked by lower priority tasks. If the priority ceiling protocol is being used then this would be the time of the longest critical section.

If pure priority inheritance is being used then the worst case blockage time is much more complex to calculate due to the fact that transitive as well as chained blocking can occur.

Release Jitters

In addition to the blocking time, one can also account for any jitter in the starting of tasks. Jitter can occur due to interrupt latency, different techniques of putting a task onto the priority queues etc. It can be shown that the completion time expression can be modified to account for this as follows [14]:

$$\begin{aligned} W_i^* &= C_i + B_i + \sum_{j \in hp(i)} C_j \left\lceil \frac{W_i^* + J_j}{T_j} \right\rceil \\ W_i &= W_i^* + J_i \end{aligned} \quad (10)$$

Arbitrary Deadlines

Another relaxation of the basic assumptions is to allow tasks to have arbitrary deadlines (i.e. deadlines greater than periods). Under this condition Theorem 5-2 is no longer valid – a task meeting its first deadline is not guaranteed to meet all its successive deadlines, and neither the Rate-Monotonic or the Dead-line Monotonic algorithms are optimal anymore. The results of [12] were generalised for the case in which $D_i = kT_i$ with $k = 1, 2, \dots$ (the same constant k for all tasks) [16]. Tindell and al. [14] extended the Completion Time Test, providing an exact test for tasks with arbitrary deadlines.

For references and information the scheduling of sporadic tasks, sporadically periodic tasks and irregular tasks refer to [10] (available for my Web site:

<http://www.ee.newcastle.edu.au/users/staff/reb/>)

Intertask Communications

A facility that any multi-tasking operating system must provide is some means of allowing tasks to communicate with each other. Rarely can a task operate in isolation from the rest of the system; indeed most real-time systems are composed of a number of tasks that operate together to perform some function. This implicitly means that the tasks must be sending information to each other about the progress of their particular job in the system.

Some of the simpler forms of intertask communication have already been introduced. In the section “Task Synchronization” on page 4-6, the concept of a synchronization semaphore was introduced. This is also a very primitive form of task communication, where the right to proceed is being communicated between tasks. This operation is essentially supported by the operating system, since semaphores are an operating system entity.

When most of us think of intertask communication we probably think of sending messages between tasks. There is a number of ways of doing this, and the choice of a method for a particular operating system is a matter of the personal preferences of the operating system designers. I shall outline a few of the techniques that have been used emphasizing their advantages and disadvantages.

Common Data Buffer Technique

Probably the most common method of communication between tasks is to use shared memory. This involves setting aside an area of memory which is made accessible to both tasks. In a non-protected mode operating system this involves making the buffer area’s name available to both the tasks. For a protected mode operating system the common data area has to be mapped into the data address spaces of both the tasks. Data is written into the common buffer area by one task and read by the other. Obviously this communication technique is intrinsically bidirectional. It should also be clear that the read and write operations to this area constitute a critical section, and consequently have to be protected by a critical section semaphore.

The merits of the common buffer technique are:

- It is simple implement.
- It is fast - i.e. it allows the fast and efficient communication of large amounts of information.
- It is inherently bidirectional.
- It allows asynchronous operation of the sending and receiving tasks – i.e. the sending task does not have to wait until the receiving task gets the message.

The disadvantages of the technique are:

- It does not include a technique to indicate that new data is in the buffer. This would have

to be built into the data passed in the buffer.

- If a task is waiting on data it would have to carry out a **wait** on an auxiliary semaphore for the sending task to put the data into the buffer and then do a signal on the semaphore
- If repeated messages are to be written into the buffer by the sending task then additional synchronization has to be implemented to prevent the sender from overwriting the data before the receiver has received the data.
- Because the data area is common to the tasks using it there is no way to prevent a task from accessing the data area, bypassing the normal access techniques. This is important when large and complex systems are being built by a number of programmers. There is always someone that will not use the correct access techniques.

Some of the above limitations can be overcome by “packaging” the extra synchronization into access routines that a user calls. However, even with this the basic problem of having a single shared data area remains. This can result in the sender being blocked for indeterminate periods of time until the receiver picks up the data.

Circular Buffer Technique

The circular buffer is one of the most used data structures in software systems. It is commonly used to buffer the input from devices such as serial channels. The data that is written to and read from circular buffers is usually of the character variety. Therefore if large amounts of data have to be sent between tasks it is not a particularly efficient technique. The basic operation of a circular buffer was explained in the section “Example: Producer-Consumer Problem” on page 1-8, and will not be repeated here.

The circular buffer technique, depending on the implementation, can be considered to be a variation on the common data area technique. The circular buffer structure itself could lie in a common data area shared by two tasks. An alternative implementation could have the buffer area in the operating system space, thereby only allowing access via the interface routines.

The circular buffer approaches merits are:

- Allows a user configurable degree of buffering between the tasks so that the sender and receiver can essentially operate asynchronously from one another.
- The data is inherently read from the buffer in FIFO order.
- Properly constructed interface routines hide all the underlying semaphore operations and pointer manipulations from the user. Therefore the synchronization between the sender and receiver is automatic as far as the user is concerned. Obviously this depends on how the interface routines are implemented, but the standard task implementation of these routines have these properties.
- Very suited for single character communications.

The disadvantages of the technique are:

- It is not suitable for transferring large amounts of information due to the overhead of transferring the information into the buffer byte by byte, and then reading it from the buffer in the same fashion.
- Not very suitable from communications coming from a large number of tasks to a single task. For example, if a task has associated with it a circular buffer, and data is put into that buffer from more than one other task, then there is no mechanism to prevent the data streams from the multiple inputs from becoming garbled.
- A single circular buffer is not an inherently bidirectional entity. Therefore bidirectional

communications between tasks requires two circular buffers.

Mail Box Technique

Mail box communications is a simple extension of the circular buffer, therefore it has the same advantages of the circular buffer technique. It differs from the circular buffer approach in that it sends groups of characters instead of a single character. These groups of characters are stored in a circular buffer of messages. The advantage of the technique above the circular buffer technique is that the messages are indivisible. Therefore it is possible to have messages arriving into a mail box from multiple sources and each message will be uncorrupted. Of course it is necessary to tag each message in some way so that the receiving routine will know where each message comes from. It would be possible to do the same with the characters in the conventional circular buffer technique, but the amount of tag information would be considerably more than the message being sent.

The disadvantages of the technique are:

- As with the circular buffer technique it is not suitable from transferring large amounts of information, as it involves copying information into and out of a buffer. In addition the transfer of large amounts of data would require the allocation of large amounts of memory for each message.
- It is not inherently bidirectional.

Pointer Transfer Technique

This technique involves transferring a pointer to a data area between tasks, and not actually copying the data itself. In many ways it is very similar to the common data area technique, since the receiving routine simply gets access to a piece of memory that contains some information. It differs in that one is not restricted to the one data area, but a number of different pointers can be transferred to any receiving routine.

By combining this technique, say with the mail box mechanism described above, one can obtain a very good technique for transferring large amounts of data. The mail box is modified so that instead of mailing conventional messages, pointers are instead sent. These pointers point to the memory areas that contain the data. If the operating system supports dynamic memory allocation then these data areas can be allocated from the heap as needed by the sender task. When the receiver has finished reading the data from the memory area it is its responsibility to free the dynamically allocated memory. The dynamic memory technique has the advantage that the memory area with the message is effectively owned by the receiver. Therefore the sender will not get blocked waiting for the data area to be cleared, as is the case with the common data area model.

The main advantage of the pointer transfer technique coupled with the mail box system is efficiency when transferring large amounts of data. The data may have to be transferred into the dynamically allocated area (although in some cases it may be a data structure being used in the sender), but it will not have to be copied into another buffer at the receiver.

In summary this technique is able to capture all the good points listed for the mail box communication technique, whilst at the same time capturing the good properties of the common data area technique. The circular buffer transfer technique is effectively a subset of this technique. The only problem with this technique are:

- It is slower when transferring small messages, mainly due to the overhead of allocating and deallocating dynamic memory.
- The constant allocation and deallocation of memory from the heap, especially if for

small messages, could possibly lead to heap fragmentation problems.

Examples of Intertask Communication in Some Real Operating Systems

Unix

The UNIX operating system is a very commonly used system in today's workstations. It is a general purpose multi-tasking/multi-user operating system. It was not designed for real-time applications, although there are real-time versions of it commercially available. The intertask communications in the system is mainly based on semaphore signalling and pipes. The semaphore communication is along the lines of that outlined in the general discussion in section "Intertask Communications" on page 6-1. The pipes are character based communication channels between tasks in the system. They effectively work like the circular buffer communication channels explained above, although the actual internal implementation may be quite different. For example, in some systems pipes open a file on the disk which is written to by the sender and read from by the receiver. It is the file system routines that carry out the buffer pointer updating and critical section protection etc.

UNOS

UNOS 1.5a currently has explicit support for the mail box communication mechanism described above. This of course does not prevent one from also using the common data buffer technique for large transfers. The sending of messages involves the use of two routines:

- (i) **send_mess**
- (ii) **rcv_mess**

See "SEND_MESS DESCRIPTION" on page C-36 and "RCV_MESS DESCRIPTION" on page C-40 for detailed descriptions of how these functions work.

The overall operation of the mail box system has been designed to be very simple to use from the users point of view. When a task is created a mail box for the task is automatically create as well. The number of message slots in the mail box and the size of each slot in terms of characters is specified by the user at task creation time. In order to identify the mail box each task is given a text name (i.e. the name is independent of the task number), the pointer to which is the address of the mail box. Therefore, any task which has been created can be communicated with by simply sending a message to the task's address. Because a text name is used, its address is invariant with the addition or deletion of tasks from the system (assuming that the task exists at all). A task can only read from its own mail box – the **rcv_mess** routine fixes the address to the task's own name.

In order to map the task name pointer to a task number, which in turn is used to index into a table of task name structures (which contain amongst other fields the task number), a chained hash table is used in the function called **mail_exchange**. This hash table is built at task creation time. If the retrieval of the task number associated with a task name is successful, then it is used to index into the table of mail box structures. Once this is available then the message can be written into the mail box. Note that the **mail_exchange** function is only required when sending messages to a task. When receiving, the mail system routines can simply find the task number from the current task tcb. Figure 6-1 shows diagrammatically this address mapping process. Once the index into the mail box pointer table is produced then address of the mail box structure for the particular task is known.

The mail box structure itself is of some interest. It is composed of a base level structure with a variety of fields, including another nested structure. Figure 6-2 shows the structure of a mail box and the relationship of all its components. The envelope structure encapsulates all the

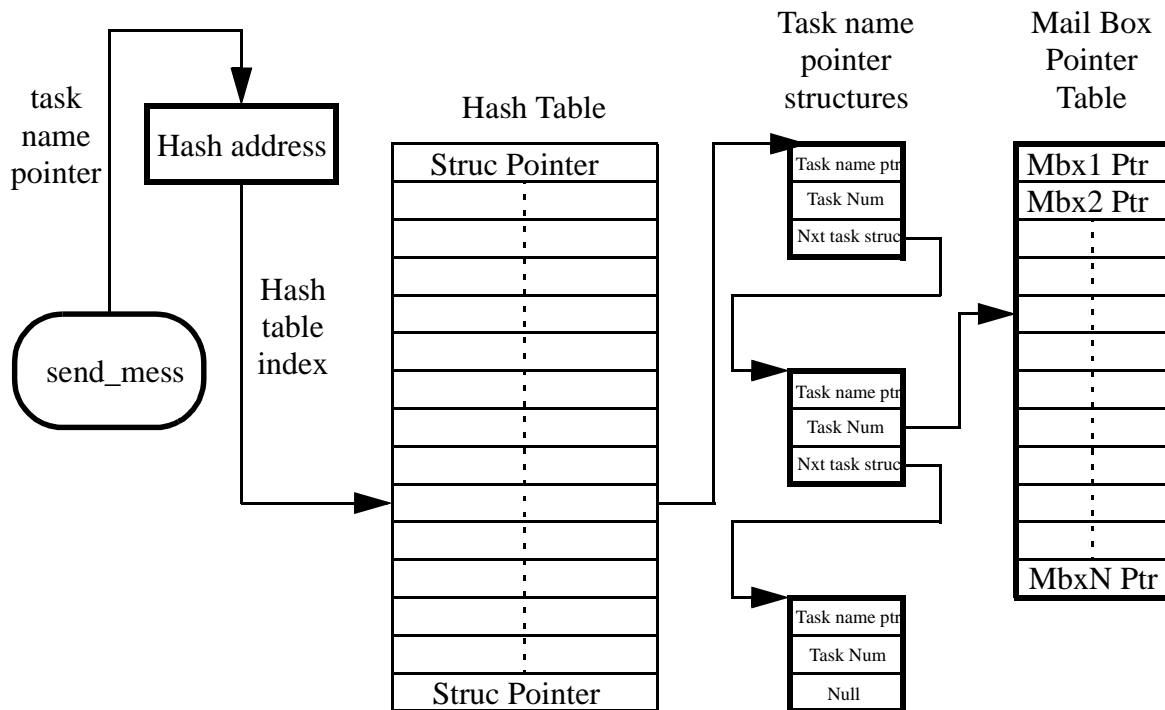


Figure 6-1: Mail Box Address translation

information to do with a specific message. One particularly interesting feature of the envelope is that it has a component called “return address”. This is used to store the name pointer of the task that is sending a message. This allows the receiving task to discern who sent a particular message. Remember, that one of the features of the mail box communication system is that messages can be sent indivisibly from any task in the system to any other. The receiving task could, for example, run a separate protocol decoder for each task that is communicating with it. It could logically simultaneously decode all these separate inputs by using the return address field to select the correct set of decoding tables.

The quick message envelope is a normally free envelope that is reserved for the sending of emergency messages between tasks. If a quick message has to be sent from one task to another, the **send_qik_mess** routine puts the message at the front of the message queue if possible. If the queue is full then it uses the special quick message envelope. The **rcv_mess** routine looks to see if the quick message envelope has something in it, and if so then the message contained in it is return to the calling task.

The mail box system in UNOS is used as an interface to most physical devices in the system. For example, the interface to the serial devices in the system is via mail boxes. If a task wishes to send a character out of a serial channel it sends the message via a mail box to the serial transmit task. The read data from a serial channel is a slightly different process. Because tasks can only read from there own mail box, a task which wishes to receive serial data has to make a connection to the serial receiver task. There are a number of ways that this could be carried out, but in UNOS this is achieved by sending a message to the serial receiver task. The message is not important, but the return address automatically appended by the mail box mechanism is used to make the connection to the task.

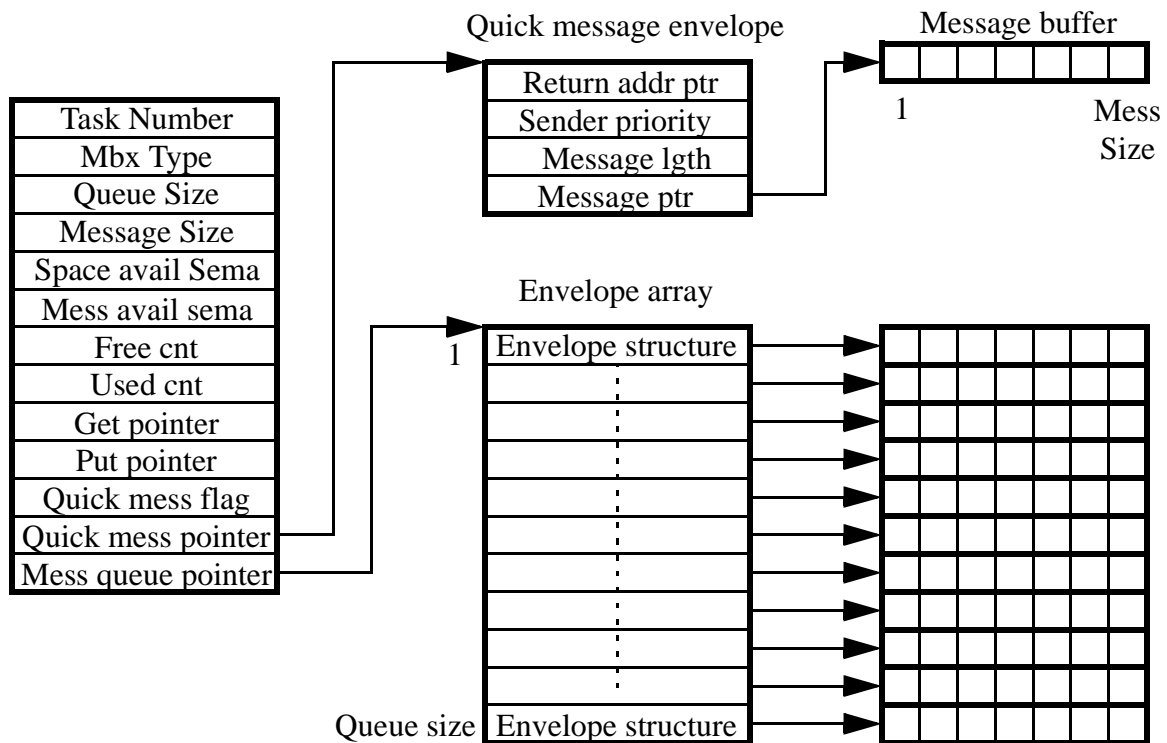


Figure 6-2: Mail box structure used in UNOS

The above procedure for connection to physical devices is very flexible. Although not currently implemented, it is possible to make multiple connection to the one device. In the case of the serial channel this would mean that an input character would be sent to several connected tasks. The messages sent by tasks to the serial receiver task could then request connection to the serial receiver, or request to be disconnected, at runtime.

One of the main advantages of using a mail box approach to interface to I/O devices is that it is easy, during debugging stages, to simulate the hardware using a task. When the software is installed on the real system, then only the mail box address has to be changed to the actual device driver mail box, for the existing software to work. The use of mail boxes provides a high degree of isolation between tasks, the only connection being an agreed serial communications protocol. This also aids software development when a large number programmers are involved.

Another advantage of the mail box approach to intertask communications is that it allows simple implementation of loosely coupled multiple processor systems where tasks are actually running on separate processors. UNOS 1.5a does not support this feature, as it also requires that the kernel calls also be implemented as mail box calls.

For block oriented I/O the combination of the pointer transfer strategy, outlined above, with the mail box mechanism is a very elegant approach. The more traditional device driver approach used in most general purpose operating systems is outlined in Chapter 8.

Timing in Real-time Systems

Real-time systems, by nature, involve tasks executing a precise times. Consequently it is important that the kernel provide facilities to allow the accurate time control of task execution. The fundamental time component in the kernel is the “tick” routine. This is usually an interrupt

routine that is invoked by pulses from a crystal driven hardware timer. The routine, in its most basic form, will increment a real-time clock count location upon each interrupt invocation. It is this clock count location that measures to passing of time for the kernel.

If time slice entries to the kernel are supported, the tick routine is the entry point. After a certain number of user definable ticks a call is made from the tick interrupt to the kernel proper. The kernel then calls the scheduler to see if the current task should continue to run. The details of time slicing are explained in “Time-slicing” on page 5-15, and will not be repeated here. This feature obviously offers some time control over the execution of tasks, but it is primarily focused at sharing the processor time, and not at getting precise timing of task execution.

The remainder of this section shall look at some general requirement of a task timing system, and then look in detail at the approach taken in the UNOS kernel.

Time Delays

In a real time operating system one of the most important features is to have some method of obtaining precision delays in task execution¹. For example, there are many applications where a task has to be put to sleep for a specified period of time, wake up and execute a small piece of code and then go back to sleep again. Obviously most sampling and control applications work in this fashion.

There are a number of ways of implementing the task sleep ability. I shall describe the standard technique and then have a look at the method used in the UNOS real time kernel.

A sleeping task is neither blocked or ready. Therefore many operating systems create a new queue for sleeping tasks. Unlike most of the kernel queues discussed so far the sleep queue is not a FIFO queue. Therefore for maximum efficiency a different queue structure should be used, however generally the same structure described previously is used because the sleep queue is usually short. This also has the secondary benefit that the kernel queue handling is kept uniform.

For sleeping tasks the **time_to_go** field mentioned in “Time-slicing” on page 5-15 is used to record how long to sleep. The most obvious way to tick the time away is for the clock routine to go all the way through the queue of sleeping tasks and decrement the **time_to_go** field. When a count reaches zero then move the task or tasks from the sleep queue to the appropriate priority queue. Note that this would mean that doubly linked tcbs would be required because tcbs would need to be, in general, removed from the middle of the queue. The main problem with this approach is that the clock routine execution is slowed down by having to decrement the counters for a number of tasks. If high accuracy timing is required, the clock tick routine may be executing at a high frequency, therefore considerable time would be wasted in this routine.

A more efficient implementation, which is commonly found in real time operating systems, is the delta time queue. The sleeping tasks are kept in time order on the sleep queue, which means that only one task, the task on the top of the queue, has to have its **time_to_go** counter decremented. All the other tasks on the queue have their **time_to_go** counters implemented as the difference between the sum of all the **time_to_go** counters for the previous tcbs on the queue and the absolute time that the task has to be made ready again. The idea behind this approach is that the very regular clock tick operation has become faster, but the infrequent

1. Note that precise delays are all that can be achieved. When a task goes to sleep it is effectively removed from the task ready queue. The timing mechanisms will accurately determine the time of this removal. However, once a task has been made ready again, after the time delay, the time of execution of the task is determined by its priority in relation to the other tasks in the system.

queuing of a task onto the time queue has become slower (since the queue has to remain in time order).

UNOS timers

The UNOS kernel uses a different approach to implement its time operations. It supports time operations at the kernel level by having a kernel data structure known as a “timer”, which has a number of associated procedures to allow them to be manipulated. The main advantage of this approach is the versatility it offers, whilst still being elegant and efficient to implement. The timing of task execution is only one of the facilities that it offers. The timer data structure has to following format:

```
unsigned char status;  
unsigned char type;  
unsigned long init_time_cnt;  
unsigned long delta_time_cnt;  
struct time_struc* prev_timer_ptr;  
struct time_struc* next_timer_ptr;  
void ( *timer_handler ) ( void* );  
void* timer_handler_data;
```

A brief description of the components are as follows:

- **status** – A byte which is used to indicate the current status of the timer. The states the timer can be in are: active (i.e. on the active timer queue or in the process of timing out); inactive (i.e. on the inactive timer queue or in the process of being moved from the inactive queue to the active queue). 0=>inactive, 1=>active (the queues are explained in the section “UNOS timer queues” on page 6-9).
- **type** – Single shot or repetitive. A single shot timer will execute the handling function when it times out and then be returned to the inactive timer queue. A repetitive timer is returned to the active timer queue before the timer handler is executed. This mode of the timer is ideal for repetitive and accurate time events. 0=>single shot, 1=>repetitive.
- **init_time_cnt** – Absolute value of the time before the timer times out in the number of clock ticks of the operating system tick routine.
- **delta_time_cnt** – Location used to keep the delta count value when a timer is active.
- **prev_timer_ptr** – This is a pointer to the previous timer structure on whatever queue the timer is on (active or inactive). If the timer is at the top of the queue then this pointer contains a NULL pointer.
- **next_timer_ptr** – This is a pointer to the previous timer structure on whatever queue the timer is on (active or inactive). If the timer is at the end of a queue then this pointer contains a NULL pointer.
- **timer_handler** – This is a pointer to the function which will carry out an action when the timer times out. Generally this pointer has to be set up by the user to point to the routine which he or she wishes to carry out a function when the timer times out.
- **timer_handler_data** – This is a pointer to an arbitrary data structure which can be used by the handling routine. This allows a generic handler to be used for a number of different time-out occurrences. Sometimes this location is simply used to store a data value which is not a pointer.

There are basically two types of timers within the system:

- single shot timers
- repetitive timers

Single shot timers are, as the name implies, timers which, when timed out, cease to function until they are explicitly initialised again. Repetitive timers, on the other hand, automatically reinitialize themselves when they have timed out. In this way a precisely timed repetitive action can be carried out.

The basic timing source for the timers is the timer interrupt. Since this frequency is decoupled from the operating system time slice frequency, one can for most practical purposes obtain any required resolution for the timers.

UNOS timer creation

In order to use a timer it must firstly be created. This is done through the routine **create_timer**. This routine can be called either at initialisation time (which is probably the best place to create the timer) or in the routine which is going to use the timer. In either case once a timer has been created it can not be destroyed. The reason for creating the timer at initialisation time is that a timer creation overhead is not incurred in real time when the system is running.

The **create_timer** routine simply allocates memory for the timer. This memory allocation is done dynamically on the heap. The routine then places the timer onto the inactive timer queue (discussed below).

UNOS timer queues

Once a timer structure has been allocated then the next task is to place the timer on a queue. The system has two queues for the management of timers:

- an active timer queue
- an inactive timer queue

The active timer queue is a queue of timers which are in the process of timing out. This queue is a delta time queue as described in the general discussion earlier. Hence, the absolute time value until a timer times out is simply the addition of the time value in the timer of interest, plus all the preceding timer values. Placing a timer in the active time queue is complex, since the timer has to be placed at the correct point in the queue, the timer's delta value calculated, and the delta of the following timer has to be recalculated.

The inactive timer queue is used to store timers which are not currently being used. Therefore it provides a pool of timers which can be used by any task. As mentioned earlier, a timer destruction routine is currently not implemented in UNOS. The reason for this is that such a facility is not particularly useful in the context of a real-time system; once a timer has been created for some purpose in most applications it will continue to be required.

Both the active and inactive timer queues are implemented as doubly linked lists. The timers on the inactive queue are only ever removed from the top of the queue, and added to the end of the queue. Therefore, from this queue's viewpoint, a singly linked list would be sufficient. However, the active timer queue requires that timers be added and removed from the middle of the queue.

UNOS timer maintenance routines

There are several routines which are used by the operating system to carry out the maintenance of the timers. These routines are **create_timer** (as mentioned previously),

add_timer_active_q and **remove_timer_active_q** and **remove_timer_top_active_q**. Much of the functionality can be gleaned from the names of these routines and their detailed behaviour will not be explained here. For more information refer to the Appendix C.

UNOS user interface routines

There are four main routines which form the user interface to the timers in the system. They are:

```
timer_struct* create_timer ( void )
```

This function can be called by the user tasks to allocate storage for a new timer. As mentioned previously the preferred place to create timers is in the initialisation code of the system. The routine returns a pointer to the timer structure. If it could not allocate storage for the timer then a NULL pointer is returned. The timer structure is placed on the list of free timers (i.e. the queue of inactive timers). It currently has no values assigned to the time period, timer type, handling routine or handling data structure. The status and link pointers are initialised when the new timer is placed in the inactive queue.

```
timer_struct* reset_timer ( timer_struct* timer_ptr )
```

This function resets the time in a timer to its initial value. The timer which is being reset must already be on the active timer queue. The parameter passed into the routine is a pointer to the timer structure which is to be reset. The value returned from the routine is an **int** which is interpreted as a boolean. If the operation is successful then the value returned is true.

```
timer_struct* start_timer ( unsigned char timer_type, unsigned long  
init_time_count, void ( * timeout_handler ) (), void* data_ptr )
```

This function takes a timer from the inactive timer queue, initialises the initial count value and timer type, updates the timer status, initialises the handling routine and data pointers and then places it into the correct place in the active timer queue. The parameters which are initialised are passed as parameters to the function in the following order – timer type, initial count, pointer to handling routine and pointer to handling data structure. If the starting of the timer is successful then the function returns a pointer to the timer started, else it returns a NULL pointer.

```
timer_struct* stop_timer ( timer_struct* timer_ptr )
```

This function takes a timer from the active timer queue and places it back onto the inactive timer queue. The parameter passed to the routine is a pointer to the timer in question. If the operation is successful then the routine returns an **int** which is interpreted as a boolean. If the operation has been successful then the value returned is true, else it is false. The parameter passed into the routine is:

The **timeout_handler** function, that is executed by the timer when time-out occurs, can be almost any routine. This function is not associated with any task in the system. In effect the

function is part of the kernel. A more typical use of the **timeout_handler** function is to execute a small function which calls the kernel signal routine. This could then make some periodic task become runnable (i.e. it is moved to the appropriate priority queue).²

The timers in the system can also be used transparently by the kernel itself (see below for more detail). In the case of the timed semaphore wait the **timeout_handler** is a function within the kernel itself.

UNOS timed semaphore wait

The timed semaphore wait is the other main use of the timers in the system. This is an example of a transparent operating system use of a timer. When a **timed_wait** is requested by a task then a timer is retrieved from the inactive timer queue and initialised with the required data for the wait. The requesting task itself is blocked on a semaphore. In this particular case, the timer handler routine is a kernel routine which will remove the requesting task from semaphore queue when the timer times out. If a signal is made to the semaphore before the timer has timed out, then from the users point of view, the semaphore behaves as a normal semaphore. Transparently, the timer associated with the semaphore is removed from the active timer queue before it times out, and placed back into the inactive timer queue.

The obvious use of the **timed_wait** is to prevent deadlocking if a **signal** is not made to a semaphore within a certain period. However, it can also be used intentionally to block tasks for certain periods of time. If a semaphore is created with an initial value of zero, then any **timed_wait** call using that semaphore will result in the blockage of the calling task on the semaphore. The task will be placed onto the blocked queue for the semaphore and will remain there until the time period has expired (assuming that no other task will **signal** the semaphore). In this way the timed_wait operation provides an elegant “sleep” facility for tasks.

The **timed_wait** operation is also used in the mail box **rcv_mess** routine. It is possible to call this function with a time limit. If a message is not received within the requested time then a return from **rcv_mess** occurs with an appropriate code indicating that there has been a time out on the receive message semaphore.

Applications of UNOS timers

One particular application of the UNOS timers has already been mentioned – the timed semaphore wait. The most common explicit user use of a timer would be repetitive activation of a task. In this particular case the timer is created as a repetitive timer. The handling routine is a procedure which carries out a signal on a semaphore which the repeating task is blocked on. The advantage in this approach as opposed to using **timed_wait** in the tasks is that the repeat rate is very accurate because upon time out the timer is immediately placed back on the time queue (before the next clock tick). Using a **timed_wait** to achieve the same function causes unpredictable errors in the timing of the task because there can be an interrupt between the exit from one **timed_wait** invocation and the next.

Another possible use of the timers is execute particular procedures at specific instances in time. This is achieved by making the handling routine a procedure which carries out some task. When this is done the procedure has properties similar to an interrupt routine, however there is no need to save the registers since this is already done when the tick routine is entered. If one, for example, wanted to flash a light on and off to indicate tick routine activity this could be achieved by writing a procedure to carry out the flash operation. This procedure is then called

2. A call to the signal routine from a timer is treated as a special case by the kernel, and reentrancy problems into the kernel are avoided.

at a frequency determined by a timer.

The other major use of the timers is for time-outs. Using the timers in UNOS makes this very easy and elegant to do. The user explicitly starts a timer with a handling procedure which will carry out some action if the time period expires. If the activity which is being timed out finishes within the time-out period then the users code explicitly stops the timer. For example, a software watchdog timer routine could be implemented using such a feature. In this situation one would make use of the retriggering facility offered by the timers. The `reset_timer` routine would have to be called within the time-out period to prevent the timer handler from being activated. If the timer handler was activated then this would indicate that some part of the software system was not executing with correct timing.

UNOS Heap Management

Although not part of the communications or timing system, this may be an opportune point to say something about the UNOS heap management facilities.

It is possible to construct real-time system with totally static memory allocation, however it is more flexible and elegant to allow dynamic memory allocation. In the case of UNOS most operating system structures are effectively statically allocated at initialisation time, even though the dynamic allocation routines are used to carry out the allocation. The main advantage of using the dynamic allocation routines is that the calls to the functions which create the systems data structures only require parameters such as the number of tasks in the system, to create the data structures required. The operating system routines themselves do not have to be modified. If static allocation was used then changes would have to be made to the operating system routines to create at compile time the data structures required.

Dynamic memory management in UNOS is based on the heap management routines developed in the book “The C Programming Language” by Kernighan and Ritchie. These routines assume that a memory space has been made available by an operating system (such as UNIX, for example), and they then manage the allocation of chunks of memory within this space. In the UNOS case, it is not running above an underlying operating system, therefore the initial chunk of memory can be allocated by the programmers knowledge of the memory map of the machine that the software is operating on, or by a call to some other routine. For example, if UNOS is running on an IBM-PC or compatible, it can call the ‘C’ library memory allocation routine before UNOS starts executing (this is possible because UNOS is simply an ordinary ‘C’ program at this stage). This routine requests memory from MS-DOS, the PC “operating system”. The pointer returned is then used as the starting point for the UNOS heap.

Once the heap memory space has been obtained, by whatever means, memory can be allocated to requesting tasks from it. The memory is allocated in units that are the size of the header used to store the information about the size of the memory chunk allocated (this header is part of a piece of allocated memory, but is transparent to the user of the memory). The idea of allocating the memory in header size units is to make the heap management system more portable. By using some tricks with the ‘C’ language it is possible to ensure that certain memory alignments can be maintained in the allocated memory. This can be important for memory efficiency reasons on some processors. For example, in the Intel 80960 processor the stacks have to lie specific boundaries for the hardware to use it properly. The stacks for the tasks are allocated from the heap, therefore it is essential that the required memory alignment be maintained for the processor to work.

When memory is freed, the heap management routines try to concatenate adjacent memory chunks together to form larger pieces of memory. This is designed to prevent heap fragmentation.

In the first section of these notes a brief discussion of interrupts and their importance in operating systems was given. In this section we shall explore the role of interrupts in operating systems in detail.

Basic interrupt mechanism

An interrupt is a means by which a physical I/O device can let the processor know that it wants attention. The requests for this attention can arise from a number of sources—user generated requests (e.g., a disk access interrupt, terminal input), external requests (e.g., some external device wants some action by the processor), hardware errors etc.

The programmer has control over whether the processor will take any notice of the interrupt requests. This is usually done by means of a interrupt enable flag. When interrupts are enabled most processors will only allow interrupt checking between instructions. It should be noted that some processors will allow interrupt checking in the middle of some instructions, usually ones which can take a long time to complete execution. An example of this is the string move instruction in the Intel 8086 line of processors.

In many respects an interrupt can be considered to be a hardware generated procedure call. The question arises as to how the processor knows what procedure to call upon an interrupt, since in general there are a number of interrupting devices connected to the processor. In early processors all interrupts caused the processor to begin execution at a fixed memory location. The interrupt procedure invoked would have to work out what caused the interrupt by polling all the possible devices which could have caused the interrupt (example 6502). A later variant of this approach was that the processor had several interrupt lines each one causing an interrupt procedure to be invoked at different fixed memory locations. This allowed several devices to have their own interrupt procedures (an example of this approach is the Intel 8085).

Most modern processors allow each interrupt to have its own interrupt procedure by using an *interrupt vector table*. In some processors this table is in a fixed location in memory, in others it can be relocated by the use of a interrupt vector table base pointer register which is user programmable. To see how an interrupt table is used consider the following typical interrupt sequence.

- (i) The device makes an interrupt request by putting a signal on the “interrupt request” line of the control bus.
- (ii) The processor completes the execution of the current instruction, and then it samples the interrupt request line. It finds that an interrupt request is present and if the interrupts are enabled then it will move to the next step else it will simply ignore the interrupt.
- (iii) The processor sends an interrupt acknowledge. At this time most processors will automatically disable interrupts.
- (iv) The interrupting device detects the acknowledge, and responds by putting its “interrupt number” onto the data bus. The interrupt number of a device is defined at the hardware

design time or in some systems can be assigned by software at system initialization time. Each device “interrupt number” is unique.

- (v) The processor reads the interrupt number from the data bus. At the same time, or perhaps earlier, the processor saves two words on the stack: the processor status word, and the current value of the program counter.
- (vi) The processor uses the interrupt number to select an entry in the interrupt vector table in main memory. The value stored in the interrupt vector table is loaded into the program counter so that execution will occur from this point. In some processors a new value of the processor status word is loaded from the interrupt table.

All of the above steps are performed by hardware. At the conclusion of step (vi) the processor will resume executing instructions at the current value of the program counter, which happens to be set up at the start of the interrupt routine.

Finer details of the interrupt mechanism

The above section itemized the basic operation of the hardware upon an interrupt. There are some fine details however, which are important in order to understand the overall operation of interrupts. Some of these are listed below.

- (i) Upon completion of the interrupt routine, it automatically restores the processor status register and the program counter. The latter is what returns execution to the point in the code where the interrupt was acknowledged. In most processors the remainder of the registers are not saved upon an interrupt. Therefore it is the responsibility of the programmer to make sure that the registers used in the interrupt routine are saved on the stack and then restored before the return from interrupt.
- (ii) As noted in the previous section most processors disable interrupts via hardware upon the receipt of an interrupt. Generally the status of the interrupt enable is kept in the processor word. Consequently when the interrupts are disabled the interrupt enable bit is cleared. However this is done after the processor status word has been saved on the stack. Therefore when the processor status word is restored at the end of the interrupt routine the interrupt enable is reinstated.
- (iii) Some processors do not save the interrupt enable bit in the flags. Therefore an explicit enable must be put as the last instruction of the interrupt routine. The effect of the enable instruction is delayed until after the execution of the next instruction (the return from interrupt) thereby preventing the possibility of recursion back into the interrupt routine.
- (iv) On processors with user and supervisor modes an interrupt routine may have the side effect that the mode of the processor changes from user to supervisor mode. This may also have the further side effect that a new stack is switched to (as opposed to using the current task stack). This process is reversed upon the return from interrupt.
- (v) When interrupts are disabled the interrupt request is not lost. The physical device requesting an interrupt will keep the interrupt request line asserted until the interrupt acknowledge is received. If a further request occurs after the interrupt acknowledge then this interrupt will remain pending until the current invocation of the interrupt routine is complete. Upon exiting the interrupt routine the interrupts will be reenabled and the request will then be attended to (all this assumes that interrupts are disabled whilst an interrupt routine is being executed).

Hardware structure of interrupts

Some simple ways of connecting interrupting devices to the processor have already been described. However, usually in an operating system environment more sophisticated hardware interrupt management is used. For example many processors use an “interrupt controller” between the CPU and the interrupting devices.

An interrupt controller at the expense of more hardware complexity (and therefore cost) offers much more control over interrupt requests. Below are listed some of the typical features which are offered by interrupt controllers.

- (i) Priority levels—each interrupt line is given a priority. When an interrupt occurs at any priority equal to or lower than the priority of the current interrupt then that interrupt is held pending (i.e., it does not activate the interrupt request line to the processor) until the current interrupt has finished (this must be signalled by an explicit output to the interrupt controller by the interrupt routine). If an interrupt of a higher priority occurs then an interrupt request will be output to processor. If the interrupts have been reenabled by the interrupt routine this enables a high priority interrupt to interrupt a lower priority interrupt routine.
- (ii) Allows a single point connection for the interrupt acknowledge. In the description of the basic interrupt mechanism no mention was made about how the interrupt acknowledge was connected to the interrupting device. If multiple devices generate an interrupt at the same time then one could not have them all connected in a “wired OR” configuration, or else all the interrupting devices would attempt to place their interrupt vectors onto the data bus simultaneously. If an interrupt controller is present this problem is solved by having the interrupt acknowledge going only to the interrupt controller. It is the controller which supplies the interrupt vector upon the receipt of the acknowledge. The actual hardware device receives an effective interrupt acknowledge when the source of the interrupt is removed by the interrupt service routine.
- (iii) Some interrupt controllers allow a different selection of scheduling algorithms (which is implemented in the controllers hardware). For example the Intel 8259 interrupt controller offers several different interrupt priority structures: - *nested priority* – the interrupt requests are ordered from 0 through 7 (0 highest). When an interrupt is acknowledged the highest priority request is determined and its vector is placed on the bus. All interrupts of the same or lower priority are inhibited until the microprocessor issues the end of interrupt to the 8259; *automatic rotation* – this is useful when there are a number of interrupting devices of equal priority. In this mode a device, after being serviced, receives the lowest priority. Therefore in the worst case a device requesting service will have to wait for 7 other devices to be serviced once; *specific rotation* – this allows programmable control of the lowest priority. For example, in the case of the 8259 interrupt request 5 (IR5) can be programmed as the lowest priority, which then causes IR6 to be the highest priority. This reprogramming may or may not be achieved as part of the end of interrupt command.
- (iv) Interrupt controllers give the programmer control over whether an interrupt will be enabled from a specific device. This is usually done by means of an interrupt mask register in the controller. The Intel 8259 has a mode known as *special mask mode*. This allows an interrupt routine to control which interrupt priorities are enabled under software control. This can be done before the issue of the end of interrupt command (i.e. it can bypass the normal nested interrupt mode). The mode allows an interrupt routine to dynamically alter which interrupt levels are enabled at various points in execution.

- (v) Many interrupt controllers allow the programmer to select whether an interrupt should be level sensitive or edge triggered. This can be important in situations where a source of interrupt will only be present for a short period of time—in this case the interrupt should be edge sensitive to ensure that the interrupt is latched. On the other hand if a number of devices are sharing an interrupt line or if there is a possibility that another interrupt could occur before a current interrupt has been acknowledged then level sensitive interrupts are necessary.
- (vi) Many interrupt controllers allow cascading of interrupt controllers to allow large numbers of interrupts into a system to be handled in an orderly fashion. The Intel 8259 for example allows the cascading of eight 8259's giving up to 63 interrupt lines.

The first diagram below (Figure 7-1) shows an interrupt system structure using an interrupt controller. The second diagram shows the interrupt structure required for multiple vectored interrupt devices without an interrupt controller. Note that in this case the devices themselves have to be capable of supplying the interrupt vectors. Note the daisy chain for the interrupt acknowledge and the “wired OR” for the interrupt line.

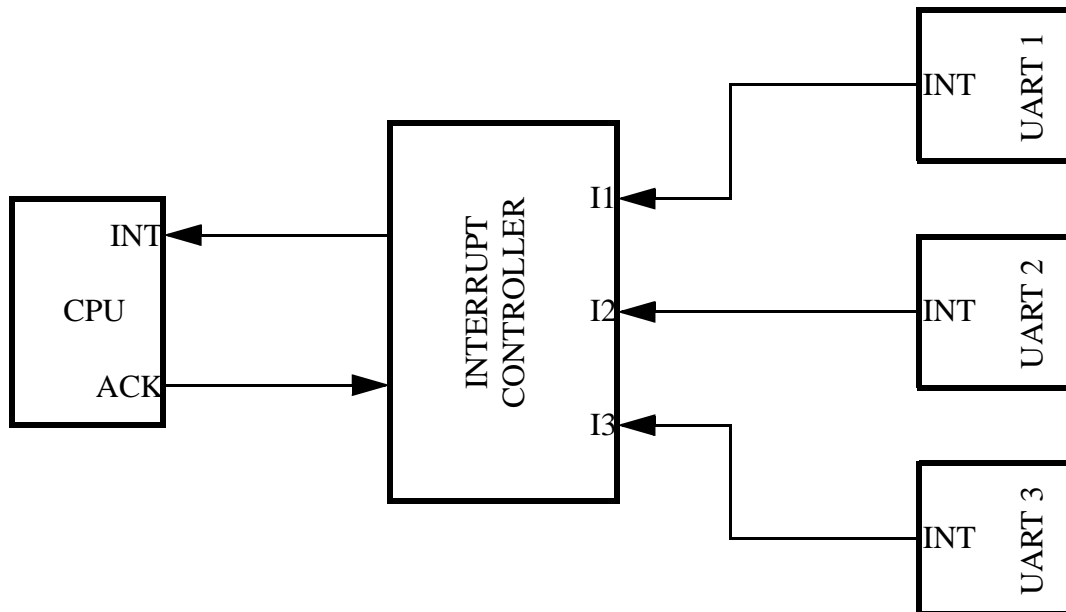


Figure 7-1: Interrupt controller based system structure

Other types of interrupts

Thus far we have only considered “normal” interrupts. However most processors have at least two other types of interrupts—nonmaskable interrupts and internal interrupts.

A nonmaskable interrupt as the name implies is an interrupt which cannot normally be masked or disabled. In all other respects it behaves as a normal interrupt. Due to the fact that they cannot be disabled, a nonmaskable interrupt has very limited use. It is clear that in most applications disabling of interrupts is required for critical section protection. Therefore an interrupt

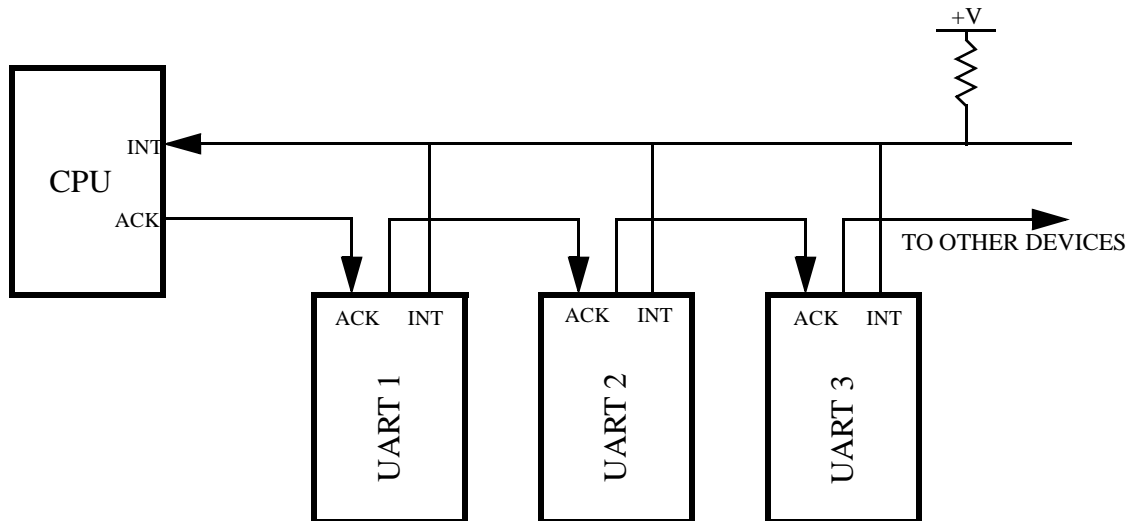


Figure 7-2: Daisy chained interrupt structure

service routine which is entered via a nonmaskable interrupt must satisfy the following conditions:

- (i) It must not refer to variables which are shared with any other code in the system since there is no way shared code can be protected from corruption.
- (ii) It must not take any action which could result in a task switch since the interrupted thread of execution could have been in the kernel. The result would be reentrancy into the kernel which would result in a corruption problem.

Due to the above limitations, only special devices should be connected to the nonmaskable interrupt. Usually it is reserved for a signal of “catastrophic failure” such as a power fail. Under these conditions it is acceptable that any current operation be interrupted so that the system can do its best to have an orderly shutdown.

The other main type of interrupt is the internal interrupt. These interrupts are instigated by the action of software as opposed to a piece of hardware demanding attention. There are three main cases of these interrupts:

- (i) Software instruction interrupt. This is a explicit instruction which causes the processor to carry out most of the actions of a hardware interrupt. Obviously it makes no sense for a software interrupt to place an interrupt vector onto the bus. Instead the vector is generally provided as an operand of the software interrupt instruction.

Software interrupts provide a commonly-used method by which user programs can call system routines. Parameters are often pushed onto the stack before the interrupt instruction. These parameters indicate what service is required. One advantage of the interrupt call technique is that it hides all the operating system routines behind a protocol—the user does not need to know procedure names to call. An additional advantage in processors with user and supervisor modes is that it provides a mechanism to change from user mode to supervisor mode. Upon return from the interrupt the user mode is reinstated.

- (ii) Single step interrupts. These interrupts are present in some processors to aid in debugging. An interrupt occurs after each instruction which causes control to go to a debug monitor where the user regains control and can then examine variables.
- (iii) Software trap interrupts. These interrupts occur when the processor detects some error

in the execution of the current instruction. An example of this which most people are familiar with is a floating point trap due to say a divide by zero. The interrupt handling routine generally prints a message to the user and then returns control back to the command interpreter. Other examples of these occur in protected mode operating systems—illegal address error. Software interrupts can also occur in this environment which are not error conditions, for example a page fault.

It is important to note that whilst on many processors the content of the stack upon a software interrupt is the same as a normal hardware interrupt, there are exceptions. In fact in some processors the content of the stack may differ depending on the type of software interrupt—for example, error related traps may push an error number on the stack.

Interrupt priorities

The concept of interrupt priorities was introduced in the previous discussion of the hardware of the interrupt controller. As indicated in that discussion the main consequence of interrupt priorities is that low priority interrupts may be interrupted by high priority interrupts.

In most processors the priority of an interrupt is determined by the physical wire that connects the interrupting device to the processor or interrupt controller. In processors such as the Intel 8086 series the priority of an interrupt is essentially encapsulated in the interrupt controller which is usually a separate component to the processor. The processor itself does not contain any priority component in its flag register. Other processors, such as the PDP-11 and the Intel 80960, have a component in their flag register which indicates the processor's current interrupt priority. If an interrupt occurs which is lower than or equal to this priority then that interrupt will be ignored until the processor priority is lowered to a level equal to or lower than that of the interrupt. In this system, the processor priority becomes that contained in the interrupt vector of the current interrupt. The processor priority of an interrupted code stream is restored upon a return from interrupt. Because the processor priority is so important, instructions which allow it to be changed are usually only executable in supervisor mode. Often these processors do not have an explicit disable instruction, but instead carry out the same function by raising the processor priority to its highest level.

One pitfall that the inexperienced fall into in a priority based interrupt scheme is that they think that interrupt routines can be made longer. This can lead to problems in systems where there are large numbers of interrupts at varying priority levels. The moral of this is that interrupt routines should always be kept short regardless of whether priority levels exist or not.

A digression for a moment to illustrate a common error which many hardware vendors have designed into their computer systems in relation to priority interrupts. Generally speaking it is dangerous to allow a task switch while an interrupt is executing, except if the task switch occurs after the interrupt has finished its work. In most systems, a clock interrupt can cause a task switch, therefore all interrupts should run at a priority of at least the clock interrupt. However in some systems hardware designers have made the clock interrupt a high priority interrupt. The only easy solution to the above mentioned task switch problem is to disable all interrupts during an interrupt routine. One therefore loses the ability to have nested interrupts. (A example of such a system is the PDP-11).

Interrupt tasks

Interrupt procedures as described so far do not behave any differently from interrupt procedures in a non-operating system environment. In the operating system environment they operate in the context of the task that has been interrupted, since they use the stack of this task. This implies that the interrupt procedure inherits some of the privileges of the interrupted task. For

example if the computer system uses hardware memory management then the interrupt procedure has access to the same memory segments as the interrupted task.

In a multi-tasking system it is likely that the interrupt procedure bears no relation to the task which has been interrupted. Therefore the fact that the interrupt procedure shares the context of the interrupted task can cause problems in the security of the operating system. For example a poorly written interrupt routine could cause a totally unrelated task to crash. Therefore in principle it would be better if each of the interrupt routines were effectively isolated from all the other tasks and interrupt routines in the system.

In the Intel 80286, 80386 and 80486 processors provision has been made for interrupt tasks. This is supported through a generalization of the interrupt vector table described earlier. This table which is known as the *interrupt descriptor table* contains entries which determine how the interrupt is to be handled. One handling mode is to call an interrupt procedure as described previously. Alternately an interrupt task can be initiated. This is achieved by entering a *task gate* which is effectively a call to a task state segment. The task state segment essentially contains all the information about a task (very much like the task control block as described previously). The task state segment mechanism allows the task switch to occur via hardware (i.e. the task switch is implemented as a single instruction). This then a new stack to be used, the registers are saved in the current task state segment and the registers for the interrupt task are restored from the new task state segment. When the return from interrupt is executed the instruction can determine whether the return should be for an interrupt procedure or an interrupt task and carries out the appropriate action. It should be noted that an interrupt task is like any other task in that it should be implemented as a loop. Execution begins at the point it left off upon the next interrupt. Therefore the return instruction is usually at the bottom of the loop within the task.

The advantage of the interrupt task is the security that it gives. However there are some disadvantages:

- (i) Interrupt tasks cannot call any operating system routines. For example it is common for the **signal** routine to be called from a normal interrupt routine. This is the usual technique to get the kernel dispatcher to start a handling task. However in the case of an interrupt task this does not make sense because an interrupt task usually does not have a task control block. Therefore if the **signal** results in a task switch the system would crash. One technique of overcoming this problem is to have a special signal whose effect is delayed until after the end of the interrupt task execution.
- (ii) Programmers can be lulled into a false sense of security in that they consider the interrupt task to be like any other and therefore make the task execution too long (it is probably a little harsh calling this a problem with interrupt tasks).

Overall the advantages of interrupt tasks outweigh the disadvantages. Therefore if the hardware supports interrupt tasks they should be used for all the interrupt handlers.

Interrupt handlers and the dispatcher

Although an interrupt handler is logically a task (in the sense that it runs asynchronously with other tasks in the system) it is usually not known to the dispatcher. In the case of a “normal” interrupt procedure it runs in the context of the interrupted task and therefore becomes effectively a part of this task during its execution as far as the dispatcher is concerned. However the dispatcher has no way of determining or controlling on a task priority basis the execution of a interrupt procedure. Similarly an interrupt task is completely unknown to the kernel.

The “mysterious” or “phantom” operation of interrupts has some implications in relation to

critical section protection. Consider the most common example of the use of interrupts, namely device drivers. These usually consist of two main sections—a user callable routine and an interrupt routine. The user task calls the procedure interface to retrieve or place some data in a buffer and the interrupt routine does the same in response to the interrupts from a hardware device. Obviously there is shared data between these two components. Which will require critical section protection. Semaphores cannot be used for this protection, the reasons varying depending on the interrupt implementation.

If the interrupts are handled by “normal” interrupt routines and if a **wait** was used for the critical section protection the interrupt routine could possibly become blocked. In this case a task switch would occur and an indeterminate time could elapse before the interrupt routine could continue (the task which would **signal** to unblock the task need not be the next task to run. Indeed the task interrupted may be the one which would carry out the **signal**, therefore deadlock would occur). Furthermore, because a task switch has occurred interrupts would be reenabled thereby allowing another interrupt to occur from the same device (especially probable in the light of the extended time before the interrupt would finish). This is a problem because interrupt routines are generally nonreentrant in nature. The final problem is that if a **wait** is executed and a blockage occurs then the interrupted task would be blocked, even though it most likely has nothing to do with the blockage condition. This is most unfair.

If interrupts are handled by interrupt tasks and semaphores are used for critical section protection then a **wait** would again cause a possible task switch. As mentioned in the previous section many implementations of interrupt tasks do not include having a task control block (tcb). Hence a task switch is not possible because the dispatcher does not know of the existence of the interrupt task. One could envisage implementations where a tcb does exist for the interrupt task. In this case the same problems would arise as for the interrupt procedure case.

In both the above if any interrupt occurs which could cause a task switch then there would be the same problem as explicitly calling the **wait** procedure. Therefore on a single processor system the golden rules for interrupt procedures or tasks are:

- (i) The interrupt handler must disable all interrupts which can allow re-entrant execution of the interrupt handler or allow a task switch. If either of these occur then the integrity of the critical section will be breached. The easiest way to ensure this is simply to disable all interrupts.
- (ii) The user interface part of the device driver can protect its critical section by disabling the interrupt which would cause the associated interrupt handler to be executed. Whilst this seems all right theoretically problems can also arise from this approach, namely if a task switch occurs after a devices interrupts have been disabled. This could result in the interrupt for the particular device being disabled for a long period. Therefore, in my view it is better to disable all interrupts during the critical section, thereby preventing any interrupts as well as a task switch. This ensures that interrupts are enabled for most of the time for all devices.

It should be noted that the restriction of (i) above does not remove the ability to use interrupt priorities if the interrupts are organised correctly. The interrupts which can cause a task switch should be made the lowest priority interrupts in the system and the others which will not result in a task switch higher priority. In this way a task switch will not affect the performance of the system since the fast response interrupts cannot be disrupted and the low priority interrupts occur at a low frequency. Unfortunately this is often thwarted by hardware designers who make the clock interrupt the highest priority in the system (this being the most likely interrupt to cause a task switch).

As mentioned in the previous section there are cases with normal interrupt procedures that one can call semaphore routines, namely the **signal** routine. This is done as a means of informing the operating system that the interrupt routine has run and some action is now required by a normal task. It is not being done as part of the procedures critical section protection. If a **signal** is called in the wrong place then all the problems cited above will occur. However if the **signal** is called at the end of the interrupt procedure then it does not matter if a task switch occurs since the critical data has already been modified and the time critical part of the interrupt procedures execution has been completed.

One is tempted to think that if a task switch occurs as a result of a call to **signal** then the interrupts will be left disabled. This is not the case because the interrupt status is part of the context that is saved for each task and is therefore restored upon a task switch. Hence if a task switch occurs as a consequence of a **signal** which has been called from within an interrupt routine (with interrupts disabled), then the interrupt status after the task switch will be whatever the interrupt status was the last time that task ran.

Relaxing the “no task switch” condition

The “no task switch” condition of the last section is rather stringent. It rules out semaphore operations in an interrupt handler (excepting a **signal** at the end of an interrupt procedure). It also makes it very difficult to organise priority interrupts on a sensible basis. Therefore it is desirable to find some way of relaxing this condition.

The three basic reasons why a task switch is undesirable can be summarised as follows:

- (i) A task switch might be unfair to the interrupted task. This is only a problem if the interrupt routine becomes blocked. If another task is scheduled it has to be higher priority than the task interrupted. However even under this condition the “priority” of the interrupt itself may be usurped.
- (ii) If device interrupts are enabled at the time of the task switch it may allow the interrupt handler to be reentered before it has finished processing the first interrupt. This is dangerous because interrupt procedure are rarely reentrant. Interrupt tasks will generate a fatal error if re-entry is attempted since the same task is being reentered at the current point of execution. This will again result in corruption problems.
- (iii) If device interrupts are disabled at the time of a task switch then subsequent interrupts may be lost. This can occur because particular device interrupts are not kept as part of the tasks context. Therefore the device interrupts will not be enabled by a task switch. When the original interrupt completes its execution then the interrupts for the particular device will be reenabled.

All of the above problems can be put down to the fact that the dispatcher does not know of the existence of the interrupt routine. Therefore it cannot distinguish between the interrupted routine and the interrupt routine. Problems (ii) and (iii) can be solved by a modification which would allow the dispatcher to recognise that an interrupt routine is executing. The sequence which would be followed would be roughly as follows:

```
procedure task_switch ( T : task_ptr );
  (* Performs a task switch from the current task      *)
  (* to task T. It is assumed that the descriptor      *)
  (* for the current task has already been put on      *)
  (* the appropriate kernel queue. If an               *)
  (* interrupt procedure is running, it is             *)
  (* allowed to continue to run in the context of     *)
  (* the new task T. The return from interrupt        *)
```

```
(* complete the task switch to T.                                *)
begin
  if an interrupt procedure is running then
    push the program counter and processor status
      of task T onto the stack for task T, in
      a format expected by a "return from
      interrupt" instruction;
    identify the portion of the current stack which
      belongs to the interrupt routine and move
      it to the stack of task T;
    (Note that this would consist of such things as the
      interrupt routine stack frame and the address pushed
      onto the stack when the kernel was entered. It is this
      address which will ensure that the interrupt routine
      which is reentered upon exit from the kernel).
    copy the return address and processor status
      from the current task's stack to the saved
      task state of the current task.;
    (Note that this step is saving the return address and
      processor status in the format required for a non-interrupt
      task switch. This will allow the interrupted task to be
      scheduled normally when the dispatcher chooses to run
      it again).
    swap to the stack for task T
  else
    perform a normal task switch as documented in
      the previous chapter on task management.
end; (*task_switch *)
```

As can be seen from the above description this mechanism would require some very careful manipulation of the registers to ensure that they appear in the right order on the stacks. The net result of the above sequence is that the exit from the kernel after the dispatcher operation is to the interrupt which was running when the kernel was entered. The interrupt routine therefore will complete its execution. Upon completion the “return from interrupt” will now complete the task switch operation to the new task. Since the technique allows the interrupt context to complete before the task switch operation has been completed then there is no possibility of reentrancy and the interrupt routine will run to completion and reenables device interrupts within a short amount of time.

The above approach is probably not justifiable for interrupt procedures. However it is for interrupt tasks because it will allow a **signal** to be called from within the interrupt task. The task will then complete before the task switch is actually carried out thereby avoiding the task re-entry problem.

UNOS-V2 takes a different approach to this problem. It allows interrupts to be enabled throughout most of the kernel code. Therefore interrupts, which may carry out signal operations, can occur whilst the kernel is being executed. This is handled by forming a queue of operations for the kernel to execute. All requests for a kernel service will result in a request being placed in the kernel request queue. The kernel has a loop in it which takes requests from the request queue and executes them. Therefore when an interrupt occurs which makes a kernel request, this request is merely added to the request queue – it is not immediately acted upon. Upon return from the interrupt the kernel continues with the processing that it was doing prior to the interrupt. After finishing this it then retrieves the next kernel service request from the request queue and executes the appropriate handler. This continues until the kernel queue is cleared. Then the kernel exits.

What is a device driver?

A device in an operating system context is a piece of hardware with which the operating system has to interact in order to carry out some external action. For example input terminals, disk drives, tape drives, printers and so on are typical devices.

A device driver is the software entity which interacts directly with a device. Because of this direct interaction, it is a piece of device dependent code. It is called by higher level device independent code to allow user tasks to communicate with the device. The device driver handles all the low level manipulation of the device registers and interrupts transparently to the higher level routines. A good device driver will supply a device independent interface to the higher level routines.

Structure of a device driver

Device drivers generally consist of two main components – a user interface component and a device interface component. The user interface component as the name implies provides a means for the user tasks to communicate with the device. The device interface component usually consists of the interrupt routine which is reacting to the hardware interrupts generated by the device. The diagram below captures the logical organization of these components.

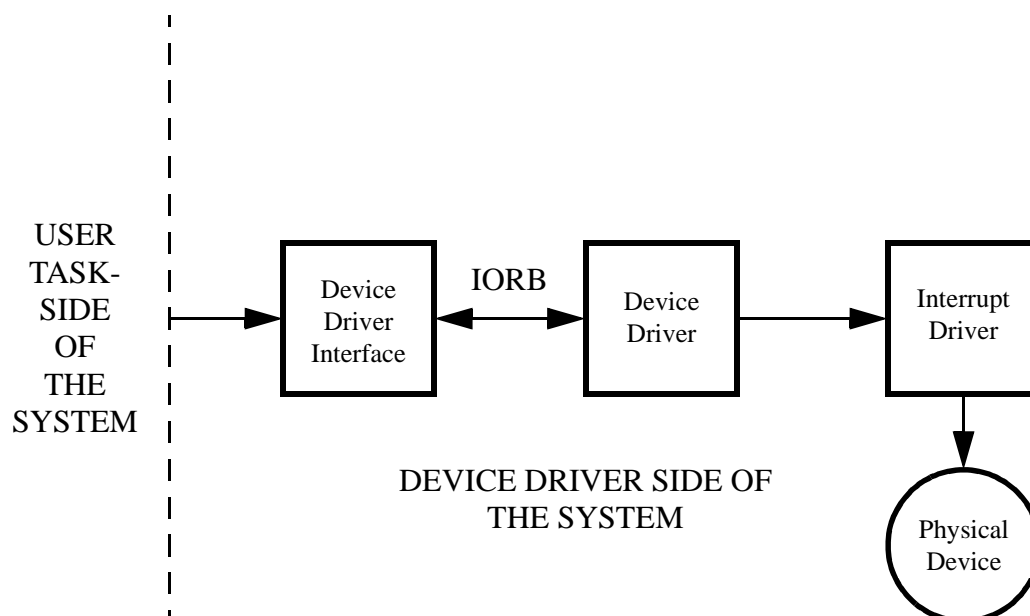


Figure 8-1: General structure of a device driver.

Traditionally the general area of device drivers has been considered to be one of the less elegant areas of operating systems design because of the number of *ad hoc* techniques used. The reason for this is the large number of different devices which a system has to cater for. Specifically, devices may vary in the following ways:

- (i) **Speed.** There may be several orders of magnitude between the data transfer rates of various devices. For example a disk drive transfers data at rates of the order of 10 – 400 Megabits/sec, whereas serial terminals and printers operate at speeds of the order of 10000 bits/sec.
- (ii) **Unit of transfer.** Data may be transferred in units of characters, words, blocks or records, according to the peripheral used.
- (iii) **Data representation.** An item of data may be encoded in different ways on different I/O media.
- (iv) **Permissible operations.** Devices differ in the type of operations they can perform. For example serial storage devices can be rewound, whereas a terminal cannot.
- (v) **Error conditions.** Failure to complete a data transfer have different consequences for different devices. For example a parity error on a RS232 communications line will invoke a different response compared to an error on a disk I/O operation.

In the following section one possible technique for the uniform handling devices will be outlined.

Design objectives and implications

- **Character code independence.** A user program should not be concerned with the particular character codes required by various peripherals. For example, the serial terminals may use different coding strategies, but the user programs should not have to be altered because of this. Character code independence implies the existence of some internal data representation which is translated (via a table for example) to the particular code required for a device.
- **Device independence.** This has two aspects:— independent of the particular one of a type of device to which it is allocated; independent of the type of device it is using. The later constraint is the more difficult of the two and is not always obtainable (for example it is no good trying to write to a CD-ROM). Device independence leads to the concept of virtual or logical devices (sometimes called *streams*). A program never communicates directly with a physical device but inputs and outputs to a particular stream. The association of a stream to a particular device occurs at the operating system level after the user initializes the association.
- **Efficiency.** Since the I/O operations tend to be a bottleneck in a system it is desirable to perform them as efficiently as possible.

One of the major implications arising from the above is the concept of the device control block (DCB), also known as the device descriptor. The purpose of the DCB is to associate the characteristics of the particular devices with the devices themselves rather than with the routines that handle them. Using this approach it is possible for device handlers to show great similarity and their differences to be derive solely from parametric information. The types of information about a device which may be stored in the DCB are:

- (i) the device identification.
- (ii) the instructions to operate the device.

- (iii) pointers to character translation tables
- (iv) the current status of the device – i.e. whether the device is busy, free or on-line.
- (v) a pointer to the TCB of the task which is using the device if it is busy.

In a manner similar to the TCB's, the DCB's are stored as a linked list of structures pointed to from the central table.

Anatomy of an I/O request

The structure of the I/O system in a general purpose operating system can be best understood from the following extended example.

When a user task is created by the operating system a number of default streams are usually made available to the user. For example, in a UNIX system the standard streams are stdin (standard input), stdout (standard output), stderr (standard error output) amongst others. Usually these streams are connected by the operating system to the users video display terminal. Streams can be associated by the user with other nonstandard physical devices. The equivalence of streams and device types for a particular task are recorded in a list of *stream descriptors* pointed to from the TCB (see diagram below).

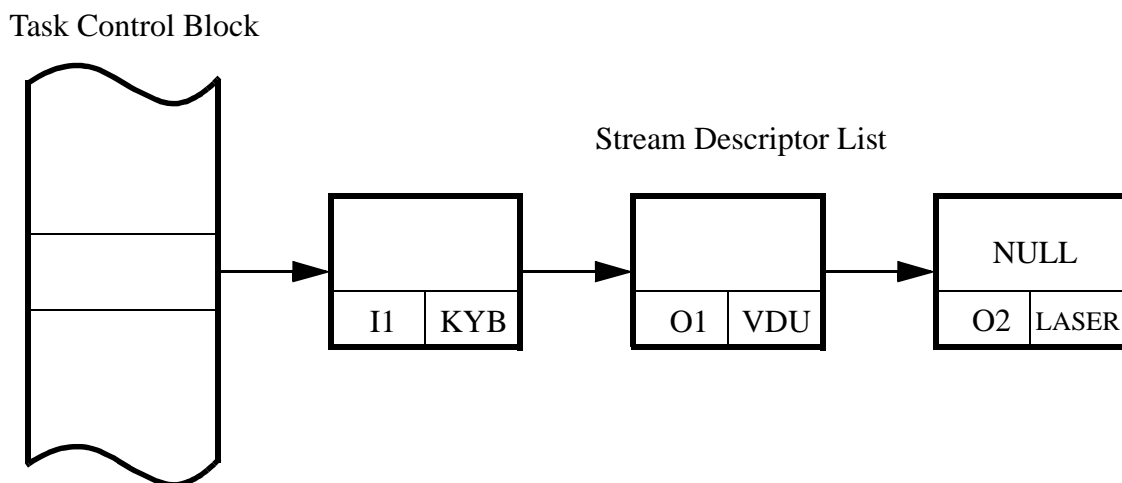


Figure 8-2: Device and stream information for a task

A typical request from a task to the operating system to do some I/O operation will take the general form of a call to a procedure (doio):

doio (stream, mode, amount, destination, semaphore)

where the parameters in the call are:

- **stream**— is the number of the stream on which the I/O is to be done.
- **mode**—indicates what operation, such as data transfers or rewind, is required. It may also indicate if appropriate what character code is to be used.
- **amount**—is the amount of data to be transferred, if any.
- **destination**—(or source) is the location into which (or from which) the transfer (if any) is to occur.

- **semaphore**—is the address of the semaphore to be signalled when the I/O request has been serviced.

The procedure **doio** is a reentrant procedure which carries out the following functions:

- (i) Map a stream number to the appropriate physical device. The stream descriptor list is used to carry out this function.
- (ii) To check the consistency of the parameters supplied to it. This can only be carried out once the physical device has been identified. For example, the **mode** would be checked to see if it is possible with the selected physical device. If an error occurs in any of the parameters then an error code would be returned to the caller. All information to be checked is contained in the DCB. The association of the stream number with a stream descriptor causes the further association with the DCB.
- (iii) To initiate the device service request. This is achieved by assembling a new data structure known as the I/O request block (IORB).

The IORB data structure is usually linked onto the end of a queue of IORB's for a particular I/O device. The other IORB's in the queue could come from the same task or could be from other tasks in the system. The *device handler* services the IORB's in the order of the queue. The *device handler* is notified that a request has been made by signalling a semaphore **request pending** which is contained as part of the DCB. When the I/O operation has been completed for a particular tasks request then a signal on the **request serviced** semaphore is done by the device handler. This semaphore address was passed to the *device handler* by the IORB, which in turn obtained it from the **doio** parameter list.

To summarise, the **doio** procedure outline would be as follows:

```
procedure doio (stream, mode, amount, destination, semaphore)
begin
    associate stream with device via TCB and the stream descriptor;
    check parameters against devices capabilities
    if error then
        exit with error number or message;
    else
        begin
            assemble IORB;
            add IORB to device request queue;
            signal (request pending);
        end;
    end;
```

The consistent structure of the DCB's enables the **doio** function to be used reentrantly by all the device drivers in the system. This has major benefits in terms of executable program size and software reliability.

The device handler

The device handler is the task which is responsible for servicing the requests on the device request queue. There is a separate device handler for each of the devices in the system, however because of the uniformity imposed by the DCB/IORB structure, device handlers all have a very similar structure and can share a number of programs.

The general structure of a device handler input operation is as follows:

```
repeat
begin
    wait (request pending);
```

```

pick an IORB from the request queue;
extract details of request;
initiate I/O operation;
wait(operation complete);           operation complete signalled by
                                   the interrupt handler.
if error then generate error information;
else
    begin
        translate characters if necessary;
        transfer data to destination;
        signal(request serviced);
        delete IORB;
    end;
end;
until kingdom_come;

```

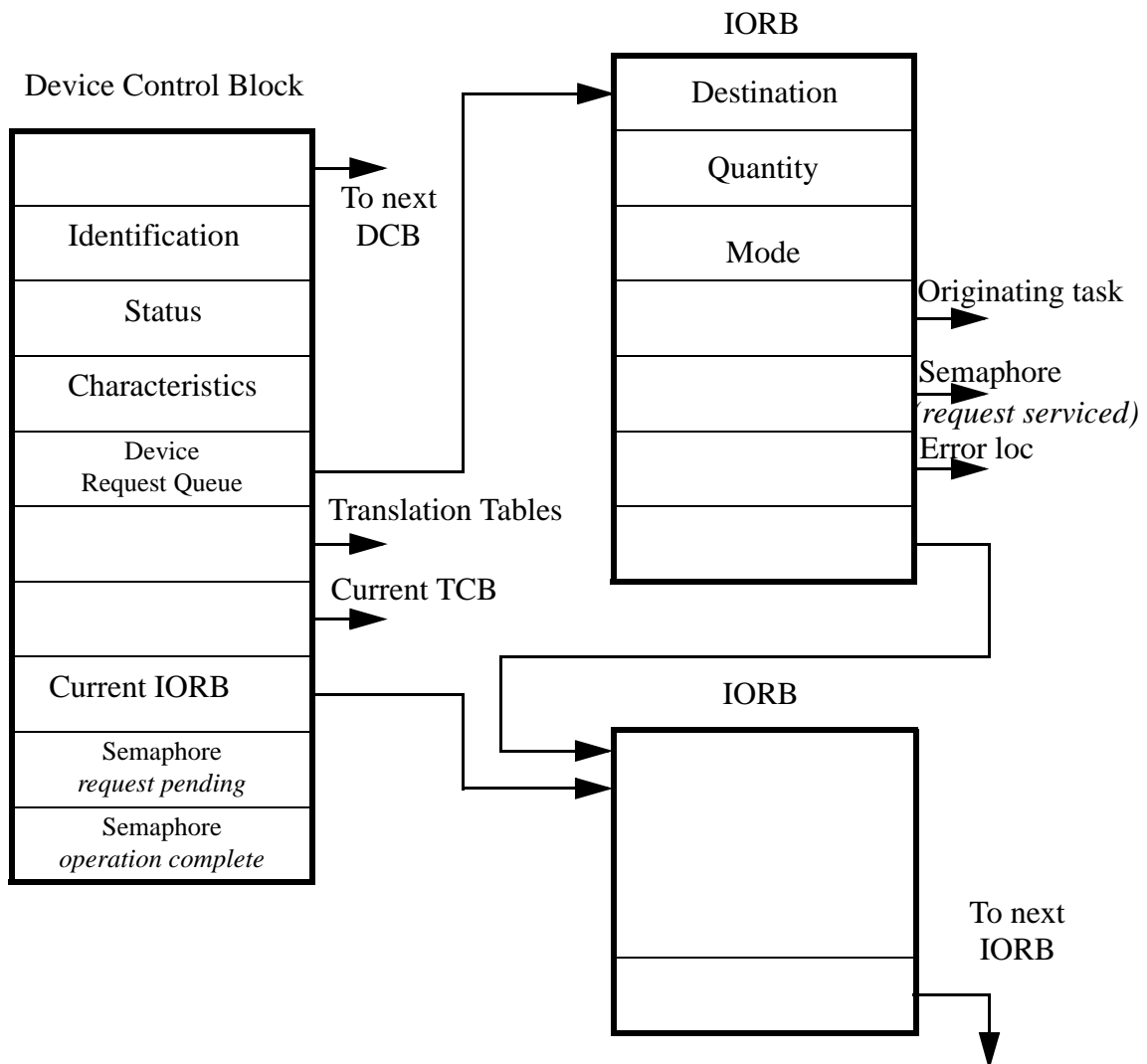


Figure 8-3: Device Control block (DCB) and device request queue.

The following discussion relates to the Figure 8-3 above:

- (i) The **request pending** semaphore is signalled each time a IORB is placed on the request queue for a particular device.
- (ii) Even though the normal queue strategy for the IORB's is that they are added to the end of the queue, this is not the only policy that can be undertaken. For example the queue may be ordered based on the priority of the originating task.
- (iii) All relevant data to carry out the I/O request on the physical device is contained in the DCB for the particular device.
- (iv) The semaphore **operation complete** is signalled by the interrupt routine when an I/O operation has been completed. This then allows the device handler to begin operation on another IORB.
- (v) If there is an error in the I/O operation (which is determined by checking a status location in the physical device) then an error location is set appropriately. The address of the error location is contained in the IORB.

The synchronization and flow of control between the task requesting I/O, the I/O procedure, the appropriate device driver and the interrupt routine is shown in the Figure 8-4. The solid arrows represent transfers of control, whilst the dashed arrows represent synchronization via semaphores. Note that as the diagram is drawn the requesting process can continue execution asynchronously with the I/O. The requesting process will only be halted if the **request serviced** semaphore has not been signalled when the **wait(request serviced)** is executed. A potential problem with this arrangement is that the user must take the responsibility of ensuring that the data is ready before an attempt is made to read. An alternative technique would be to carry out the **wait** semaphore operation in the operating system itself. The data read operation effectively becomes a single high level instruction as far as the user routine is concerned. However one loses the concurrent operation of the user task and the I/O operation.

Buffering

The description of the I/O handling as described in the above section is often very inefficient. In the case of the above every I/O operation requires that the interrupt handler be activated and the entire operation be complete before the device handler is allowed to continue. Such a strategy can result in excessive calls to the kernel semaphore primitives at both the device driver level and the user task level (via the **wait(request serviced)**).

The technique called buffering was developed to overcome the above problem. A buffer is a storage area in memory which can be used to store the incoming data. A call to the DOIO procedure in this case would simply get the data directly from the buffer instead of generating an IORB and activating the device handler. If on an I/O request the buffer is empty then a request is made to the device handler to get some more data. In this case an IORB is generated and the device handler works much as described previously.

When buffering is implemented then *double buffering* is the usual situation (especially when block oriented I/O devices are being used). The reason for this is due to the fact that in the case of input the occurrences which produce the input are usually asynchronous. Therefore when the interrupt routine is activated the input data has to be stored. Since this generally wouldn't coincide with the activation of the device driver requests, the data has to be stored. This is done in the interrupt buffer. It is from here the device driver would retrieve the data when an IORB is generated.

Similar arguments can be mounted for output I/O. Here the main consideration is efficiency,

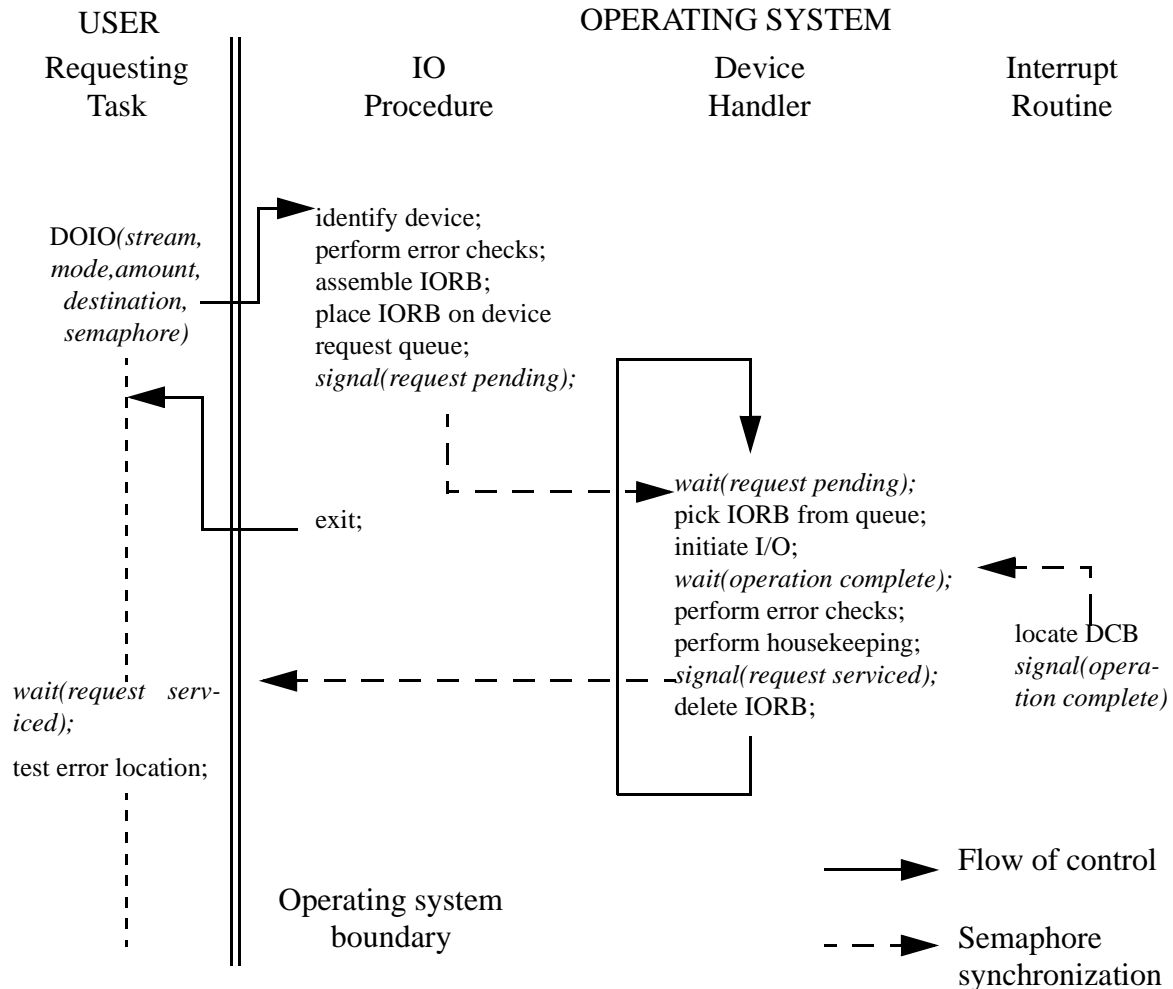


Figure 8-4: Sketch of the control flow in an I/O system

since data cannot be lost as the user task is producing it.

Issues related to efficiency

The buffering techniques used in the I/O system can have an enormous impact on the efficiency of the system. The data structure used to store the incoming or output going data is also influenced by the hardware structure of the system.

One of the most common data structures used for I/O is the circular buffer. This data structure is used as the buffer when the device operates as byte-at-a-time up to moderate rates. Typical devices which may use a circular buffer would be keyboard devices. These are suited to circular buffers because the input rate is low and the data is often handled by the user tasks on an individual byte basis. Furthermore, delays associated with the accumulation of blocks of data are unacceptable.

When data rates become high the circular buffer data structure may not be the most suitable. The reason for this can be several fold depending on the implementation details. Often the placement of a character into the interrupt routine results in a **signal**. This can lead to excessive task switching, resulting in poor efficiency. Coupled with this one also has the overhead of an interrupt for each character received and sent. There are several ways of overcoming these problems. A feature which is used in some serial input hardware is a hardware queue. As char-

acters come into the system they are stored in the hardware FIFO (first in first out) structure. If the data reaches a software programmable character number limit then an interrupt is generated and the hardware FIFO is flushed. If a number of characters are received but the software programmed limit to generate an interrupt is not reached then an interrupt is generated in any case after a programmable elapsed time. A similar scheme can be used for output of characters. One of the main advantages of this approach is that the interrupt loading is significantly reduced.

For very high speed I/O even the above hardware buffering approach is not sufficient to ensure reasonable interrupt and kernel operation loadings. In cases such as this Direct Memory Access (DMA) transfers are used. This lowers the interrupt and kernel call loading because data is sent in blocks without any software involvement. The software overhead is incurred to set up the block of data to be transferred and to program the hardware which will actually send the data. Once the transfer is initiated the hardware of the DMA device directly transfers the data byte by byte to/from memory from/to the I/O hardware. Only when the block of data has been transferred is an interrupt initiated and software becomes involved again. For character by character input operations (such as occur from terminals) the hardware FIFO technique as described above is used. However DMA techniques are commonly used for output operations, even when sending characters to a terminal. For operations such as disk subsystem I/O DMA is almost always used.

This chapter is not designed to be an exhaustive dissertation on all aspects of file systems. The reason for this is that these notes emphasize real-time aspects of operating systems, and file systems are associated more with soft real-time (e.g. multi-user) operating systems. Nevertheless, files are sometimes used in real-time applications, especially if there is data logging involved. In addition, file systems are usually covered in traditional operating systems courses, therefore students should have some basic knowledge about them.

The references used to compile the information in this Chapter are [1], [4] and [5].

Logical Structure of File Systems

In most general purpose computer systems there is a requirement for storing and retrieving information, both system information, and user information, between power-downs of the computer system. The ideal way of achieving this would be to have a gigantic memory in the computer that could store all the information that a user generates, as well as store the system utilities and routines required by the system itself. These persistent storage techniques still haven't found their way into main stream computer systems, although research is continuing into the techniques required to implement such a system. As random access memory costs drop, and chip densities increase the feasibility implementing a commercial persistent storage based computer is increasing.

For the immediate future most computer systems use a conventionally designed file system. For this reason we shall concentrate on the design of these systems. The term file system is derived from the analogy with a normal paper file system where one has a filing cabinet (which is analogous to a disk drive), and within the filing cabinet there are a number of folders, each folder containing a number of files. It is these files that actually contain the useful information as far as the user is concerned. The folder structure is necessary to allow the user to organize the files so that they can be easily found.

For a computer file system to be useful from a users point of view it should have the following attributes:

- (i) allow the creation and deletion of files
- (ii) allow files to be written to and read from the file system.
- (iii) make the management of the secondary storage system transparent as far as the user is concerned.
- (iv) allow reference to files using symbolic names. Since the user has details of the secondary storage effectively hidden from them, then the user should be able to access the device using only names of there own making.
- (v) protect files against system failure. Different file systems have varying compliance with this desired trait. Protected file systems generate user confidence in the systems relia-

bility.

- (vi) allow the sharing of files between co-operating users but protect files against access from unauthorized users.

All file systems aim for device independence. This means that access to a file is the same no matter what or where the device is that it is stored on. For example, in most operating systems the same file operations can be performed on a hard disk and a floppy disk, and as far as the user is concerned there is little or no difference in the actions that need to be carried out in order to access the file.

Some file systems are more device independent than others. For example, in UNIX a file system can be mounted anywhere in the directory tree, and a file can be accessed without any concern by the user as to where the file is physically located. An example of this is when a file system is not even mounted on a local disk drive, but is on a drive somewhere else on a network.

MS-DOS on the other hand requires the user to specify the device that the file resides on. For example, if one wished to access a file on drive D, then the path has to be prefaced by D:. This is not always required, as MS-DOS has the concept of a default drive.

Basic Implementation

This section will consider, firstly in a very general way, how the above logical file system properties can be achieved. The term logical file system properties refers to the view that the user or application has of the file system, as opposed to the actual physical structure on the disk of the file system. There is no necessity for there to be a connection between the logical and physical structure, although in some cases there is a connection.

Figure 9-1 illustrates three common file organizations. The first is a simple sequence of bytes – this is the structure that UNIX uses. The second is a sequence of fixed size records. Arbitrary records can be read or written, but records cannot be inserted or deleted in the middle of a file. The old CP/M operating system (which ran on the 8085 and the Z80 microprocessors in the late 1970s and early 1980s) used this file structure. Finally the third way is a tree of disk blocks, each block holding n keyed records. Records can be looked up by key, and new records can be inserted anywhere in the tree. If a record is added to a block that is full, then the block is split into two blocks, both of which are then added to the tree in their correct alphabetical sequence. This method of is used on many main-frames, where it is called ISAM (indexed sequential access method).

One problem that arises in many of the common file organizations is how to insert data into the middle of the file. If the file is organized as a sequential file, a common technique is to open a second file when data is to be input into the primary file. This second ‘overflow’ file is used to store the data to be inserted, and this is eventually merged with the primary file when the file is closed. Other tree based file organizations do not have this problem, as the file is connected together by the links that make up the tree. Therefore new data can be easily inserted by simply changing the links.

It should be noted that the above file organizations are usually implemented in a hierarchical manner, and may not necessarily reflect the intrinsic underlying organization of the file system. However the above logical file organization exists as far as applications are concerned. For example the lowest level of the file system can be implemented as a sequential byte stream. To allow access based on records, higher level routines translate the record number into the number of bytes from the start of the file. This is then used by the lower layer of the software to move the file pointer to the correct byte of the beginning of the record. Clearly this would be a

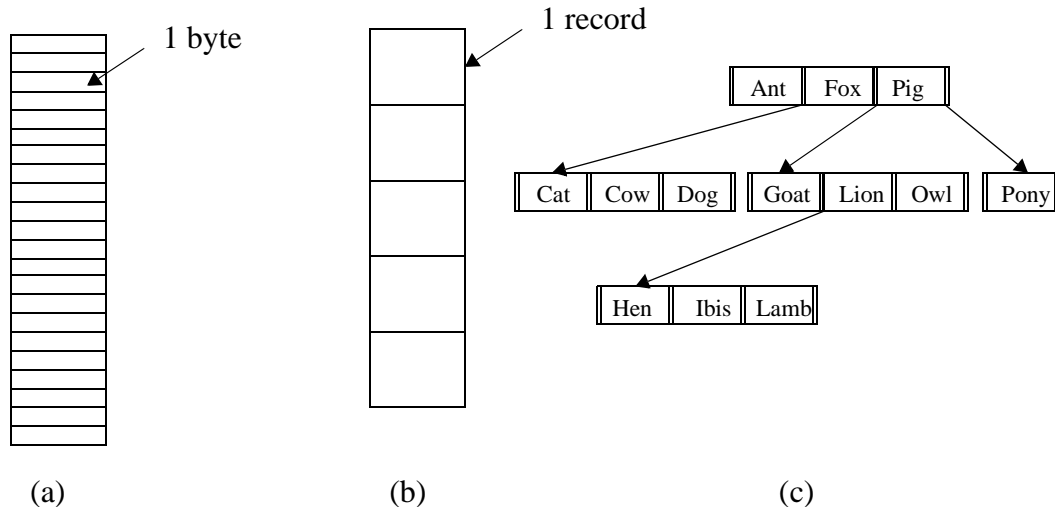


Figure 9-1: Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree

very inefficient way of accessing these types of files.

For example, in the MS-DOS and UNIX file systems, files are organized as a sequence of bytes. However, one can also access records. The underlying BIOS allows the record number and the size of the record to be specified. The organization of the underlying file system operation allows specific records to be located without having to go sequentially through the file. However, in both these operating systems the basic file system is essentially a sequence of bytes, with some access functions that allow quick access to parts of the file. This will become more apparent after we look in detail at their operation.

It should be noted that the underlying physical organization on the disk is very often different from the low level programming interface. For example, in MS-DOS the low level interface is a sequential file of bytes. However, data is read, written and stored on the disk in sector size chunks. A sector is typically 512 bytes. Therefore, when a byte is read from the disk a minimum of one sector is read into an internal buffer, and the particular byte is located in this buffer. In fact in MS-DOS the minimum size addressable unit on the disk is a cluster, which is a collection of several sectors. The size of a cluster is dependent on the size of the disk. Clusters are required due to limitations with the directory structure for MS-DOS.

When data is arranged as sequential records on the disk, the records themselves usually are not arranged sequentially on the disk. This is because the records have fields in them allowing them to be arranged as a linked list. This helps overcome problems with disk fragmentation, allowing a file to be stored on the disk even if there is not enough contiguous free space at least equal to the file size.

Sequential Files

This is the simplest type of file structure. In this file structure a fixed format is used for all the records in the file. The records are stored as far as the user is concerned in a sequential order (although the physical storage does not necessarily follow this pattern as mentioned above). Access to the file is usually carried out in a sequential manner, making these files particularly suitable for batch processing applications such as billing or payrolls.

For applications where access to the file is random in nature they give poor performance. To

find a particular record with a key field the file has to be searched sequentially, with the key being checked against the desired key for each record read. Records are typically arranged in these files in the order of their key. For example for numeric keys the records are arranged from the lowest key number to the highest. For text keys they are arranged in alphabetical order.

Indexed Sequential Access Method (ISAM)

A variant of the sequential file is the index sequential file. Records are still arranged based on a key field, but an index file is added to structure. This file provides a quick look-up capability so that the vicinity of a record can be located without a search, and then a sequential search can occur from this point to locate the correct record.

The operation of this file is best demonstrated by an example. Consider a single level index (see Figure 9-2), that contains a number of the keys into the file evenly distributed from the beginning to the end of the file. The index file is a simple sequential file. In order to locate a particular record based on say a text key, then a search is made through the index file to find the key that is the highest that is equal to or lower (i.e. comes before the desired key in the alphabet) than the desired key. The associated pointer is then retrieved from the index file, and this is used as the base position to begin a sequential search into the main file. Therefore the search can begin in the vicinity of the required data using this technique.

The concept of an overflow file is also easily catered for in this file structure. If entries are to be added in the middle of the file, then they are put into the overflow file. The index entry is changed to point to the approximate position in the overflow file, and then a sequential search is initiated from this position. If the index initially points to the main file, but the subsequent search is going to go into the overflow file, this is handled by there being a pointer in each record to the next record. In this way the overflow file is seamlessly integrated into the main file. Of course the overflow file and the main file have to be integrated when the main file is to be closed.

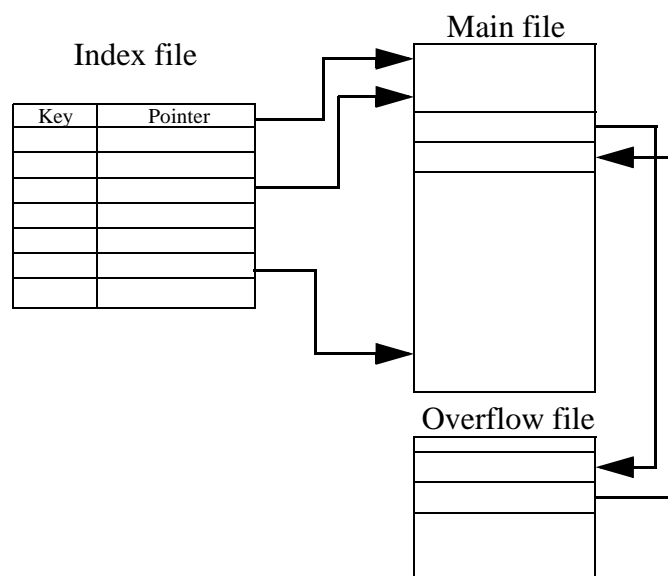


Figure 9-2: Indexed sequential file access

It should be again noted that the underlying structure of the file system may not necessarily reflect the sequential index structure. For example, in an inherently sequential file system the above accessing process would result in a byte file position pointer and the underlying byte

sequential file would be read until the particular byte position in the file was obtained. However as far as the application that is making the file request is concerned the file is an indexed sequential file. The mapping from the indexed file type to the underlying byte sequential file type is essentially carried out by the type of pointer information that is contained in the index file. Naturally if the underlying structure of the file did mirror the indexed sequential file then the accesses would be more efficient in that the particular record could be accessed directly without having to read in the previous records as in a sequential file system. This functionality very much depends on the physical structure of the directory information for the file system. We shall return to discuss this further later.

Indexed file

A further enhancement to the indexed sequential file is the fully indexed file. The advantage of this file type is that access to the file is no longer confined to only one key (as with indexed sequential access). The concept of sequential access is no longer used, instead every record in the file has to have an index entry. There are multiple indexes for the different fields that are going to be used to access the file. With this organization there are no longer any restrictions on the placement of records. In addition not all records need to contain the same keys, meaning that not all records will appear in all the indexes. The records themselves need not be of the same length, as length information can be contained in the index entries. The addition of a record to the file means that an entry has to be made in all the appropriate indexes for which the record has a key.

The index is usually organized as a two level structure. The outer most level basically contains categories for the information in the second index file. These categories contain pointers into the secondary exhaustive index file, which is then searched sequentially from the indexed point to find the matching key that will have an associated pointer to the appropriate record in the file.

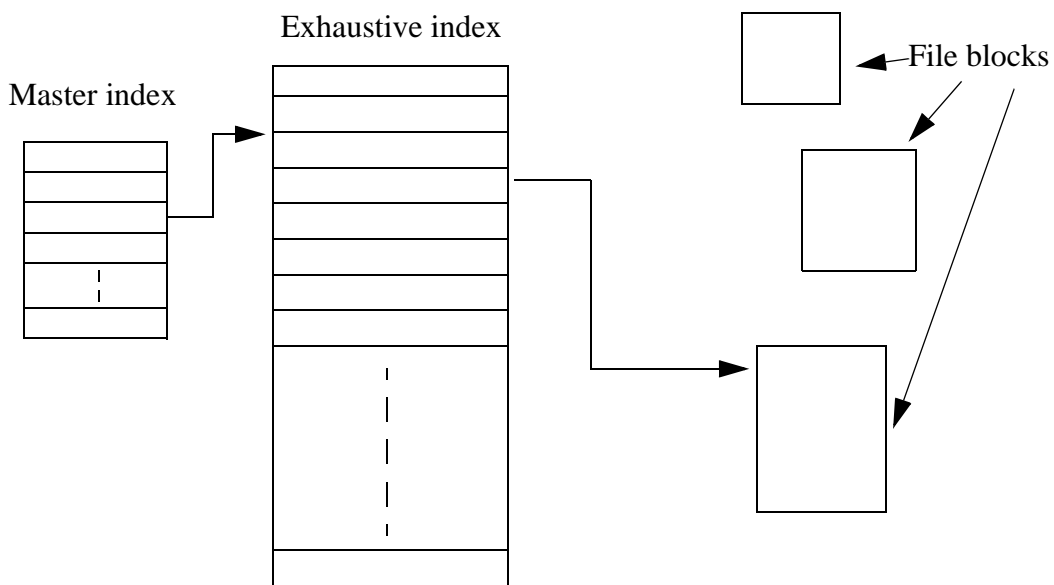


Figure 9-3: Fully indexed file system.

Again the complexity of mapping the desired record to an actual physical disk block depends on the characteristics of the directory structure of the particular operating system being used.

Operating Systems that use file systems of this nature are ones where random access to large amounts of information is required in reasonably short periods of time. Examples are airline reservation and inventory systems.

Remark: All of the above file access schemes are logical schemes as seen by the user or the application software. However how these logical access methods are mapped to the physical storage patterns on the disk will become clearer after the following sections on the directory structure and the physical structure of the file system.

Directories

One of the most basic features of a file system is the ability to map a symbolic name to a physical file on the secondary storage device. This functionality is provided by a file system entity called the **directory**. In many systems the directory is a file itself.

In its simplest form a directory is an index containing a number of entries, each one corresponding to a file. Each entry contains a number of pieces of information (see Figure 9-4). This

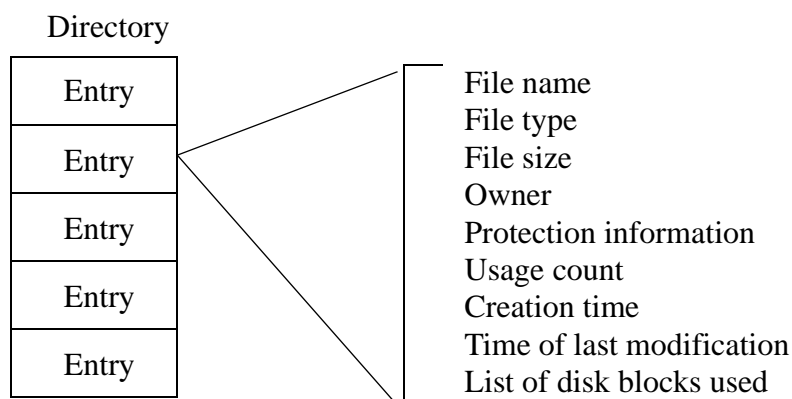


Figure 9-4: Simple single level directory structure

simple structure is totally unsuitable for a multiuser computer system, as one would soon have conflicts resulting from different users giving their files the same names. In fact, such a system is really not very suitable for a single user computer system, since all the users files would be stored together. With modern hard disks one would soon have hundreds if not thousands of files stored in the directory. Finding files which are logically connected together would be very difficult. It is analogous to having a filing cabinet in which all items are stored under miscellaneous. Such a flat directory structure was common with the early microprocessor operating systems (e.g. CP/M, Apple DOS), since these commonly used floppy disks which were limited to 128 kilobytes. The number of files stored on disks of this capacity was not very large. Furthermore these were single user operating systems.

A much more flexible organization for the directories is the hierarchical directory structure. This is the structure that is used by most modern operating systems, and should be familiar to most computer literate people. Figure 9-5 shows the basic organization of this scheme. Often the directory and subdirectories are stored as sequential files. This is not always optimal, particularly when the directories contain a large number of entries. In these situations hashing schemes are commonly used to enable files to be quickly found.

The directory structure in Figure 9-5 is also commonly used in multiuser operating systems, as well as single user systems. In the case of the multiuser system the master directory might con-

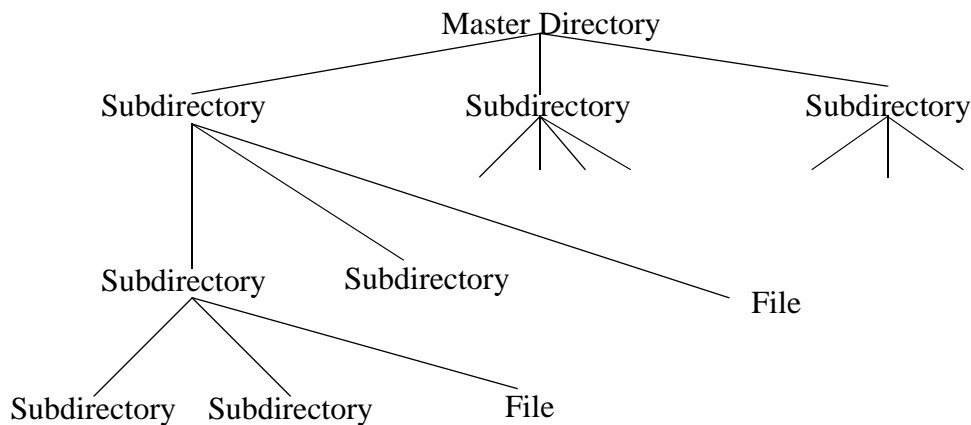


Figure 9-5: Hierarchical directory structure

tain the different users in the system, and the contents of the directory entry for this user points to a subdirectory that is effectively the user home directory.

Symbolic Naming

Users require that files be addressed by a symbolic name. In a multiuser system it is important that the file names are unique, else one would have to enforce the ridiculous constraint that no user could name a file with the same name as another user.

Fortunately, the hierarchical directory structure shown in Figure 9-5 implicitly allows this requirement to be satisfied. Any file in the system can be located by following a path name, which consists of a listing of the directories from the master directory, ending finally with the file name itself. Typically the master directory will contain a number of subdirectories, one for each user in the system. These directories are typically given the names of the users. Since the user name in a typical multiuser system is unique, then the path name to a file will be unique, regardless of whether the file name at the end of the path is unique or not.

To allow a user to access their files without specifying the complete path to the file, most computer file systems have the concept of the *working directory*. The working directory consists of the path to the directory of the file that the user wishes to manipulate. This “default path” is automatically, but transparently appended to all the file accesses.

Within a directory there is typically two special files that cannot be deleted – “.” and “..”. The “.” file is actually the current directory, and “..” is the previous directory in the directory tree. Typically the directories are stored as sequential files. Therefore one could consider that “.” and “..” are the names of the sequential directory files for the current and previous directories in the file system.

Access Rights

Almost all file systems contain the concept of file access rights. This even applies to single user file systems such as MS-DOS. In this file system each file has several bits that allow a user to specify if the file should be a read-only, hidden, or a system file. In addition MS-DOS also has an archive bit that is used by backup software to determine whether a file should be backed up.

Access rights perform a more important function in a multiuser file system. Typically a partic-

ular user does not wish to have their file available for other users to delete, read or modify in some way. On the other hand, there may be occasions where such access is legitimate. Therefore there needs to have access bits that allow the user to control the access to their files by other users.

In the UNIX operating system, the user can control user access through the use of group access rights. The groups supported are: the user group, a work group, and the rest of the world group. The user group consists of the owner of the file, the work group can be a grouping of different users. In each of the groups the owner of the file can specify whether a user belonging to one of these groupings can read, write or execute the file. Also associated with each file is the concept of the files owner. The owner of the file is the user that created the file. It is the owner that can change the access rights to the file.

In addition to files being protected by access rights, directories can also be protected. In this situation all the files and subdirectories in the directory are also protected.

Typical information contained in a directory entry is shown in Table 9-1. How this information

Table 9-1: Contents of a typical directory entry^a

Basic information	
File name	Name chosen by the creator (user or program). Must be unique within a specific directory.
File type	For example: text, binary, executable etc.
File organization	For systems that allow different file system organizations this entry indicates the organization of this file.
Address information	
Volume	Indicates the physical device that the file is stored on.
Starting address	Starting physical block on the secondary storage device (e.g., cylinder, track, and block on the hard disk).
Size used	Current size of the file in bytes, words or blocks.
Size allocated	The maximum size of the file.
Access control information	
Owner	User who is assigned control of the file. This is the person that can grant or deny access to other users and changes these privileges
Access information	A simple version of this element would include the user's name and password for each authorized user.
Permitted actions	Controls reading, writing, executing, transmitting over a network.
Usage information	
Date created	When the file was first placed in the directory.
Identity of creator	Usually but not necessarily the current owner.
Date of last read access	Date of the last time a record was read.

Table 9-1: Contents of a typical directory entry^a

Identity of last reader	User who did the last read.
Date file last modified	Date of the last update, insertion, or deletion.
Date of last backup	Date of the last time the file was backed up on another storage medium.
Current usage	Information about the current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk.

a. From “Operating Systems, by William Stallings, Prentice-Hall, 1995, ISBN 0-13-180977-6.

is stored varies widely among different operating systems. For example some information may be stored in a header to the file itself. This keeps the directory entries smaller, allowing most of the directory information to be kept in memory.

Physical Structure of File Systems

Thus far we have discussed the structure of the file system as seen from the user or applications point of view. However, in the depths of the file system code the logical structure has to be mapped to physical data stored on the disk. This section will discuss various techniques to achieve this.

Physical Structure of Disks

A disk is physically laid out as a sequence of cylinders and sectors. The cylinders are a set of concentric rings starting on the outside of the disk and moving to the centre (see Figure 9-6). On each of the cylinders there are a number of sectors, each sector being a number of bytes in length. The number of bytes in a sector depends on the format of the disk. Typical values are 128, 256 and 512 bytes. The sector is the absolute minimum addressable unit on a disk system. This block access nature is what makes a disk fundamentally different from random access memory. It should be pointed out that sectors may not be the minimum addressable unit for a particular disk access method.

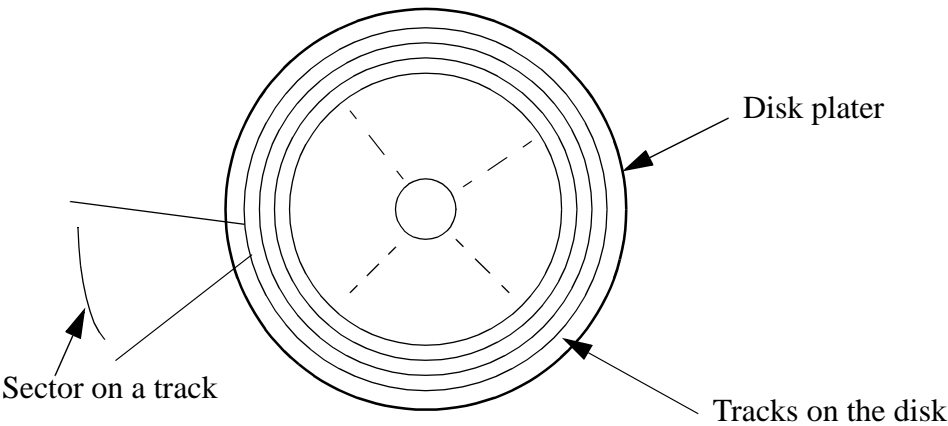


Figure 9-6: Format of a typical disk surface.

The task of the lower portions of the disk file system software is to map the logical records onto this physical disk structure. There are two main approaches to be taken. The obvious one is to store data as a contiguous set of bytes in contiguous sectors. The obvious disadvantage of

this approach is that as a file increases in size there may not be enough contiguous space in the current area, and the file would have to be moved to another area of the disk where there was. Another equally serious problem is that one would suffer file system fragmentation. This is best demonstrated by an example. If one wishes to write a file to the disk of a particular size. This operation is occurring after there has been a lot of other writes and deletions of data on disk. It may be that no single free area on the disk is large enough to store the file, even though the addition of all the free storage on the disk is more than enough to store the file. The disk is said to be fragmented when this occurs. For a file system requiring contiguous locations on the disk for file storage this situation can become disastrous.

The approach that is taken to the physical layout of most file systems is to split the file in blocks which do not have to be contiguous. This overcomes the fragmentation problems that were cited above. The next question is how large should these blocks be, and how are the probably discontinuous blocks of one file connected together.

The size of the blocks is a trade-off of speed against disk utilisation. For example one block allocation decision maybe to allocate one disk cylinder to a block. Assuming that a block of this size contains a number of logical records, then a large number of records for a file will be read into a memory based disk buffer in one access. Subsequent file accesses will most probably find the record they required in the memory. However, the downside of this decision is that a file of one byte in length is going to require a single block to store it, this being one cylinder. Therefore there is a large amount of wasted space.

The alternative is to allocate a very small block size. For example a block is the size of one physical sector would not be an unreasonable decision. In this case if one were to read a number of logical records in from disk then this is going to involve a number of sectors to be read in from the disk. Depending on the timing of the reading, then would involve a number of separate accesses to the disk drive, each involving a rotational latency and possibly a seek time. Therefore accessing the file will be a slow operation.

Generally the block size is chosen to be an integral number of sectors. For example if the block size is 1K bytes and the sector size is 512 bytes, then every access to the disk will involve reading two physical sectors, as the block is the minimal addressable unit.

Disk allocation table

Assuming that a block size has been chosen then the next task is to devise a technique of keeping track of which blocks are used and which are free on the disk. There two main techniques that are used to accomplish this: a list of free block numbers, and a bit map. The list of free blocks is simply kept in a special file on the disk. This file contains the numbers of the blocks on the disk that are not being used by any file. The problems with this approach are:

- (i) The amount of space taken for the list can be large. For example when the disk is nearly empty, the number of free blocks will be large. If each disk block number takes 16 bits, then the space taken on the disk will be reasonably large if the disk is nearly empty, but more importantly the whole free table cannot be loaded into the memory.
- (ii) As free blocks are used then they have to be taken off the list of free blocks. This will necessitate some complex procedure for updating the free block list.

The alternative strategy is to keep a bit map of the free locations. In this scheme there is one bit for each block on the disk. If the block is used then the corresponding bit is one, else it is a zero. Clearly this will result in tracking the free memory with up to one 16th the storage requirement on the previous scheme. This scheme usually allows the free block table to be buffered in the main memory of the machine.

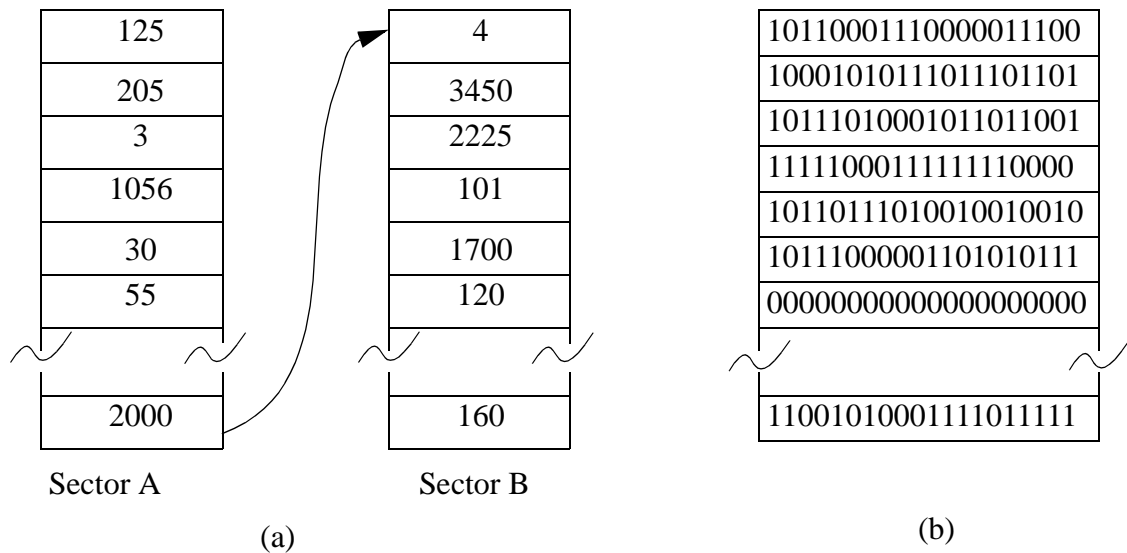


Figure 9-7: Disk allocation tables: (a) Block number free list. (b) Bit map free list.

Record blocking

Another design decision that has to be made by the file system designer is how the logical records are to span the blocks on the disk. There are basically three choices for this:

- (i) *Fixed blocking*: Records of fixed length are arranged so that there are an integral number stored in a block. If a record does not fit into a block then there is unused space at the end of the block. This is a commonly used technique for sequential files with blocks of fixed length. Records are limited to the size of a block.
- (ii) *Variable-length spanned blocking*: Variable length records are used with no unused space in a block. This means that if a record does not fit into a block, then as much as will fit is put into the block, and the remainder is put into the next block. Pointers are used to connect these records together. This is a very flexible file structure, but is the most difficult to implement. Records do not have any limitations on their size.
- (iii) *Variable-length unspanned blocking*: Variable-length records are used, but spanning is not used. In other words if a record does not fit into a block then the remainder of the block is left unused. Records are limited to the size of a block.

Clearly all these techniques have their strengths and weaknesses.

File allocation table

Given that one has a technique for finding free memory blocks and a policy has been decided upon for the records, then one has to decide how to allocate blocks to a file and how to keep track of the blocks allocated.

In many ways the allocation of blocks to a file parallels the logical organization of files. For example, one can allocate files using contiguous blocks (see Figure 9-8). This will lead to a very simple file allocation table structure in that it only needs to contain a pointer to the starting block of the file, and the length of the file. The file allocation table (abbreviated to FAT) is the means by which the operating system keeps track of the blocks that are used to store a file. The contiguous allocation strategy has a number of problems; (a) the size of the file has to be known at file creation time. With many files this is impossible to know, (b) one often ends up

with a large amount of fragmentation in the file system, (c) fragmentation means that a contiguous block of memory large enough for the file is often impossible to find even though the total free memory on the disk is large enough to accommodate the file.

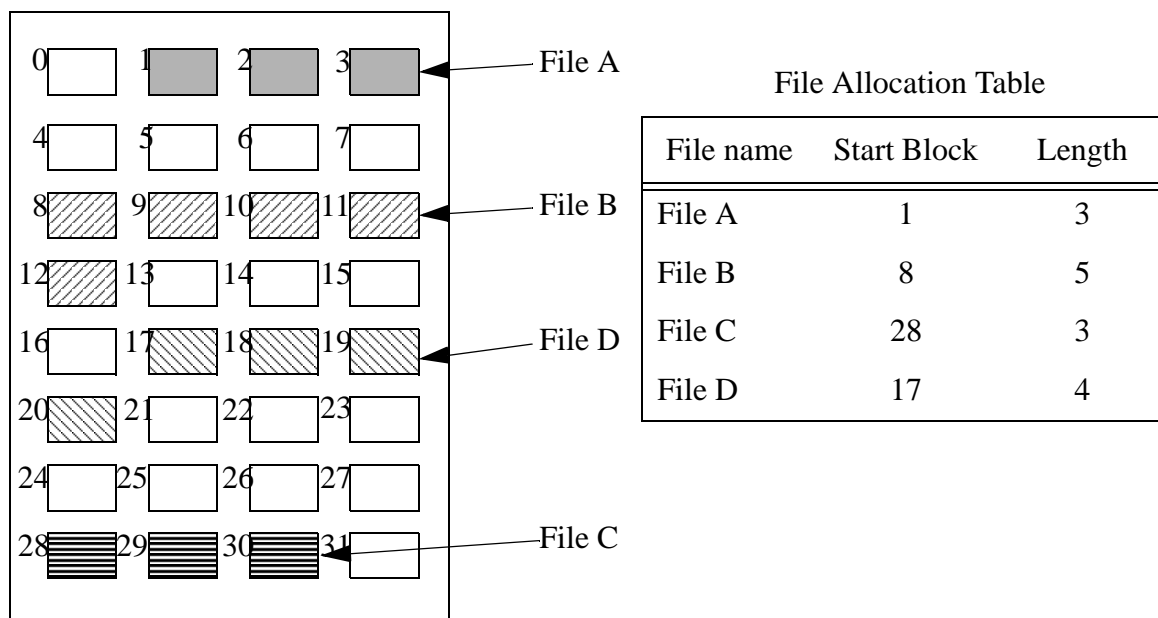


Figure 9-8: Contiguous file allocation scheme.

At the other extreme from the contiguous allocation scheme is the chained scheme (see Figure 9-9). In this scheme files are allocated on an individual block basis. The blocks are connected together using a pointer. Therefore the blocks of a file form a linked list. The file size does not have to be known when the file is created, free blocks being added to the file as it grows. The extra blocks required can be allocated from the free list as required. There is not macro scale fragmentation to worry about using this scheme. On the down side though, blocks of any file can be scattered anywhere on the disk, and this can slow down file accesses compared to a file that is stored sequentially on the disk (due to the time required to move the head around the disk). In addition to this problem to locate a particular block the file chain has to be traversed until the required block is found. This also obviously requires a lot of disk accesses.

A scheme which attempts to overcome the problems cited above for the contiguous and chained schemes is the indexed chain scheme. This file allocation scheme uses a single level index that points to the various components of a file. The file allocation entry for a particular file has a pointer to the index block of the file, which is typically the first block of the file. The index block contains a list of all the blocks in the file in the order that they occur (see Figure 9-10). With this scheme any particular block of a file can be found without having to search through all the chained blocks to find it, and one still has the other advantages of the chained approach.

Some examples

MS-DOS file allocation

MS-DOS uses a FAT system similar to the indexed file allocation technique mentioned above. There are two different FAT organizations used for MS-DOS disks: the 12 bit FAT and the 16 bit FAT. The 12 bit FAT was used on floppy disks and early hard disks. The 16 bit FAT was

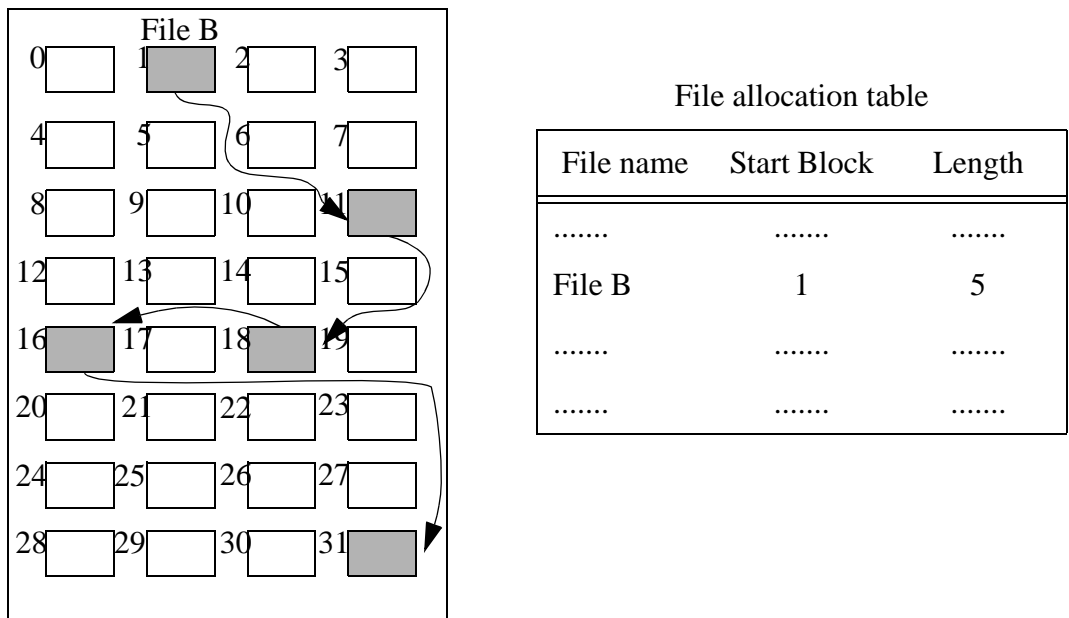


Figure 9-9: Chained file allocation scheme.

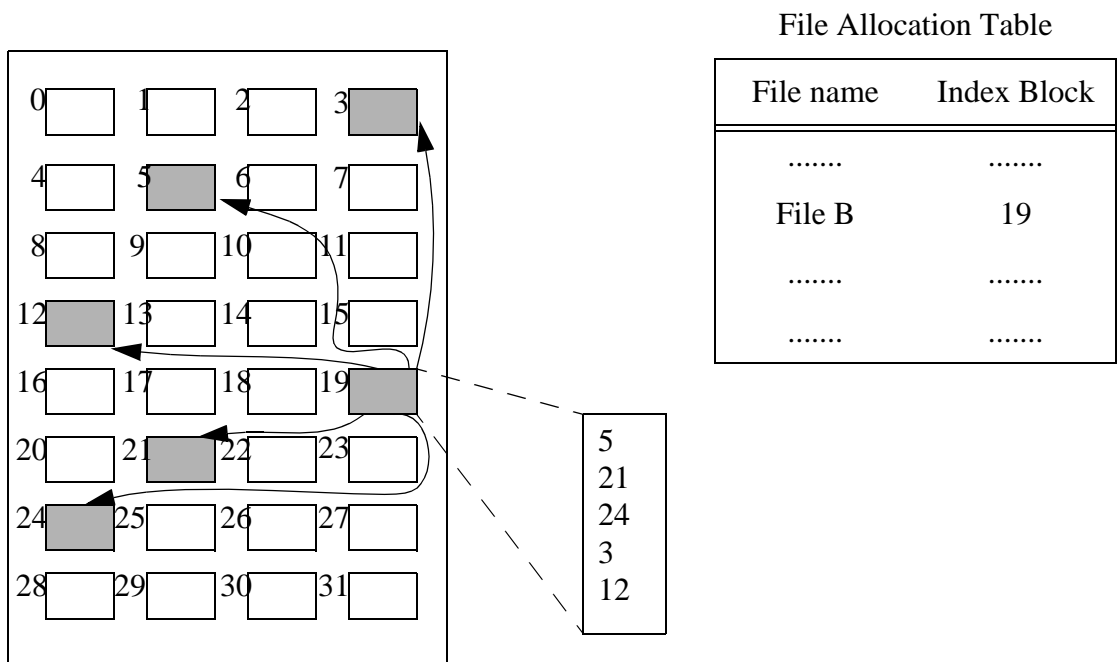


Figure 9-10: Indexed file allocation scheme.

used on the later MS-DOS based file systems, and the 32 bit FAT is currently used in Windows 95, 98 and 2000.

Let us consider the 12 bit FAT (the 16 bit FAT is a simple extension of this). The FAT consists of a table of 4096 numbers (the 16 bit FAT is a table of 65,536 numbers) ranging from 0 to 4095 (16 bit 0 to 65,535). The number of entries in the FAT determines the size of a cluster on the disk. In the MS-DOS file system the cluster size is the minimum unit of the disk storage

that can be addressed. For example, if one has a 500Mbyte disk, and a 16 bit FAT then the cluster size is calculated as follows:

$$\text{Cluster size} = \frac{512,000,000}{65,536} = 7,812.5 \text{ bytes}$$

Therefore the cluster size for this size of disk drive is 8KBytes, since there has to be an integral number of sectors in a cluster. If the disk drive is 2GB in size (which is small by today's standards, but large compared to what was available when MS-DOS was designed) then the cluster size is 32KBytes. This means that a file of one byte in size will consume an entire cluster. Therefore, it is better to partition a single physical disk drive into several logical disk drives of 500MB or less. Clearly from the above explanation, a single cluster consists of several sectors on the disk. For the 500MB disk case a single cluster consists of 16 sectors. If there is a bad sector within a cluster the whole cluster is deemed to be bad and is recorded as such in the FAT.

As mentioned in the previous paragraph the FAT consists of a set of consecutive numbers corresponding to each of the clusters on the disk. Associated with each of these entries in the FAT there is information describing the use of the cluster. Consider Table 9-2 which shows the structure of the FAT for a 12 bit MS-DOS FAT based disk.

Table 9-2: FAT for an MS-DOS file system.

FAT entry (Cluster #)	Dec value	HEX value	Description
0	253	FD	Disk is double-sided, double density, 9 tracks per sector.
1	4094	FFE	Entry unused and not available.
2	3	3	File's next cluster is 3.
3	5	5	File's next cluster is 5.
4	4087	FF7	Cluster is unusable; bad track.
5	6	6	File's next cluster is 6.
6	4095	FFF	Last cluster in a file; end of the file's space allocation chain.
7	0	0	Entry unused; available.

As can be seen from Table 9-2 the cluster numbers are arranged starting from zero up to the total number of clusters available plus one. The entry for cluster zero is reserved to indicate the type of disk that the file system is on. Therefore the first possibly available cluster starts at the cluster 1 point in the FAT. In the example of Table 9-2 the number appearing in this entry is a reserved number FFEH, which indicates that this cluster is not usable but is empty. Similarly FAT entries 4 and 6 have special reserved numbers in them.

To see how the FAT is used consider the situation where one wishes to sequentially read a file. The directory entry for the file contains the start cluster. Assume that this is 2. This start cluster is used to index into the FAT. Entry 2 contains the number 3 in this particular case. This value is interpreted as the next cluster in the allocation chain for the file. This value can then be used as an index into the FAT to locate the next cluster. In this case this is cluster 5. Again use this as

an index into the FAT. The fifth entry contains 6, indicating that the 6th cluster is the next cluster in the file allocation chain. Finally the 6th entry in to FAT contains FFF, which indicates that there are no more clusters in the file allocation chain.

The other question that naturally arises is “how is the cluster number mapped to the disk sectors?”. With the MS-DOS system this is very simple. The clusters are allocated from the beginning of the disk and increment sequentially. Therefore cluster one would cover sectors 0 to the number of sectors in a cluster. Cluster 1 would continue for the next set of clusters, and so on. The situation gets a little more complicated with multi-platter multi-head disks, but the principle is the same. Therefore, the particular track and sector on the disk can be located by taking the cluster number-1, multiplying by the size of the cluster in sectors, and then dividing by the number of sectors on a track. The quotient is the track number, and the remainder is the sector on the track.

MS-DOS file system naturally supports the concept of the record. This occurs as a natural by-product of the FAT based file system. In an MS-DOS record based system call the number of the record that one wishes to access and the length of the records in the file are passed during the call. Using the record number and the record length the number of bytes to the required record can be calculated. By the dividing this by the sector length the number of sectors can be calculated. The number of clusters can be calculated by dividing this number by the number of sectors in a cluster. The trail of clusters from the starting cluster can then be followed through the FAT structure until we get to the number of clusters calculated. This will then allow the appropriate record to be located. Therefore, using this mechanism the required record is located and can be read or written to without reading the whole file to get to that point.

Remark: *It should be noted that the underlying file system is still a sequential byte stream. For example, if one wishes to insert a new record in the middle of the file it is still difficult to do in this file system because the file is still sequential. The FAT simply allows one to index quickly to a particular point in the sequential file.*

A directory entry in the MS-DOS system has the format shown in Figure 9-11. The directory entry is 32 bytes long and contains the file name and the first cluster number amongst other things. The initial cluster number provides the entry point into the FAT to track the file clusters.

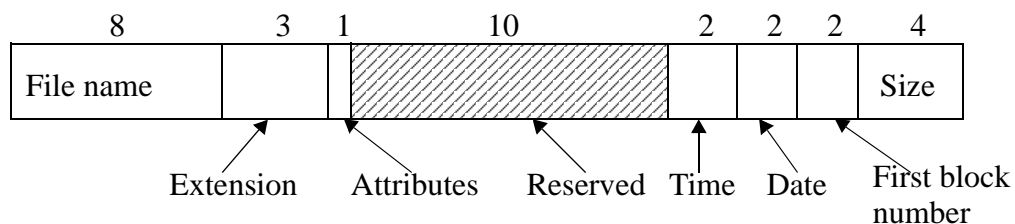


Figure 9-11: A MS-DOS directory entry

FAT32 File System

Microsoft, with the introduction of the later versions of Windows'95 and more recently with Windows'98 and 2000, introduced an enhancement to the old FAT16 file system. This new system was denoted as FAT32. The main enhancement was to increase the indexing into the FAT to a four byte value, allowing up to 2^{32} possible entries. This allows the size of the clusters to be reduced on the large disk drives that are prevalent today. For example, one can have 4K clusters for drives up to 8GB. The file system will handle drives up to 2 terabytes in size.

In addition to the ability to use large drives more efficiently, FAT32 also has the ability to relo-

cate the root directly and use the backup copy of the FAT instead of the default copy. The boot record on FAT32 drives has been expanded to include a backup of critical disk data structures, making the file system less susceptible to single point failures compared to FAT16.

UNIX file allocation

Each file in the UNIX operating system has associated with it a little table (stored on the disk) called an **i-node**. The structure of the i-node is shown in Figure 9-12. This structure contains lists of disk block numbers as well as things such as accounting and protection information. Let us consider the 10 disk block numbers and the indirect pointers in the i-node. For files up to 10 disk blocks the blocks can be stored in the i-node structure.

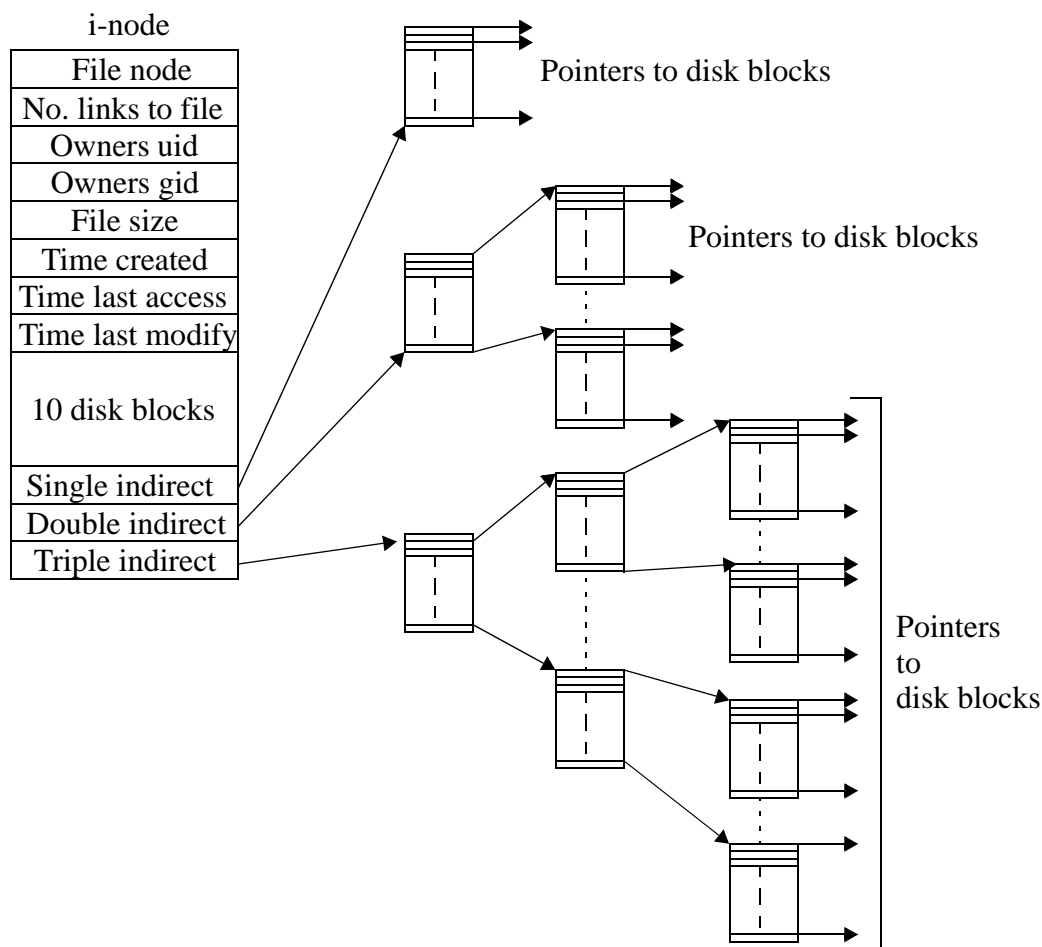


Figure 9-12: Structure of the UNIX i-node

When a file grows above 10 disk blocks, then the single indirect pointer is used. This pointer points to another disk block that contains block numbers. Therefore if the block size is 1 Kilo-bytes, and each block number is 32 bits, then a single disk block can contain 256 block numbers. If the file is bigger than 266 blocks, then the double indirect pointer becomes active. This pointer points to a disk block containing 256 pointers that in turn point to 256 disk blocks containing file block numbers. Therefore $266 + 256^2 = 65,802$ blocks are the maximum file length. If the file grows even longer then the triple indirect pointer comes into play. This pointer points to a block containing 256 double indirect pointers. Therefore the maximum file size becomes blocks, or 16 Gigabytes (since each block is 1Kbytes).

Notice that the UNIX file system only uses the indirect pointers if they are needed. In any case then is a maximum of three disk accesses to locate the disk address for any block for even the largest file.

Remark: The use of *i-nodes* associated with each file means that only the *i-nodes* for open files need to be kept in the memory. In the FAT system the whole of the FAT needs to be kept in RAM. For files as large as those that can be handled by the UNIX system the size of the FAT would be prohibitively large.

The directory structure of for the UNIX system is extremely simple. Figure 9-13 shows the structure of a directory entry. As can be seen it only contains a file name and an *i-node* number. The directories in UNIX are all files and can contain an arbitrary number of these directory entries.

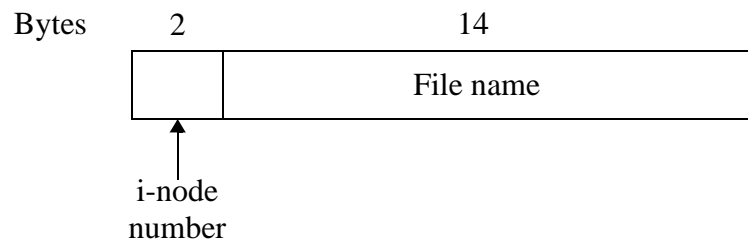


Figure 9-13: Structure of a UNIX directory entry.

Let's consider what happens when a file is opened. The following discussion is with reference to Figure 9-14. Assume that we wish to access a file with the following path:

`/usr/staff/test.txt.`

The process starts with the system looking up the *i-node* for the root directory. This file is located at a special place on the disk. This *i-node* is searched for the name *usr*. From the directory entry for *usr* in the root the system is able to locate the *i-node* for the directory file for */usr*. The *i-node* then contains the block number for the directory file for *usr*. This directory file then contains a number of subdirectory entries. These are searched until the *i-node* number for the directory file for *staff* is found. This *i-node* contains the block number for the directory file for *staff*. This is searched until the file name *test.txt* is found, and then the associated *i-node* number gives the *i-node* for the file. This *i-node* then contains the block numbers that form the file.

Remark: The UNIX file system is still a sequence of bytes. It does allow the files to be accessed as records (similar to MS-DOS), but the records are simply a faster way to get access into a section of the sequential byte stream. Inserting of new records into the middle of a file is still problematic. Additional logical structure of the file system is the responsibility of applications.

Miscellaneous File System Issues

Links

Links are designed to allow several users to share files. In order to allow sharing it is often convenient to have a shared file appear simultaneously in different directories, belonging to different users.

There are two main ways that this has been achieved. In the UNIX system linking is easily achieved because of the use of *i-nodes*. Consider that directory 'A' and directory 'B' are to con-

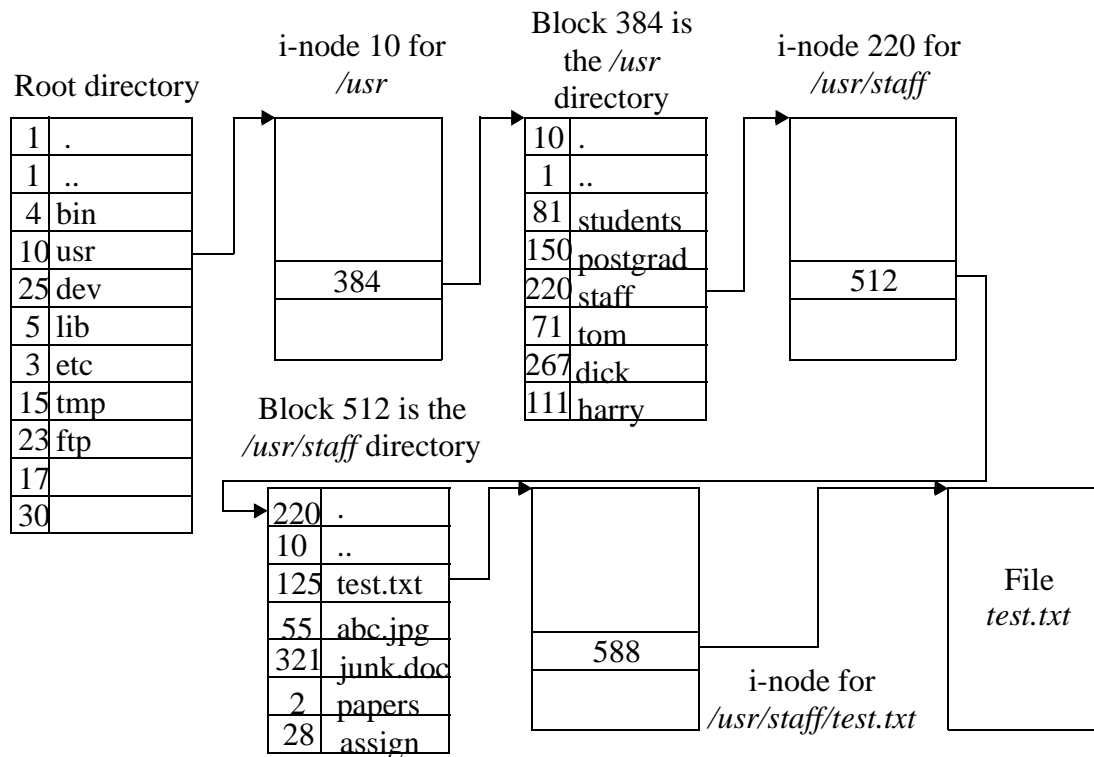


Figure 9-14: Locating a file in the UNIX file system.

tain a shared file. The directory structure for directory 'A' contains the file name and an i-node number. Similarly the directory structure for directory 'B' contains the same file name the same i-node number. Therefore both directories 'A' and 'B' have access to the file through the common i-node structure.

The other way of implementing a link is to create a special file called a LINK file. This new file type simply contains the path to the shared file. Therefore if the this file is accessed the operating system sees that it is the special file type LINK, and then interprets the information in the file as a path to another file. This form of link is known as a symbolic link. This form of the link has the advantage that files can be linked on different machine by including the network path as part of the path name.

Both approaches can have problems. In the case of the i-node linking, if the owner of the file deletes the file then the i-node is cleared. Therefore the i-node entry in the LINK will then point to an invalid i-node. Alternately the i-node could have been allocated to another file, and the link will now point to another file. This is solved by having a link count in the i-node. When a file is linked to the i-node the count is incremented. Whilst this entry is greater than zero the i-node cannot be deleted or reallocated. Therefore even if the owner of the file has deleted the directory entry to the file, the i-node will remain until the link has been deleted.

The above problem does not occur with symbolic links, since if the owner removes the file then the path stored in the link will not be able to be resolved. This will not cause any major problems. However, symbolic links have a problem with efficiency, since they are a file containing a path. Consequently the path has to be parsed and followed until the file is found.

File System Consistency

If the system crashes whilst it is in the middle of writing out blocks for a file then it is possible to end up with file system inconsistency. Most operating systems have software to find and repair these inconsistencies. Most of these work by setting up a table for all blocks on the disk,

each entry containing two fields – one for used blocks and one for free blocks. The software then searches through all the files on the disk, incrementing the used table entries for all the different blocks used by the files. A similar thing is then carried out for all the free blocks. If the file system is consistent, then there should be either a 1 in the table entry for the blocks in use, or a 1 in the free block table entry, but not both. Consider Figure 9-15. Notice that a 1 only

0	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7	8
1	0	1	1	1	0	0	1	0	Blocks in use	1	0	1	1	2	0	0	1	0
0	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7	8
0	1	0	0	0	1	1	0	1	Free blocks	0	0	0	0	0	1	1	0	1

(a) (b)

Figure 9-15: File system consistency check results: (a) consistent file system;
(b) inconsistent file system

appears in the blocks used table or the blocks free table if the file system is consistent. Consider Figure 9-15 (b) which is an example if inconsistency occurs. For example block 1 does not have an entry in either table. Therefore it is neither free nor used. This is a lost block. These do not do any damage to the system, but they do result in lost disk space. To fix this problem the block would be assigned to the free list.

The other inconsistent case is more serious. Consider block 4. The blocks used entry for this contains the number 2 meaning that two files (or the same file for that matter) have used the same block twice. This means that two different files could write to the same block resulting in corruption. If one of the files is deleted then the block will be put onto the free list, and the result is that the block will appear on the free and used lists at the same time. In this situation the consistency checker will allocate a free block, copy the offending block into it, and then allocate this block to one of the files that is using it. In this way the double allocated block will appear in only one file, and the newly allocated block with the copy of the data will belong to the other file. One or both of the cross-linked files will be garbled by this process, but at least the file system is now consistent.

The other type of checking that occurs is directory checking. In a UNIX system this check is carried out by traversing through the directory tree, and for each i-node there is a counter which is incremented for each file that uses the i-node. The number in this table should equal the number in the link field of the i-node. Clearly two different errors can occur; the link count can be too high or too low. If it is too high then the error can be fixed by setting the link count to the correct value. If the count is low, then when the link count goes to zero there will be still files using the i-node even though it can now be reallocated. This is potentially disastrous if not fixed. The fix is simple; set the link count to the correct value found by the consistency check software.

Real-time issues

One important property of a file system for real-time systems is that it be interruptible. If it wasn't interruptible then critical timing may be upset. In most cases the majority of the software in the file system is interruptible. However, it is important that the low level device dependent software also be interruptible. To understand how problems can occur here consider the following: the software is writing a sector of data to the disk using software. This operation cannot be interrupted without corrupting the disk data, therefore interrupts would have to be

disabled for the duration of the sector write. Such delays will cause jitter in a real-time operating system. The consequences of this jitter is implementation dependent. Such problems are usually overcome by appropriate hardware support. For example, data may be written to the disk using Direct Memory Access (DMA), which does not involve interaction with the CPU. On IBM-PC's and compatibles data is transferred to the hard disk using a string move instruction. This instruction is interruptible, but no corruption occurs because the IDE hard disk controller buffers a complete sector before writing to the disk.

The other issue that is of relevance when considering real-time systems is the average worst case data rate onto the disk. This is particularly relevant when real-time data logging is occurring. Clearly if the rate at which data is being sampled is on average greater than the rate at which it is written to the disk then there is going to be data overflow, with consequent lost data. However if the disk data rate is \geq the sampling rate then a buffering solution will allow the data to be stored without loss in real-time.

At any instant of time a multiprocessing computer is running a number of tasks, each with its own address space. It would be too expensive to give each task a complete machine address space to work with, therefore a technique is required so that the tasks can share a smaller amount of physical memory. One way of achieving this, *virtual memory*, divides the memory up in sections and allocates them to different tasks. Inherent in such an approach is the concept of *protection* so that one task can not access the block of memory allocated to another task.

Sharing is not the original reason for the development of virtual memory. In former days if a program became too large to fit into the physical memory of a computer system then the programmer had to divide his program into a set of mutually exclusive pieces known as overlays. When a section of code for an overlay was required, and if that overlay was not in RAM then the overlay was loaded from disk (the code to do this was put into the code stream by the compiler). It was the programmers responsibility to carry out all the memory management. As one can imagine this could lead to complex and error prone programs, especially when data had to be overlaid into RAM. Virtual Memory relieved the programmer from all these memory allocation worries by automatically and transparently loading sections of executable code or data from the disk into main memory as required. Therefore the secondary storage device could be viewed as *virtually* being part of the RAM of the machine.

The objectives of memory management can be summarised as:

- (i) **Relocation:-** It is not possible for a programmer to know in advance where a task will be loaded into the memory of the system (since he/she does not know what tasks will be running in the system at load time). Therefore the program has to either have its addresses modified at load time to run at the correct location (a software relocation solution) or there must be hardware support for relocation. The memory management system provides this hardware support.
- (ii) **Protection:-** As indicated above protection of one task from another is almost inherent in the concept of virtual memory. Compile time checks on memory accesses are inadequate to ensure the security required, since dynamic calculation of addresses bypasses any of these checks. The memory management system provides hardware which checks every memory access by a task to ensure that it is within the area of memory allocated to that task. At first it may seem that only write accesses need to be checked, and this is the case if corruption is the only issue. However if one wants to be sure that a task is executing its own code then read accesses should also be checked. Privacy considerations would also require this.
- (iii) **Sharing:-** There are occasions where two tasks should be allowed to access the same area of memory. For example it is desirable to allow reentrant execution of the same code in order to save memory. In other cases, tasks are required to communicate with each other. The memory management system should allow controlled access to shared

memory without compromising the essential protection.

- (iv) **Logical organization:-**Most computers have a flat linear address space with memory numbered sequentially from zero to an upper limit. This organization does not mirror the way programs are actually written - for example programs are usually divided into procedures and modules which imply that certain pieces of data and code are modifiable, readable and executable as seen externally from the procedure/module. If such logical divisions are reflected in a corresponding *segmentation* of the address space then several advantages are obtained:- (a) segments can be coded independently and intersegment references can be filled in by the system at runtime.; (b) With little overhead it is possible to apply protection at the segment level - e.g. segment is read-only, executable etc.; (c) It is possible to develop mechanisms whereby segments can be shared amongst tasks.
- (v) **Physical organization:-**We have already touched on this issue. Due to the high cost of large amounts of memory most computer systems employ a two level memory system - i.e. high cost but fast random access memory (RAM), and lower cost large capacity but slow access disk storage. As discussed previously one of the prime motivations for the development of the virtual memory concept was to offer a seamless blending of the primary and secondary storage, so that as far as the programmer was concerned the two appeared to be one large virtual RAM.

Virtual addressing - basic concepts.

The concept of the *virtual address* and the *physical address* has to be understood in order to understand virtual addressing. The virtual address is the address used by the assembly language programmer or the compiler. In the absence of a virtual to physical address mapping then the physical and the virtual address are the same. The virtual to physical address mapping is carried out by a Memory Management Unit (MMU) which sits between the CPU and the memory as shown in the Figure 10-1 below.

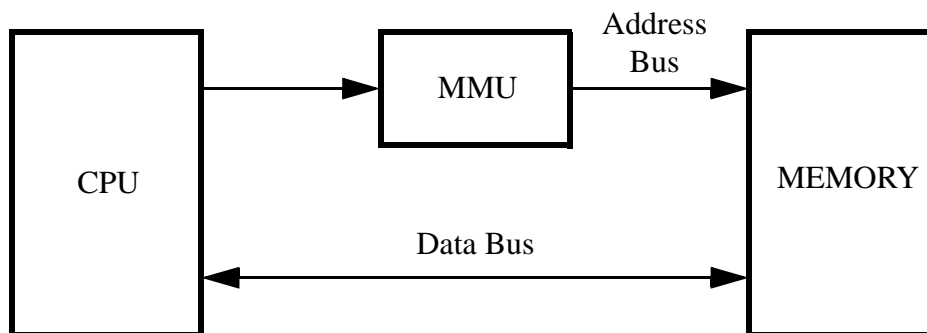


Figure 10-1: Virtual to Physical Address Mapping

The MMU translates every virtual address which comes from the CPU into a physical address which is applied to the memory. This translation occurs every time the processor fetches or writes data from the memory, therefore it can be appreciated that the translation process must be done efficiently if the MMU is not to impact on the performance of the machine. An interesting observation is that because the MMU is usually located next to the CPU (and in many cases these days it is actually on the same physical chip) then DMA devices (such as disk con-

trollers) which use the address bus must use physical addresses and not virtual addresses.

Virtual memory models

There are two main types of virtual memory implementations in common use - *segmentation* and *paging*. They both provide relocation, sharing and physical organisation, but they differ in philosophy and implementation.

In the *segmentation* approach to memory management, a program is made up of a number of different segments, the divisions being made based on the logical design of the program. For example some segments may be read only, others execute only etc. In tasks share memory then the shared area can also be made a segment. If the programmer is writing in assembly language then the divisions into segments is at his discretion. In the case of a high level language the decisions are usually made by the compiler. The main motivation for segmentation is that the protection can be tailored to the logical organisation of the code.

The *paging* approach breaks up the memory into equal size blocks or pages. These divisions are not based on any logical view of the code as in the segmentation approach. Because the *pages* are of equal size (defined by the hardware of the system) then *swapping* (which will be discussed later) is relatively simple to implement. The *page* is not a natural unit of protection since the page boundaries do not correlate with user defined attributes such as the boundary between procedures or modules.

Segmentation - basic principles

The simplest form of segmentation is the base - limit relocation strategy. The basic concept is illustrated in the diagram below. The MMU hardware performs the following calculation:

```
if VA > LIMIT then
    illegal address trap;
else
    PA := BASE + VA;
```

where **VA** is the virtual address (or the address used by the programmer) and **PA** is the physical memory address. The **BASE** and **LIMIT** values are kept in two internal registers which can only be altered by the operating system (see Figure 10-2). Upon a task switch these registers are altered so allow the code for the newly switched in task to be executed.

This scheme implements what is known as a “single segment name space” – that is there is only one segment per task. Whilst it allows relocation and protection of tasks, it is worth noting that the address space mapped by this scheme is linear and its size is less than or equal to the size of the memory space. Thus objectives (iii), (iv) and (v) above are not achieved.

A modification of this scheme is to provide two pairs of registers rather than one. One pair is used to demarcate the memory space occupied by reentrant code, and the values in them are the same for all tasks using the code. The other pair is used to demarcate the code and data areas for each particular task and consequently have different values for each task. The older DEC System-10 is an example of a machine which employed this technique.

To turn the base-relocation scheme into a more elaborate *segmentation* scheme every virtual address must be interpreted as having two components: a *segment selector* and an *offset* within the segment. This is illustrated in the Figure 10-3.

In the segmented memory case the segment selector is not directly used as a base but instead is an index into a table which contains the base and the limit for a particular segment number. The number of bits in the segment selector section of the virtual address determines how many

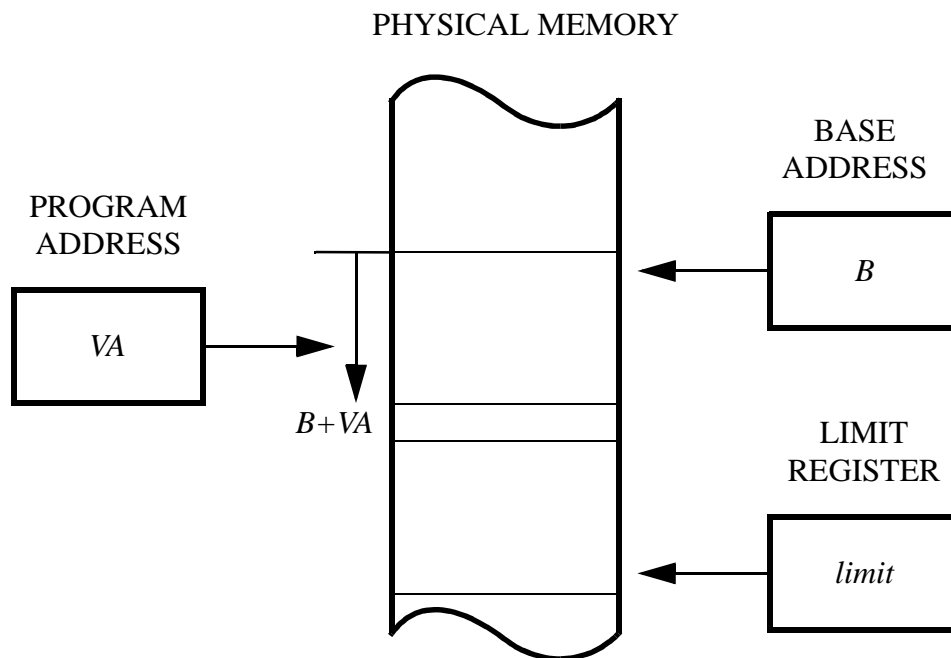


Figure 10-2: Base-limit relocation scheme



Figure 10-3: Format of segmented virtual address

segments a single task can have. Similarly the number of bits in the offset will determine the maximum size of a segment. The address translation for this scheme is shown in the Figure 10-4.

The entries in the segment table are often called segment descriptors, and the table is known as the *segment descriptor table*. The bits blanked out in the segment descriptor are usually reserved for protection purposes. For example they may contain codes saying that the currently addressed segment is read-only. Later on we shall look at some specific examples of segmented addressing.

Paging - basic principles

In the base - limit address scheme described above the size of the address space is necessarily less than or equal to that of the memory space. In order to eradicate the distinction between the main (RAM) and secondary (disk) storage as far as the programmer is concerned the concept of *paging* was developed. The *one-level store* concept can be traced back to the Atlas computer at Manchester University around 1960, and since then has had a profound influence on compu-

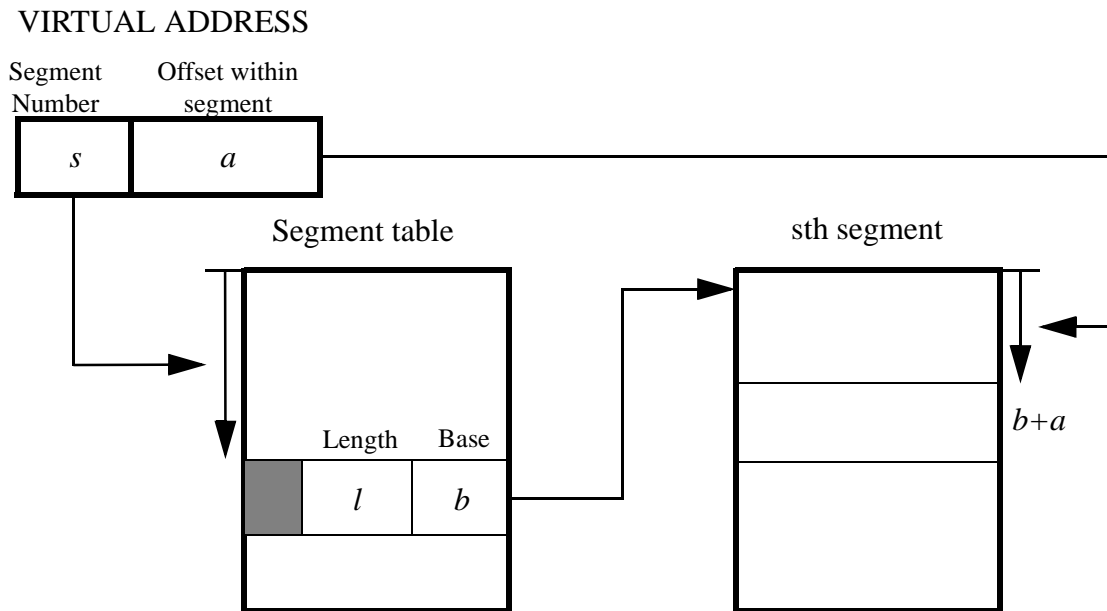


Figure 10-4: Addressing for full segmentation

ter design.

In a *paged* system the virtual address space is divided into *pages* of equal size (e.g. on the Atlas 512 bytes), and the main memory is similarly divided into *page frames* of the same size. The virtual address has the format illustrated below. The high order n bits of the address are interpreted as the page number and the low order address bits are the byte within the page. At first glance the form of the page virtual address is the same as that of the segmented virtual address. However the paged virtual address is treated very much as a linear address - i.e. it is essentially a single binary number. In the address arithmetic if a carry is generated out of the *address within page* section of the virtual address it is propagated to the *page number* part of the virtual address. In other words the addresses simply run off one page and into the next. This is very different to segmentation where running off the end of a segment is considered to be an addressing error. This difference is obviously due to the primary motivation behind each virtual addressing scheme - segmentation being motivated by intratask and intertask protection considerations, whereas paging is motivated by the *one-level storage* concept.

The size of the pages is usually defined by the hardware of the system. For example, the DEC VAX has a program address of 32 bits (i.e. its virtual address space is 4 Gigabytes). The page size is 512 bytes - i.e. the least significant 9 bits of the address represent the address within a page. That means that the top 23 bits represent the page numbers (i.e. there are 2^{23} pages in the virtual memory). On the other hand a typical machine may only have 2^{16} pages (32 Megabytes) of physical memory. The resolution of this mismatch between the virtual pages and the number of physical pages is elegantly solved by the paging mechanism as discussed below.

Figure 10-5 is a block diagram showing the basic page address mechanism.

The address from page and address to physical memory locations is made by means of a *page table*, the p th entry of which contains the location p' of the page frame containing page number p . The address within the page is then added to p' to give the actual physical address of the required virtual address a . The various components of the paged address are calculated as follows:

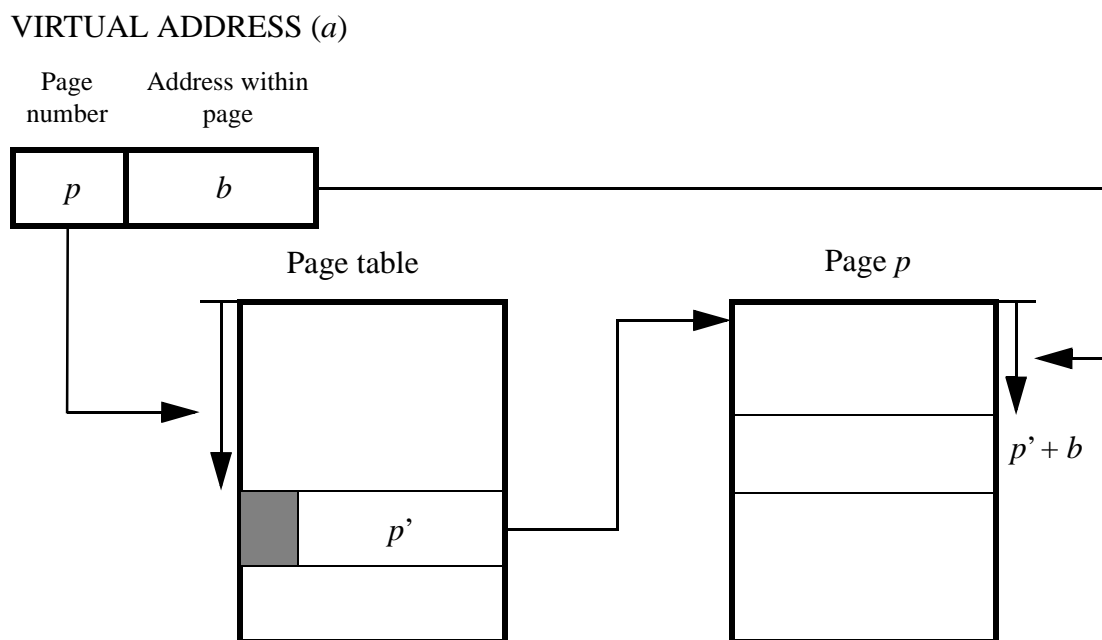


Figure 10-5: Address mapping for paging

$$p = \text{integral part of } \frac{a}{Z}$$

$$b = \text{remainder part of } \frac{a}{Z}$$

where a is the virtual address, p is the page number, and Z is the page size.

As mentioned previously the number of virtual pages is often far larger than the actual number of physical pages. Therefore it is quite possible that the mapping operation shown above will attempt to access a page which is not in the physical memory. In this case the corresponding entry in the page table will be empty and a *page fault* will occur. This results in an interrupt to the operating system so that it can fetch the required page from the secondary storage. This process is totally transparent to the user (except for the side effect that the task requiring the page will be blocked until the required page is fetched). In many implementations the location of the page on the secondary storage device is stored in the page table itself. A bit in the page table entry is needed to indicate whether or not the page table is present in main memory or not, and consequently whether the page table entry is to be interpreted as a page frame address or a secondary storage location.

Some implementation issues

The above discussion has outlined the basic principles of segmentation and paging. Let us consider segmentation for a moment. The segment tables are normally kept in a *segment descriptor table* in the main memory of the machine. If the processor had to access the main memory in order to get the *segment descriptor table* entry on each memory access then the address mapping would impose an intolerable performance penalty on the processor. In order to over-

come this problem all implementations of segmented virtual memory employ a cache of some description to store the most recently accessed segment descriptors. This cache is present in the MMU (or in the case of processors with an on board MMU on the processor chip itself).

Let us consider the Intel 80286 processor as an example. This processor implements a very sophisticated segmented memory protection system. The MMU is on board the same chip as the processor. The processor has 4 registers whose function is to store the segment numbers of the 4 currently active segments. The descriptor tables themselves are located in memory, their base being indicated by three registers (one each for the global descriptor table, the local descriptor table and the interrupt descriptor table). When the segment registers are loaded the descriptor indexed to by the registers contents is automatically loaded into an on board descriptor cache (known as an *address translation cache*). This cache has the capacity to hold 4 segment descriptors - one for each of the segment registers. Since segment numbers are rarely changed, the caching is very effective at lowering the address translation time.

As in the case of segmentation, page tables are also kept in main memory. For the same performance reasons a cache is employed to carry out the page translation. This cache is known as a *translation lookaside buffer*. For many systems with a large virtual memory and small pages the size of this cache would be prohibitively large (e.g. the VAX example cited earlier). A common technique adopted to solve this problem involves adding to the machine an *associative store* which contains a small set of *page address registers* (PARs), each of which contains the page number of an active page (i.e. a page which is currently being used). Since the number of pages which are being used is usually much smaller than the total number of pages in the virtual address space then the size of the associative register space is much smaller than the number of pages in the virtual memory.

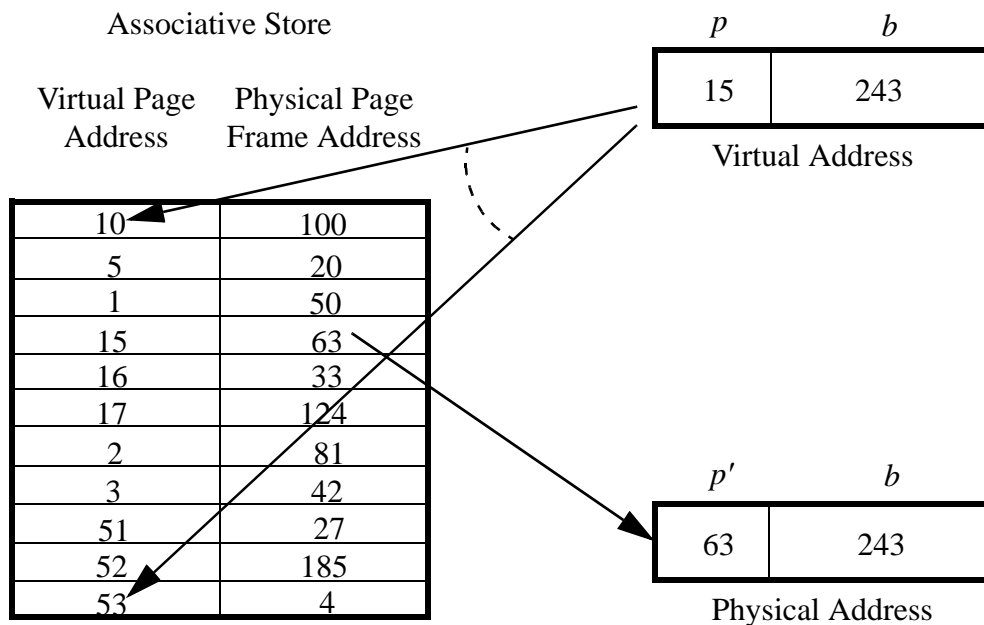


Figure 10-6: Page mapping via associative store

Figure 10-6 indicates how page mapping occurs using an associative store. An associative store has the property that all locations are searched simultaneously for the contents of the *key*. In this particular application the key is the virtual page number. The physical *page frame* is the information retrieved from the associative store. In a match is not found within the associative

store then the main memory has to be accessed. If the required page table entry indicates that the page is not in the main memory then the page must be fetched from the secondary storage. The flow of control is shown in the Figure 10-7. Note that in this scheme that adjacent pages in the virtual memory do not have to map to adjacent pages in the physical memory.

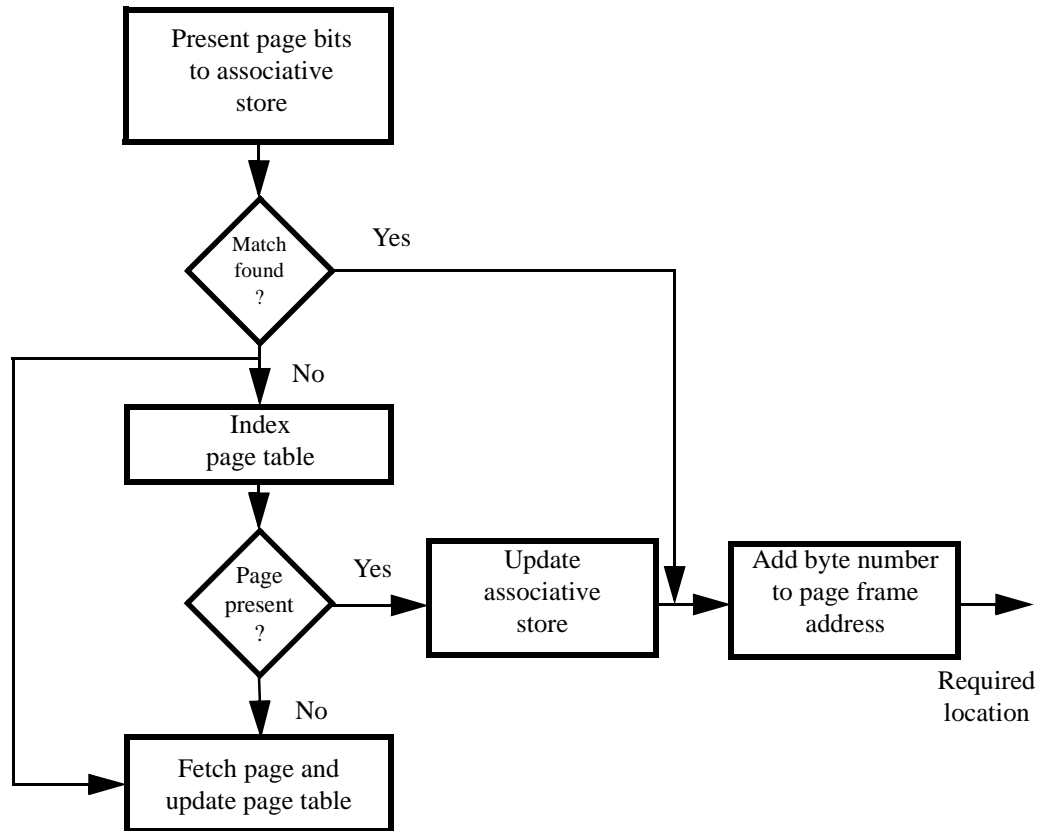


Figure 10-7: Page algorithm with associative store.

One problem not illustrated in these figures is that of distinguishing in the associative store between the pages belonging to different tasks. One solution is to extend the PARs to include task number as part of the page number. A simpler solution is to extend the PAR by one bit which is used to indicate whether the page belongs to the current task.

One other point to note is that page tables may take up a considerable amount of memory. For example, the DEC VAX has 8,388,608 pages of 512 bytes each. Therefore the complete set of page tables would take up 32 megabytes at 4 bytes per page table entry. The problem is tackled by placing the page tables in a special part of the memory (the *system area*) and then paging the page tables themselves.

Some processors (e.g. the Intel 80386) use the *two-level store* technique of storing the page tables. This idea is based on the fact that most tasks only use a small number of the total pages available in the virtual address space. The top few bits of the address are used to index into a *page table directory* which in turn points to a page table. The remaining bits in the page section of the address are then used to index into this page table. Therefore the number of entries in the page table are significant less than without this scheme for the average task. If one had a very large task then it would simply use multiple page tables (as indicated by the page directory bits).

Another approach to reducing the page tables is to reduce them to the actual pages used, rather

than the number in the address space. This is achieved by accessing them via *hashing* rather than indexing. Each time a new page is brought into main memory the page number and corresponding page frame address are entered in the page table at a point determined by a suitable hash function. The same hash function can then be used to locate the page during the addressing operation. The use of the hash function is fundamental to this technique since it means that the page table does not have to be an exhaustive array.

The price paid for reducing the page table size by the use of the hashing idea is increased time for memory accesses. This increased time arises due to two factors:- the computation of the hash function; some references may result in a hashing collision which means that further time is required to resolve the collision. These timing costs can be reduced to acceptable levels by the use of an associative store (as discussed previously) which can reduce page table accesses to 1% of the total accesses, and the implementation of the hashing function in hardware.

The best of both worlds - combined paging and segmentation

The advantages of paging arise from the fixed size blocks used. This allows memory allocation to be both simple and efficient. When memory is allocated to a task any page is as good as any other. The flexibility of not having to map tasks into consecutive pages allows the memory to be almost fully utilized (except for *internal fragmentation* which means that one page per task may not be fully utilized since a task may not take up an integral number of pages)

In a system using segmentation memory allocation is more difficult because the segments come in various sizes. In order to allocate memory for a segment it is necessary to find a piece of contiguous memory large enough to hold the segment. The obvious problem which could arise is that the total amount of free memory is large enough to hold a new segment but no one block has enough contiguous storage for the segment. This problem is known as *fragmentation*.

Segmentation was motivated mainly by high level protection considerations whereas paging was motivated by ease of low level memory management. It therefore makes sense to try and combine the two in some way to get the “the best of both worlds”. There are two reasonable ways to do this – “paging within segments” and “paging after segmentation”.

The “paging within segment” approach is probably the most obvious. Here the virtual address space for a task is broken up into a number of segments, then each segment is divided into fixed-size pages. Therefore the virtual address has the format shown in Figure 10-8. In the address arithmetic the carries are allowed to propagate from the “address within page” section to the page number (as with pure paging) but they are not allowed to propagate from the page section to the segment section as this would constitute an addressing error (trying to access outside the maximum addressing limit for the machine).

The steps taken in the virtual to physical address translation are as follows:

- (i) Each task has a descriptor table in main memory which consists of an array of descriptors, one for each segment of the task. This descriptor table is pointed to by a register contained in the MMU, which must be updated upon every task switch.
- (ii) The segment section of the virtual address is used to index into the current segment descriptor table. The descriptor holds the address and length of the page table for the segment (the length of page table effectively becoming the limit of the segment). The segment descriptor also holds the protection information for the segment.
- (iii) The page number field selects an entry in the page table. This gives the physical starting address of the page. Any page attributes will also be present in the page table.

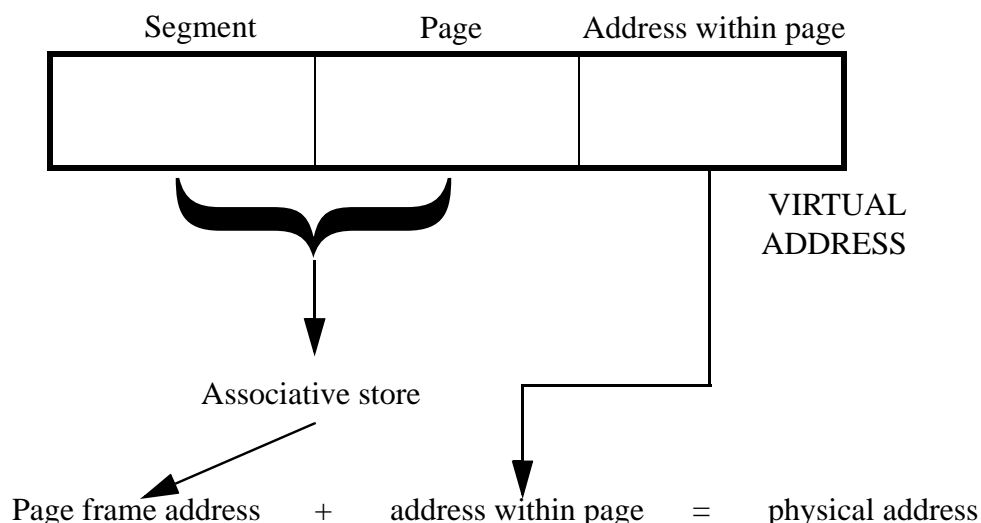


Figure 10-8: Address mapping for paging within segments

- (iv) The “address within page” field is appended to the page frame address obtained from the page table to obtain the final physical address of the memory reference.

In practice an associative cache has to be employed in order to make the above process efficient. The cache is presented with the segment and page number and if a match is found then the page frame address can be found immediately. If a match is not found then the process described above is carried out and the resultant entry is stored into the associative cache.

One subtle problem with the above scheme of paged segmentation is that the address protection can be violated. Generally in the above implementation the segment size is made an integral number of pages. However the size of a logical segment will usually not be an integral number of pages. Therefore it is possible for a task segment to address a location outside the logical segment (i.e. a location in the unused part of the last page of a segment) without generating an addressing error. This problem can be overcome by employing the second of the paged segmentation approaches - “paging after segmentation”.

This alternative approach is taken by the Intel 80386 and above processors. These machines have conceptually independent segmentation and paging units. This allows pure segmentation, pure paging and the combination of the two to be obtained. The virtual address consists of a segment and offset part as described previously. These are translated by the segmentation unit as described in pure segmentation. Any addressing errors would be picked up by this process. Notice that the segment does not have to be an integral number of pages in length. The *linear address* obtained from the segmentation unit can then be considered to have a page part and an address within page part. If paging is enabled then this address is passed to the paging unit which implements pure paging as described before.

The advantages of implementing a paged segmented address space are twofold. Firstly, the entire segment need not be in memory – only those pages currently in use need be accommodated and those that are not can be easily “paged” from secondary storage when required. Secondly, it is not necessary that pages occupy contiguous areas of memory – they can be dispersed through the physical memory wherever they can be found. Against a paged segmented address space is the complexity of the address mapping and the overhead of the paging tables.

The best of both worlds - combined paging and segmentation

As can be deduced from our discussion of paging its main function is to efficiently implement the *one-level store* concept and flexible address mapping for a dynamic environment such as a time shared computer system. Segmentation on the other hand is useful to have even if the above is not important (such as in an embedded real-time system).

A modern microprocessor that embodies many of the protection and memory management features mentioned in Chapter 10 is the Intel 386 series of microprocessors (this includes the 486 and Pentium extensions to the family). The following discussion is not intended as a complete description of the functionality of this processor family, but will instead concentrate on the memory management and protection features of the processor. For more detail on this processor refer to the “Intel 386 DX Microprocessor Programmer’s Reference Manual”, and for detail on how to use the processor for system software refer to the “80386 System Software Writer’s Guide”. Since the 80386 is a superset of the Intel 80286, the “iAPX 286 Operating Systems Writer’s Guide” is also a useful publication.

Before looking at the memory management and protection details of this machine it would be beneficial to have a brief look at the register model of the machine from the application programmer’s viewpoint. Differences from the 8086 will be highlighted.

Application Programmer’s Register Model

The major difference between the 8086 and the 80386 microprocessors is that the 8086 is essentially a 16 bit microprocessor, whereas the 80386 is a 32 bit microprocessor. These differences extend to both the internal register configuration as well as the memory addressing.

The 8086 general purpose registers are 16 bits in size, and with a few instructions two can be connected together to form a 32 bit register. The memory addressing is also largely 16 bit orientated, with the offset register being 16 bits in size (thus limiting the segment size to 64 kilobytes). The segment size limitation has been a major contributing factor to the demise of the original 8086 as a popular microprocessor.

The 80386 was designed to be a superset of the 8086, extending the addressing and internal register model to 32 bits. In addition, the processor offered the memory management and protection features required to implement sophisticated multi-tasking operating systems. It is ironic that these features were essentially left unused for approximately 5 years after this processor was introduced. This was partly due to the inertia produced by the amount of real mode software already written, the lack of protected mode development tools available, and the lack of knowledge in the applications programming community about protected mode operation.

Figure 11-1 below shows the register set available to the application programmer. The same 8 bit and 16 bit registers that were present in the 8086 are still supported, but they have been extended to allow 32 bit values. The control register has also been extended to allow the extra flag bits required for the protected mode operation.

The 80386 microprocessor, similarly to the 8086, uses a segmented memory model. Whereas the 8086 segmented memory model was mainly inspired by the chip density limitations when it was designed, the 80386 segmented structure is motivated by the protection facilities it offers. The segment selector for the 386 is 16 bits as in the 8086, but the offset was increased

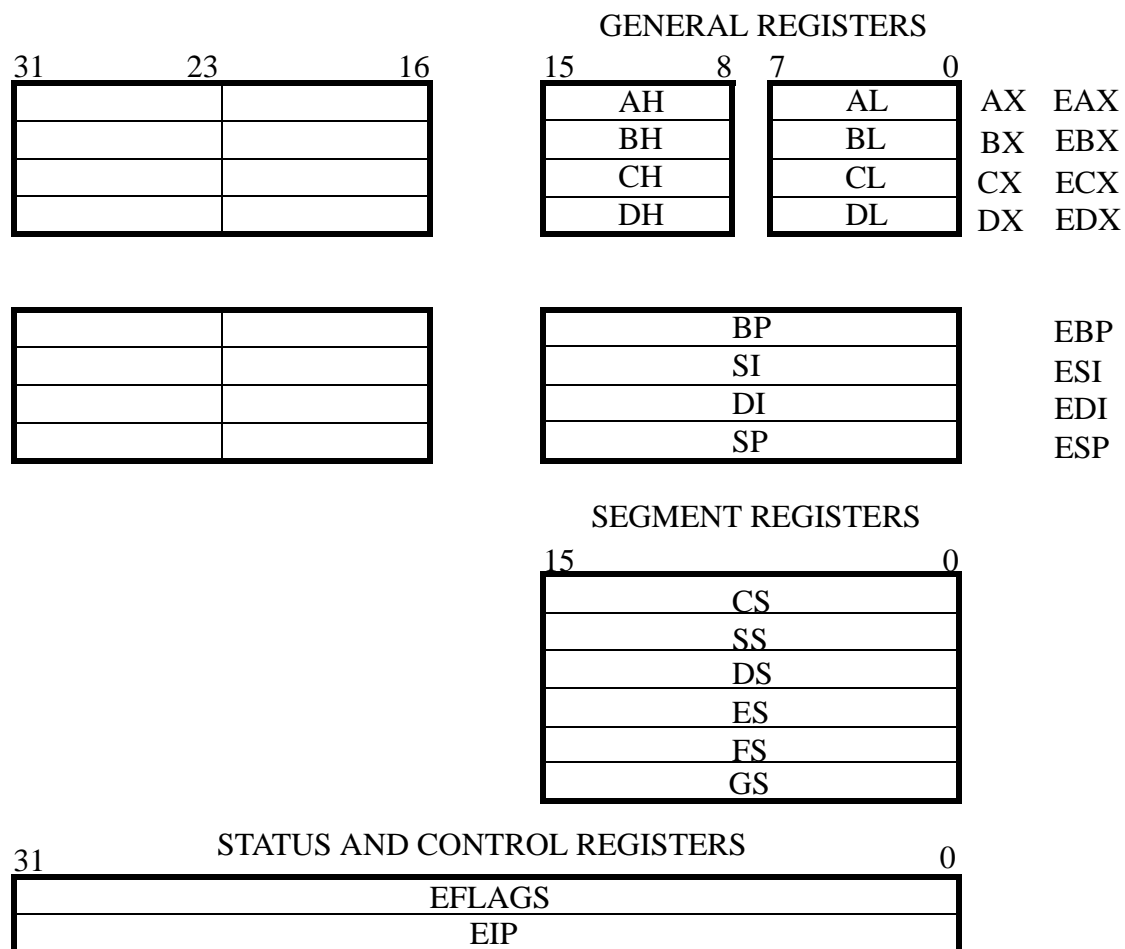


Figure 11-1: Application Register Set for the Intel 80386 Microprocessor.

from 16 bits to 32 bits, thereby allowing a flat segment of 32 bits (which is also the maximum physical memory address space for the processor). Notice from Figure 11-1 that the number of segment registers was increase from 4 to 6 in the 80386, thereby allowing more data areas to be simultaneously accessible without having to change the data segment register values.

System Programmer's Register Model

We will now look at the new features of the 80386 processor that have been specifically designed to support system level programming. These features are usually not accessed by the application programmer; in fact most systems deliberately restrict user access to them.

The registers intended for use by system programmers fall into the following categories:

- EFLAGS Register
- Memory Management Registers
- Control Registers
- Debug Registers
- Test Registers

Let us now briefly examine the function of each of these register groups.

System Flags

These flags are present in the EFLAGS register. This register also contains the normal flags present in the 8086 processor. The system flags control I/O, maskable interrupts, debugging, task switching, and virtual-8086 mode. They are generally not changeable by the application programmer, and in most systems any attempt to do so will result in the generation of an exception. The EFLAGS register is shown in Figure 11-2. The new bit flags in the register

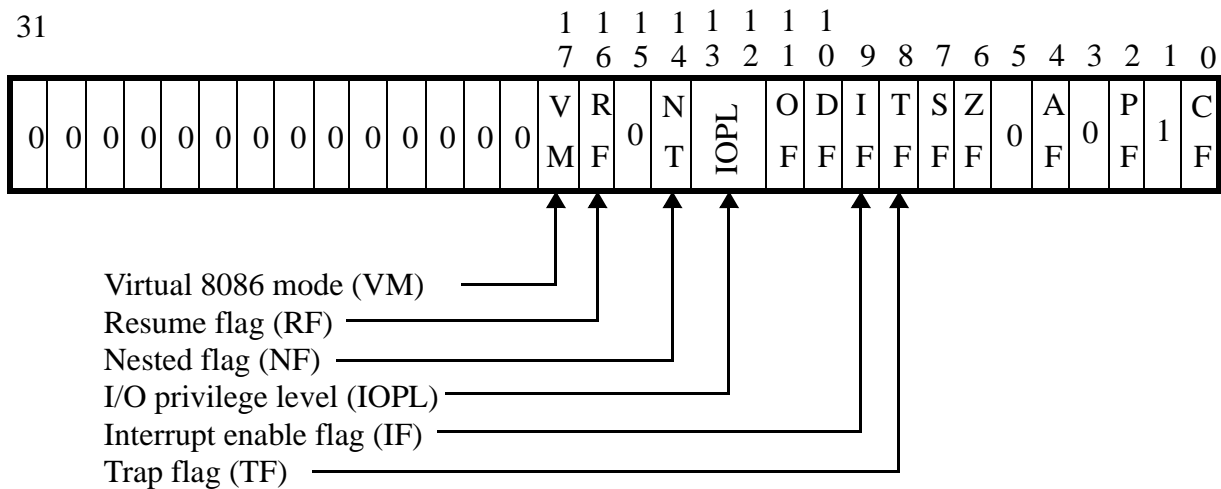


Figure 11-2: System Flags for the 80386 Processor

have the following functions:

VM (Virtual-8086 Mode, bit 17): Setting this flag places the processor in virtual-8086 mode. This mode allows the 80386 to emulate the real mode operation of the 8086 whilst still operating in a protected mode system.

RF (Resume Flag, bit 16): This flag temporarily disables debug exceptions. The main use of this flag is in debuggers, where the debug exceptions have to be disabled so that the debugger doesn't recursively call itself until a stack overflow occurs. It is also used to allow instructions to be restarted after a debug exception without immediately causing another debug exception.

NT (Nested Task, bit 14): This flag controls the chaining of interrupted and called tasks. The flag affects the operation of the IRET instruction.

IOPL (I/O Privilege Level, bits 12 and 13): These flags are used by the protection mechanism to control access to the I/O address space. The privilege level of the code segment currently executing (Current Privilege Level – CPL) and the IOPL determine whether this field can be modified by POPF, POPFD, and IRET instructions.

IF (Interrupt-Enable Flag, bit 9): This flag determines whether the processor will respond to maskable interrupts. It does not affect the nonmaskable interrupts or system traps. The CPL and the IOPL determine whether this field can be modified by the CLI, STI, POPF, POPFD and IRET instructions.

TF (Trap Flag, bit 8): Setting the TF flag puts the processor into single-step mode for debugging. When this flag is set a debug exception is generated after each instruction so that the debugger can take control. If an application sets the TF flag using POPF, POPFD, or IRET instructions a debug exception is generated.

Memory-Management Registers

There are four registers in the CPU that keep track of where the data structures that control the operation of the segmented memory reside. Special instructions have been added to the 80386 instruction set to allow these registers to be loaded and stored. The Global Descriptor Table Register (GDTR) and the Interrupt Descriptor Table Register (IDTR) may be loaded with instructions which get a six-byte block of data from memory. The Local Descriptor Table Register (LDTR) and the Task Register (TR) may be loaded with instructions which take a 16 bit segment selector as an operand. The remaining bytes of these registers are then loaded automatically by the processor from the descriptor referenced by the operand. Figure 11-3 shows the structure of these registers.

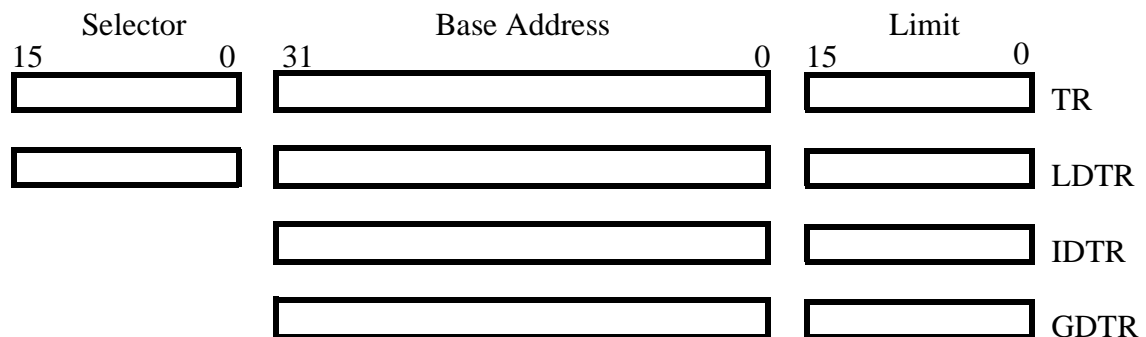


Figure 11-3: Memory Management Registers

A brief description of the function of these registers follows:

GDTR (Global Descriptor Table Register): This register holds the 32 bit base address and the 16 bit segment limit for the global descriptor table (GDT). Note that the limit refers to a limit on the size of the descriptor table, not a limit on the size of the segment being referenced. When a reference is made to data in memory, a segment selector is used to find a segment descriptor in the GDT or the LDT (table referenced is indicated by a flag in the segment selector itself). The segment descriptor contains the base address of a segment. The structure of the descriptor is described later.

LDTR (Local Descriptor Table Register): This register holds a 32 bit base address, a 16 bit segment limit, and a 16 bit segment selector for the local descriptor table in the GDT. The segment containing the LDT has a descriptor in the GDT. It is this descriptor that is being referenced by the selector in the LDTR. The limit and base address values are loaded into the LDTR from the contents of the GDT descriptor. The limit value, as with the GDT limit, refers to the limit on the size of the LDT.

IDTR (Interrupt Descriptor Table Register): This register contains a 32 bit base address and a 16 bit limit for the Interrupt Descriptor Table (IDT). When an interrupt occurs, the interrupt vector is used as an index to get a gate descriptor from the IDT. The gate descriptor contains a far pointer used to start up the interrupt routine.

TR (Task Register): This register holds the 32 bit base address, 16 bit segment limit, and 16 bit segment selector for the task currently being executed. The selector references a task state segment (TSS) descriptor in the GDT. The limit and base address values in this descriptor are copied into the base address and limit components of the TR. The limit refers to the limit on the size of the TSS itself.

Control Registers

Figure 11-4 shows the format of the four control registers present in the CPU. These registers control modes that apply to the general operation of the processor, rather than a specific task. The registers are read or written via MOV instructions from general purpose registers.

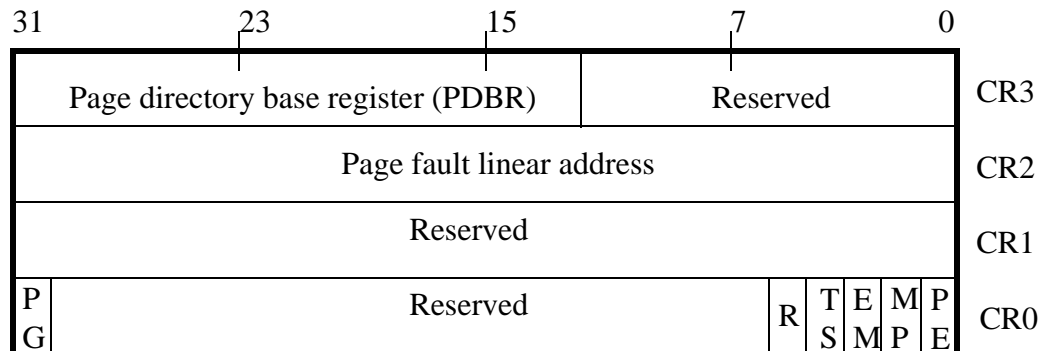


Figure 11-4: Control Registers in the 80386 Microprocessor.

A brief description of the functionality of the bits within the CR0 register follows:

PG (Paging, bit 31): This bit enables paging when set, and disables paging when clear.

R (Reserved, bit 4): This is a reserved bit and should not be altered. In fact none of the reserved locations should be changed when programming the CR's.

TS (Task Switch, bit 3): The processor sets the TS bit every task switch and tests it when interpreting coprocessor instructions.

EM (Emulation, bit 2): When set this bit indicates that the coprocessor instructions are being emulated in software.

MP (Math Present, bit 1): This bit controls the function of the WAIT instruction, which is used to synchronize with a coprocessor.

PE (Protection Enable, bit 0): Setting this enables the protection of segments and pages.

The remaining registers have the following functionality:

CR2: This register contains the 32 bit linear address that led to a page fault.

CR3/PDBR (Page Directory Base Register): When paging is being used this register contains the 20 most significant bits of the address of the first level page table (page directory). The page table must be aligned to a page boundary(4kilobytes), consequently the bottom 12 bits are assumed to be zero and therefore are not stored in the PDBR.

There are other system level registers related to debugging - the debug registers and the test registers. Since these registers are not directly related to memory management or protection they will not be discussed any further.

The general principles of segmentation and paging have been discussed in Chapter 10 and will not be repeated here. This section will highlight the pertinent memory management aspects of the 80386 processor.

The 80386 contains both segmentation and paging hardware. The memory management can be organised to use pure paging, pure segmentation or a combination of segmentation and paging. The page size on the 80386 is 4 kilobytes and the segment size has a limit of 4 Gigabytes.

Segmented Address Models in the 80386

This memory model offers the maximum protection for the software in the system, since each component of the software in the system can operate in its own protected memory space. One of the first decisions that has to be made is the type of segmented space one wants. These segmented models differ in the amount of protection offered.

Flat Model

This is the simplest segmented model. It maps all the segments into the one memory space which is the entire physical address space (i.e. the segment limit is set to 4 Gigabytes). This model essentially removes the segmentation from the architecture seen by both the system designer and the application programmer. The usual situation when this segmentation model is used is when segmentation is to be disabled and paging enabled. This memory model is used for UNIX operating systems.

At least two segment descriptors must be created for a flat model, one for code references and one for data references. The segment selector for the stack may be mapped to the data segment descriptor.

Protected Flat Model

This is very similar to the flat model described above, except that the segment limits are set to only cover the available memory present. Therefore, a general protection fault will be generated on any attempt to access nonexistent memory.

Multi-Segment Model

The most sophisticated model is the multi-segment model. The full capabilities of the segmentation mechanism are used. Each program is given its own table of segment descriptors, and its own segments. The segments can be completely private to the program, or they can be shared with specific other programs. Access between programs and particular segments can be individually controlled.

As mentioned in “Application Programmer’s Register Model” on page 11-1, the 80386 processor has six segments available for immediate use (the selectors are contained in the six segment registers). Other segments are accessed by loading their segment selectors into the segment registers. It is the responsibility of the operating system software to set up the segment descriptor values so that the segments have the ranges in the physical memory. It is possible to set the segments up so that the physical memory areas are overlapping – for example if a ROM contains code and data then the code segment and data segment registers would be set up with the address ranges overlapping.

Segment Translation in the 80386

The translation of a logical address to a physical address for the 80386 closely follows the classical segmented model translation as described in Chapter 10. The segment selector references into either the GDT or the LDT depending on the setting of a Table Indicator (TI) bit (see

Figure 11-5) in the segment selector itself (see later for a detailed description of the segment selector). The descriptor contained in the GDT or the LDT contains the actual physical base address of the segment and the limit on the offset within the segment. This limit is checked against the offset section of the logical address prior to addition to the physical base address.

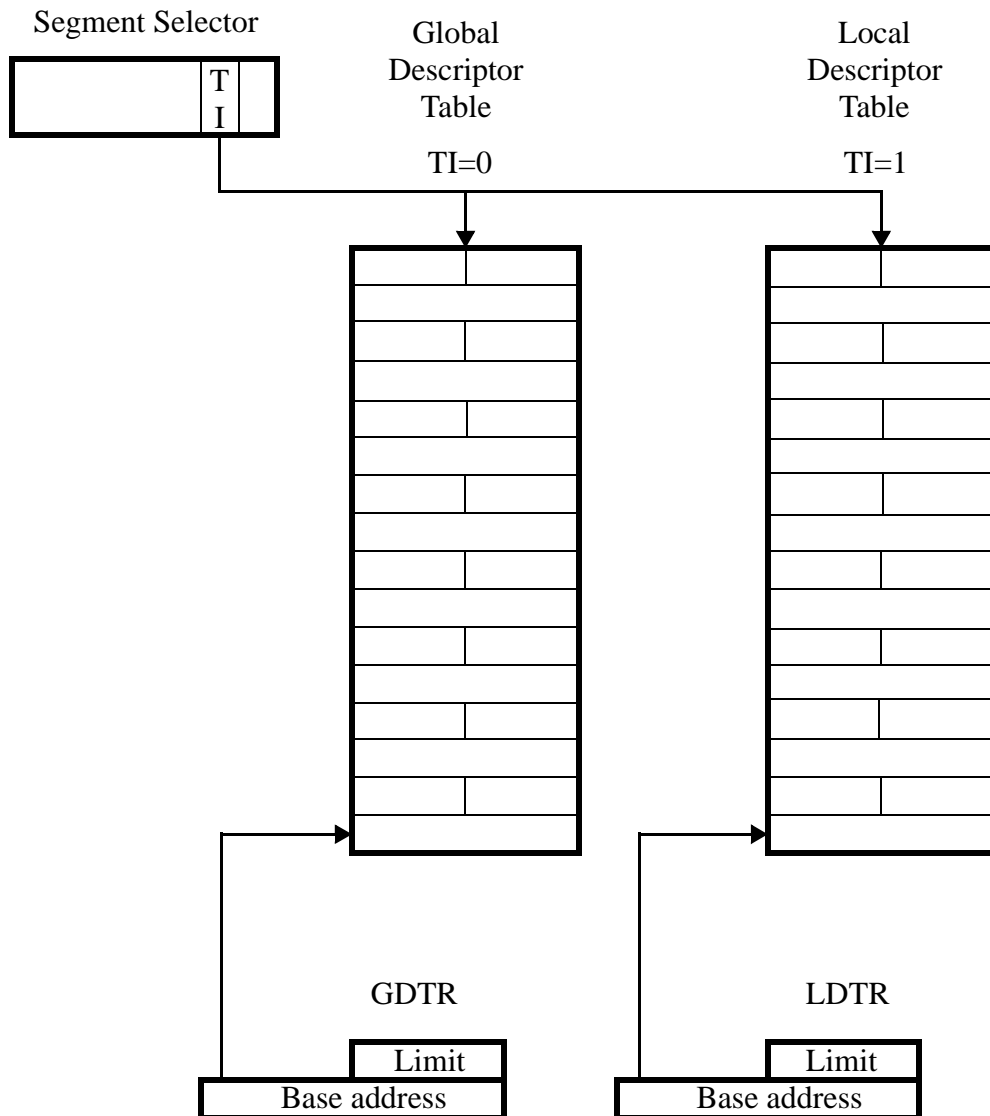


Figure 11-5: Location of the segment descriptors via the GDTR and the LDTR.

Segment Selectors

The segment registers themselves have a “visible part” and an “invisible part”. The visible part can be accessed via MOV instructions, and the invisible part is maintained by the processor. The invisible part essentially acts as an onboard descriptor cache. When a new selector value is loaded into a segment register the descriptor referenced by the selector is loaded into this cache by the processor. The value is then onboard for all future accesses to the segment, thus saving any extra memory cycles required to access segments. The value is held in the cache until the

particular segment register is reloaded with another selector value.

The segment selector contains a 13 bit index into one of the descriptor tables (meaning that there can be 8192 segments for a particular descriptor table). The index is scaled by eight (the number of bytes in a segment descriptor) and then added to the 32 bit base address contained in the GDTR or the LDTR, depending on the TI bit in the selector. This then locates the segment descriptor.

Before looking at the structure of the segment descriptors, we shall define the components of the segment selector itself. As mentioned above, one component is the index into the descriptor table. In addition there are three bits in the least significant bits of the selector. Figure 11-6 shows the layout of the selector.



TI Table indicator (0=GDT, 1=LDT)
RPL Requested Privilege Level
 (00=Most privileged, 11=least)

Figure 11-6: Block diagram of the segment selector.

The Requested Privilege Level (RPL) field is used to enforce protection in relation to memory access. For example, if this field contains a privilege level greater than the privilege level of the program that is to use the segment (i.e. the selector is less privileged than the program using it), then the program will make the memory access at the privilege level of this segment. If the RPL field is greater than the CPL of the task that generated the selector, then the RPL is adjusted to the CPL. This situation can occur if the selector is being passed as a pointer to a function. This feature is designed to stop less privileged programs from using more privileged programs to access protected data.

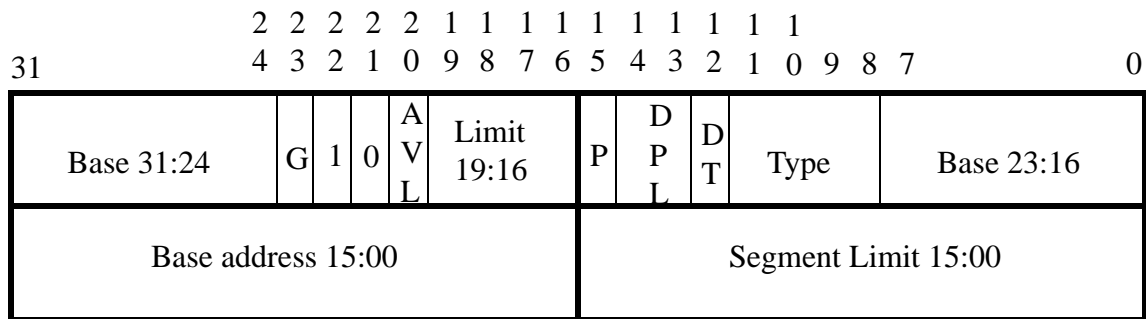
Segment Descriptors

The segment descriptor is a data structure in memory which provides the processor with the size and location of the segment in physical memory (when pure segmentation is used). In addition it contains some control and status information. The segment descriptors are typically created by compilers, linkers, loaders, or the operating system, but *not* by application programs. Figure 11-7 shows the descriptor format used by the 80386.

The components of the segment descriptor are as follows:

Base: Defines the location of the segment within the 4 gigabyte physical address space. The three base address fields are pieced together to form the 32 bits of the address.

Granularity bit: This turns on scaling of the Limit field by a factor of 4096. When the bit is clear then the scaling factor is one and the limit is interpreted in units of one byte, when the bit is set then the limit is interpreted as units of 4096 bytes. When set, the 12 least significant bits are not tested when the limit tests are carried out. This, to some degree, lim-



- AVL Available for use by system software
- BASE Segment base address
- DPL Descriptor privilege level
- DT Descriptor type (0=system, 1=application)
- G Granularity
- Limit Segment limit
- P Segment present
- Type Segment type.

Figure 11-7: Block diagram of a segment descriptor.

its the “tightness” of the protection mechanism. Note that only the limit field is affected, the base address remains byte granular.

Limit: Defines the size of the segment. The limit is formed by combining together the two limit fields in the descriptor to form 20 bits. If the granularity bit is clear then this means that the segment size is limited to 1 megabyte in increments of 1 byte. If the granularity bit is set then the segment size is 4 gigabytes in increments of 4 kilobytes. The limit can be interpreted in two different ways – for normal segments the logical address offset can range from 0 to the limit, for expand down segments the offset can be any value except those in the 0 to limit range.

DT field: The descriptors for application segments have this bit set. This bit is clear for system segments and gates.

Type: The interpretation of this field depends on whether the segment is an application segment or a system segment. System segments have a slightly different descriptor format.

Data segments – the three lowest bits of the type field can be interpreted as expand-down (E), write enable (W), and accessed (A). An expand-down segment changes the way the limit is interpreted as discussed in the section on limits. A data segment can be read only or read/write. This is controlled by the W bit. The accessed bit is set whenever the segment is accessed. This bit is used by some operating systems software to aid in deciding what segments should remain in memory and what segments should be swapped out to disk.

Code segments – the three lowest bits are the conforming (C), read enable (R), and accessed (A) bits. A transfer into a more privileged conforming segment keeps the current privilege level. A transfer into a nonconforming segment at a different privilege level results in a general protection fault, unless a task gate is used. System utilities that do not manipulate protection facilities are often conforming segments (e.g. math library routines, code translation routines, some exception handlers). A code segment may also be execute only or execute/read, depending on the state of the R bit. An example of where it may be execute/read is when the code with data is on ROM. By using a segment override on the instructions the data could also be read from the ROM using the code segment. An alternative technique is to load the DS register with a selector to a readable, executable segment.

DPL (Descriptor Privilege Level): These bits determine the privilege level of the segment.

P (Segment Present): This bit is used to indicate whether the segment is physically present in the memory of the machine. If the bit is clear when the segment selector for the descriptor is loaded into the segment register then an exception is generated. Segments may not be present in memory if they have been swapped out of memory onto disk in order to free memory. Therefore this bit may be used to implement a segment oriented virtual memory system. When the segment-not-present bit is clear the Base, Limit and some of the control bit fields can be used to store operating system specific information (e.g. where the segment resides on the disk system).

Paging in the 80386

The paging model used in the 80386 is the classical paging model described in the “Paging - basic principles” on page 10-4, except that it uses a two level page table. The paging mechanism is enabled by setting bit 31 (the PG bit) in the CR0 register. In the 80386 the 32 bit logical linear address is broken into 3 parts:

- two 10 bit indexes into the page tables – one for the page directory and the other for the page table.
- a 12 bit offset into the page addressed by the page tables (i.e. the page size is 4 kilobytes).

The top 20 bits of the linear address are used to index into the page table. If a single level page table was used then this would mean that 4 megabytes of memory would be consumed by the page tables ($4 \text{ bytes per page table entry} \times 2^{20}$). In order to avoid this a two level page table is used. The top level page table is called the page directory, and the second level is called the page table. This structure saves memory because the page tables themselves can be paged. Therefore, if certain page table entries are not being used then they can be swapped out of memory onto disk. A single page swapped from memory contains 1024 page table entries, and each page referenced has a size of 4096 bytes. Therefore such a page table would be capable of mapping to:

$$1024 \times 4096 = 4,194,304 \text{ bytes}$$

Therefore, usually very few pages have to be kept in memory at any point of time. The structure of the page translation process for the 80386 is shown in Figure 11-8.

As mentioned in “Control Registers” on page 11-5, the physical address of the current page directory is stored in the CR3 register, also known as the page directory register (PDBR). The system software has the option of creating a page directory for each task in the system, or using a single page directory for all tasks. The entries in the page table are of the form shown in

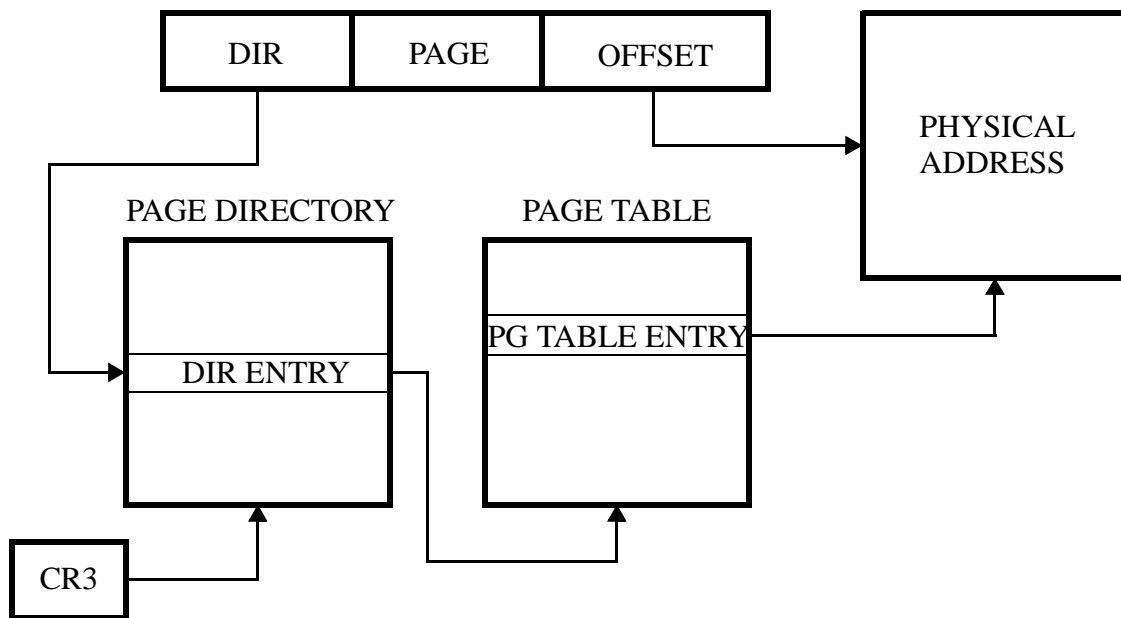


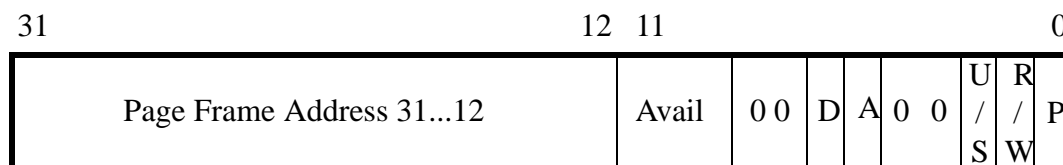
Figure 11-8: Page translation mechanism in the 80386 microprocessor.

Figure 11-9.

The entries in the page table have the following functions:

Page Frame Address: These bytes form the top 20 bits of the address in physical memory of the page. The lower 12 bits of the address come directly from the offset part of the linear address. If the page frame address is in the directory table then it is addressing another page table, and not a page of code or data.

Present bit: This bit indicates whether the page frame address in the page table maps to a physical page in memory – i.e. the page is in memory. If the bit is clear then the page is not in the memory, and the rest of the page table entry is available for system use. It is normally used to hold information indicating where the page is. If the bit is clear then a page fault exception is generated if an attempt is made to access the page. This invokes a series of events that get the page into memory and sets up the page table entry with the appropriate physical address and control bits. It should be noted that the translation



P Present
R/W Read/Write
U/S User/Supervisor
A Accessed
D Dirty
Avail Available for system programmers

Figure 11-9: Format of a page table entry

lookaside buffer (TLB) should be flushed when a page table entry is changed, in order to ensure that the TLB entry is updated. Obviously a paging operation is quite an expensive operation in terms of system performance, since all page table entries being used will have to be cached again into the TLB.

Accessed and Dirty Bits: These bits provide data about page usage in both levels of page tables. The accessed bit is used to report that there has been a read or write access to a page directory or second level page table. The dirty bit is used to report write access to a page. With the exception of the dirty bit in the directory page table entry (which is not used since this table does not directly reference memory pages), these bits are set by hardware. They are not cleared by the hardware. The accessed bit is set on both levels of the pages tables before a read or write operation to a page. The processor sets the dirty bit in the second level page table before there is a write operation to an address mapped by that page table entry.

The accessed bit is used to help the operating system decide which pages should be kept in memory and which pages can be swapped to disk. The dirty page bit is used to determine whether a page in memory has to be rewritten to the disk when the page is to be swapped out. If the page has not been written to (the dirty bit is zero) then there is no need to write the page to disk, since the memory image and the disk image match.

Read/Write and User/Supervisor Bits: These are used for protection checks applied to pages which the processor performs during the address translation process. Note that if pure paging is used there are only two modes the processor can be in – user and supervisor modes. This is compared to the segmented protection model where there are four levels of protection.

Translation Lookaside Buffer.

The translation lookaside buffer (TLB) is an onboard cache used for page table entries. The size of the TLB is such that most of the page references are found in the TLB. The TLB is invisible to application programmers, but not to operating system programmers. As mentioned previously, if any changes are made to the page table entries the TLB must be flushed to ensure that the cached entries are consistent with these changes. This flushing operation is carried out every time there is a change made to the CR3 register.

Combined Segment and Page Translation

One of the main attributes of the 80386's memory management system is that it offers the capability of combining both segmentation and paging, thus gaining the strengths of both memory management approaches.

The Flat Paging Model – Getting rid of Segmentation

If the 80386 is used to run software written without segments, one can map the stack, code and data spaces to the same range of linear addresses. This effectively removes the effects of the removal of segmentation. The processor does not have a mode bit for disabling segmentation. Paging can be used within this single segment space. If one desires to have protection between tasks in an operating system then each task can use a different set of page tables, which effectively means that they will operate in their own address space.

Segmentation and Paging Together

Segmentation and paging may be combined in a number of ways. A logical address is firstly

checked by the segmentation hardware and a linear address is produced if the check does not cause any protection violations. The linear address is then passed to the paging hardware, which then maps the linear address to the physical address, as described in the “The best of both worlds - combined paging and segmentation” on page 10-9. Figure 11-10 is a block diagram showing the overall segmentation and paging address mapping mechanism. Notice that the combined address space size for segmentation and paging would be:

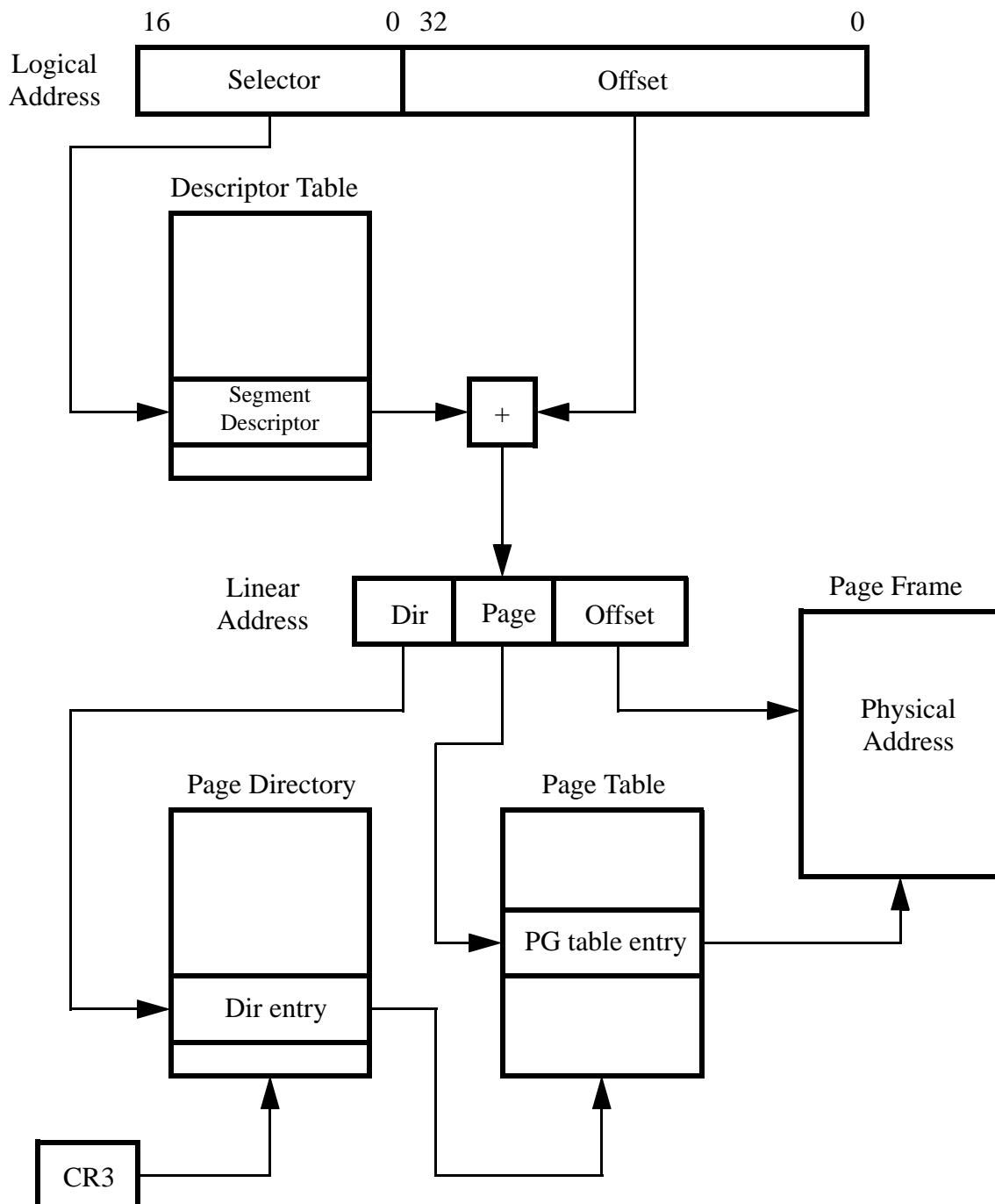


Figure 11-10: Combined segmentation and paging address translation

$$2^{14} \times 2^{32} = 2^{46} = 64 \text{ Terabytes}$$

Note that the 2^{14} comes from the 13 bits of the index in the segment selector, plus the 1 bit for the 2 descriptor tables (which effectively creates an extra bit).

The 80386 does not enforce any alignment between segments and pages. For example, a segment can span many pages. Therefore, accesses to data in such a segment does not require the whole segment to be in memory; only the page containing the data needs to be memory. Pages can span several segments (albeit that these segments would be very small (< 4 kilobytes)).

The boundaries of the segments and pages may occur anywhere in relation to one another. For example a page can contain the end of one segment and the beginning of another. This does not cause any problems with the protection, since this is mainly handled at the segment level and not the page level. Obviously the dual of this is valid for segments.

One approach to the memory management when paging is combined with segmentation is to give each segment its own page table (see Figure 11-11). This means the each page directory entry is associated with a segment, and this in turn points to a page table with 1024 entries in it. Therefore, a segment would be limited to $1024 \times 4096 = 4,194,304$ Bytes in size using this approach (which in practice is probably not a limitation). Because a page table is associated with a segment, if the segment does not use the full 4 megabyte of space available then the remaining page table entries cannot be used. In addition this approach also does not allow the full range of segment selectors to be used since there are only 1024 entries in the page directory (thereby allowing 1024 selectors).

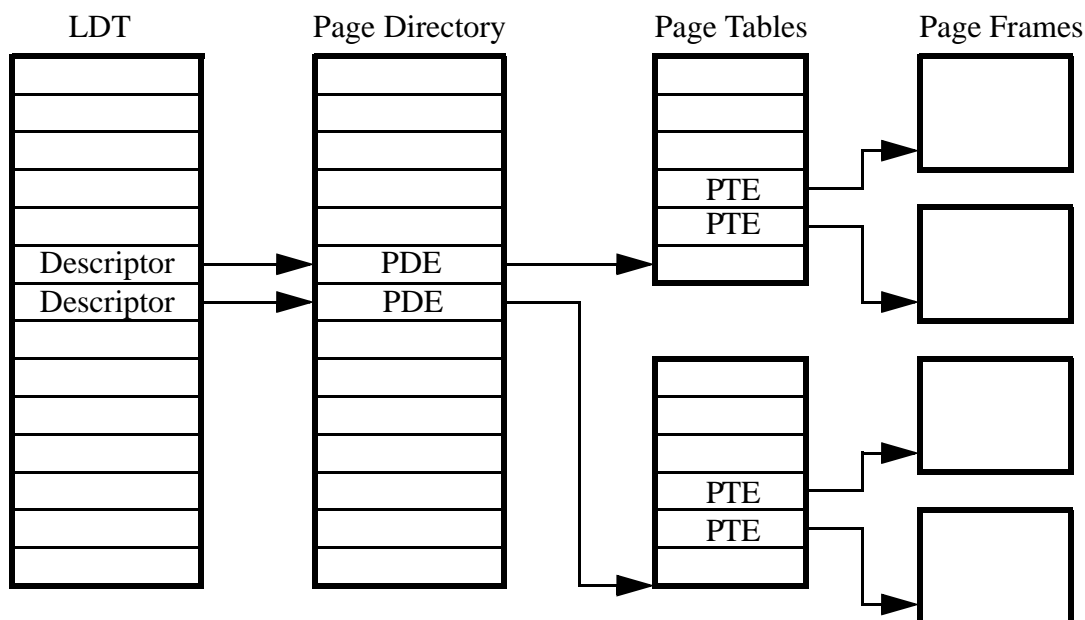


Figure 11-11: Segmentation and paging with each segment having its own page table.

Protection in the 80386 Processor

Protection and memory management are very closely related, since protection is basically about protecting the memory space of one task in an operating system environment from another. The segmented memory management technique was mainly developed with protection in mind. Paging was mainly developed as a virtual memory technique, but has been modified since to also give protection.

Protection is a valuable aid in the development of reliable software. During software development protection violations can be used to locate software bugs, since the nature of protection means that the debug software will remain runnable because it is protected against damage by the misbehaving application software. For production software, protection can make software more reliable by giving the system the opportunity to recover from protection faults. Even if this doesn't occur, the diagnostic information obtained from the protection fault can be invaluable in locating a latent fault in production software.

Segment-level Protection

The protection checks are activated before every memory reference. If a protection violation occurs then the memory reference is halted and a protection exception occurs. Because the checks are carried out in parallel with the address translation, there is no performance penalty.

Five protection checks are carried out:

- (i) Type check
- (ii) Limit check
- (iii) Restriction of addressable domain
- (iv) Restriction of procedure entry points
- (v) Restriction of instruction set

Segment Descriptors and Protection

Figure 11-12 shows the fields of the segment descriptors which are used by the protection mechanism. Note that the individual bits in the type field have their names associated with their function. The protection bits are set in the descriptor when it is created. When a program loads a segment selector into a segment register, the contents of the associated descriptor is loaded into the invisible part of the segment register, including the protection information. This allows protection checks to be made without a performance penalty.

Type Checking

Descriptors are not only used for application code and data segments, but also for system segments and gates. The descriptors are data structures used for managing tasks, exceptions and interrupts. Note, not all descriptors are associated with segments; gate descriptors hold pointers to procedure entry points.

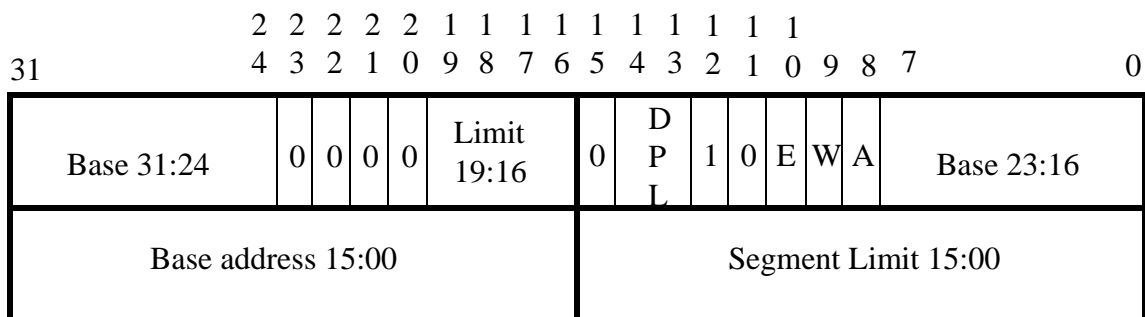
The type fields for code and data segment descriptors include bits which further define the purpose of the segment:

- The Writable bit in a data segment descriptor controls whether programs can write to the segment.
- The Readable bit in an executable segment descriptor specifies whether programs can read from the segment (e.g. to access constants stored in the code space). This may be achieved in two ways:
 - (i) With the CS register, by using a CS override prefix.
 - (ii) By loading a selector for the descriptor into a data segment register (the DS, ES, FS, or GS registers).

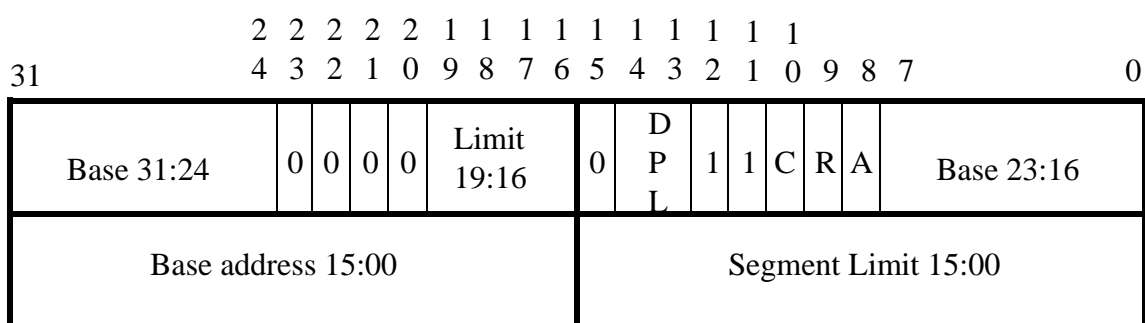
The processor examines type information on two kinds of occasions:

- (i) When a selector for a descriptor is loaded into a segment register. Certain segment registers can contain only certain descriptor types; e.g.:

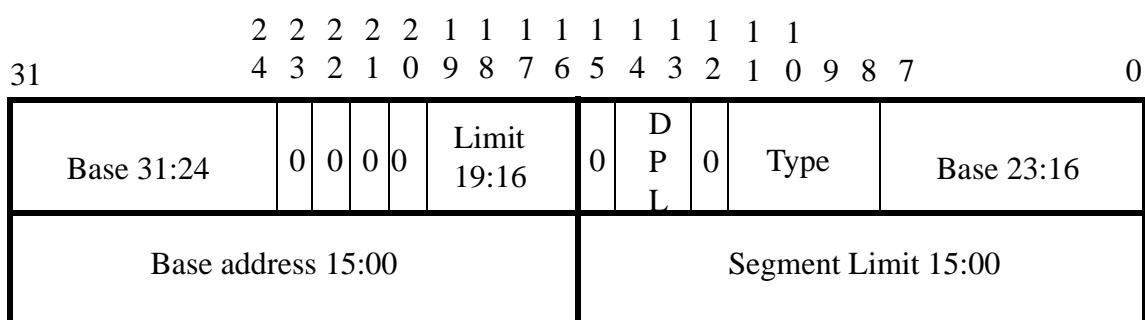
DATA SEGMENT DESCRIPTOR



CODE SEGMENT DESCRIPTOR



SYSTEM SEGMENT DESCRIPTOR



A Accessed
 C Conforming
 DPL Descriptor Privilege Level
 E Expand-down
 R Readable
 Limit Segment limit
 W Writable

Figure 11-12: Descriptor fields used for protection.

- The CS register can only be loaded with a selector for an executable segment.
 - Selectors of executable segments which are not readable cannot be loaded into data segments.
 - Only selectors of writable data segments can be loaded into the SS register.
- (ii) Certain segments can be used by instructions only in certain predefined ways; e.g.:
- No instruction may write into an executable segment.
 - No instruction may write into a data segment if the writable bit is not set.
 - No instruction may read an executable segment unless the readable bit is set.

Limit Checking

As discussed in the section “Segment Descriptors” on page 11-8, the limit field is used to check the validity of the address being accessed; normally (depending on the E bit) the address must be between the base address of the segment and the base address plus the limit.

The effective value of the limit depends on the G (granularity bit) and the E (expand down bit). If the G bit is clear, then the limit has a 20 bit (or 1 megabyte) range. When the G bit is set then the limit value is scaled by 2^{12} bytes. Therefore the limit has a 4 gigabyte range. Under this situation the limit checking is not as tight as for the former case since the lower 12 bits are not checked; for example, if the limit is zero then the valid addresses are from 0 to 4095. Therefore there is a slackness of 4 kilobytes in the protection validation. It should be noted that this also occurs in the paging protection mechanism.

It should be noted that the size of the data being accessed is also accounted for by the protection hardware. A protection fault occurs for the following accesses:

- Attempt to access a memory byte at an address $>$ limit
- Attempt to access a memory word at an address $>$ (limit-1)
- Attempt to access a memory double word at an address $>$ (limit-3)

With expand down segments the valid address lie in the range (limit+1) to $2^{32} - 1$. Therefore the maximum size segment is obtained with a segment limit of 0.

In addition to applying limit checking on segments, there is a limit check on the descriptor tables. Both the GDTR and LDTR register contain a 16 bit limit value. This is used to prevent programs from selecting a segment descriptor outside the descriptor table. The value should be a multiple of 8, since the descriptors have a size of 8. For N descriptors the limit should be $8N-1$. Another interesting point in relation to descriptors is that the selector with an all zero value is referring to the first descriptor in the GDT. This is a reserved descriptor, which may be loaded into a data selector, but will cause a general protection fault if an attempt is made to reference memory with it. If the selector is loaded into the CS register then a general protection fault will result. This is a handy feature with pointer orientated languages like ‘C’, enabling uninitialised pointers to be easily detected by the hardware. It is also used with the call gate mechanism.

Privilege Levels

When the full segmented model of protection is being used the 80386 supports four privilege levels, numbered from 0 to 3 (0 being the highest privilege). Privilege essentially works by generating a general protection fault if a program attempts to access a segment using a less privileged level than that applied to the segment. Figure 11-13 shows a protection ring model that may be established using this feature.

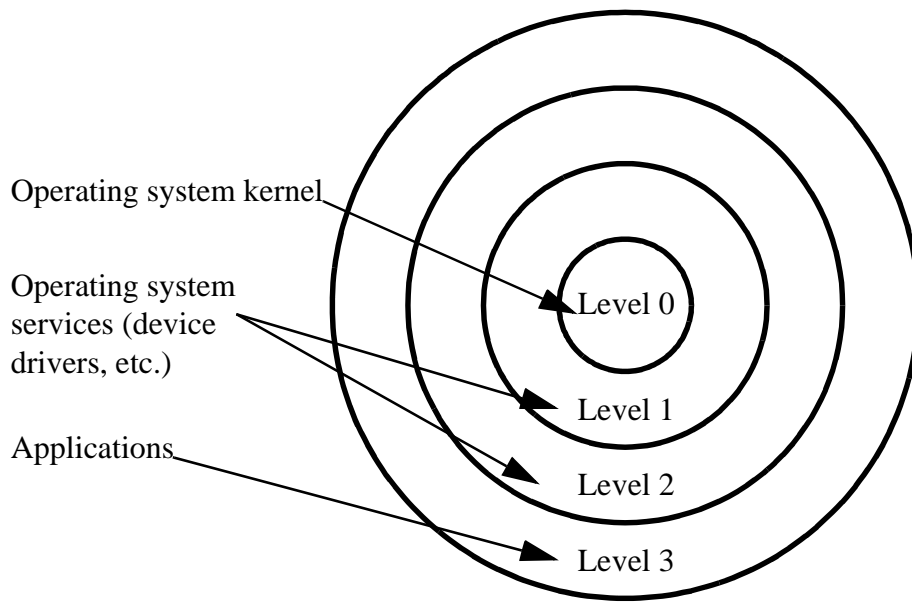


Figure 11-13: Protection ring structure

The presence of the four privilege levels allows a layered operating system model to be implemented with hardware protection between the layers (the 80386 is the only microprocessor that provides the hardware support for this model). The crucial operating system components are operated at the most privileged level, and therefore are protected from damage by other tasks and procedures in the system. If some other section of the system crashes this allows at least a diagnostic message to be generated. The less trusted sections of the operating system are confined to the outer levels of the ring structure, with the applications layer at the outer most level. In relation to system software, device drivers can be operated at ring 1 and 2 privilege. Examples would be the network drivers, disk drive drivers etc. In many operating systems these pieces of software operate at kernel privilege, therefore bugs in them can bring the whole system down. The likelihood of this happening is increased by the fact that these pieces of the system are quite large and there almost always have some bugs in them.

The OS/2 operating system is structured on a model that has all operating system routines at privilege ring 0. The protection model used in this operating system is a two ring one (since it does not employ segmentation). The Microsoft Windows NT operating system also has a similar protection set up (see Figure 1-2), except that it moves the operating system components that would normally operate at ring 1 and 2 to the user level (which is equivalent to ring level 3). In this way, the operating system itself is protected from faults in software components such as network drivers, operating system interface components etc. Both OS/2 and Windows NT use the paging flat memory model instead of segmentation mainly for efficiency reasons, and in the case of Windows NT, also for portability reasons. The performance argument put forward goes something like this – every time there is a segment change, a new descriptor has to be loaded into the descriptor cache on the processor. Even if segmentation is implemented at the module level (it can, if desired be implemented right down to the procedure level), then the loading of descriptor information and carrying out the protection checks can have significant performance implications.

It is true that the times required for calls via call gates, between privilege levels, using a seg-

mented memory model, can be lengthy compared to a real mode call. However, intersegment calls between segments at the same privilege level (which do not need a call gate) are not excessively long (approximately twice as long in clock cycles as compared to an intrasegment call). The calls between privilege levels (which take of the order of $2\mu\text{secs}$) would be very much in the minority, therefore the performance hit with these types of calls should not have a great influence on system performance. One viewpoint that can be taken is this; the graphic user interface probably has the greatest performance impact on computer systems (from the user responsiveness viewpoint), yet users still use them because of their ease of use. The use of protected mode segmentation has a far less significant effect on the performance of the system, with potential gains in terms of system robustness and shortened software development times.

Restricting Access

In order to make an operating system secure, restrictions must be applied to accessing data as well as the right to call certain procedures. Obviously if data can be freely accessed then this circumvents the whole protection philosophy. Similarly if control transfers between privilege levels is not controlled then problems can arise because a higher privilege procedure may execute a less privileged procedure, thereby compromising the integrity of the higher privilege procedure. Let us now look at each of these cases separately.

Restricting Access to Data

Data is normally referenced using a segment selector for a data segment (i.e. in one of DS, ES, FS, GS, or SS registers). When a data segment register is loaded with a segment selector a privilege check is made to see if the data access is going to violate any privileges. Three different privilege values enter into this privilege deliberation:

- (i) The CPL (current privilege level) of the program. This is held in the bottom 2 bits of the CS register (the code segment selector register).
- (ii) The DPL (descriptor privilege level) of the segment descriptor. This value is held in the access rights section of the segment descriptor. This is normally in the invisible part of the segment selector after the selector has been loaded into the segment register.
- (iii) The RPL (requested privilege level) of the selector used to specify the segment containing the data. This is held in the bottom two bits the data selector that would be in one of the data segment registers cited above. This value is usually set to the value of the procedure that creates the selector. The RPL value is a way that a data selector can carry with it the privilege of who originated selector. This is particularly important when a data pointer is passed into more privileged procedures.

The basic rule is that a procedure may only load a segment register if the DPL of the segment is \geq the maximum of the CPL of the procedure or the RPL of the selector (i.e. the segment is equal to or less privileged). Note that the RPL does not have to match the CPL or the DPL, since the RPL is usually loaded with the value of the CPL of the creating procedure. This is not necessarily the procedure accessing the data, since the selector may have been passed to another procedure with a different privilege level before an attempt is made to use the selector.

Given the above rules, a procedure with CPL privilege 0 should be able to access any data (assuming that it creates the selector) because all the privilege levels are \leq its privilege. Similarly a procedure with privilege 1 can only have access to data at privilege levels 1, 2 and 3. Clearly, user applications at privilege level 3 can only directly access data at their own privi-

lege level. To access data at other privilege levels user procedures have to call procedures with

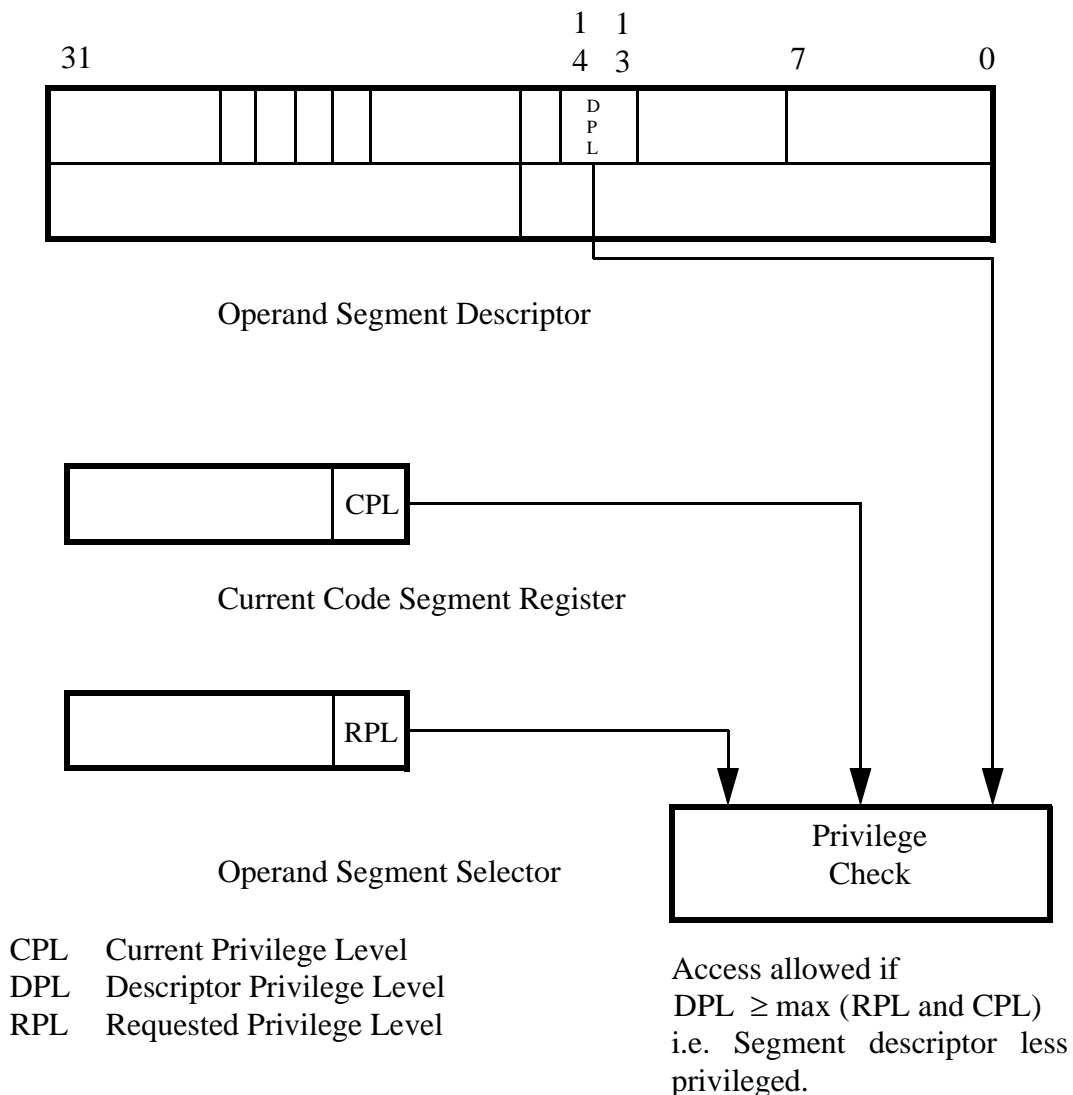


Figure 11-14: Privilege Checks required for access to data.

the necessary privilege to access the data required. These control transfers are carefully controlled in order to prevent any violations of the protection in the system.

As mentioned on several previous occasions data may also be accessed in the code segment. This is a legitimate thing to do, especially in ROM based systems where constant data is stored in the code segment. The techniques to achieve this have been previously outlined (see “Segment Descriptors” on page 11-8).

Restricting Control Transfers

As eluded to in the previous section, not only does direct access to data have to be restricted, but also access to data using other procedures. The mechanism by which the first level of protection can be violated in a poorly protected system is best demonstrated by some examples.

Consider a ring 3 procedure (i.e. the lowest privilege level) which calls a ring 0 procedure. The calling ring 3 procedure passes a selector to a segment for which it does not have the necessary

privilege (this could be on purpose or accidental). The ring 0 procedure does have privilege to access the data, and therefore can return the requested privileged data to the caller. This obviously violates the protection, since a less privileged procedure has gained access to privileged data via a control transfer mechanism.

Another situation one can imagine is the following – a more privileged procedure is allowed to call a less privileged procedure. This action would immediately compromise the integrity of the more privileged procedure because it is being allowed to execute a part of itself as a less trusted, and therefore less privileged, procedure.

The 80386 processor has 5 different instructions that allow control transfers to occur – JMP, CALL, RET, INT, and IRET, as well as the hardware interrupt and exception mechanisms. In this section we shall only discuss the JMP, CALL and RET transfers.

The intrasegment forms of the JMP, CALL and RET instructions are the “near” form of the instructions. Because they are occurring within the one code segment they are subject only to the normal limit checking on segments.

The intersegment or “far” forms of the JMP and CALL instruction refer to other segments, so the processor has to perform privilege checking in addition to the limit checking. There are two ways that a JMP or CALL instruction can refer to another segment:

- (i) The operand selects a descriptor for another segment.
- (ii) The operand selects a call gate descriptor. These are special descriptors that contain information to control the transfer between segments. Call gates are investigated in detail in a following section.

Two privilege checks enter into a privilege check for a control transfer which does not involve call gates (see Figure 11-15):

- (i) The CPL (current privilege level).
- (ii) The DPL of the descriptor of the destination segment.

Under normal conditions the DPL is equal to the CPL of the current procedure. However, in the case of conforming segments the CPL may be less than the DPL of the segment that is executing. The name of this segment type comes from the fact that the privilege level of the segment is dynamic in the sense that it “conforms” to the calling procedure. Conforming segments are mainly reserved for system level procedures that have no data of their own, but are used to manipulate data of the caller. Therefore, these procedures do not need to run at high privilege. In addition it means that the segments do not need a call gate for a transfer between levels.

If a segment is not a conforming segment, then it can only be directly called from procedures at the same privilege level. Interlevel transfers are only allowed using call gates, which are described in the next section. The JMP instruction can only transfer control to non-conforming segments with the segment DPL equal to the CPL (this applies even if call gates are used for the JMP's).

Gate Descriptors

These are a special form of descriptor used to provide protected control transfers among executable segments. There are four kinds of gate descriptors:

- Call gates
- Trap gates
- Interrupt gates

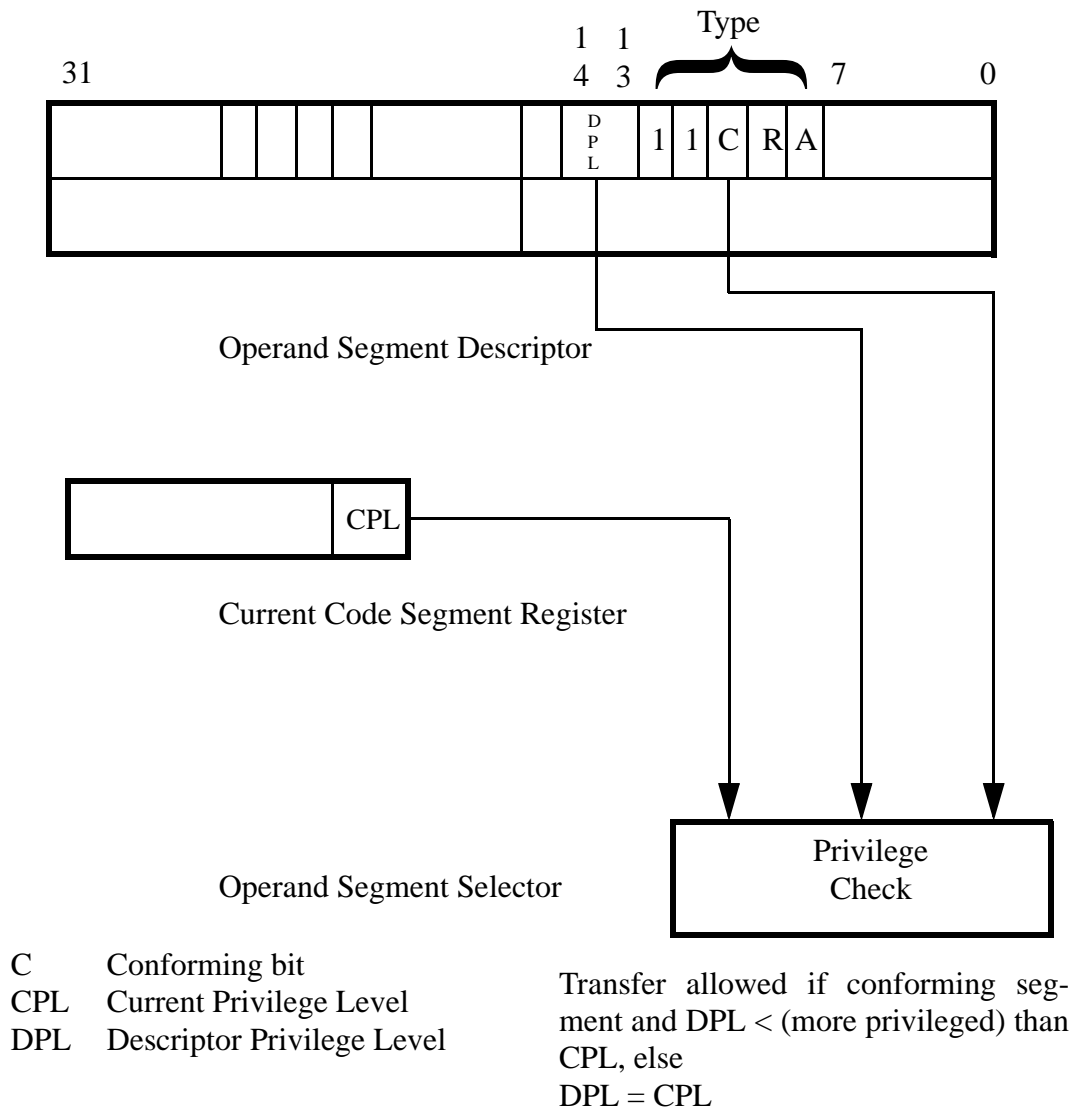


Figure 11-15: Privilege Checks required for control transfers without a Gate.

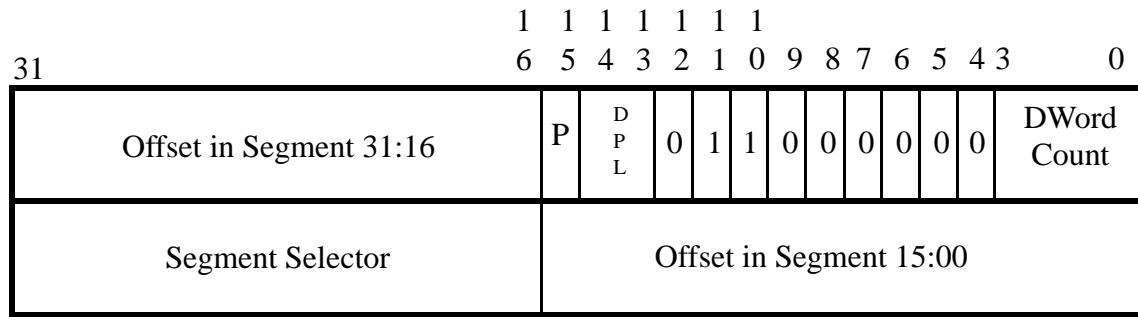
- Task gates

The latter three of these are considered in future sections; we shall concentrate on call gates here. Call gates are used for control transfers between different privilege levels. A call gate has two main functions:

- To define an entry point of a procedure.
- To specify the privilege level required to enter a procedure.

Call gate descriptors are used by CALL and JMP instructions in the same manner as code segment descriptors. The hardware recognises that the selector refers to a call gate descriptor, and not a code segment descriptor, and the control transfer is then controlled by the contents of the gate descriptor. The gate descriptor resides in the GDT or the LDT, but not in the IDT (interrupt descriptor table). Figure 11-16 shows the format of the gate descriptor.

The selector and offset components of the call gate form a pointer to a procedure entry point.



32 Bit Call Gate

Figure 11-16: Call Gate Descriptor

The DPL of the gate is used to determine what privilege levels are able to use the gate. Basically the DPL of the gate has to be greater than or equal to that of the CPL of the procedure instigating the call (i.e. the gate has to be equal or less privileged than the caller).

The selector operand of the control transfer instruction is used to index into the descriptor table (GDT or LDT). The offset part of the operand is ignored. Figure 11-17 is a block diagram of the call gate control transfer mechanism.

The formal rules for the use of the jump and call gate mechanism are:

For a JMP instruction to a nonconforming segment, both the following privilege rules must be satisfied, otherwise a general protection exception fault is generated.

MAX(CPL,RPL) ≤ gate DPL
destination code segment DPL = CPL

For a CALL instruction (or JMP instruction to a conforming segment), both of the following privilege rules must be satisfied, otherwise a general protection exception fault will be generated.

Figure 11-18 and Figure 11-19 show the rules for intralevel and interlevel control transfers. The essential feature is that the gate DPL has to be as privileged or less privileged than the calling procedure. The segment that control is being transferred to has to have privilege equal or greater than the calling procedure. This implies that calls can only occur in towards the centre of the ring protection model. Consequently RET instructions will only return control from a more privileged level to the same privileged level or a less privileged level. Jumps are more restricted, only being able to transfer to segments of the same privilege.

The restriction of only allowing calls to more privileged levels is to make sure that a privileged function cannot be compromised by executing code at a less privileged level as part of itself. This rule is defined by the hardware design and cannot be circumvented.

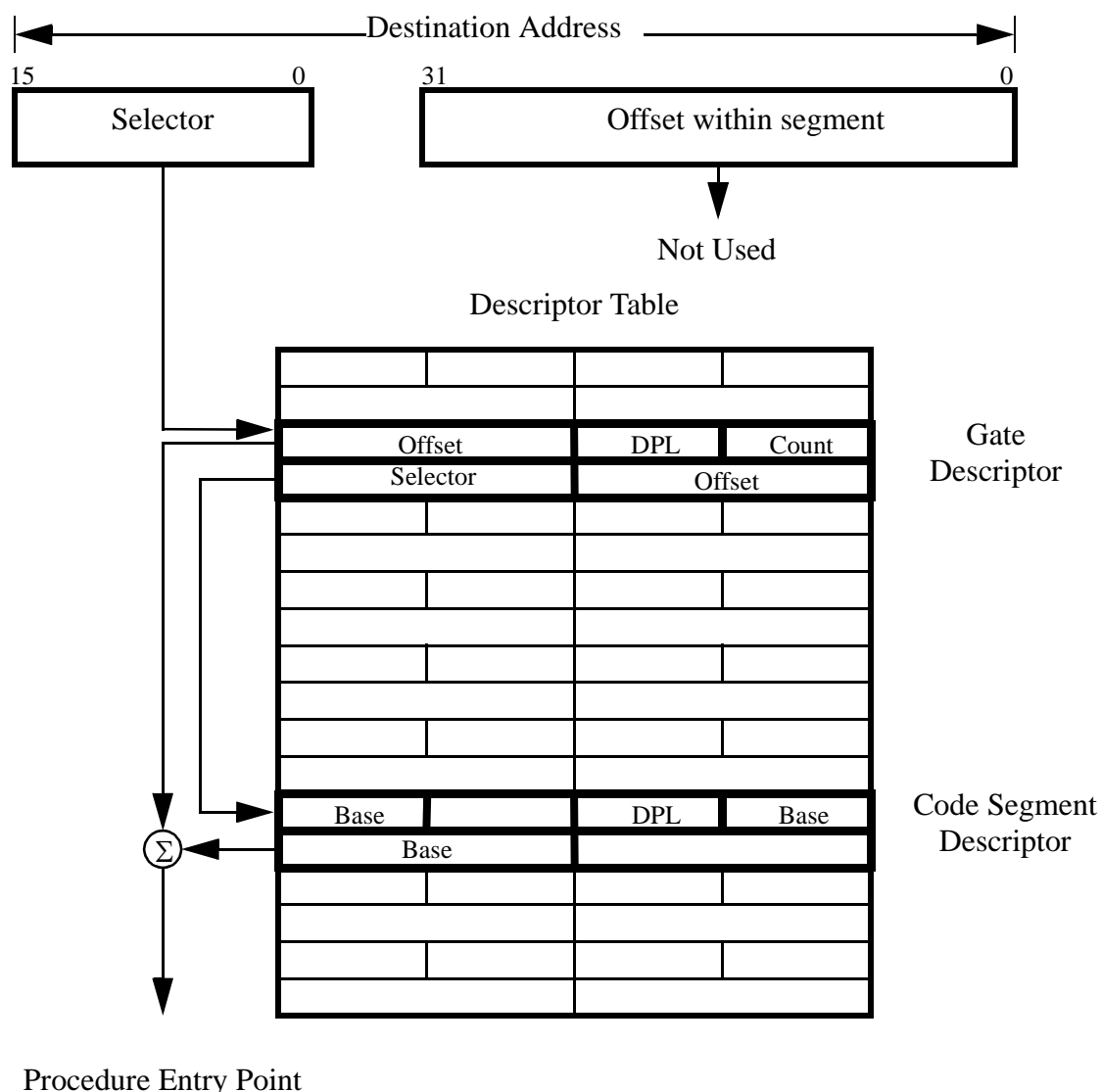


Figure 11-17: Call Gate Mechanism

Stack Switching

When a procedure call occurs to a more privileged level the following occurs:

- (i) Changes the CPL.
- (ii) Transfers control to the new executable.
- (iii) Switches stacks to a new stack.

The latter of these three occurrences, the switching to a new stack, is carried out in order to enhance the overall protection of the system. The new stack is allocated so that the privileged procedure does not have to rely on the less privileged procedure to provide a stack. If the less privileged procedure does not supply a large enough stack, for example, then the more privileged procedure may crash. Therefore the isolation of the more privileged procedure would be compromised. The other reason for using the separate stack is so the less privileged procedure cannot get access to more privileged information. This can possibly occur, if there wasn't a separate stack, by examining old information on the stack. When there is a separate stack, the

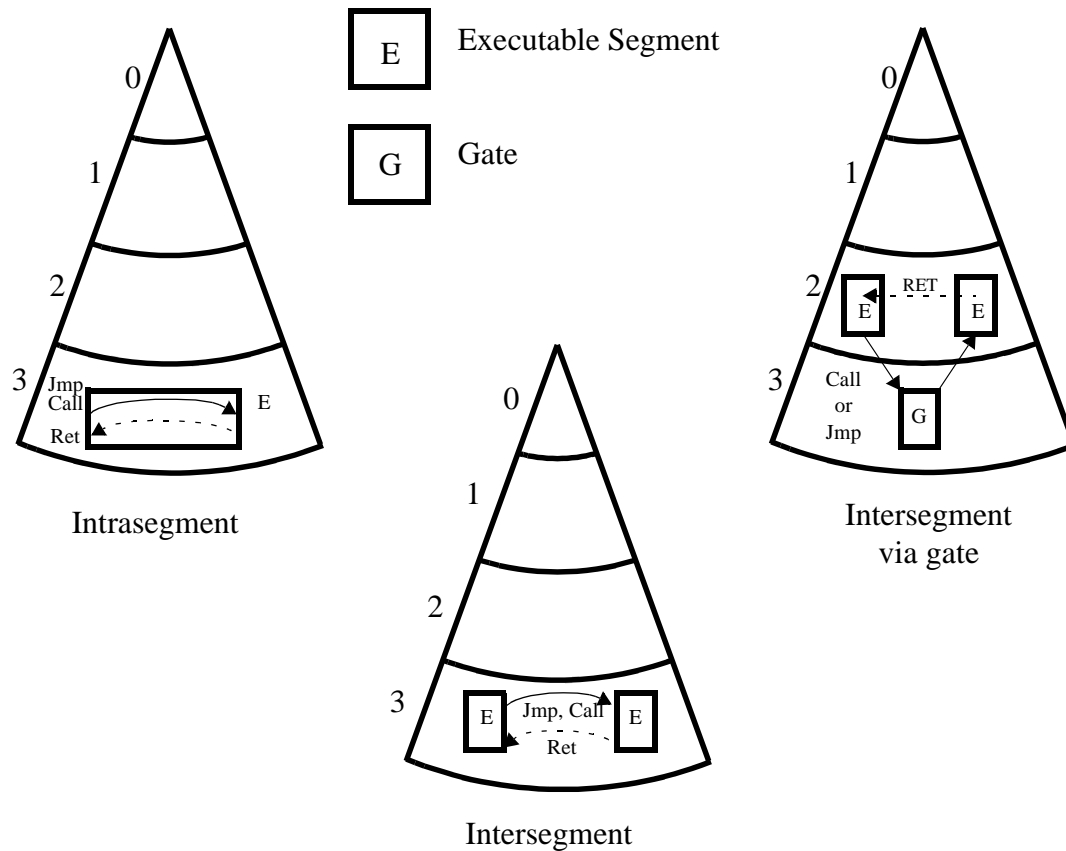


Figure 11-18: Intralevel Control Transfers

stack and its contents are effectively destroyed as far as the less privileged procedure is concerned when a return is executed from the more privileged level.

The space for the stacks that are created on calls to more privileged routines is found using information in the Task State Segment (TSS). This segment will be discussed in detail later, suffice to say at the moment that it contains information required in a multitasking environment (such as registers, current task stack pointers and segment), which includes the initial stack pointers and segments for the inner protection levels. For a task at ring 3 privilege this would mean that there is 3 sets of stack selector and pointer values, one each for ring 2, 1, and 0. The DPL of these stacks must equal the CPL of the procedure that is being called. It is the responsibility of the operating system to supply the stack segment descriptors for all the privilege levels. These segment descriptors must specify enough space (in the form of the Limit field) to hold the SS and ESP of the current stack, the return address, and the parameters and temporary variables required by the called procedure.

The parameters that are placed on the old stack are copied to the new stack. The count field of a call gate tells the processor how many double words (up to 31) to copy from the old stack to the new stack. If more parameters are required, then one the parameters passed can be a pointer to a data area, or the old stack can contain the extra parameters and can be accessed via the old stack selector and stack pointer stored on the new stack (this is all right since a more privileged procedure can access less privileged data).

The sequence of events that occur when executing a call are:

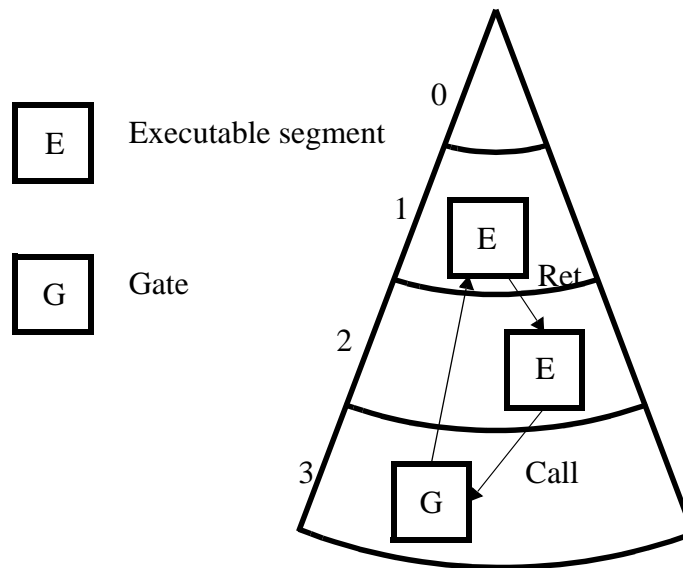


Figure 11-19: Gate Interlevel Call and Return

- (i) The stack of the called procedure is checked to make sure that it is large enough to accept all the parameters and saved registers – if not then a stack exception occurs.
- (ii) The contents of the old SS and ESP are copied onto the new stack as two double words.
- (iii) The parameters are copied onto the new stack of the called procedure.
- (iv) The return address (CS and EIP registers) are pushed onto the stack. The SS and ESP of the new stack point to this return address after the whole call sequence has been completed.

It should be pointed out that a call gate does not check the values of the words copied onto the stack. It is the responsibility of the called procedure to check the parameters for validity. The 80386 has specialised instructions to carry out these operations. These will be discussed later.

Returning from a Procedure

A RET instruction may or may not cause a change in privilege level. The “near” form of the RET is returning within the current segment, and is only subject to the normal Limit checks on the value popped into the EIP register. The “far” form of the RET instruction pops the return address which was pushed onto the stack by an earlier far CALL instruction. Normally the value popped off the stack is a valid pointer, since it was placed on the stack by an initial CALL or INT instruction. Privilege checking is carried out anyway because the stack may have been tampered with or corrupted. The RPL of the CS popped off the stack should be equal to the CPL of the calling procedure.

When a RET occurs towards a less privileged procedure (i.e. the RPL of the CS register saved on the stack is numerically greater than the CPL of the currently executing procedure at the time of the RET) then the following checks are carried out:

- (i) A number of checks are carried out (15 checks in total), some of which are – top of stack must be within the segment limit, RPL of the return code segment must be greater than the CPL, Return code segment descriptor must be within descriptor table limit,

Return segment descriptor must be a code segment, etc. (see the “386 DX Microprocessor Programmer’s Reference Manual” for more details of these checks).

- (ii) The old contents of the SS and ESP (which are on the top of the current stack) are adjusted by the number of bytes indicated in the RET instruction. The resulting value of ESP is not checked against the limit until the next stack operation occurs.
- (iii) The contents of the DS, ES, FS, and GS segment registers are checked. If any of these registers refer to segments that have a DPL less than the new CPL (i.e. the CPL of the calling procedure), excluding conforming segments, then the NULL segment selector is loaded into the segment. This does not cause a protection fault unless an attempt is made to use the segment selector. This process is designed to prevent less privileged programs from getting access to the data of more privileged programs by using the selectors left in the registers by more privileged programs. The normal privilege checking does not prevent this, because the privilege checks are carried out when the segment selector is loaded into the segment register.

Pointer Validation

As mentioned previously, pointers passed as parameters to procedures have to be validated to ensure that no privileges are violated. The steps carried out for pointer validation are:

- (i) Check if the supplier of the pointer is allowed to access the segment.
- (ii) Check if the segment type is compatible with its use (automatically carried out by the hardware).
- (iii) Check if the pointer offset exceeds the segment limit (automatically carried out by the hardware).

The first check is especially important when more privileged procedure is being passed data referencing parameters. It must be explicitly carried out by the called procedure. The ARPL (Adjust Requested Privilege Level) is provided for this purpose. The last two steps above can also be carried out in software – the advantage of this approach is that problems in the passed pointers can be captured before an exception is generated, and gracefully handled. The exceptions are then a last resort capture of the problem. The LAR (load access rights), LSL (load segment limit), VERR (verify for reading), and VERW (verify for writing) are extra instructions provided for parameter validation purposes.

The LAR and LSL instructions are used for loading sections of the segment descriptor. The descriptor must be readable at the current privilege level. In the case of the LAR instruction, the descriptor section can be loaded and the access rights tested. With the LSR instruction a 32 bit limit calculated using the G granularity limit is loaded.

Descriptor Validation

The VERR and VERW instructions are used for descriptor validation. Basically the instructions are used to determine whether a segment selector points to a segment which can be read or written using the CPL. If the test fails a protection fault does not result, but the ZF flag is cleared to indicate there has been a problem accessing the segment.

Pointer Integrity and RPL

The RPL privilege level stored in the segment registers is used to protect more privileged code from damage by pointers passed from less privileged code.

Normally the RPL bits in a segment selector are loaded with a value equal to the CPL of the

generating procedure. At the time of the load the processor automatically performs a check to ensure that the segment selectors RPL loaded into the segment register allows access to the segment. However this does not make sure that the RPL equals the CPL – it could in fact have a higher privilege in the RPL and pass this test.

If a selector passed to a higher privilege routine has an elevated RPL level then the calling procedure may gain access to a segment for which it does not have the privilege. In order to prevent this from occurring the called procedure need only ensure that the selector value passed to it has an RPL level equal to or greater than the CPL of the calling procedure. This guarantees that the passed selector does not have privilege greater than the caller.

The ARPL instruction is incorporated in the instruction set to carry out an adjustment operation on the RPL. It has two operands – one a word with the selector containing the RPL to be adjusted, and the other a register or memory location containing the data with which the RPL is compared. The segment selectors RPL is adjusted to the least privileged of the RPL fields in the two operands. Usually the second operand is the CS register stored on the stack of the called procedure, therefore the above operation would ensure that the RPL is adjusted to be greater than or equal to the RPL of the caller.

The ARPL instruction should be used in all system routines that accept pointers as parameters to ensure complete protection.

Page Level Protection

In the paging linear memory model much of the complexity of the segmented approach is gone. However, in the process the stringent checking is also lost to some degree. Some of the features lost are:

- The multi-ring protection model is lost – the paged only model only uses a supervisor/user protection model. Operating systems designed using this model tend to have large amounts of system code at supervisor privilege, therefore the critical system level routines tend to be less robust because of the large amount of code having high privilege. The code at the same privilege level tends to be less protected (unless the procedures are implemented as separate tasks with separate page directories).
- This model loses the concept of the executable segment – segments can only be read or write, or read and write. Code is treated no differently than data. Therefore, it is possible for execution of data to occur.
- Single tasks and programs cannot be broken down into small segments, each operating in its own address space. Therefore fine grained protection cannot be achieved. Consequently, total tasks will crash if there is a problem without any indication (except perhaps an address) as to where the problem occurred. Often the crash will occur at some time in the future relative to the time of the corruption.
- The call gate mechanism is used mainly as a means to call the supervisor mode routines (this is the means that a mode change occurs to get into supervisor mode). Call gate transfers between procedures at the same privilege level are lost (unless one starts to introduce segmented addresses again).

The protection is implemented at the page directory level as well as by the use of the page tables. Each reference is checked to verify that it satisfies the protection checks prior to a memory cycle starting. There are two page level protection checks:

- (i) Restriction of addressable domain
- (ii) Type checking

Restriction of Addressable Domain

There are two privilege levels for pages – user and supervisor mode. The privilege levels used for segmentation are mapped into these levels, with levels 0, 1, 2 mapping to supervisor mode, and level 3 being user mode. Figure 11-9 shows the format of a page table entry, and the various fields within this structure are discussed in “Paging in the 80386” on page 11-10. The privilege tends to work much the same as with the segmented approach (as one might suspect as the system is using very grainy segmentation), and the supervisor mode can gain access to all pages.

In order to enforce protection between tasks, the page directories for each task are different. The page directory is changed by the operating system upon the task switch. This means that each task can use the same linear addresses but the page tables will map the addresses to different physical addresses. Each task has the operating system pages mapped into its address space by the paging table, but direct access to these areas is prevented by the supervisor/user mode bit in the page table entries¹. Therefore the operating system data and code is protected from the user privilege code by page level protection and not segment protection.

Because of the protection limitations with paging outlined in “Privilege Levels” on page 11-17, alternative operating system designs have been developed recently, Windows NT being the best known example (see Figure 1-2). The main reason for opting for the paging protection model is for portability – very few processors support segmentation. Windows NT introduced many other overheads in order to get the system robust, these overheads probably being more costly in performance terms than a fully segmented memory model. However, since one of the main design objectives of Windows NT was to have an operating system that could be easily ported to different CPU architectures, the paging protection approach was the only viable alternative.

The IBM OS/2 Version 2.x operating system also employs the paging model. It was never designed to be a portable operating system, therefore the only reason for adopting the model appears to be a predisposition towards UNIX like memory models². Perhaps the “performance improvement” compared to segmentation was also a reason. Since OS/2 hasn’t adopted a design similar to Windows NT, it has many of the potential problems mentioned at the beginning of this section.

Figure 11-20 shows the segment set up for a paging only system. Note that the GDT is the only one used, since each task in the system shares a common set of segments. However, as mentioned above each task has a separate page table directory. Note the use of the gate descriptor for the transfer from user to supervisor privilege. In the process, the system selector, which points to the system descriptor is loaded, this descriptor having privilege 0.

Multitasking Support in the 80386

The 80386 processor provides hardware support for multitasking. The tasks can be invoked via the following means:

- (i) Interrupt
- (ii) Exceptions (or traps)

1. Note that the system procedures are in a different segment from the user code, but the segment is exactly aliased with the user segment – i.e. the segment starts at address zero and has a 4 gigabyte limit. By setting the segment up like this it has access to the same address space as the user privilege level has. Therefore using this model it is not possible to use the segment descriptor to protect the system data areas.

2. It should be pointed out that although never designed to be a portable operating system (as far as I know), OS/2 has been ported to the IBM R6000 series of RISC processors.

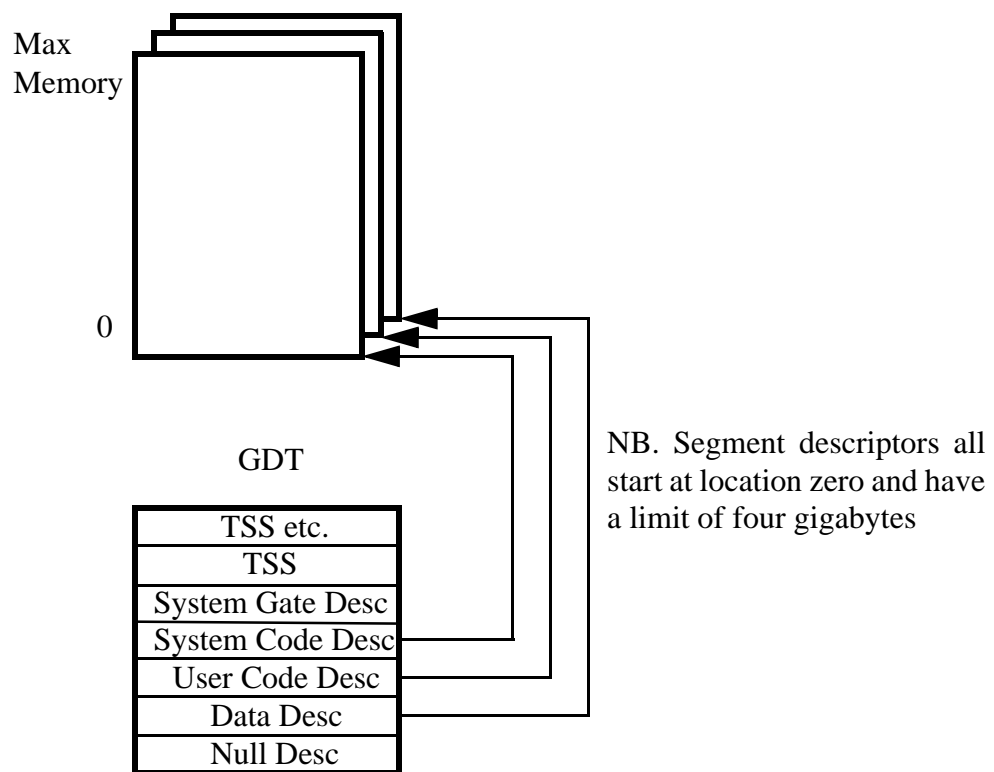


Figure 11-20: GDT set up for a paged, flat memory model.

(iii) Jumps

(iv) Calls

When one of the above is carried out with a special segment called the task state segment as the target, then a task switch occurs. This involves saving the state of the current task in the current task state segment, and then starting a new task by retrieving its saved state from its task state segment.

The registers and data structures that support multitasking are:

- Task state segment (TSS)
- Task state segment descriptor
- Task register
- Task gate descriptor

In addition to task switching, the 80386 offers two other task management features:

- Interrupts and exceptions can cause task switches – i.e. interrupt tasks are supported (in addition to the normal task concept).
- Switches to other LDT's or page directories can occur on a task switch. This allows each task to operate in its own address space, isolated from all other tasks.

The use of the inbuilt multitasking mechanism is optional – in many cases it may be more efficient not to use it, and instead use a software task switch technique. This is the approach taken

in systems like OS/2 V2.x for example.

Let us now look in detail at the registers and data structures listed above.

Task State Segment

This is a special segment that is used to store all the information required to save and restore a task. Figure 11-21 shows the layout of a Task State Segment (TSS). The fields in the TSS can

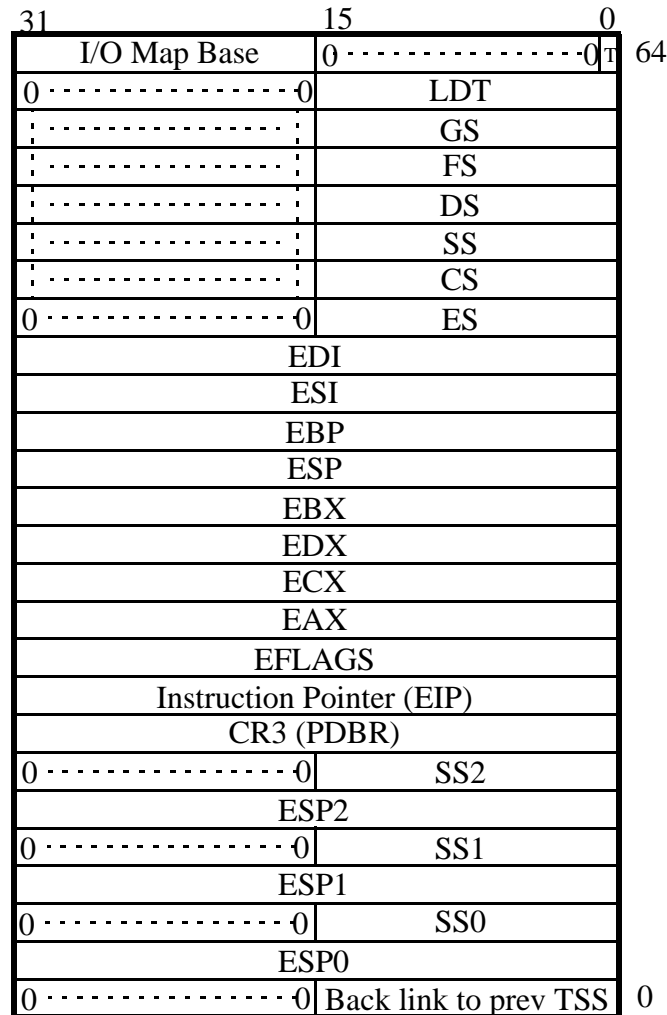


Figure 11-21: Diagram of a Task State Segment.

be divided into two main sections:

- (i) Dynamic fields the processor updates with each task switch. These fields store the following information:
 - The general registers (EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI).
 - The segment registers (ES, CS, SS, DS, FS, and GS).
 - The flag registers (EFLAGS).
 - The instruction pointer (EIP).
 - The selector for the TSS of the previous task (updates only when a return is expected).

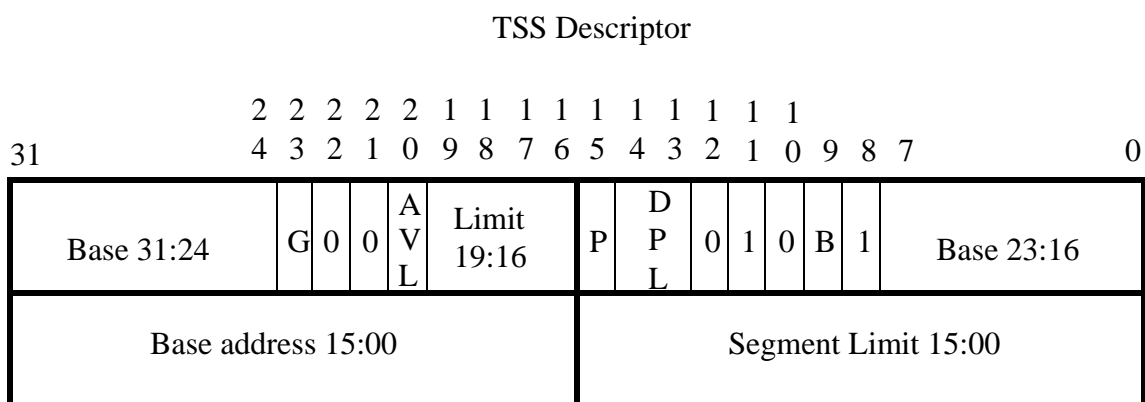
(ii) Static field the processor reads, but does not change. These fields are set up when the task is created. These fields store:

- The selector for the tasks LDT.
- The logical address of the stacks for privilege levels 0, 1, and 2.
- The debug bit (T bit) which, when set, causes the processor to raise a debug exception when a task switch occurs.
- The base address for the I/O permission bit map (see later for details in I/O permissions). The base address points to the beginning of the map, which is stored as part of the TSS at higher addresses.

If paging is being used it is important that the TSS does not straddle two pages. A general protection exception will result if a page fault occurs during a read of the TSS. If one allows this situation then the two pages have to be locked in memory, and cannot be paged.

TSS Descriptor

The task descriptor has a format very similar to other segment descriptors. Figure 11-22 shows the layout of the descriptor. The Busy Bit indicates whether the task is busy (i.e. actually running or on a ready list). The Busy Bit is used to detect whether the operating system is attempt-



- AVL Available for use by system software
- B Busy bit
- Base Segment Base Address
- DPL Descriptor Privilege Level
- DT Descriptor Type
(0 = system; 1 = application)
- G Granularity
- Limit Segment Limit
- P Segment Present
- Type Segment Type

Figure 11-22: TSS Descriptor Layout

ing to reentrantly run a task. Tasks are not reentrant entities.

The remainder of the fields in the descriptor are much the same as those defined in the data segment descriptors. The Limit field has a minimum value of 67 Hex, which is one less than the minimum size of the TSS. The maximum size is an application specific value. An attempt to switch to a TSS with a limit less than the minimum value will cause an exception.

The TSS descriptor can only reside in the GDT. It is written to by aliasing a data segment descriptor to the same table position. The only procedures that can activate a task switch using the TSS are those with privilege greater than or equal to that of the DPL of the TSS.

Task Register

The task register (TR) is an internal processor pointer to the current TSS. This is how the microprocessor locates the currently executing task, since the TSS is the identification of such entities. In order to make accesses to the current TSS efficient, the TR caches parts of the TSS descriptor in an invisible part of the TR; namely the base address of the current TSS and its segment limit. Figure 11-23 shows the access path to the TSS.

The processor has two instructions for manipulating the TR; LTR (Load Task Register) and STR (Store Task Register). LTR loads the selector into the visible portion of the TR. In the process the TSS descriptor is accessed and the invisible portions are loaded. The LTR instruction can only be used from code operating at CPL 0. The STR instruction allows the TR selector to be stored in memory or a register. It is not privileged.

Task Gate Descriptor

In a manner similar to the call gate descriptor for procedures, the task gate descriptor provides a protected reference to a TSS. A procedure may not carry out a task switch using a task gate unless the RPL of the selector for the gate and CPL of procedure are less than or equal to the DPL of the task gate (the DPL of the TSS descriptor is not used when a task gate is used). The task gate is provided for the following reasons:

- (i) Only one TSS descriptor for a task exists in the system, and it is usually in the GDT and has a privilege level of 0. The task gate allows the TSS descriptor to be accessed in a controlled way from other privilege levels in the system. The task gate descriptor, for example, can reside in the LDT of the task that can have access to the TSS descriptor. One TSS descriptor can have multiple task gates.
- (ii) Interrupt routines and exceptions can cause task switches. The task gate descriptor can appear in the Interrupt Descriptor Table (IDT) and can be referenced upon an interrupt or exception.

Task Switching

The task switch operation can be carried out by any one of the following operations:

- (i) The current task executes a CALL or JMP to a TSS descriptor.
- (ii) The current task executes a CALL or JMP to a task gate.
- (iii) An interrupt or exception indexes to a task gate in the IDT.
- (iv) The current task executes an IRET when the NT flag³ is set.

Note that the JMP, CALL and IRET instructions are normal instructions in the 80386, and not

3. The NT flag (Nested Task Flag) indicates that the current task has been invoked from another task, and that control is to be returned to the previous task if an IRET instruction is executed in the current task.

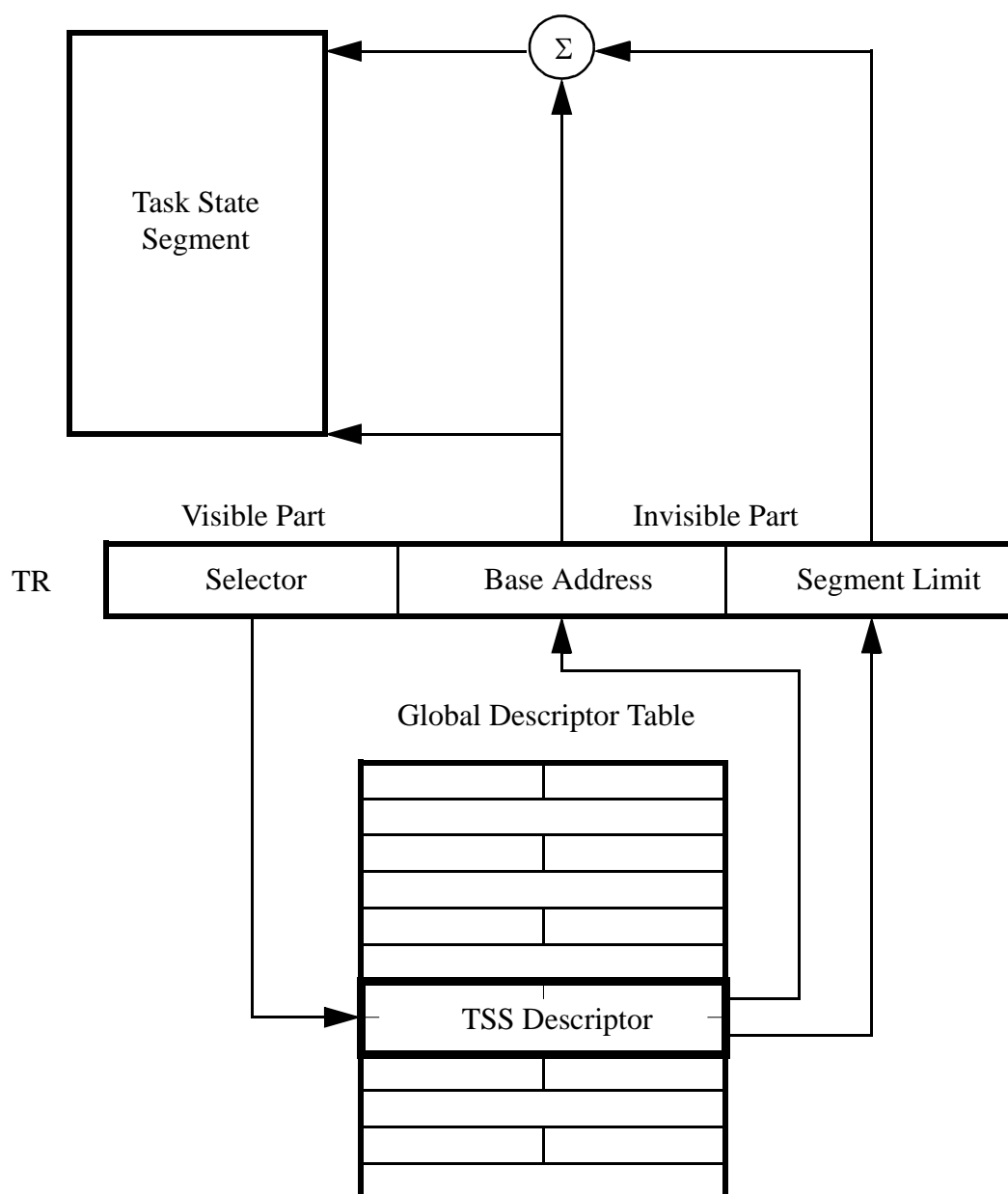


Figure 11-23: Location of the TSS using the Task Register.

necessarily concerned with task switching. In the case of the JMP and CALL it is the descriptor which is pointed to by the selector in the instruction that determines whether there is a task switch or not. In the case of the IRET, the NT flag will determine whether a normal IRET or a task switch IRET will occur.

The following steps⁴ are involved in a task switch:

- (i) Check that the current task has the privilege to switch to the new task – i.e. the DPL of the TSS descriptor and task of the gate selector. Excep-

4. These steps are mainly executed by the hardware and do not require any software.

tions, interrupts and IRET instructions are permitted to switch tasks regardless of the DPL of the destination task or TSS descriptor.

- (ii) Check the TSS descriptor of the new task is present and that the limit value is valid. Note that any errors occurring to this time are in the context of the current task. Any error will result in the restoration of the processor state prior to the error (i.e. just before the error producing instruction has been executed). This allows the exception error to return to the instruction that caused the error, hopefully after fixing the condition that caused the error. This feature is necessary to handle page fault exceptions.
- (iii) Save the state of the current task – basically copies all the registers into the TSS indicated by the TR.
- (iv) Load TR with the selector of the new task's TSS descriptor, set the Busy Bit in the TSS descriptor, and set the TS bit in the CR0 register. The selector loaded is either the operand of a JMP or CALL instruction, or taken from a task gate.
- (v) Load the new task's state from its TSS and commence execution of the new task. Any errors that occur here are in the context of the new task. If there is a problem restoring the state then it appears to the exception handler that the new task has not executed an instruction.

Task Linking

The link field of the TSS is used for nested tasks to allow reverse traversing of the nested chain of TSS's. The NT flag within the TSS indicates whether the link field contains valid information – i.e. whether the task is a nested task.

The typical situation which results in a nested task is an interrupt or exception occurring, which results in a task switch. In these situations the execution of the current task is suspended and an interrupt task or exception handler task starts running. When the interrupt task completes execution, the desired situation is that the task that was interrupted should continue. In other words, the interrupt task should behave in a similar fashion to the interrupt routine. Figure 11-24 shows a nested task structure and how the linking works. The current task releases control back to the previous task by executing an IRET. The hardware examines the NT flag, and if set it uses the back link component of the current TSS to indicate which task to switch to. The nested task feature can also be used to form child tasks that are not interrupt tasks.

Busy Bit

The purpose of the Busy Bit is to prevent loops of tasks from forming. Normally if a task carried out a task switch to itself no harm would be done; the task will simply resume immediately where it stopped (in other words the task switch has achieved nothing, except waste some time). However, if one has a chain of nested tasks, a call to a task that is already in the chain would corrupt the back link so that the chain would be broken.

When a switch to a new task occurs the Busy Bit is set in the TSS descriptor. If the task is then added to a chain of tasks the Busy Bit remains set, otherwise when the task switch to another task occurs the Busy Bit is cleared. If the task being switched to has the Busy Bit set then a general protection exception will occur, thereby preventing the nest task chain from being broken.

Task Address Space

By using the LDT field in the TSS and PDBR in the CR3 register each task can be given its

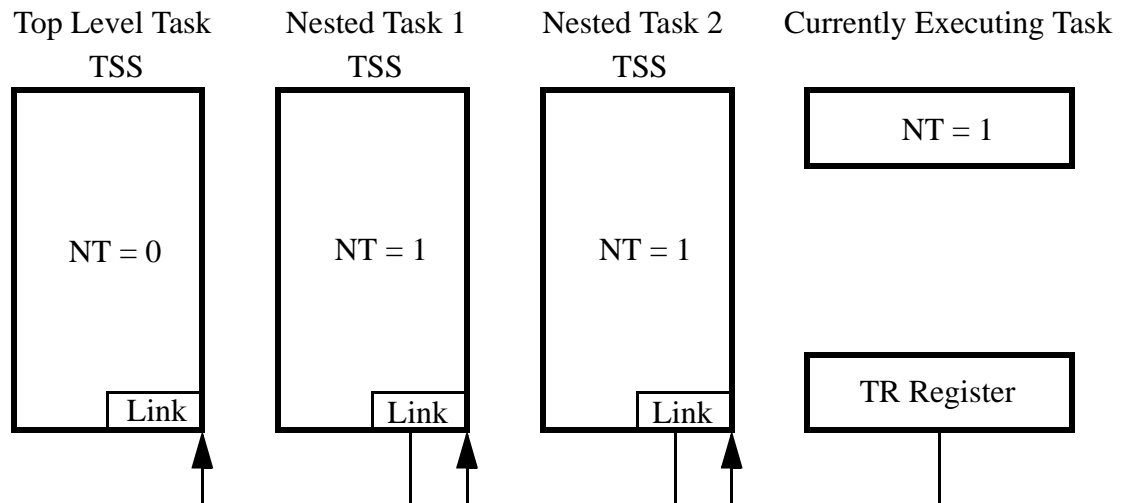


Figure 11-24: Nested TSS's and the NT bit status.

own address space. The separate LDT's for the tasks are the task's connections with the segments it uses. Similarly, if paging is enabled each task can have its own set of page tables for mapping linear addresses to physical addresses.

It is possible to have tasks sharing memory by allowing them to have the same LDT. This is an efficient way to allow tasks to communicate with one another without dropping the protection barriers of the whole system. In addition, all tasks have access to the GDT, so it is possible for them to share segments via this as well.

Protected Input/Output

Just as memory accesses have to be protected, so do direct accesses to I/O devices. In a multi-tasking environment it is important to restrict direct I/O access only to those tasks with sufficient privilege, these generally being system tasks. If user level tasks were allowed direct I/O access then I/O hardware programming could be corrupted. Furthermore, there would no longer be any system control on the access to the I/O hardware, so corruption would occur due to logically simultaneous access by different tasks.

I/O Addressing in the 80386

The I/O ports can be addressed in two ways in this processor:

- Through a separate I/O address space accessed through special I/O instructions.
- Through memory-mapped I/O, where I/O ports appear as physical memory locations.

The I/O address space approach means that no physical memory is consumed by I/O, and the full memory addressing capability of the machine is available. However, there are only a limited set of I/O instructions, compared to those available under the normal memory addressing instruction set. It very much comes down to personal preference as to which philosophy one uses, as there is no clear winner.

The I/O address mapping approach is supported by special I/O protection. If memory-mapping is used, then the normal segmentation and paging protection is used (since the I/O space does

not look any different than memory). The AVL fields in segment descriptors or page table entries may be used to mark pages containing I/O as unrelocatable and unswappable.

Protection and I/O

The I/O architecture has two protection mechanisms:

- (i) The IOPL (I/O privilege level) in the EFLAGS register controls access to I/O instructions.
- (ii) The I/O permission map in the TSS. This controls access by specific tasks to specific ports.

Note that the above mechanisms are only available if the I/O space mapping for I/O is being used. Protection for memory-mapping is provided by the normal segmentation and paging protection.

I/O Privilege Level

In a system where I/O protection is used, access to I/O instructions (e.g. IN, OUT, INS, OUTS) is controlled by the IOPL field of the EFLAGS register. The basic privilege control is that no task can use the I/O instructions unless its $CPL \leq IOPL$. The IOPL can only be changed by procedures running a ring 0 privilege. Usually ring 0 and ring 1 procedures have IO privilege, and lower privilege routines in the system carry out their I/O by calling, via a call gate, the required I/O procedures.

I/O Permission Bit Map

This further restricts the access of tasks to ports when the task has already achieved I/O privilege. The I/O Permission Map is pointed to by a base address in the TSS segment for a task. It contains a bit map, one bit for each port in the I/O address space. If the bit for a corresponding port is set then I/O permission is denied for that port, and a general protection exception is generated upon a attempted access. If the bit is clear then permission is allowed.

It is not necessary for the I/O bit map to represent all the ports. Any ports not represented are given default values of one (i.e. access is not enabled). Any attempt to access them will cause a general protection exception.⁵ Note that the I/O bit map does not allow discontinuous arrays of ports. If one is to use this feature then it is best for the ports to start from location zero in the I/O address space. If a task has access to ports in a range at some offset from the zero address then the bits corresponding to zero address to the beginning of the address range would have to be set to ones.

Conclusion

This chapter has outlined the main features in the 80386 series of microprocessors that are related to memory management and protection. The presentation has attempted to give enough detail to understand how the mechanisms work, but at the same time not get too bogged down. If one wishes to know more about the operation of these processors refer to the Intel manuals cited at the beginning of Chapter 1.

5. An exception would be generated in any case because this would imply an attempted read of the I/O bits past the end of the segment. Therefore an exception would be generated because the segment Limit value has been exceeded. In fact this is the only way to limit the I/O permission bit map.

Introduction

The UNOS operating system is a small real-time kernel. It is not an operating system in the sense of UNIX or Windows with a full fledged file system, a large suite of device drivers, dynamic creation of tasks, and perhaps support for multiple users. Instead it is a library of functions that provide basic support for multi-tasking, which can be incorporated into user programs.

For a user to utilize UNOS they link the UNOS library routines with their own routines to form a single program. This program is then executed (as is any other program) on an operating system such as DOS, and then UNOS takes over the machine. Alternatively, in other environments the resultant UNOS program would be located using a locator program (which assigns absolute addresses to the code and data) and then downloaded into the target hardware which is under the control of a resident ROM based monitor. In this situation the target hardware does not have any other operating system running on it prior to UNOS running.

UNOS is designed so that multi-tasking code can be written for embedded applications. Therefore it is essential that it has a small memory footprint, that it is fast (i.e. low interrupt latency and fast task switch times), and flexible (so that all common programming applications can be handled), and portable so that it can be easily ported to different hardware.

UNOS, because it is targeted towards embedded applications, has a rich set of real-time features to enable easy and reliable development of real-time code. For example, it will handle priority inversion problems automatically, it has a rich set of accurate timing primitives, it allows interrupts to be enabled whilst the kernel is executing to achieve low interrupt latency, it allows up to 256 task priorities for fine tuning of task execution, it allows finer timing control of the execution of tasks within the same priority level. Refer to the UNOS reference manual for a full explanation of all the user interface routines to the kernel.

One of the strong points in UNOS from the users viewpoint is the simple and elegant mail box task inter-communication mechanism. All tasks when they are created are given a mail box. Other tasks that can then communicate with a task by sending messages to its mail box. The messages can be multi-byte strings, and these are indivisible. Because a return address is always appended to a sent message, the receiving task can differentiate between messages sent to it from multiple tasks.

This appendix cannot possibly cover all the features of UNOS. Instead, it will develop a simple UNOS based application, and in the process explain the features being used. The application is an artificial one so that a reasonable number of features will be used. The overall structure of a UNOS based application will be emphasized.

The PC based versions of UNOS were originally intended to be run from DOS. This was so that once UNOS started it had complete control of the machine, and DOS effectively was no longer there. However, in the years since UNOS was developed DOS has come and gone as a popular operating system. Fortunately it lives on to some extent in the DOS boxes of

Windows 3.1, 95, 98 and NT. However, in the latter three of these the DOS boxes are running in a highly protected mode of operation, and the host operating systems are still running even when UNOS is running. Therefore the highly accurate timing that UNOS is supposed to have is compromised by the presence of the other operating system. This is especially true in the case of Windows NT where the timer interrupts are emulated by protected mode software traps. Therefore, these operating system environments are alright for running test code and examples, but if one desires true real-time performance then UNOS must be run from DOS, or be running on ones own hardware after being downloaded.

General Layout of UNOS Based Software

Before getting into the details of writing a UNOS application, it is beneficial to have an overview of the organisation of the modules used to get UNOS running. This is best understood with reference to Figure A-1.

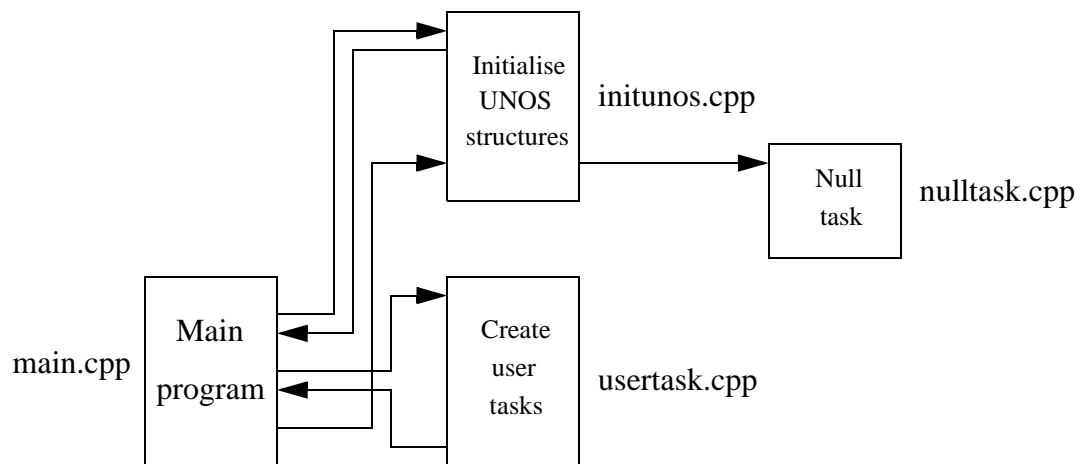


Figure A-1: Arrangement of the initialisation modules.

A UNOS system requires three main modules in order to initialise itself and get the null task running. Since a UNOS application is simply a program as far as the host operating system is concerned, then it begins with a main module (as does any ‘C’ program). In the example this main module is called “main.cpp”, but one can call it whatever one likes. This module *must* contain the function **main**. The **main** function is the starting point from which the UNOS application is initialised. The **main** function is a very short routine which calls three functions: **init_unos**, **create_user_tasks** and **go_unos**. The calling sequence of these functions is shown diagrammatically in Figure A-1, as one goes from the top of the main.cpp clock vertically downward. Let us briefly describe the function of each of these routines.

The **init_unos** function (which resides in the `initunos.cpp` module) has, as the name implies, the job of initialising UNOS. It calls several routines which create most of the internal data structures required by UNOS in order to operate, as well as program up the hardware required by the kernel (usually the timer chip required for timing operations and the time slice interrupt), and set up any interrupt vectors that are to be used (the kernel interrupt vector for example). It also creates some system tasks – namely the keyboard task and the keyboard multiplexer task. Upon returning from **init_unos** to **main** many of the OS data structures have been established. It should be noted that the `initunos.cpp` modules contains a number of **#defines** that set up the initial values for many kernel structures. For example, the number of priorities in the system, the maximum number of semaphores, numbers of timers etc. There-

fore the user will be required to customize these variables to tailor the system for their requirements. In addition any special application specific hardware initialisation may also occur in this module. See a later section for details on this.

After control is returned from `init_unos` back to `main`, the `create_user_tasks` function in `usertask.cpp` is called. This function consists of a series of calls to the kernel function `create_task`, whose purpose is to create a user task in the system. These tasks are created in a state ready to run when the kernel begins to operate (at this stage the kernel multi-tasking code is not running). In the process of creating the user tasks the creation routine can run task initialisation routines. These allow variables to be defined or semaphores to be created prior to the task starting. An example of where this is necessary is when tasks immediately become blocked on a semaphore when they start. The semaphore must exist prior to the starting of the tasks.

Upon return from `create_user_tasks` all the user tasks in the system have been defined and placed on the appropriate queues ready to be scheduled. At this stage of the initialisation process the kernel is still not running. The final step is to start the multi-tasking kernel. This is carried out by calling the function in the `init_unos.cpp` module called `go_unos`. This function carries out two functions (which are transparent to the user). It firstly creates the null task (i.e. the task that runs when there is nothing else to run). It does not queue this task on any of the kernel queues, but instead runs it after enabling the hardware interrupts. It is this final step that enables the multi-tasking code of the kernel. On the first timer interrupt the kernel code is entered and the highest priority task amongst those created will be selected to run. At this stage the UNOS kernel is in full control of the tasks running. The code execution from this point is no longer linear, and therefore cannot be shown diagrammatically.

Detailed Example of the Use of UNOS

In order to flesh out the general discussion of the previous section we shall look in detail at creation of a sample UNOS application. This application does not do anything in particular, but it does demonstrate various aspects of the UNOS kernel.

The approach taken in this section is to look at each of the modules and annotate the code listing with comments so that each step can be understood (the program annotation is in italics). This has the advantage that all the details will be covered. It should also be noted that any tutorial has limitations, in that it is only one example. Clearly one cannot show every permutation of what can be done, but it at least it is a start to understanding what is going on in one particular case.

The Main Module

The first section of this module is the section that includes the header files that contain the function prototypes for the routines called by the main function.

```
/*
*****
/*
/*                               MAIN.C                               */
/*
/*                               The main program                       */
/*
/*
/*
/* Programmer:    P. Moylan and Robert Betz                          */
/* Last modified: April 20, 1999                                       */
/* Status:        Working                                              */
/*
/*
*****
#include "..\globalh\initunos.h" /* FROM InitUNOS IMPORT InitUNOS, GoUNOS */
```

Appendix A — UNOS-V2.0 TUTORIAL

[illegible]

The variables below are static variables used in the window C++ classes. The first contains the overall background colour of the screen. The second variable contains a flag to indicate whether the constructor for the screen class has been called. The variable prevents the constructor from reinitialising the screen more than once (since it is called each time a window is created). See the documentation on the screen handling routines for details on this.

```

/* Static variables used in the windowing routines */
backgrd_colour screen::scrn_backgrd_colour;
int screen::screen_created = FALSE;

/*
=====
|
| main
|
| This function is called to start the UNOS kernel
|
| Parameters:   none
|
| Returns:     nothing
|
=====
*/

void main (void)
{

```

The first function called in the main function is to create all the UNOS data structures. For example the semaphore data structures are created, the central table is allocated, etc. In addition any hardware that is required by the kernel is initialised, such as the timer interrupt rate. The kernel interrupt vectors are also initialised. See the section on the **initunos** module for more details.

```
/* Initialise the UNOS internal data structures */
init unos ();
```

The next function called is in the usertask module. Its function is to create all the user tasks in the system, and therefore is a point of significant modification by the user. If the function returns with a false value then there has been some problem creating a task

```
/* Now create the user tasks in the system */
    if (!create_user_tasks())
```

```
{  
    /* Enter here if there has been some problem creating the tasks */  
    cprintf("Tasks creation failure\r\n");  
    exit(EXIT_FAILURE);  
}/*if*/
```

*Everything should be set up if we reach this point in the code. Therefore start the operating system. The **go_unos** function sets up the null task prior to enabling interrupts so that the first time slice entry into the kernel can occur.*

```
/* Now ready to start UNOS */  
    go_unos ();  
}
```

The Initunos Module

The module is called early in the UNOS initialisation phase. Its function is to initialise the fundamental data structures of UNOS – i.e. it dynamically allocates storage for the UNOS heap, from which storage for semaphores and timers are then allocated.

```
#include <stdio.h>
#include <stdlib.h>      /* FROM stdlib IMPORT exit, atexit, EXIT_FAILURE */
#include <dos.h>
#include <conio.h>
#include <alloc.h>
#include <math.h>
#include ".\globalh\general.h"
#include ".\globalh\unos.h"
#include ".\globalh\hwpc.h"
#include ".\globalh\kbddrv.h"
#include ".\globalh\time.h"
#include ".\globalh\tasks.h"
#include ".\globalh\taskdefs.h"
#include ".\globalh\configos.h"

/*****
/*
/*
/*          University of Newcastle Operating System          */
/*
/*          ( UNOS )                                          */
/*
/*          Version 1.5                                       */
/*
/*          MAIN PROGRAM MODULE                               */
/*
/*          by                                                */
/*
/*          Robert Betz                                       */
/*          Department of Electrical and Computer Engineering */
/*          University of Newcastle                           */
/*
/*          ( Copyright 1989 )                                */
/*
/*
/*          Last edited: March 7, 1999                        */
/*
*****/

/*
HISTORY

27/3/89
Began typing in the preliminary version of a test system for the 'C' version
of UNOS. This initially consists of two tasks with no time slice interrupts
running. The task switches are to occur via direct operating system calls to
reschedule.
*/

/*****
/*
/*          TAILORING FOR COOK ISLANDS PROJECT              */
/*
/*          PJM 25/2/92: eliminated task1, task2, task3 from the demo program */
/*          as a first step towards building a version suitable for the Cook   */
/*          Islands project. In future should also eliminate task0, but I'm    */
/*          leaving it in for now in order to have something interesting        */
/*          happening on the screen until I have some further tasks running.   */
/*
/*          PJM 26/2/92: added InitialiseModules()           */
/*
/*          PJM 28/2/92: Split up the main procedure into two procedures called */
*****/
```

Detailed Example of the Use of UNOS

```
/* InitUNOS and GoUNOS, and renamed this module INITUNOS.C; that is,    */
/* the main program is no longer contained in this module.                */
/*                                                                           */
/* PJM 19/3/92: Eliminated the test task (task 0 or task 1, depending      */
/*           on which bit you read).                                       */
/*                                                                           */
/* RHM 15/6/92: Add an _setcursortype(_NORMALCURSOR) to the exit          */
/*           exit procedures.                                              */
/*                                                                           */
/*****/

/*
DESCRIPTION

This module contains the main program and the associated initialisation
routines. This would be the main module modified by a user as more tasks,
semaphores and communication buffers are added to the system.

*/
```

```
extern char pchKeyboardTask[];
```

```
static void initialisation (void);
static void software_initialise (void);
static void hardware_initialise (void);
void fvdRestore (void);
```

```
/***/
/*                                                                           */
/*                               EXTERNAL DECLARATIONS                      */
/*                                                                           */
/*****/
```

```
extern void null_task ( void* );
```

```
/***/
/*                                                                           */
/*                               GLOBAL DECLARATIONS                      */
/*                                                                           */
/*****/
```

The variables declared below are for the old interrupt routine vectors.

```
void (interrupt*pfvdOldKernelPositionVector)();
void (interrupt*pfvdOldTickInterruptRoutine)();
```

```
/* start of the memory pool */
char far* ptr_to_memory_pool = NULL;
```

```
/*=====*/
```

```
/***/
/*                                                                           */
/*                               HARDWARE RELATED DEFINITIONS              */
/*                                                                           */
/*****/
```

```
/*
=====
|
| fvdRestore
|
| This is the exit fucntion which is called to restore the interrupts related
| to UNOS operation - namely the kernel entry interrupt and the tick interrupt.
| In addition the screen is reset to a standard screen.
|
| Parameters      :      none
|
| Returns         :      nothing
|
=====
*/
```

*This function is called when the system is terminating. It is registered as an **atexit** function. It function is the restore the 8254 timer chip to it original programming and to restore the kernel and tick interrupt vectors back to their original values. This is to ensure that DOS can run properly when UNOS terminates. The keyboard interrupt is restored through another **atexit** function defined in the `kbdrv.cpp` module.*

```
void fvdRestore (void) {

    /* restore the timer back to its original mode */
    disable ();
    outputb (I8254_CTRL_REG, I8254_SQUARE_WAVE_MODE_COUNTER_0);
    outputb (I8254_COUNTER_0, 0xff);
    outputb (I8254_COUNTER_0, 0xff);

    /* Now restore the interrupt vector back */
    fpvdSetVector (KERNEL_ENTRY, pfvdOldKernelPositionVector);
    fpvdSetVector (8, pfvdOldTickInterruptRoutine);
    window (1, 1, 80, 25);
    textcolor (LIGHTGRAY);
    textbackground (BLACK);
    clrscr ();
    _setcursortype(_NORMCURSOR);

} /* end of fvdRestore */

/*-----*/

/*
=====
|
| hardware_initialise
|
| The function of this routine is to initialise all the hardware related
| items in the system.
|
=====
*/
```


*/

This function sets up various pieces of hardware used by the system. This function would normally be modified by the user if the hardware configuration and/or initialisation is changed.

```
void hardware_initialise ( void )
{
    unsigned int uin8254CountValue;

    /* now set up the 8254 timer so that the desired tick rate is obtained
    for the system.
    */
    uin8254CountValue = (unsigned int)ceil(1 / (1 /
        (double)(I8254_INPUT_FREQ) *
        (double)(TICK_INTERRUPTS_PER_SEC)));

    outportb (I8254_CTRL_REG, I8254_RATE_MODE_COUNTER_0);
    outportb (I8254_COUNTER_0, (unsigned char)(uin8254CountValue));
    outportb (I8254_COUNTER_0, (unsigned char)(uin8254CountValue >> 8));
}
```

/*-----*/

/*

```
=====
|
| software_initialise
|
| The function of this procedure is to initialise all the software entities in
| the operating system.
|
| Parameters : - none
|
| Entry via  : - initialisation
|
|=====
*/
```

The software initialisation function sets up various data structures used by the system.

```
void software_initialise ( void ) {
    int inI;

    /* set up all the OS main data structures */
```

*The function below is called to set up all the major data structures for the kernel. Most of the parameters passed to this function are initialised via the **#defines** at the top of the module.*

```
if ( !setup_os_data_structures ( KERNEL_ENTRY, ENTER_KERNEL_VALUE,
    NUM_OF_PRIORITIES, MAX_NUM_SEMAPHORES, MAX_NUM_OF_TASKS,
    ptr_to_memory_pool, MEMORY_POOL_SIZE ) ) {
```

Appendix A — UNOS-V2.0 TUTORIAL

```
    cprintf ("\n Problem setting up the OS data structures\n" );
    exit (EXIT_FAILURE);
} /* if */
```

*Now create the timers available for the user and for the system. The particular number of timers created is set by a user configured **#define**. What that number should be is basically a guess.*

```
/* now create the timers */
for (inI = 0; inI < MAX_NUM_TIMERS; inI++) {
    if (create_timer () == NULL) {
        cprintf ("\n\rProblem creating timers");
        exit (EXIT_FAILURE);
    } /* if */
} /* for */

/*
initialise the interrupt vector for the kernel entry
*/
```

We now disable interrupts to ensure that no interrupts can occur whilst the interrupt vectors are being changed. Note that the old values of the interrupt vectors are stored so that they can be returned at the conclusion of the kernel.

```
disable ( );

/*
set up the kernel entry interrupt
*/

pfvdOldKernelPositionVector =
    (void(interrupt*>()) fpvdSetVector (KERNEL_ENTRY, kernel);

/*
now set up the tick routine interrupt vector
*/
pfvdOldTickInterruptRoutine =
    (void (interrupt*())fpvdSetVector (8, tick);

/* now set up an exit routine */

atexit (fvdRestore);
} /* end of software_initialise */

/*-----*/

/*
=====
|
|  initialisation
|
|
```

```
| The function of this procedure is to initialise all the software and
| hardware entities in the system related to the operation of the OS software.
| Note that the initialisation of the software data structures related to all
| the user tasks is carried out in the user tasks.
|
| Parameters : - none
|
| Entry via  : - main routine
|
|=====
*/
```

The entrance point for the hardware and software initialisation functions.

```
void initialisation ( void ) {
    software_initialise ( );
    hardware_initialise ( );
}

/*****
/*
/*                               MAIN PROGRAM                               */
/*
/*                               */
*****/
```

This is the main entrance point to the module. This routine is called from the main program.

```
void init_unos ( void ) {

    /* Carries out all of the UNOS initialisation EXCEPT for creating    */
    /* the null task and running the system.                               */

    /* have to get some memory from DOS before doing anything */
```

This is where DOS is asked for some memory. By getting a pool of memory in this fashion we ensure that UNOS will not overwrite any of the data structures used by DOS. Therefore DOS should run OK upon exit from UNOS. Note that this memory pool is used for the allocation of all the data structures used by the UNOS as well as any dynamic memory allocates requested by the user during user code execution.

```
ptr_to_memory_pool = (char far*)farmalloc ( MEMORY_POOL_SIZE );

disable ();

/* Firstly initialise all the software data structures and the    */
/* necessary hardware in the system.                               */

initialisation ();

/* Now set up system level tasks */

/* Set up the keyboard task */
```

A couple of tasks are created now - these tasks allow UNOS to use the keyboard.

Appendix A — UNOS-V2.0 TUTORIAL

One of the tasks carries out the translation of the scan codes from the keyboard to ascii codes used by the applications. The other task is a multiplexer task that allows multiple tasks to interface to the keyboard. These tasks use a protocol to request that they be connected to the keyboard.

```
create_task ( pchKeyboardTask, PRIORITY_1, 0, TASK_RUNNABLE,
             PRIORITY_Q_TYPE, 0, TASK_STACK_SIZE,
             MESS_Q_SIZE_KEYBD_TASK, MESS_SIZE_KEYBD_TASK,
             fvdInitScanCodeTranslatorTask,
             fvdScanCodeTranslatorTask, NULL );

/* Set up an associated keyboard multiplexer task. This task has the job
of sending characters from the keyboard to intercepting tasks.
*/
create_task ( pchKeyboardMultiPlexerTask, PRIORITY_1, 0, TASK_RUNNABLE,
             PRIORITY_Q_TYPE, 0, TASK_STACK_SIZE,
             MESS_Q_SIZE_KB_MULTI_TASK, MESS_SIZE_KB_MULTI_TASK,
             fvdInitKbMultiplexerTask,
             fvdKbMultiplexerTask, NULL );

} /* end of init_unos */

/*-----*/

/*
=====
|
| go_unos
|
| This is the function called from the main module of the program. When this
| routine is called it is assumed that all the users tasks have been created and
| the system is nearly ready to start. The last remaining thing carried out by
| this function is to create the null_task for the system and start this task.
| During the initialisation phase of the null_task the interrupts for the system
| are enabled.
|
=====
*/
```

This function is called when everything is set up. Its function is to create the null task and then enable interrupts to allow time slicing to start.

```
void go_unos (void)
{
    /* Creates the null task, and starts UNOS. It is assumed that      */
    /* init_unos has already executed successfully.                  */

    /* Set up the null task */

    create_task ( name_null_task, NULL_PRIORITY,
                  0, TASK_RUNNABLE, DONOT_Q_TYPE, 0, 0, 0, 0,
                  NULL, null_task, NULL );

    /* Now start the null task and make all the other tasks start up. */
```

The start_tasks function starts executing the null task. As well is sets a flag indicat-

ing that the tasks are started and enables the interrupts so that the time slice scheduler will be called.

```

    start_tasks ( null_task );
} /* end of go_unos */

```

```
//<<<<<<<<<<<<<<<<< END OF FILE >>>>>>>>>>>>>>>>>
```

Usertask Module

As the name implies this module is the module that all the user tasks are created in. Therefore this module is customized for each application of the operating system.

The module is largely self explanatory – it essentially consists of a number of calls to the **create_task** function in the kernel, passing in the appropriate parameters to set up each task.

```
#include "..\globalh\unos.h"
#include "..\globalh\usertask.h"
#include "..\globalh\taskdefs.h"
#include "..\globalh\serinit.h"
#include <stdlib.h>
/*****
/*
/*
/*          University of Newcastle Operating System          */
/*
/*          UNOS TASK CREATION MODULE                        */
/*
/*          by                                                */
/*
/*          Robert Betz                                       */
/*          Department of Electrical and Computer Engineering */
/*          University of Newcastle                           */
/*
/*          ( Copyright 1989,90,91,92,93,94,95,99 )           */
/*
/*
/*
/*
*****/

/*
DESCRIPTION

This modules contains the coded required to create all the user tasks in the
system.

NB: The total number of tasks should not exceed the MAX_NUM_TASKS define in
the init_unos routine.
*/

/*----- Define the user task names -----*/
/* each task in the system is given a unique name which is used by
other tasks as the mailing address when using the mailbox mechanism.
The name can be of any length up to the limit set by the 'C' compiler
for string variables. NOTE: it is the string name which is used as the
address by the mail box routines.
*/

The declarations of the character strings below is very important. These declara-
tions define the name of each task. It is the address of these character strings that
become the addresses of the mailboxes for the task (not the character string itself).
The character strings are filled with a meaningful name for the task to aid in
debugging. When looking via a debugger at the task control blocks (tcbs) of the
system, one can immediately see what tcb one is looking at by looking at the task
name component of the tcb structure – one will see the name string.

char name_task0[] = "Task 0 - test task 0";
char name_task1[] = "Task 1 - test task 1";
char name_task2[] = "Task 2 - test task 2";

/*----- Definitions required for task creation -----*/
```

```
extern void task0 ( void* );
extern void init_task0 ( void );
extern void task1 ( void* );
extern void task2 ( void* );
extern void init_task1 ( void );
extern void init_task2 ( void );

#define MESS_Q_SIZE_TEST1_TASK 8
#define MESS_SIZE_TEST1_TASK 80

/*
=====
|
| create_user_tasks
|
| This function is called from the main function. It contains all the calls
| to create_task to create all the user tasks in the system.
|
| Parameters:    none
|
| Returns:      TRUE if the task creation is successful
|              FALSE if the task creation is not successful
|
=====
*/

BOOL create_user_tasks(void)
{
```

Note that here we are calling a function in `serinit.cpp` to create and initialise the serial channels to be used by the kernel. These serial routines are set up for the IBM PC or compatibles. Support for other UART types is also provided in the module, but they have not been tested.

```
/* Create the system serial channels */
if (!fintSerialChannelInitialise())
{
    return FALSE;
}/*if*/
```

*Now enter here where all the user tasks are created. If the call to `create_task` is successful then the routine returns a **TRUE**, else it returns a false. The parameters passed to these routines are detailed in the UNOS reference manual, but it may be beneficial to go through them at this time.*

The `create_task` parameters are (in the order of declaration):

- **char *task_name_ptr:** This is a pointer to the task name string for the task to be created.
- **unsigned char priority_of_task:** This number contains the priority of the task. The priorities are in a number of **#defines** to make them simpler to use. Note the priority 1 is the highest priority.
- **int task_tick_delta:** This parameter contains the number of clock ticks that are added or subtracted from the standard number of clock ticks before a time slice entry for the kernel. This location effectively defines a

micro level of priority amongst tasks of the same priority.

- **unsigned char status_of_task:** *This parameter specifies the initial state of a task. The possibilities are: **task_runnable** – task is on a priority queue, **task_blocked** – task is on a semaphore queue and is therefore blocked. These equates are defined in “general.h”.*
- **unsigned char q_type:** *This location determines the queue that the task is placed on initially. The possibilities are: **PRIORITY_Q_TYPE** – put task on the priority queue whose priority is one of the other parameters; **SEMAPHORE_Q_TYPE** – place on a semaphore queue, the semaphore number is passed in as the next parameter; **DONOT_Q_TYPE** – don’t place on any queue (used for the null task).*
- **unsigned int task_stk_size:** *This location is used to determine the size of the tasks stack. It can be set individually for each task, or one can use the system wide generic value of **TASK_STACK_SIZE**, which is defined in “taskdefs.h”.*
- **unsigned int mess_q_size:** *This is the length of the mail box queue – i.e. it refers to how many messages the queue can hold before filling up.*
- **unsigned int mess_size:** *This location contains the size of each individual message that can be stored in the mail box of the task.*
- **void (*init_task) (void):** *This is the pointer to the function that will be called during the task creation phase. This function is typically used to initialise any variables that should exist prior to the task beginning to execute. A typical example is if a semaphore needs to exist before the system starts. If there is no initialisation function then this pointer is a **NULL_PTR**.*
- **void (*task) (void*):** *This is the pointer to the function that is the beginning of the task. This is the function that is called when the system first starts up the task. Typically this function will never return.*
- **void* local_var_ptr:** *This is a pointer to an undefined location. This pointer can be initialised to an arbitrary data structure which will be used in the task. This pointer would be cast to the correct data type in the task function. If there is no data to be passed then this pointer is set to a **NULL_PTR**.*

```
/* Test task 1 */
if (!create_task ( name_task0, PRIORITY_1, 0, TASK_RUNNABLE, PRIORITY_Q_TYPE,
                  0, TASK_STACK_SIZE, MESS_Q_SIZE_TEST1_TASK,
                  MESS_SIZE_TEST1_TASK, init_task0, task0, NULL ))
{
    return FALSE;
}/*if*/

if (!create_task ( name_task1, PRIORITY_4, 0, TASK_RUNNABLE, PRIORITY_Q_TYPE,
                  0, TASK_STACK_SIZE, MESS_Q_SIZE_TEST1_TASK,
                  MESS_SIZE_TEST1_TASK, init_task1, task1, NULL ))
{
    return FALSE;
}/*if*/

if (!create_task ( name_task2, PRIORITY_7, 0, TASK_RUNNABLE, PRIORITY_Q_TYPE,
```



```
        0, TASK_STACK_SIZE, MESS_Q_SIZE_TEST1_TASK,  
        MESS_SIZE_TEST1_TASK, init_task2, task2, NULL ))  
    {  
        return FALSE;  
    }/*if*/  
  
    return TRUE;  
}  
/*end create_user_tasks*/
```

Task Modules

This section contains the listing of all the tasks in the demonstration system. Note that this code does not do anything much, but it does serve to demonstrate some of the features of the operating system.

Task0 Module

This modules demonstrates the following aspects of UNOS tasks:

- (a) Use of the simple windowing system.
- (b) Connection of a task to the keyboard multiplexer task.
- (c) Use of the non-blocking SEND_MESS_NB macro to send mail box messages.
- (d) The use of the command decoder and its associated data structures and handling routines.
- (e) The use of lock and unlock primitives for mutual exclusion in critical sections.

As with the previous section the code shall be annotated to explain its operation.

```
#include <string.h>
#include <stdlib.h>
#include "..\globalh\general.h"
#include "..\globalh\unos.h"
#include "..\globalh\pcscrn.h"
#include "..\globalh\tasks.h"
#include "..\globalh\kbddrv.h"
#include "..\globalh\comdec.h"
#include "..\globalh\task0.h"

//=====//
//                                                    //
//                                                    //
//                      Demonstration Task 0          //
//                                                    //
//                      by                            //
//                                                    //
//                      Robert Betz                   //
//                      Department of Electrical and Computer Engineering //
//                      University of Newcastle        //
//                      Copyright (1999)              //
//                                                    //
//=====//

/*
DESCRIPTION

This is a simple task that does not do anything. Its purpose is to demonstrate
a number of features of the UNOS operating system.

*/

// LOCAL FUNCTION PROTOTYPES

static void fvd_kb_char_handler(dec_input_struct* dec_st);
static void fvd_illegal_kb_char_handler(dec_input_struct* dec_st);
static void fvd_task1_char_handler(dec_input_struct* dec_st);
static void fvd_illegal_task1_char_handler(dec_input_struct* dec_st);
static void fvd_com2_char_handler(dec_input_struct* dec_st);
static void fvd_illegal_com2_char_handler(dec_input_struct* dec_st);
static void fvd_illegal_task_handler(dec_input_struct* dec_st);
```

Declare a global **lock** data structure. Note that it is **static** so that it is global only within this module.

```
static LOCK lk_count;
static unsigned long ulg_count = 0;
```

In this module the windows to be created are created as global objects (within the module). One could also create them as local objects within the main function of the task. However, one could only use the windows from this function if this was done. See the chapter on the simple window system for an explanation of the parameters passed to these routines.

```
// Now create the three winds on the screen for this task.
// NB The borders around the windows have to be accounted for when butting
// the windows. The size passed to these routines is the actual active size
// of the windows. If the window has a border then one has to add two
// characters to the width and height.

// Window for keyboard characters
static window wind_kb(1, 1, 20, 10, DOUBLE_LINE, white_char, black_backgrd,
    "Tk 0: KB chrs", white_char, light_gray_backgrd);

// Window for characters from task 1
static window wind_t1(24, 1, 54, 10, SINGLE_LINE, light_blue_char,
    black_backgrd, "Tk 0: Task 1 chrs", light_red_char,
    light_gray_backgrd);

// Window for characters from serial com2
static window wind_c2(58, 1, 78, 10, NO_LINE, white_char, black_backgrd,
    "Tk 0: Com2 chrs", white_char, light_gray_backgrd);

//-----
```

*The section below sets up all the data structures for the command decoder. The command decoder is table driven, therefore there is necessarily a large number of tables. There are several layers of tables due to the reentrant nature of the command decoder. The tables immediately below are the lowest level of the tables. They are the tables that finally vector to the handling routines to carry out some action. These tables are arranged in groups of two – one table of the table of valid data at this point in time (denoted as the **vdt** tables), and the other companion table contains the vectors to the functions that carry out the handling of a particular piece of data (denoted as the **vt** tables).*

*Note the use of the **CHAR_RANGE** macro in the vdt tables to denote character ranges, and the **ASC_END_TABLE** to denote the end of the table. For more specifics on the use of the command decoder refer to the appropriate Appendix.*

```
// Now define the command decoder structures. This is used in this demonstration
// program to decode characters arriving from the keyboard and another task.

// Now set up the decoding tables for the various decoders

// Define the decode structures for each of the legal tasks communicating with
// this task
```

```
// Decoder table for the keyboard - basically allows any of the normal printable
// characters to be input

unsigned char vdt_kb[5] =
{
    CHAR_RANGE(' ', '~'),
    CR,
    LF,
    ASC_TABLE_END
};

void (*vt_kb[4])(dec_input_struct*) =
{
    fvd_kb_char_handler,
    fvd_kb_char_handler,
    fvd_kb_char_handler,
    fvd_illegal_kb_char_handler
};

// Decoder tables for data arriving from task 1

unsigned char vdt_task1[5] =
{
    CHAR_RANGE(' ', '~'),
    CR,
    LF,
    ASC_TABLE_END
};

void (*vt_task1[4])(dec_input_struct*) =
{
    fvd_task1_char_handler,
    fvd_task1_char_handler,
    fvd_task1_char_handler,
    fvd_illegal_task1_char_handler
};

// Decoder tables for data arriving from serial channel 2

unsigned char vdt_com2[5] =
{
    CHAR_RANGE(' ', '~'),
    CR,
    LF,
    ASC_TABLE_END
};

void (*vt_com2[4])(dec_input_struct*) =
{
    fvd_com2_char_handler,
    fvd_com2_char_handler,
    fvd_com2_char_handler,
    fvd_illegal_com2_char_handler
};

// Decoder table if there is an illegal task sending info
unsigned char vdt_illegal_task[1] =
{
    ASC_TABLE_END
};

void (*vt_illegal_task[1])(dec_input_struct*) =
{
    fvd_illegal_task_handler,
};
```

The next highest level of decoding tables appear next in the code. These are known as the decoder input structures, and there is one for each of the decoding streams. Each decoding stream (which are all separate from each other but reentrantly use the decoder) needs to have its own data structure that carries the current context of the decoding operation. See the Appendix on the command decoder for the details of what the components are in this structure.

```
// Decoder for keyboard
static dec_input_struct keybd_decoder_struct =
{
    ASCII_DECODING,
    0,
    vdt_kb,
    vt_kb,
    NULL_PTR,
    NULL_PTR
};

// Decoder for task 1 communications
static dec_input_struct task1_decoder_struct =
{
    ASCII_DECODING,
    0,
    vdt_task1,
    vt_task1,
    NULL_PTR,
    NULL_PTR
};

// Decoder for serial channel 2
static dec_input_struct ch2_serial_chan_decoder_struct =
{
    ASCII_DECODING,
    0,
    vdt_com2,
    vt_com2,
    NULL_PTR,
    NULL_PTR
};

// Decoder if an illegal task communicates with this task
static dec_input_struct illegal_task_decoder_struct =
{
    ASCII_DECODING,
    0,
    vdt_illegal_task,
    vt_illegal_task,
    NULL_PTR,
    NULL_PTR
};

//-----
```

The tables that follow are the second highest level of the command decoding system. These tables decode if the command to be decoded is from a valid task that is allowed to send messages to this task. If it is valid then the correct decoder input structure is selected to carry the decoding to the next level. If the sending task is not valid then an error handler can be called.

```
// Set up the address decoding tables.
```

Appendix A — UNOS-V2.0 TUTORIAL

```
static char huge* valid_task_names[4] =
{
    pchKeyboardMultiPlexerTask,
    name_task1,
    ch_1_rx,
    NULL_PTR
};

static dec_input_struct* task_decode_structs[4]=
{
    &keybd_decoder_struct,
    &task1_decoder_struct,
    &ch2_serial_chan_decoder_struct,
    &illegal_task_decoder_struct
};
```

The table below is the highest level of the decoding system It contains pointers to the tables that will decode the valid sending tasks.

```
// Now set up the tasks address decoding structure
static addr_struct task1_address_map =
{
    NULL_PTR,
    valid_task_names,
    task_decode_structs,
    NULL_PTR,
    0
};

//-----
```

*There are a number of functions below. These functions are the handling functions that are vectored to from the vector tables above. The input to the functions is a pointer to a **dec_input_struct**. Usually the handling functions will modify the table pointers in this structure as the command context changes (although this may not happen all the time). The **dec_input_struct** always contains the character that has been decoded by the command decoder.*

```
//
// DECODER HANDLER ROUTINES
//

//=====
//
// fvd_illegal_task_handler
//
// This function if the task sending a message to this task is not a legal
// task. This function could signify an error message, set some error locations
// or whatever. The current version simply ignores the problem and does
// nothing
//
//=====

static void fvd_illegal_task_handler(dec_input_struct* pst_dec_st)
{
    (dec_input_struct*)pst_dec_st;
}

//fvd_illegal_task_handler
```

```
//-----

//=====
//
// fvd_kb_char_handler
//
// This function is called via the vt_kb table. Its function is to handle
// the response for all the printable characters from the keyboard.
//
//=====

static void fvd_kb_char_handler(dec_input_struct* pst_dec_st)
{
    char str[2] = {0,0};
    // print the character out in the appropriate window
    str[0] = pst_dec_st->data_byte;
    wind_kb.print(str);

    // Send the character back down the serial channel line
    SEND_MESS_NB(str, 1, ch_1_tx);

    // One would normally change the valid data table and the vector table
    // here to values that would decode the next character in the command
    // context. However, in this particular case the context does not change
    // since we are simply going to continue to print out the characters that
    // arrive on the screen.
}

//fvd_kb_char_handler

//-----

//=====
//
// fvd_illegal_kb_char_handler
//
// This function is only executed if an illegal character is detected from
// the keyboard input stream. These would be the control characters and the
// DEL character. The routine simply dumps the character - i.e. it does
// nothing.
//
//=====

static void fvd_illegal_kb_char_handler(dec_input_struct* pst_dec_st)
{
    (dec_input_struct*)pst_dec_st;
}

//fvd_illegal_kb_char_handler

//-----
```

```
//=====
//
// fvd_task1_char_handler
//
// This function is called via the vt_task1 table. Its function is to handle
// the response for all the characters sent from task1.
//
//=====

static void fvd_task1_char_handler(dec_input_struct* pst_dec_st)
{
    char str[2] = {0,0};
    static unsigned int i;

    // print the character out in the appropriate window
    str[0] = pst_dec_st->data_byte;

    if (str[0] == '\n')
    {
        switch(i%2)
        {
            case 0:
                wind_t1.set_string_colour(light_red_char);
                i++;
                break;

            case 1:
                wind_t1.set_string_colour(black_char);
                i++;
                break;

            default:
                break;
        }/*switch*/
    }/*if*/

    wind_t1.print(str);

    // One would normally change the valid data table and the vector table
    // here to values that would decode the next character in the command
    // context. However, in this particular case the context does not change
    // since we are simply going to continue to print out the characters that
    // arrive on the screen.

}//fvd_task1_char_handler


//-----

//=====
//
// fvd_illegal_task1_char_handler
//
// This function is only executed if an illegal character is detected from
// the task1 input stream. These would be the control characters and the
// DEL character. The routine simply dumps the character - i.e. it does
// nothing.
//
//=====
```



```
static void fvd_illegal_task1_char_handler(dec_input_struct* pst_dec_st)
{
    (dec_input_struct*)pst_dec_st;
}

//-----

//=====
//
// fvd_com2_char_handler
//
// This function is called via the vt_com2 table. Its function is to handle
// the response for all the characters sent from com2.
//
//=====

static void fvd_com2_char_handler(dec_input_struct* dec_st)
{
    char str[2] = {0,0};
    // print the character out in the appropriate window
    str[0] = dec_st->data_byte;
    wind_c2.print(str);

    // One would normally change the valid data table and the vector table
    // here to values that would decode the next character in the command
    // context. However, in this particular case the context does not change
    // since we are simply going to continue to print out the characters that
    // arrive on the screen.
}

//-----

//=====
//
// fvd_illegal_com2_char_handler
//
// This function is only executed if an illegal character is detected from
// the com2 input stream. These would be the control characters and the
// DEL character. The routine simply dumps the character - i.e. it does
// nothing.
//
//=====

static void fvd_illegal_com2_char_handler(dec_input_struct* pst_dec_st)
{
    (dec_input_struct*)pst_dec_st;
}

//-----
```

```
//=====
//
// fulg_return_count
//
// This function provides access by other routines to the counter being
// incremented in this task.
//
// Note that it uses LOCKS for critical section protection.
//
//=====
```

This function allows semaphore controlled access to the ulg_count variable. It uses lock primitives to provide this control.

```
unsigned long fulg_return_count(void)
{
    unsigned long ulg_temp;

    lock(lk_count);
    ulg_temp = ulg_count;
    unlock(lk_count);

    return ulg_temp;
} //ful_return_count
```

```
//-----
```

```
//
// MAIN TASK ROUTINE
//
```

```
//=====
//
// task0
//
// This is the function that is called to start the task. The function is
// an infinite loop, so it never returns.
//
//=====
```

```
void task0 ( void* local_var_ptr )
{
```

This function is the task itself.

```
    char ach_message[30];

    /* Put the next line in to prevent a warning
    */
    (void)local_var_ptr;

    // initialise the mess_ptr section of the task1_address_map
```

```
task1_address_map.mess_ptr = (unsigned char*)ach_message;
enable();
```

Initialise some values related to the windows used by this task.

```
wind_kb.go_window_xy(0,0);
wind_t1.go_window_xy(0,0);
wind_c2.go_window_xy(0,0);
wind_kb.set_string_colour(yellow_char);
wind_t1.set_string_colour(black_char);
wind_c2.set_string_colour(light_red_char);
```

The macro below is used to send messages between tasks. In this particular case the message is being sent to the keyboard multiplexer task requesting that this task receive characters being typed on the keyboard. The characters are sent to this tasks mail box.

```
/* Now connect to the keyboard
*/
SEND_MESS_NB(CONNECT_KB, 1, pchKeyboardMultiPlexerTask);
```

The message below is used to connect this task to the serial channel. This means that the serial channel task (for channel 1 in this case) will send all characters received to this tasks mail box.

```
/* Now connect to the serial task receiver */
SEND_MESS_NB('C',1, ch_1_rx);
```

The while loop below is the main loop of the task. This is an infinite loop, therefore control never leaves this loop.

```
// Enter the infinite loop of the task.
while(1)
{
```

The decode_mbx_mess routine below waits for a character to arrive in this tasks mail box and then it decodes it using the decoder tables at the top of this module. It should be noted that the information arriving in the mail box can come from several tasks in the system, and the decoder maintains several decoding contexts associated with the current context of decoding data from these different sources.

Until data arrives in the mail box this task is asleep.

```
    decode_mbx_mess(&task1_address_map, 0);

    // Now increment the number of times around the loop
    lock(lk_count);
    ulg_count++;
    unlock(lk_count);

} /* while */
} /* end of task0 */
```

//-----

```

//=====
//
// init_task0
//
// This function is called during the task initialisation process. It is
// used to set up anything for task 0 that should exist prior to any tasks
// running in the system.
//
//=====

```

This function is called during the task creation process. It can carry out any initialisation required prior to the starting of the task.

```
void init_task0 ( void )
{

    // Create a mutual exclusion lock
    lk_count = create_lock();


} /* init_tasks */
```

[illegible]

Task1 Module

This task module is much the same as the previous task module in general layout. The details of what it does are different.

```
//=====//
//                                                    //
//                                                    //
//                Demonstration Task 1                //
//                                                    //
//                by                                    //
//                                                    //
//                Robert Betz                          //
//                Department of Electrical and Computer Engineering //
//                University of Newcastle              //
//                Copyright (1999)                    //
//                                                    //
//=====//

/*
DESCRIPTION

This is a simple task that does not do anything. Its purpose is to demonstrate
a number of features of the UNOS operating system. See the task function
for a better description.

*/

    Denote a variable of the semaphore type. Note that this does create a semaphore,
    but simply generates an instance of the location that is used to store the semaphore
    handle once created.

static SEMAPHORE timer_semaphore;

// Create a window to write the count value read from task 0 in.
static window wind_task1(1, 14, 10, 23, DOUBLE_LINE, white_char, black_backgrd,
    "Tk 1", white_char, light_gray_backgrd);

//=====//
//
// task1
//
// The main function of the task.
//
// This is a very basic task that simply sleeps on a timed wait and then
// wakes up and sends a message to task0. It then gets the value from
// a shared location (protected by locks) and writes to the screen
// in its own window.
//
//=====//

void task1 ( void* local_var_ptr ) {

    /* Put in the following line to prevent a warning
    */
    (void)local_var_ptr;

    unsigned long ulg_count = 0;
```

Appendix A — UNOS-V2.0 TUTORIAL

```
char ach_strg[30];
enable();
```

```
wind_task1.set_string_colour(black_char);
```

As with task 0 we enter an infinite loop for the main task section.

```
while ( 1 ) {
```

Below shows the use of a timed wait semaphore. The parameters passed to the function are the semaphore to wait on and the number of clock ticks to wait. Because the semaphore was created with a value of zero then the task will always become blocked on the semaphore.

```
    // wait for a while
    timed_wait(timer_semaphore, 10);
```

The send_mess macro below is sending a message to task 0.

```
    // send a message to task 0
    SEND_MESS("Task1 calling\r\n", sizeof("Task1 calling\r\n"),
              name_task0);

    // Now get the count value from task0 and print it out
    ulg_count = fulg_return_count();
    sprintf(ach_strg, "%u\r\n", ulg_count);
    wind_task1.print(ach_strg);

} /* while */
} /* end of task1 */
```

```
//-----
```

```
//=====
//
// init_task1
//
// Initialise task1 prior to any tasks running. Called during the task
// creation process.
//
// Note that this initialisation creates a semaphore prior to the
// task running. This is not absolutely necessary since one could
// create the semaphore when the task first starts in this case
// without anything funny happening.
//
//=====
```

```
void init_task1 ( void ) {
```

The line below actually creates an instance of the semaphore. The handle for the semaphore is stored in the location we defined at the beginning of the module. Note that the semaphore value is initialised to a value of zero.

```
    timer_semaphore = create_semaphore ( 0 );
} /* init_task1 */
```

```
#include <dos.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "..\globalh\general.h"
#include "..\globalh\unos.h"
#include "..\globalh\pcscrn.h"
#include "..\globalh\kbddrv.h"
#include "..\globalh\tasks.h"
#include "..\globalh\task0.h"
```

```
//=====//
//
//
//          Demonstration Task 2
//
//
//          by
//
//
//          Robert Betz
//
//          Department of Electrical and Computer Engineering
//
//          University of Newcastle
//
//          Copyright (1999)
//
//
//=====//
```

*** /**

```
static unsigned int timed_wait_semaphore;
```

/*-----*/

A-31

This is the main task function of the module.

```
void task2 ( void* local_var_ptr )
{
    (void)local_var_ptr;
```

Set up the window to be written into.

```
window wind_t2(16, 14, 40, 23, DOUBLE_LINE, white_char, black_backgrd,
               "Tk 2", light_blue_char, light_gray_backgrd);
unsigned int i =0;
unsigned long ulg_count;
char ach_strg[30];
```

```
enable();
```

```
while ( 1 )
{
    timed_wait(timed_wait_semaphore, 32);
```

The following switch statement simply allows the writing of different coloured lines to the screen on each run through the task.

```
switch(i%2)
{
    case 0:
        wind_t2.set_string_colour(black_char);
        i++;
        break;

    case 1:
        wind_t2.set_string_colour(yellow_char);
        i++;
        break;

    default:
        break;
}/*switch*/

wind_t2.print("This is task 2 talking");
wind_t2.print('\r');
wind_t2.print('\n');
```

*The task accesses a shared variable via the function **fulg_return_count** and prints it out onto the screen. This function provides protection of the critical section via a **lock** primitive.*

```
    ulg_count = fulg_return_count();
    sprintf(ach_strg, "Count is: %u\r\n", ulg_count);
    wind_t2.set_string_colour(light_blue_char);
    wind_t2.print(ach_strg);

} /* while */
} /* end of task2 */

/*-----*/
```



```
void init_task2 ( void )
{
    timed_wait_semaphore = create_semaphore(0);
} /* init_task1 */
```

A-33

Header Files

There are a number of header files used in UNOS. The main ones are alarm.h, comdec.h, fpx.h, general.h, initmodu.h, kbdrv.h, pccolour.h, pcscrn.h, serial.h, serinit.h, serint.h, unos.h, user-task.h, unosasm.h. Clearly there are a lot of header files. I will not attempt to list all of them, but instead will list the major ones and indicate what the function of the other header files are.

The main header file is unos.h. This file would be included into all tasks in the system that are accessing UNOS functions.

unos.h

```
#ifndef __UNOS_H_
#define __UNOS_H_
#include "..\globalh\general.h"

/*****
/*
/*
/*          UNOS HEADER FILE          */
/*
/*          by
/*
/*          Robert Betz
/*          Department of Electrical Engineering and Computer Science
/*          University of Newcastle
/*
/*          ( Copyright 1989 )
/*
/*
*****/

/*
HISTORY

This file was created for version 1.5 of UNOS.

Updated for version 2.0 on the 15/1/95 by Robert Betz

Minor modifications 14/2/99 by Robert Betz

*/

/*
DESCRIPTION

This file contains the function definitions for the UNOS kernel file. In
addition definitions used in the kernel module are defined.

*/

/* LOCK related definitions */

/*----- Define the lock primitive handle as used by the user -----*/
#define LOCK lock_type*

/* Define the states of the lock */
#define LOCKED TRUE
#define UNLOCKED FALSE

/* Semaphore definition - this is simply to make the software more readable */
#define SEMAPHORE unsigned int
```

```

/*****
/*
/*          DATA STRUCTURE TYPES          */
/*
/*
*****/

/*----- Lock structure type definition -----*/
typedef
    struct lock_struct
    {
        int locked;
        struct tsk_ctrl_blk *holder;
    } lock_type;


/*----- timer structure type definition -----*/

typedef
    struct time_struct
    {
        unsigned char status;
        unsigned char type;
        unsigned long init_time_cnt;
        unsigned long delta_time_cnt;
        struct time_struct* prev_timer_ptr;
        struct time_struct* next_timer_ptr;
        void ( *timer_handler )( void* );
        void* timer_handler_data;
    } timer_struct;


/*----- task control block structure type definition -----*/

typedef
    struct tsk_ctrl_blk
    {
        unsigned int task_num;
        char *task_name_ptr;
        unsigned char priority;
        unsigned int ticks_before_kernel_entry;
        unsigned int ticks_to_go;
        unsigned char task_status;
        struct tsk_ctrl_blk* prev_task_ptr;
        struct tsk_ctrl_blk* next_task_ptr;
        struct time_struct* timer_ptr;
        char timeout_flag;
        char q_type;
        LOCK waiting_for;
        unsigned int cent_tab_index;
        unsigned int sema_blocked_on;
        unsigned int task_stkbase;
        unsigned int task_stkptr;
        unsigned int task_baseptr;
        unsigned char fp_save_area [ 150 ];
    } task_control_block;


/*----- Kernel Control Block (kcb) definition -----*/

typedef
    struct kcb_struct
    {
        unsigned char entry_type;
        unsigned int semaphore_index;
    }

```

Appendix A — UNOS-V2.0 TUTORIAL

```
        unsigned int preemption_status;
        unsigned int dectime_signal_flag;
        unsigned int task_number;
        unsigned char new_priority;
        kcb_struct* kcb_ptr;
    } kcb_structure;

/*----- semaphore structure type definition -----*/

typedef
    struct sema_struct
    {
        char *creating_taskname_ptr;
        unsigned int semaphore_value;
        unsigned int semaphore_multiplier;
        unsigned int semaph_bgn_q_index;
        void ( *semaph_queue_handler )(int, kcb_structure*);
    } semaphore_struct;

/*****
/*
/*
/*          MAIL BOX RELATED DEFINITIONS
/*
/*
*****/

/*----- TASKNAME_MAP STRUCTURE DEFINITION -----*/
/* this structure contains the connection between a task name and the
task number associated with a task. The structure also contains a pointer
to allow the structures to be placed in a linked list so that chained
hashing can be implemented.
*/
typedef
    struct taskname_struct
    {
        char *task_name_ptr;
        unsigned int task_number;
        struct taskname_struct *nxt_taskname_map;
    } taskname_map;

/* Define the envelope structure */

typedef
    struct
    {
        char *rtn_addr_ptr;
        unsigned int sender_pri;
        unsigned int mess_lgth;
        char* message_ptr;
    } envelope;

/* Define the mail box structure */

typedef
    struct
    {
        unsigned int mess_addr;
        char mbx_type;
```

```
    unsigned int q_size;
    unsigned int mess_size;
    unsigned int spce_avail_sema;
    unsigned int mess_avail_sema;
    unsigned int free;
    unsigned int used;
    unsigned int get_ptr;
    unsigned int put_ptr;
    char qik_mess_flag;
    envelope* qik_mess_ptr;
    envelope* mess_q_ptr;
} mbx;

/*****
/*
/*          MEMORY ALLOCATION STRUCTURES          */
/*
*****/

/*----- HEADER STRUCTURE DEFINITION -----*/
/* The structure defined below is the header which is stored at the
beginning of every allocated block and every free block of storage. The
structure is defined as a union so that the header will be aligned according
to the most restrictive alignment type. The header structure contains a
pointer to the next free block and the size of the current free block.
The list of free blocks is arranged in order of memory address. The linked
list is circular in nature with the last free block being linked back to
the first free block.
*/

typedef
    int align; /* forces alignment to int boundaries */

union header_struct
{
    struct
    {
        union header_struct huge* nxt_blk_ptr; /* pointer to next free block
        */
        unsigned long blk_size;                /* size of this free block */
    } header;
    align x; /* force alignment of blocks */
};

typedef
    union header_struct blk_header;

/*****
/*
/*          DEFINITIONS FOR MAIL BOXES          */
/*
*****/

/* Indication that a non-blocking send_mess has been done on a full mail box */
#define FULL_MBX (-1)

/* Definitions of types of send_mess's */
```

Appendix A — *UNOS-V2.0 TUTORIAL*

```
#define BLOCK_SM 0      /* Blocking send_mess */
#define NO_BLOCK_SM 1  /* Non-blocking send_mess */

/* Define some macros for sending messages */

/* Normal send message macro */
#define SEND_MESS(mess_ptr, mess_lgth, mess_addr_ptr) \
    send_mess((unsigned char*)mess_ptr, (unsigned int)mess_lgth, \
              (char*)mess_addr_ptr, BLOCK_SM)

/* Non-blocking send message macro */
#define SEND_MESS_NB(mess_ptr, mess_lgth, mess_addr_ptr) \
    send_mess((unsigned char*)mess_ptr, (unsigned int)mess_lgth, \
              (char*)mess_addr_ptr, NO_BLOCK_SM)

/* OTHER DEFINITIONS */

/*----- Queue types used in the task creation routine -----*/

#define PRIORITY_Q_TYPE 0
#define SEMAPHORE_Q_TYPE 1
#define DONOT_Q_TYPE 2

/*----- Types of mail boxes -----*/

#define FIFO_TYPE 0
#define PRIORITY_TYPE 1
#endif

/*****
/*
/*
/*          FUNCTION PROTOTYPE DECLARATIONS
/*
/*
*****/
// #ifdef __cplusplus
// #define EXTERN extern "C"
// #else
#define EXTERN extern
// #endif

EXTERN void interrupt kernel ( void );

EXTERN void interrupt tick ( void );

EXTERN void wait ( unsigned int semaphore_num );

EXTERN void usignal ( unsigned int semaphore_num );

EXTERN unsigned int rtn_current_task_num ( void );

EXTERN void reschedule ( void );

EXTERN void stop_time_slice ( void );

EXTERN void start_time_slice ( void );

EXTERN int change_task_priority ( char *task_name_ptr,
                                unsigned char new_priority );

EXTERN unsigned int rtn_task_priority ( char *task_name_ptr );

EXTERN void chg_base_ticks_per_time_slice
    ( int new_base_ticks_per_time_slice );

EXTERN int chg_task_tick_delta ( char *task_name_ptr, int new_tick_delta );
```

```
EXTERN void start_tasks ( void ( *null_task ) ( void * ) );

EXTERN BOOL create_task (  char *task_name_ptr,
                          unsigned char priority_of_task,
                          int task_tick_delta, unsigned char status_of_task,
                          unsigned char q_type, unsigned int semaphore_num,
                          unsigned int task_stk_size, unsigned int mess_q_size,
                          unsigned int mess_size, void ( *init_task ) ( void ),
                          void ( *task )( void* ), void* local_var_ptr );

EXTERN void init_semaphore ( unsigned int sema_num, unsigned int sema_value,
                             unsigned int multiplier_value );

EXTERN void preemptive_schedule ( void );

EXTERN timer_struct* start_timer ( unsigned char timer_type,
                                   unsigned long init_time_cnt,
                                   void ( *timeout_handler ) (void *),
                                   void* data_ptr );

EXTERN timer_struct* reset_timer ( timer_struct* timer_ptr );

EXTERN timer_struct* stop_timer ( timer_struct* timer_ptr );

EXTERN int timed_wait ( unsigned int sema_num, unsigned long timeout_value );

EXTERN timer_struct* create_timer ( void );

EXTERN int send_mess ( unsigned char* mess_ptr, unsigned int mess_lgth,
                      char *mess_addr_ptr, int block_action );

EXTERN int send_qik_mess ( unsigned char* mess_ptr, unsigned int mess_lgth,
                           char * mess_addr_ptr );

EXTERN char *rcv_mess ( unsigned char* mess_ptr, unsigned int* mess_lgth,
                       unsigned long time_limit );

EXTERN unsigned int size_mbx ( char *mbx_addr_ptr );

EXTERN unsigned int size_mbx_mess ( char *mbx_addr_ptr );

EXTERN unsigned int free_mbx ( char *mbx_addr_ptr );

EXTERN unsigned int used_mbx ( char *mbx_addr_ptr );

EXTERN void flush_mbx ( void );

EXTERN char setup_os_data_structures ( unsigned char kernel_interrupt_num,
                                       int clock_ticks_kernel_ent,
                                       unsigned int num_of_priorities,
                                       unsigned int max_num_semaphores,
                                       unsigned int maximum_num_of_tasks,
                                       char huge* ptr_to_mem_pool,
                                       unsigned long memory_pool_size );

EXTERN char huge* umalloc ( unsigned long num_bytes );

EXTERN void ufree ( char huge* blk_ptr );

EXTERN char huge* ucalloc ( unsigned long num_obj, unsigned long size_of_obj );

EXTERN unsigned long ret_free_mem ( void );

EXTERN unsigned int return_semaphore_value ( unsigned int sema_num );

EXTERN unsigned int create_semaphore ( unsigned int semaphore_value );

EXTERN char *rtn_current_task_name_ptr ( void );
```

Appendix A — *UNOS-V2.0 TUTORIAL*

```
EXTERN char return_start_flag (void);

EXTERN void enable_task_switch (void);

EXTERN void disable_task_switch (void);

EXTERN void np_signal ( unsigned int semaphore_number);

EXTERN LOCK create_lock ( void );

EXTERN void lock (LOCK);

EXTERN void unlock (LOCK);

EXTERN LOCK destroy_lock (LOCK);
```


general.h

This file contains a number of general definitions. This would usually be included into most user task modules in the system.

```
#ifndef GENERAL_H
#define GENERAL_H
#include <dos.h> // required for the MK_FP macro

// General.h file -- contains general definitions

/*----- Task status bit definitions -----*/

#define TASK_RUNNABLE 0x1
#define TASK_BLOCKED 0x0

/*----- timer related definitions -----*/

#define INACTIVE 0
#define ACTIVE 1
#define SINGLE_SHOT 0
#define REPETITIVE 1

const int TRUE = 1;
const int FALSE = 0;

const char CR = 0x0d;
const char LF = 0x0a;
const char CTRL_Q = 0x11;
const char CTRL_S = 0x13;

// Define the NULL_PTR
#define NULL_PTR MK_FP(0x0,0x0)

// Define the terminating pointer for task name tables
#define TERMINATING_PTR MK_FP(0xffff,0x000f)

#define BOOL unsigned char

#endif // GENERAL_H
```



```
/*
DESCRIPTION

This file contains a number of variables that a user would configure
to control how the operating system is built upon compiling. Most of the
options are well commented below.

*/

/*****
/*
/*      Commonly Changed Variables      */
/*
/*      */
*****/

/*----- Maximum number of tasks in the system -----*/

/* note that task numbers start from zero. Also note that is
always one task in the system - the null task. The variable
max_num_user_tasks below is the maximum number of USER tasks,
and therefore excludes the null task
*/
#define MAX_NUM_USER_TASKS 32

/*
-----
DON'T CHANGE THESE TASK NAMES
-----
*/

/* Define the task names of the system tasks created in this routine */
char pchKeyboardTask[] = "Keyboard Task";
char pchKeyboardMultiPlexerTask[] = "Keyboard Multiplexer Task";

/* null task name always defined */
char name_null_task[] = "Null task";

/*
-----
DON'T CHANGE THIS DEFINE - IT IS COMPUTED USING A VALUED FROM ABOVE
-----
*/
/* the total number of tasks in the system - user tasks
plus the null task and the keyboard tasks.
*/
#define MAX_NUM_OF_TASKS ( MAX_NUM_USER_TASKS + 3 )

/* now define the size of the mail boxes to be used by the keyboard task
and the keyboard multiplexer task.
*/
/* number of mail box messages */
#define MESS_Q_SIZE_KEYBD_TASK 8
#define MESS_Q_SIZE_KB_MULTI_TASK 32

/* size of each mail box message */
#define MESS_SIZE_KEYBD_TASK 2
#define MESS_SIZE_KB_MULTI_TASK 3

/*----- Maximum number of semaphores -----*/
```

```

/* Note that the semaphore numbers start from 0. The system requires
a minimum of num_tasks * 2 semaphores to support the mailbox
mechanism. The additional semaphores that the user creates are
stored as a pool of semaphores which can be obtained by tasks that
require semaphores.
*/

```

```
#define MAX_NUM_SEMAPHORES 200
```

```
/*----- Interrupt number for kernel entry -----*/
```

```
#define KERNEL_ENTRY 96
```

```
/*----- Memory pool declaration -----*/
```

```
/* size of the memory pool. This is the pool of memory used for all
dynamic memory allocations, including those made when the kernel is
starting.
```

```
*/
#define MEMORY_POOL_SIZE 0x30000L
```

```
/*----- Timer declaration -----*/
```

```
/* define the maximum number of timers in the system */
#define MAX_NUM_TIMERS 50
```

```
/ * <<<<<<<<<<<<<<<< END OF FILE >>>>>>>>>>>>>>>>* /
```

Appendix B *A SIMPLE WINDOW SYSTEM*

Introduction

A recent improvement to UNOS (in 1999) has been the addition of a *simple* windowing system (note the emphasis on simple). The previous system which had been used for a number of years was based on the use of the Borland text mode library. This library was designed to work with MS-DOS software and consequently was not reentrant. Hence there were problems using it in a multi-tasking environment. A number of “cludges” were used to get around these problems, and whilst it was possible to generate reasonable text based screens using these routines it was not very elegant.

The “new” window system is written in C++. It will link successfully with UNOS-V2.0 if it is compiled as a C++ program. The window system is called simple as it only implements a very basic set of windowing functions:

- (i) Allows one to create a window of a desired background colour and with user defined coloured characters.
- (ii) The windows created can have no border, a single line border or a double line border.
- (iii) Windows may or may not have a title.
- (iv) The windowing code is reentrant.
- (v) The windows support scrolling.
- (vi) The windows support cursor addressing.
- (vii) One can control the overall screen background colour.
- (viii) One can control the attributes of individual character positions on the screen.

The windowing system has a few major limitations:

- (i) It does not support overlapping windows. It is the programmers responsibility to make sure that windows don't overlap – there is no checking done by the windowing system.
- (ii) Two tasks cannot write to the same window. This in practice should not be a limitation since two tasks can write to the same window by using mail boxes to send the information to a controlling task (which could be the task that owns the window).
- (iii) The windowing system is not integrated with a mouse, so it is not possible to drag windows around the screen.
- (iv) The windowing system is pure text based – it does not employ any graphics.

Despite these limitations it should be very useful in quickly interfacing UNOS tasks to the text screen. A quick look at the code will reveal that it is not particularly well written. My only excuse for this is that this was the first C++ program that I had written, and in hindsight one could have done a better job. Perhaps (when I get some free time, hah, hah) I will rewrite it so that it is more elegant and has more sophisticated features.

windows. It inherits the screen class. Again this does not obey classical class inheritance concepts, but what do you expect from a first time C++ programmer.

NOTE

This module assumes that the screen coordinates start at 0,0 in the top left hand corner, and go to 79, 24 in the bottom right hand corner.
*/

```
// Declare the border_type enumeration
```

The following enum types are used by the user when declaring borders (or lack of borders) around windows, or simply lines, as well as colours for characters and borders.

```
enum border_type {NO_LINE, SINGLE_LINE, DOUBLE_LINE};
enum line_type {single_line = SINGLE_LINE, double_line = DOUBLE_LINE};

enum char_colour {black_char = BLACK_CHAR, cyan_char = CYAN_CHAR,
                  magenta_char = MAGENTA_CHAR, brown_char = BROWN_CHAR,
                  white_char = WHITE_CHAR, gray_char = GRAY_CHAR,
                  light_blue_char = LIGHT_BLUE_CHAR,
                  light_green_char = LIGHT_GREEN_CHAR,
                  light_cyan_char = LIGHT_CYAN_CHAR,
                  light_red_char = LIGHT_RED_CHAR,
                  light_magenta_char = LIGHT_MAGENTA_CHAR,
                  yellow_char = YELLOW_CHAR,
                  bright_white_char = BRIGHT_WHITE_CHAR};

enum backgrd_colour {black_backgrd = BLACK_BACKGRD, cyan_backgrd = CYAN_BACKGRD,
                    magenta_backgrd = MAGENTA_BACKGRD,
                    brown_backgrd = BROWN_BACKGRD,
                    light_gray_backgrd = LIGHT_GRAY_BACKGRD};
```

The point class defines a number of properties and methods associated with “points” on the screen. In the case of this system a point corresponds to a character position on the screen.

The methods that will be mostly used in this class are:

set_attributes – set the character and background colour the next point to be written to on the screen.

moveto – moves to a particular point on the screen in absolute screen coordinates.

write_to_pt – writes a character to the currently addressed point on the screen. It does not advance the point location.

```
//-----
// POINT CLASS

// This is the base class used for the screen routines. It defines a point
// on the screen and some basic properties of that point. It also gives the
// user a few routines to manipulate a point.

class point {
protected :
```

Appendix B — A SIMPLE WINDOW SYSTEM

```
// point on the screen
int x;
int y;

// character value at point on screen
char point_value;

// attributes of character and background
unsigned char point_attrib;

void put_on_scrn();
point read_scrn_pt();

public :
    // Constructor
    // Sets up the position and default attributes of the first
    // character
    point(int xx = 0, int yy = 0, char pt_value = ' ',
          unsigned char point_attribute = WHITE_CHAR | BLACK_BACKGRD)
    {
        x = xx;
        y = yy;
        point_value = pt_value;
        point_attrib = point_attribute;
    } // end of constructor

    // Destructor
    // Resets the screen back to normal operation
    ~point() {
    }

    // Member functions

    // Assignment operator
    point operator=(point rv);

    // Move to an x,y coordinate on the screen. This move is carried out
    // in such a way that one cannot move to a point off the screen.
    // Returns a TRUE if the move is successful, else it returns a
    // FALSE.

    int moveto(unsigned int xvalue, unsigned int yvalue);

    // This function writes the value passed into the function at the
    // currently addressed point on the screen.

    void write_to_pt(char value);

    void set_attributes(unsigned char attribute);

    unsigned char return_attributes(void);
}; // end of point class definition

//-----
```

The screen class is related to aspects of the display that pertain the to screen as a whole. For example, the overall screen background, whole screen scrolling, clearing the whole screen, etc. It also contains routines to draw lines on the screen in screen coordinates. Notice that the screen class inherits the point class.


```
// SCREEN CLASS

// This class inherits the point class as the base class. This class offers
// some slightly higher level functions related to the screen. For example,
// the class does not allow one to write off the screen.

// NOTE:
// That any coordinates in this class are relative to the top left corner
// of the screen which is the 0,0 position.

class screen : public point {

    protected :
```

Note the static variables immediately below. Instances of these variables must be created at the global level of a module (usually in the same module as the main program). They are required so that as we create multiple windows the screen constructor only clears the screen once. The background variable is required as this is inherently a global quantity – there can only be one background colour for the screen.

```
    // This attribute is used to store the general screen background colour
    // that is currently being used. Note that this can be different than
    // the point background colour. This variable is important as it
    // determines the colour when scrolling occurs.

    static backgrd_colour scrn_backgrd_colour;

    // This variable is used to test whether the screen has already been
    // defined. If TRUE then the screen has been created and the clearing
    // of the screen in the screen constructor will not be called. This
    // variable is required so that multiple windows can be created
    // without a screen clear occurring each time a new window is defined.

    static int screen_created;

public :
    // Constructor
    screen(backgrd_colour scr_backgrd_colour = black_backgrd);

    // Destructor
    ~screen(void);
```

The following print function is a “dumb” print function as it writes to the screen at the current point (as contained in the inherited point class) using the current attributes in the point class. Note that this print function is overloaded by the same function in the window class (which obeys the boundaries of a window).

```
    // Write a string of characters to the screen. If the write is
    // successful then return a TRUE, else return a FALSE. The operation
    // may result in an error if the string attempts to write outside the
    // boundaries of the screen. In this case the routine will write what
    // it can but will stop at the boundary. The write begins at the current
    // point on the screen.
    int print(char* string);

    // Clears the current line in the current screen background colour.
    void clr_line(void);

    // Clears from the current point to the end of the line in the current
    // screen background colour.
```

Appendix B — A SIMPLE WINDOW SYSTEM

```
void clr_eol(void);

// Clears from the current point to the beginning of the line in the
// current background colour.
void clr_bol(void);

// Clears the entire screen with the screen background colour.
void clr_scr(void);

// Allows the screen background colour to be set.
void set_scrn_backgrd_colour(backgrd_colour scr_backgrd_colour);

// Draws a box on the screen. Note that the routine allows the
// border type for the box to be set. The two border types are
// SINGLE_LINE and DOUBLE_LINE. These have been declared as an
// enum type called border_type in pscrn.h. This ensures that
// no other type can be passed to the routine.
void draw_box(int left_topx, int left_topy, int right_botx,
              int right_boty, line_type type_of_line);

// These two routines draw horizontal and vertical lines on the screen.
// The lines can be either double lines or single lines.

// The line is drawn in the current y row between the passed x coords.
void draw_horiz_line(int leftx, int rightx, line_type type_of_line);

// The line is drawn in the current x column between the passed
// ycoords.
void draw_vert_line(int topy, int boty, line_type type_of_line);

// This function is called to scroll the whole screen. It has not been
// defined as a general window type scrolling routine for efficiency
// reasons. It is more complicated to scroll a window as compared to
// the whole screen because a window does not have consecutive locations
// to move whereas the screen does. The function scrolls the screen by
// not changing the background attributes of the new line at the bottom
// of the screen. The character attributes are not changed either.
// The parameters passed are the y coordinates of the boundaries of
// the area that has to be scrolled.

void scroll(int ystart, int yend);

}; // end of screen class definition

//-----
```

The window class is the main class in the system that most users will use. It provides a very basic windowing system. As noted previously it does not support overlapping windows – it is the programmers responsibility to ensure that windows do not overlap. If they do it is not disastrous, but the display will not be correct.

Note that the window class inherits the screen class, and therefore implicitly inherits the point class as well (since the screen class inherits the point class).

```
// WINDOW CLASS

// This is the window class that contains all the information required to
// control the display of data in windows. Note that this call inherits the
// screen class and consequently the user has access to all the functions
// associated with screen. In addition all the properties of points are
// inherited by this class. It is through the use of the point attributes
// that the character attributes are set for the data written to the screen.
```

```
class window : public screen {

protected :
    // Remember that the protected data from the point and screen
    // classes are inherited into this class.

    // Define the window geometric parameters
    int window_left_topx;
    int window_left_topy;
    int window_right_botx;
    int window_right_boty;

    // x,y corrdinates relative to the top left hand corner of the
    // window. This coordinate is denoted as point (0,0).
    int window_x_pt;
    int window_y_pt;

    // Define the window and border colour attributes
    backgrd_colour window_backgrd_colour;
    border_type window_border_status;
    char_colour window_border_colour;
    backgrd_colour window_border_backgrd_colour;

    // Location used to store the window title
    char window_title[81];

    // Colour of the window title
    char_colour window_title_colour;

    // String character and background colour. These values denoted the
    // character and background colours for the next string to be written.
    char_colour default_string_colour;
    backgrd_colour default_string_backgrd_colour;

    // The attributes of the particular point are contained in the
    // attributes part of the base point class.

    // This function is called when the contents of the window have to
    // scrolled.
    void scroll_window(void);

public :

    // Window constructor
    // This first routine defines a window with or without without a border.
    // The parameters passed to this routine define the top left corner and
    // the bottom right corner of the window, the type of border on for
    // the window, and its background colour.
    // Care is taken in the routine to ensure that the window does not
    // exceed the dimensions of the screen.
    //
    // Note that these routines do not check if separate windows overlap.
    // This is the responsibility of the programmer.
    //
    // NB. THE SIZE OF THE WINDOW IS THE SIZE OF THE AREA TO BE WRITTEN
    // INTO. THEREFORE THE BORDER (IF THERE IS ONE) WILL CAUSE THE WINDOW
    // DIMENSIONS TO INCREASE BY 2 CHARACTER POSITIONS.

    window(int left_topx = 1, int left_topy = 1, int right_botx = 10,
           int right_boty = 10,
           border_type type_of_border = DOUBLE_LINE,
           char_colour wind_border_char_colour = white_char,
           backgrd_colour wind_border_backgrd_colour
```

```
                                = black_backgrd,
char* wind_title_ptr = NULL_PTR,
char_colour wind_title_colour = white_char,
backgrd_colour wind_backgrd_colour = black_backgrd);

// Window destructor

~window(void);

// Clear window - This function writes space characters into the
// current window. The background colour is determined by the window
// background variable stored in the class.

void clear_window(void);

// This function allows the background colour of the current window
// to be set. This routine does not actually paint the back ground
// colour but merely sets the attribute in the structure.
void set_window_backgrd_colour(backgrd_colour);

// This function sets the screen background variable to the background
// colour and then actually changes the screen background to that
// colour.

void paint_window_backgrd(backgrd_colour);

// This function moves the window insertion point to the x,y coordinates
// passed into the function. Checks are made to ensure that the
// coordinates are not outside the window dimensions.
// If the move is successful without any limiting occuring then a TRUE
// is returned, else a FALSE is returned.

int go_window_xy(int x, int y);

// This is the print function for the window class. It overloads that
// for the screen base class. The function simply prints a string to
// the current window starting at the current x,y position. Checks
// are made to see if the string will write outside the window. If
// the string will write outside the window then only the portion of the
// string that writes into the window will be written, and the last character
// of the string will appear in the right most screen location. This allows
// one to see that something is happening when writes occur outside a window.
// The strings do not auto wrap at the end of a window line, and will
// only wrap if there is a line feed character in the string.
// Returns a TRUE if successful, else a FALSE is returned.

int print(char * string);

// This is an overload of the window print function that allows easier
// writing of a single character to the screen. It simply prints the
// character to the current window starting at the current x, y postion.
// Checks are made to see if the char will write outside the window. If
// the char will write outside the window then the character appears
// in the right most location on the screen.
// Characters do not auto wrap at the end of a window line, and will
// only wrap if it is a line feed character.
// Returns a TRUE if successful, else a FALSE is returned.
```

[illegible]

How to Use the Windowing System

This section will briefly describe how to use the windowing system. For actual examples of using the windows in real code refer to the Appendix A. We will only outline the basics in this document so that the reader has some idea of where to start to generate a window.

One of the least obvious parts of using the windowing system is to declare two static variables that are used in the screen class. One static variable is used as a flag to indicate whether or not the screen class constructor has been called. If it has been called then when more windows are

Appendix B — A SIMPLE WINDOW SYSTEM

created the normal screen class initialisation is not carried out (remember that the screen class is inherited into the window class, therefore each window will result in the screen class constructor being called). This is to ensure that the screen is not cleared as each window is created (which would result in the destruction of any previously created window on the screen). The other variable is used to store the background colour of the screen (since one can only have one background colour for the screen).

The variables in question are declared as follows (probably the best place to do this is in the global section of the module containing the main function):

```
// Variable used to store the screen background colour
backgrd_colour screen::scrn_backgrd_colour;

// Variable used as a flag to indicate that the screen constructor has been
// called previously.
int screen::screen_created = FALSE;
```

The creation of a window is a relatively straight forward process. One simply needs to declare a variable of the window class type, passing to the constructor function a number of parameters indicating the properties of the window. For example, the following line of code comes from Appendix A, “Task 0” code listing:

```
// Window for keyboard characters
static window wind_kb(1, 1, 20, 10, DOUBLE_LINE, white_char, black_backgrd,
    "Tk 0: KB chrs", white_char, light_gray_backgrd);

// Window for characters from task 1
static window wind_t1(24, 1, 54, 10, SINGLE_LINE, light_blue_char,
    black_backgrd, "Tk 0: Task 1 chrs", light_red_char,
    light_gray_backgrd);

// Window for characters from serial com2
static window wind_c2(58, 1, 78, 10, NO_LINE, white_char, black_backgrd,
    "Tk 0: Com2 chrs", white_char, light_gray_backgrd);
```

These declarations were carried out in the global section of this module, and hence the window objects are created as the software is loaded and the variable initialisation code runs. Alternatively, one could also put this code in the task function itself, so that the windows are created when the system actually starts. The window constructors do not use any operating resources, therefore it is OK to create them prior to the execution of the operating system.

For details on the parameters passed during window creation refer to the reference section at the end of this Appendix.

Once a window has been created with a certain size, background colour, border or no border, title or no title etc., then one has to actually write something into the window. This can be achieved in several ways. If one wishes to write a string to the screen then the **print** method is called as follows for **wind_kb** created above:

```
wind_kb.print(chr_string);
```

where **chr_string** is the null terminated character string to be printed into the window. The print method returns a TRUE if the string is successfully written, else it returns a false if the string is too big to fit into the line in the window. In this situation the portion of the string that can be written into the window will be written, and the character at the end of the line (right most position) in the window will be the last character in the string.

An alternative to using the string print method is to write a single character at a time using the character print method – i.e.:

```
wind_kb.print(character);
```

where **character** is the character to be printed to the window.

In order to print either strings or characters in a window some preconditions must be established – namely the position of the start of the string or the character, and the colour of the character/s to be written, and the background for the characters. By default the character background is the same as the window background, but it is possible to change the string background so that it is different from that of the window that it lies in. These three objectives are achieved by calling the following three routines before carrying out the write operations:

```
wind_kb.set_string_backgrd_colour(colour_of_backgrd);  
wind_kb.set_string_colour (colour_of_char);
```

The `colour_of_char` variable has to be one of those listed in the **char_colour** data type defined at the beginning of `pcsrn.h`. Similarly the background colour has to be from **backgrd_colour**.

```
wind_kb.go_window_xy(x, y);
```

Note that the “**x,y**” coordinates are in window relative coordinates, with the “0,0” position being the top left hand corner of the writable window.

In a manner similar to characters, the background of the window can also be controlled after the window is created via the following function:

```
wind_kb.paint_window_backgrd(colour_of_backgrd);
```

where **colour_of_backgrd** is selected from the type **backgrd_colour** in `pcsrn.h`. See Table 2-2 for a full listing of the character and background colour definitions available in UNOS.

In addition to these basic functions one can also clear the entire window or clear a single line in the window.

There are also some basic line drawing functions to draw lines in the window. Consistent with the other functions one has to set up the attributes of the background colour and the character colour for the line prior to drawing it using the **set_attributes** method. A check is also made to ensure that the line does not go outside the boundaries of the window.

A Note on Setting Character Attributes

Characters in the text mode of an IBM-PC or compatible computer have several attributes – character colour, character background colour, bright mode and flashing mode. These attributes are stored as a second byte of information associated with each character. The attribute byte is laid out as shown in Table 2-1.

In order to set the attributes one should bitwise “or” in the various options. There are a number of enum types defined at the beginning of `pcsrn.h` which define the various bit patterns to obtain the different character and background colours, as well as the highlight and flashing characters. By “oring” together the required options a byte is formed with the correct bit pattern. The colours for characters and backgrounds defined in UNOS are listed in Table 2-2 (note the case is important). These colours are defined as two **enum** types:

```
char_colour
```

and

```
backgrd_colour.
```

In order to use these character and background attributes to effect what is written to the screen then one uses the **set_attributes** method as mentioned previously. For example if one

Table 2-1: Bits for character and background colour control

Bit								Use
7	6	5	4	3	2	1	0	
1	Blinking character
.	1	Red component of background
.	.	1	Green component of background
.	.	.	1	Blue component of background
.	.	.	.	1	.	.	.	Intense character
.	1	.	.	Red component of character
.	1	.	Green component of character
.	1	Blue component of character

Table 2-2: UNOS character and background colour definitions

Character Colours	Background Colours
black_char	black_backgrd
white_char	light_gray_backgrd
red_char	red_backgrd
green_char	green_backgrd
blue_char	blue_backgrd
cyan_char	cyan_backgrd
magenta_char	magenta_backgrd
brown_char	brown_backgrd
gray_char	
bright_white_char	
light_red_char	
light_green_char	
light_blue_char	
light_cyan_char	
light_magenta_char	
yellow_char	

wishes to have a red character on a blue background from all the characters written after the invocation of the **set_attributes** method for window **wind_c2** then this is written as:


```
wind_c2.set_attributes(red_char | blue_backgrd);
```

If one wanted the next characters to be written to also blink then one would “or” in the **B** **BLINKING** attribute as follows:

```
wind_c2.set_attributes(red_char | blue_backgrd | BLINKING);
```

It should also be noted that the string print function keeps its own copy of the attributes for writing strings. Therefore it is possible to have the writing of individual characters with different attributes then the writing of strings without having to constantly reset the attributes.

In addition to the window class based functions there are also functions associated with the screen and point classes. These can also be used in order to get fancy displays.

Reference Section

This section will describe each of the methods publicly available in the pscrn.cpp module.

The Point Class

```
point::point(int xx = 0, int yy = 0, char pt_value = ' ',  
             unsigned char point_attribute = WHITE_CHAR | BLACK_BACKGRD)
```

POINT::POINT DESCRIPTION

This is the constructor for the point class. Usually this is not explicitly called, since the point class is inherited into the screen class and the window class. Therefore it is implicitly constructed when these classes are created. The default values of the parameters are shown above.

PARAMETERS

xx	This is the x coordinate of the point.
yy	This is the y coordinate of the point.
pt_value	This is the value of the point.
point_attribute	This is the attribute of the point.

RETURNS

Nothing.

```
point point::operator=(point rv);
```

POINT::OPERATOR= DESCRIPTION

This operator allows one point to be assigned to another. The assignment means that the all the components of the point class *except for the position* are copied to the assigned to point, and then the point is actually written on the screen – i.e. the screen memory at the point has the character code and attributes written to it. Therefore a change occurs on the screen.

A point class is created for every window that is created in the system. Note that even though the screen consists of a number of points, there is not a point class created for every point on the screen. The user, if they so desire, can create extra instances of the point class.

EXAMPLE

```
// Create a point at (10,20) on the screen with a character 'A'.
// Note that the point does not appear on the screen.

point point_1(10, 20, 'A', white_char | black_backgrd);

// Create a second point at (15,30) with a read character and
// light gray background colour.

point point_2(15, 30, 'B', red_char | light_gray_backgrd);

// Now make these two points appear on the screen

point_1 = point_1;
point_2 = point_2;

// Now make the character and attributes of point_1 be that of point_2
// and write the result on the screen - i.e. point_1 is now a red 'B' on
// a light gray background.

point_1 = point_2;
```

```
int point::moveto(unsigned int xvalue, unsigned int yvalue);
```

POINT::MOVETO DESCRIPTION

This method changes the current point position to the (x,y) coordinates passed to it. No change is made to anything on the screen. It simply changes the point location for other future commands related to points. The move is made so that one cannot move off the physical screen.

The screen coordinates start at (0,0) in the top left hand corner of the screen, and range to (24, 79) in the bottom left hand corner.

PARAMETERS

xvalue	This is the x coordinate to move to.
yvalue	This is the y coordinate to move to.

RETURNS

Returns a TRUE if the move is successful, else a FALSE is returned if one has attempted to move off the screen.

```
void point::set_attributes(unsigned char attribute);
```

POINT::SET_ATTRIBUTES DESCRIPTION

This method is used to set the attribute variable in point class. The attributes are formed by “oring” together the character colour, the character background colour and the blinking bit. These attributes are then applied to future single character writes and string writes using the screen class **print** function.

PARAMETERS

attribute	The contains the attribute that is to be associated with the point class.
------------------	---

RETURNS

Nothing

EXAMPLE

Use with a window class

```
window wind_examp();

// Next data to be written to the window with blinking red characters,
// on a light gray background.

wind_examp.set_attributes (red_char | light_gray_backgrd | BLINKING);
```

```
void point::write_to_pt(char value);
```

POINT::WRITE_TO_PT DESCRIPTION

This method writes the character value passed in to the current point on the screen. The character written is using the current attributes. Note that the character point is not advanced after the write operation.

PARAMETERS

value This is the character to be written to the screen at the current point and with the current attributes. The character value can be any value in the range of 0 to 255.

RETURNS

Nothing.

```
unsigned char return_attributes(void);
```

POINT::RETURN_ATTRIBUTES

This function returns the attribute byte of the currently addressed point location.

PARAMETERS

None

RETURNS

A **char** variable containing the attribute byte.

The Screen Class

```
screen::screen(backgrd_colour scr_backgrd_colour = black_backgrd);
```

SCREEN::SCREEN DESCRIPTION

This is the constructor for the screen class. The function clears the screen using the screen background colour passed in. It also clears the normal text mode cursor using a BIOS function.

PARAMETERS

scr_backgrd_colour This is the background colour for the screen.

RETURNS

Nothing.

```
int screen::print(char* string);
```

SCREEN::PRINT DESCRIPTION

This method prints a null terminated string on the screen starting at the current location using the currently set attributes. If an attempt is made to write off the screen then the portion of the string that will fit on the screen is written. If the string contains a CR LF then the appropriate actions are carried out. If the CR LF occurs on the last line on the screen then the screen is scrolled.

The

PARAMETERS

string Pointer to the string to be printed to the screen.

RETURNS

TRUE if the string is successfully written on the screen. FALSE is returned if the string will not fit entirely on the screen.

```
void screen::clr_line(void);
```

SCREEN::CLR_LINE DESCRIPTION

This method clears a whole line on the screen. The line cleared is the current line. It is cleared using the current value for the screen background colour. After the clear the current point is at the beginning of the line cleared.

PARAMETERS

None.

RETURNS

Nothing.

```
void screen::clr_eol(void);
```

SCREEN::CLR_EOL DESCRIPTION

This method clears from the current position on the screen to the end of the line. The line is cleared using the current value for the screen background. At the end of the operation the current point is left at start point of the cleared line section.

PARAMETERS

None

RETURNS

Nothing.

```
void screen::clr_bol(void);
```

SCREEN::CLR_BOL DESCRIPTION

This method clears the line from the current point to the left edge of the screen. The clear is carried out using the screen background colour. After the clear operation the current point is left at the left edge of the screen.

PARAMETERS

None.

RETURNS

Nothing.

```
void screen::clr_scr(void);
```

SCREEN::CLR_SCR DESCRIPTION

This method clears the entire screen using the current screen background. The current point is left at the top left hand character position after the clear operation.

PARAMETERS

None.

RETURNS

Nothing.

```
void screen::set_scrn_backgrd_colour(backgrd_colour scr_backgrd_colour);
```

SCREEN::SET_SCRN_BACKGRD DESCRIPTION

This method is used to set the background colour of the entire screen.

PARAMETERS

None.

RETURNS

Nothing.

```
void screen::draw_box(int left_topx, int left_topy, int right_botx,  
                      int right_boty, line_type type_of_line);
```

SCREEN::DRAW_BOX DESCRIPTION

This method draws a box on the screen. The box dimensions are passed in via the coordinates for the top left hand corner and the bottom left hand corner. These coordinates are in screen relative coordinates (i.e. start from 0,0 in the top left hand corner). The line type of the border for the box can be: NO_LINE, SINGLE_LINE or DOUBLE_LINE. The border uses the current values set by the **point::set_attributes** routine. This has to be set prior to calling this routine.

Only the portion of the box that will fit on the screen will be displayed.

PARAMETERS

left_topx	This is the x coordinate for the top left hand corner of the box.
left_topy	This is the y coordinate for the top left hand corner of the box.
right_botx	This is the x coordinate of the bottom right corner of the box.
right_boty	This is the y coordinate of the bottom right corner of the box.
type_of_line	This is the type of line for the box. The possible line types are: SINGLE_LINE – the character used to draw the line is a single line character; DOUBLE_LINE – the character used to draw the line is a double line character.

RETURNS

Nothing.

```
void screen::draw_horiz_line(int leftx, int rightx, line_type type_of_line);
```

SCREEN::DRAW_HORIZ_LINE DESCRIPTION

This method is used to draw a horizontal line on the screen. The line is drawn on the current row. The start and end points of the line are passed as parameters, as is the type of line. The colour of the line has to be set using the `set_attributes` method prior to calling this method.

PARAMETERS

leftx	This is the left x coordinate marking the start of the line.
rightx	This is the right x coordinate marking the end of the line.
type_of_line	This is the type of line to draw. The possible line types are: SINGLE_LINE – the character used to draw the line is a single line character; DOUBLE_LINE – the character used to draw the line is a double line character.

RETURNS

Nothing.

```
void screen::draw_vert_line(int topy, int boty, line_type type_of_line);
```

SCREEN::DRAW_VERT_LINE DESCRIPTION

This method draws a vertical line on the screen. The line is drawn in the current column using the current values for the character foreground and background colours. These should be set up prior to calling the routine using the **point::set_attributes** routine.

PARAMETERS

topy	This is the top y coordinate of the line.
boty	This is the bottom y coordinate of the line.
type_of_line	This is the type of line to draw. The possible line types are: SINGLE_LINE – the character used to draw the line is a single line character; DOUBLE_LINE – the character used to draw the line is a double line character.

RETURNS

Nothing.

```
void screen::scroll(int ystart, int yend);
```

SCREEN::SCROLL DESCRIPTION

This function scrolls the screen between the rows passed in as parameters. The attributes of the locations on the screen are not changed by the scroll operation. The bottom line of the scroll region is cleared using the screen background colour.

PARAMETERS

ystart	This is the coordinate of the top row of the scroll region.
yend	This is the coordinate of the bottom row of the scroll region.

RETURNS

Nothing.

Window Class

```
window::window(int left_topx = 1, int left_topy = 1, int right_botx = 10,
               int right_boty = 10,
               border_type type_of_border = DOUBLE_LINE,
               char_colour wind_border_char_colour = white_char,
               backgrd_colour wind_border_backgrd_colour =
                           = black_backgrd,
               char* wind_title_ptr = NULL_PTR,
               char_colour wind_title_colour = white_char,
               backgrd_colour wind_backgrd_colour =
                           black_backgrd);
```

WINDOW::WINDOW DESCRIPTION

This is the constructor for the window class. It has a number of parameters so that all the features required of a window can be initialised at the time of creation. If the window will not fit on the screen then a window will not be created.

PARAMETERS

- left_topx** This is the screen relative x coordinate of the top left corner of the writable area of the window.
- left_topy** This is the screen relative y coordinate of the top left corner of the writable area of the window.
- right_botx** This is the screen relative x coordinate of the bottom right corner of the writable area of the window.
- right_boty** This is the screen relative y coordinate of the bottom right corner of the writable area of the window.
- type_of_border** This indicates the type of border for the window. The possibilities are: NO_LINE – there is no border; SINGLE_LINE – there is a single line border; DOUBLE_LINE – there is a double line border.
- wind_border_char_colour** This is the colour of the foreground character for the window border (i.e. the line character colour).
- wind_border_backgrd_colour** This is the colour of the character background for the window border.
- wind_title_ptr** This is a pointer to a character string to be used for the window title. The string pointed to is a null terminated string. If the string does not fit in the area allowed for the title then no title appears.
- wind_title_colour** This is the character colour of the title. The background colour is the border background colour.
- wind_backgrd_colour** This is the background colour for writable section of the window.

RETURNS

Nothing.

```
void window::clear_window(void);
```

WINDOW::CLEAR_WINDOW DESCRIPTION

This method clears the writable area of the window. It uses the window background colour to set the colour of the cleared window.

PARAMETERS

None.

RETURNS

Nothing.

```
void window::set_window_backgrd_colour(backgrd_colour backgrd_col);
```

WINDOW::SET_WINDOW_BACKGRD_COLOUR DESCRIPTION

This method sets the background colour of the window. It does not actually paint the window, but merely sets the appropriate attribute variable in the window class.

PARAMETERS

backgrd_col This is the background colour to set the internal variable to.

RETURNS

Nothing.

```
void window::paint_window_backgrd(backgrd_colour backgrd_col);
```

WINDOW::PAINT_WINDOW_BACKGRD DESCRIPTION

This method sets the background colour to the value passed in. It not only changes the internal variable but it actually repaints the window with the new colour. It does not change any characters that are currently in the window.

PARAMETERS

backgrd_col This is the background colour to set the internal variable to, and to paint the window with.

RETURNS

Nothing.

```
int window::go_window_xy(int x, int y)
```

WINDOW::GO_WINDOW_XY DESCRIPTION

This method changes the window relative (x,y) coordinates to the values passed in. This can then affect the outcome of future methods that operate on the current point. If one attempts to move outside the window then the current point will be left with the extreme window value in the dimension/s that has gone out of bounds. The window coordinates are window relative and start with (0,0) in the upper left hand corner.

PARAMETERS

int x	The window relative x coordinate to be set as the current point.
int y	The window relative y coordinate to be set as the current point.

RETURNS

TRUE if the coordinates do not violate any limits, else a FALSE is returned.

```
int window::print(char* string);
```

WINDOW::PRINT DESCRIPTION

This method prints a string in a window. It overloads the print method in the screen class. It starts to write the string at the current location using the attributes stored in the **default_string_colour** and **default_string_backgrd_colour** locations in the window class. These are separate attributes from those used to write single characters to a location in the window. If the string will not fit in the window then it will write what will fit, and the last character of the string will be the character at the right most point of the line. The function does not auto wrap, but if there is a CR LF in the string it will carry out a window scrolling operation.

PARAMETERS

string Pointer to the null terminated string to be printed.

RETURNS

TRUE if the string write has occurred without any limits being hit, else a FALSE is returned.

```
int window::print(char chCharacter);
```

WINDOW::PRINT DESCRIPTION

This is an overload of the previous string print function that makes it easier to write a single character to the window. It essentially uses the string print function underneath, and therefore has the same limitations as it. It should be noted that it also uses the **default_string_colour** and **default_string_backgrd_colour** attributes to determine the colour of the characters.

PARAMETERS

chCharacter Character to be printed to the window.

RETURNS

TRUE if when the current pointer is advanced one position after the write and it does not hit a limit, FALSE if the advance hits a limit and cannot be made.

```
void window::set_string_colour (char_colour colour_char);
```

WINDOW::SET_STRING_COLOUR DESCRIPTION

This method simply changes the stored value for the string colour attribute variable. This variable determines the foreground colour of strings.

PARAMETERS

colour_char This is the new colour of string characters.

RETURNS

Nothing.

```
void window::set_string_backgrd_colour (backgrd_colour colour_backgrd);
```

WINDOW::SET_STRING_BACKGRD_COLOUR DESCRIPTION

This method simply changes the stored value for the string background colour attribute variable. This variable determines the background colour of strings.

PARAMETERS

colour_backgrd This the new background colour for string variables.

RETURNS

Nothing.

```
void window::scroll_window(void);
```

WINDOW::SCROLL_WINDOW DESCRIPTION

This method carries out a scroll of a window. The bottom line of the window is cleared after the scroll and the current window position is set to the bottom left hand side of the window.

PARAMETERS

None.

RETURNS

Nothing.

```
int window::window_clear_line (unsigned int line_number);
```

WINDOW::WINDOW_CLEAR_LINE DESCRIPTION

This method clears an entire line in the window. The line number in the window to be cleared is passed to the routine. The current point in the window is left at the left end of the bottom window line. If the line number is specified is outside the window then nothing happens.

PARAMETERS

line_number The line number on the current window to clear.

RETURNS

TRUE if the clear operation is successful, else FALSE is returned.

```
int window::get_top_leftx(void)
int window::get_top_leftty(void)
int window::get_bot_rightx(void)
int window::get_bot_rightty(void)
```

WINDOW::GET_TOP/BOT_LEFT/RIGHT/XY DESCRIPTIONS

These routine are virtually identical in function. Their purpose is to return to the caller the requested screen relative coordinate of the window.

PARAMETERS

None.

RETURNS

The requested screen relative coordinate of the window.

```
int window::draw_horiz_line(int line_length, line_type type_of_line);
```

WINDOW::DRAW_HORIZ_LINE DESCRIPTION

This method draws a horizontal line in the window. The line is draw from the current point and its length is passed in. The type of line is also selectable. If the line length is larger than what can fit in the screen, then the line is drawn to the window border.

PARAMETERS

line_length Length of the line to be drawn.

type_of_line The line types are: SINGLE_LINE – a line consisting of the single line character; DOUBLE_LINE – a line consisting of the double line character.

RETURNS

TRUE if the line is written without hitting a limit, else a FALSE is returned.

```
int window::draw_vert_line(int line_length, line_type type_of_line);
```

WINDOW::DRAW_VERT_LINE DESCRIPTION

This method draws a vertical line in the window. The line is draw from the current point and its length is passed in. The type of line is also selectable. If the line length is larger than what can fit in the screen, then the line is drawn to the window border.

PARAMETERS

line_length Length of the line to be drawn.

type_of_line The line types are: SINGLE_LINE – a line consisting of the single line character; DOUBLE_LINE – a line consisting of the double line character.

RETURNS

TRUE if the line is written without hitting a limit, else a FALSE is returned.

INTRODUCTION¹

The University of Newcastle Operating System (UNOS) is an operating system kernel designed primarily for real-time system applications. The first version of UNOS was developed for use in a vehicle for detecting stress corrosion cracking in underground pipelines. The hardware for this system was custom built and based on the Intel 80c186 embedded microcontroller. The development tools available resulted in this first version of UNOS being written in PL/M, a Pascal like language supported by Intel. This version of UNOS was called Version 1.0.

The PL/M implementation of UNOS meant that the operating system was confined to platforms using Intel processors. In order to allow easy porting of UNOS to other platforms it was decided to translate it into the C language. Initially this was done by hand translating the PL/M version. Subsequently this version had a number of improvements made to simplify the use of the kernel from the users point of view, and also to add some extra features. The resultant version was called Version 1.5. This version was initially used in an antenna control system for the Department of Defence dish antennae at Geraldton in Western Australia. During the development of this medium size software system several bugs in the kernel were identified and rectified. In addition several deficiencies from a users point of view were identified in the kernel. These were rectified and the result is the version of the kernel known as UNOS-Version 1.5a. It should be realised, that whilst some attempt was made to overcome priority inversion in this kernel, it did not implement a general solution that worked under all circumstances.

Subsequent to the Geraldton Project UNOS 1.5a has been used in a series of satellite tracking systems market by TUNRA Pty Ltd under the trade name of TrackSat. These systems are operational in a number of countries in the South East Asia region. In addition UNOS 1.5a was used in a truck tracking system, partially developed by the Centre of Industrial Control Sciences (CICS), for the local mining scheduling software company Australian Systems Integration (ASI) Pty Ltd. The UNOS based units are still being sold by this company.

The latest update to the kernel is UNOS-Version 2.0-Beta (from now on known as UNOS-V2.0). This is known as a “Beta” version because it has not had the extensive field testing of the previous version. This is a major upgrade to the kernel, with significant enhancements such as ‘lock’ primitives (which overcome the priority inversion problem) and enabling of interrupts inside the kernel to decrease interrupt latency. These features are designed to improve the performance of the kernel.

The kernel module has been modified so that it will now compile successfully under the Borland 4.5 C++ compiler. It should be noted that the kernel is still written in C, but by allowing compilation under C++ one can take advantage of the tighter type checking of the C++ com-

1. This document pertains to UNOS-Version 2.0-Beta. There is a separate document, available from the author, outlining the features of UNOS-Version 1.5a.

piller, and at the same time allow the user to write modules using C++ objects without having to worry about C to C++ linkage problems.

In the process of developing UNOS-V2.0 the support routines have been converted to C++ compile routines (i.e. the keyboard handler, the serial drivers and the command decoder). In addition the previous very cludgy screen I/O system has been scrapped and a new (but very basic and simple) screen system written in C++ has been added. It should be emphasized that this screen I/O system is very basic, and is not a particularly well written C++ program (it was the authors first C++ program). However, due to time constraints it will have to suffice at the moment. One advantage it does have over the previous screen system is that there are no BIOS or DOS routines used in it, therefore the code is totally reentrant.

KEY FEATURES

The main features of the UNOS kernel are:–

- The kernel is written in standard ANSI ‘C’ so that the source can be customized to add application specific special features. (From Version 2.0-beta it can be compiled under C++).
- The kernel is easily ported to different hardware platforms.
- Much of the source code development can be done on an IBM-PC or compatible using powerful software development tools such as those in Borland C++ (e.g. the Borland stand alone debugger).
- Easily programmed into PROMs or EPROMs (dependent on the target compiler being used).
- Pseudo dynamic implementation. The kernel has been implemented so that from the user's point of view most resources can be allocated dynamically. For example, semaphores are created within user tasks by calling a **create_semaphore** routine. However, internally the semaphores are centrally located structures with a runtime determined number. Other structures are similarly defined. The advantage of this implementation is that an operating system monitor can easily locate all the kernel related data structures created by the user.
- UNOS 2.0- beta onwards has a complete implementation of a full nested priority inversion strategy based on the use of critical section primitives known as “locks”.
- Low interrupt latency and fast task switch times (exact times depend on the speed of the processor).

REAL-TIME SERVICES

The UNOS kernel provides a rich set of services tailored specifically for real-time applications:–

- Task management with calls to create, manage, and schedule tasks in a multi-tasking environment. The kernel offers preemptive priority scheduling combined with optional round-robin scheduling, and is designed to allow rapid predictable scheduling of tasks. The total number of tasks supported is 65536. The user can define up to 255 priority levels. Tasks of similar priority can be effectively assigned a sub-priority by defining whether the task will execute for the default number of clock ticks before a time sliced task switch, or some other value which may be more or less than the default. Task priorities and sub-priorities can be altered by user tasks at runtime.

- Interrupts are handled by switching immediately from a user task to the interrupt routine. The interrupt routine operates below the kernel level – i.e. the kernel cannot distinguish the interrupt routine from the task that was being executed at the time the interrupt occurred. If the kernel is being executed when the interrupt occurs then the interrupt routine execution occurs immediately. If the interrupt routine calls a kernel routine (such as a usignal), then the kernel operation is delayed until the current invocation of the kernel completes. Using this approach low interrupt latency can be achieved whilst preventing reentrancy into the kernel.
- Time management facilities providing single shot and repetitive timers and a real-time clock. The timers can be asynchronously reset so that they can be used to implement watchdog timers.

The timer facilities can be used for a wide variety of applications. For example the single shot timers can be used to handle timeouts. If a single shot timer does timeout then a user supplied routine is executed. The repetitive timers can be used to execute a user supplied function at a precise repetition frequency. The user supplied routine could carry out some function in its own right, or alternatively simply signal another task to execute. Because a repetitive timer restarts itself automatically upon timeout the time drift associated with some timer implementations due to the instruction execution time between the timeout of the timer and the user software reenabling of the timer is eliminated. The result is that the repetition frequency is very accurate over long periods of time.

The tick frequency can be defined independently of the time slice frequency. This means that the tick can be defined so that the timers achieve the accuracy required, without causing excessive task switching.

- Semaphores are provided for *intertask communications*. Semaphores can be used for intertask signalling and synchronization, but they should not be used for critical section protection.

A timed *wait* primitive is supplied. This allows a timeout to be defined for a “wait” operation. A timeout on a “timed wait” is detected from the return code of the **timed_wait** function. The “timed wait” facility can be used to put a task to sleep for a user specified length of time by calling the **timed_wait** function when the associated semaphore has a value of zero and no other task will do a signal on the semaphore.

- Mailboxes are provided for more sophisticated intertask communication. They are created by the kernel at task creation time for every task in the system. A task can only read a message from its own mail box. Mail boxes can contain a system dependent maximum number of messages (usually a maximum of 65536 since the mail box length is usually stored in an unsigned integer), with each individual message also of a maximum system dependent length (usually 65536)². Currently the messages are considered to be composed of bytes. The mail boxes also allow the sending of priority messages. This is achieved by placing a priority message in front of all the other messages stored in a mail box. If there are no free locations in the mailbox then the priority message is placed in a special reserved location (one per mail box) for priority messages. Normal messages are stored in a first in first out (FIFO) order in the mail box.

2. Note that both the number of messages and the size of messages are user configurable values defined at task creation time.

Another feature of the mailbox mechanism is that the sending task address is made available to the reading task. This means that the single mail box per task implementation does not restrict the number of tasks that can communicate to a task. The address of a task is represented by a pointer to a **NULL** terminated character string that describes the task. This pointer is connected to a mail box number internally within the kernel. Therefore a user needs to know only the pointer to the task name character string in order to send a message to a task. Having the task name independent of the task number means that task creation order does not affect a task's mail box address from the user's perspective.

In order to prevent deadlock when waiting for a mail box message the system provides a "timed wait" on mail box receive message function calls. It is up to the user task which is waiting to receive the message to carry out the necessary handling in the event that there is a time out waiting on a message to arrive. The "timed wait" on a mail box message has been implemented by using the **timed wait** semaphore functions feature provided within the kernel.

HARDWARE REQUIREMENTS

As mentioned previously one of the main design criteria of the UNOS kernel was its portability across different hardware platforms. Consequently UNOS makes as few assumptions about the underlying hardware as possible.

At the time of writing this manual UNOS has been ported to the Intel 8086 series of processors, the Motorola 68000 series, and the Intel 80960KB RISC processor. Architecturally these three machines are quite different (one being a 16 bit CICS processor and the other two 32 bit processors), but the ports only took a few days to complete in most cases. The main areas of change between the UNOS implementations were in the task creation, task switch and interrupt handling sections of the kernel.

Problems associated with memory protection differences between processors are avoided because UNOS is designed for an unprotected mode processor. The preferred memory organisation is a flat linear address map, however other address maps can be accommodated (e.g. the segmented addressing technique used by the Intel 8086 series of processors).

Several prototype protected mode versions of UNOS have been developed by University of Newcastle, Department of Electrical and Computer Engineering final year project year students. These have used both the flat memory model and a segmented paging memory model. At this stage these versions are largely untested, and the development tools are very esoteric. Therefore they will not be discussed in this manual.

INTRODUCTION TO UNOS-V2.0 KERNEL PRIMITIVES

Below is a brief description of the main user callable kernel routines. The routines are organised into logical sections.

Kernel Initialisation

setup_os_data_structures

This creates all the internal data structures used by the kernel.

Task Management

create_task

create a new task and initialise the associated task control block.

change_task_priority

Change the main priority of a task.

rtn_task_priority

Returns the priority of any nominated task.

start_tasks

Start all the created tasks after all tasks in the system have been created.

rtn_current_task_num

Return the current task number.

rtn_current_task_name_ptr

Return a pointer to the task name string.

chg_task_tick_delta

This effectively changes the number of ticks that a task is allowed before it is time slice task switched. It forms a sub-priority since tasks of the same priority can get different amounts of processor time.

Task scheduling management

preemptive_schedule

Called to initiate a software preemptive task switch

reschedule

Software reschedules another task to run, even if it is a lower priority then the calling task.

start_time_slice

Starts time slice scheduling.

stop_time_slice

Stops time slice scheduling.

chg_base_ticks_per_time_slice

Changes the default number of clock ticks before a time slice entry into the kernel

Time management

create_timer

Creates the data structure required for a timing operation.

start_timer

Initialises and starts a timer timing.

reset_timer

Resets the time of a timer back to its original time value and continues timing.

stop_timer

Stops an operational timer and places it onto the queue of inactive timers.

Intertask communication and synchronization***create_semaphore***

Dynamically creates a semaphore.

init_semaphore

Allows custom initialisation of the components of the semaphore structure.

wait

Carries out a semaphore “wait” operation.

timed_wait

Carries a a “timed wait” on a particular semaphore – that is the “wait” on the semaphore has a limited duration before the calling task becomes unblocked.

usignal

Carries out a semaphore “signal” operation. Note the “u” in the name so that it does not conflict with the “signal” function name in ‘C’.

return_semaphore_value

Returns the value of the semaphore count component of a particular semaphore.

create_lock

Dynamically creates a lock critical section primitive.

lock

Puts a lock on a critical section, preventing other tasks from accessing this resource.

unlock

Unlocks a previously locked critical section, allowing other tasks to access this resource.

send_mess

Sends a message consisting of an string (ascii or binary) to a particular task.

send_qik_mess

Sends a priority message to a particular task. This differs from “send_mess” in that this message effectively goes to the front of any queue of messages.

DETAILED DESCRIPTION OF USER INTERFACE

rcv_mess

Receives a message from the mail box associated with the calling task. The message is returned as a string of bytes (ascii or binary).

size_mbx

Returns the number of messages that can be stored in a mail box.

size_mbx_mess

Returns the maximum size of an individual message in a mail box.

free_mbx

Returns the number of free message slots in a mail box.

used_mbx

Returns the number of used message slots in a mail box.

flush_mbx

Flushes all messages inside a mail box.

Memory Management

umalloc

UNOS malloc function. Allocates a number of bytes from the UNOS heap.

ucalloc

UNOS calloc function. Allocates a number of bytes equivalent to that required for a number of objects of a user specified type.

ufree

Frees an area of memory allocated on the UNOS heap via a umalloc or a ucalloc.

ret_free_mem

Returns the amount of free memory on the UNOS heap.

Miscellaneous

return_interrupt_status

Returns the current status of the processor interrupts – i.e. are they enable or disabled.

DETAILED DESCRIPTION OF USER INTERFACE

The following details the interface routines to the UNOS kernel services. The function prototypes are presented along with a brief outline of its functionality, any calling restrictions on the function, the parameters that are passed, and the nature of the return values (if any).

The following pages are organised with one function description per page. Groups of functions are placed under the headings used in the previous section.

Kernel Initialisation Functions

```
char setup_os_data_structures ( unsigned char kernel_interrupt_num,
                                int clock_ticks_kernel_ent,
                                unsigned int num_of_priorities,
                                unsigned int max_num_semaphores,
                                unsigned int maximum_num_of_tasks,
                                char* ptr_to_mem_pool,
                                unsigned long memory_pool_size );
```

SETUP_OS_DATA_STRUCTURES DESCRIPTION

This function sets up some of the major data structures in the UNOS system. All these data structures are created dynamically on the UNOS heap. In addition it defines values for the interrupt number used for entry into the kernel, the number of priority levels to be allowed, the maximum number of semaphores to be allowed, and the maximum number of tasks. The location and size of a memory pool is passed in so that these values are known to the kernel.

CALL RESTRICTIONS

The function *must* be called from the main function after a memory pool has been defined and before any other initialisation action.

PARAMETERS

kernel_interrupt_num	This is the number of the software interrupt used to enter the kernel.
clock_ticks_kernel_ent	This parameter defines the default number of entries into the kernel tick routine before a time slice entry into the kernel (if time slicing is enabled)
num_of_priorities	This parameter defines the number of priorities to be used in the system. In the current implementation a maximum of 255 priority levels are allowed.
max_num_semaphores	Sets the maximum number of semaphores that can be created in the system.
max_num_of_tasks	Defines the maximum number of tasks that can be created.
ptr_to_mem_pool	A pointer to the beginning of a previous defined memory pool. Note that the memory pool must contain contiguous memory - a fragmented memory pool is not supported in the current UNOS implementation.
memory_pool_size	The size of the memory pool which becomes the UNOS heap.

RETURNS

The routine returns a **TRUE** value if the creation of all the data structures has been successful, else it returns a **FALSE**.

Task Management Functions

```
int create_task (  char *task_name_ptr,
                  unsigned char priority_of_task,
                  int task_tick_delta, unsigned char status_of_task,
                  unsigned char q_type, unsigned int semaphore_num,
                  unsigned int task_stk_size, unsigned int mess_q_size,
                  unsigned int mess_size, void ( *init_task ) ( void ),
                  void ( *task )( void* ), void* local_var_ptr );
```

CREATE_TASK DESCRIPTION

This function creates and initialises the task control block (tcb) and creates a ready to run stack for the task name passed in. The stack is dynamically allocated on the UNOS heap. The routine also provides a mechanism to allow the global variables for a task to be initialised before any tasks in the system start to run. This mechanism is ideal to set up any semaphores that are to be used by a task. In addition a mechanism is provided to initialise local variables to a task (via the **local_var_ptr** parameter). When the task is created the routine also creates a mail box associated with the task. Since the size of the various mailbox components are passed into the routine then the mail box for individual tasks can be customized. Implementing the mail boxes in this section of code also allows the name of the task to be placed into the mail exchange so that the address for mail box messages is the created task's name as far as other user tasks are concerned. When reading information from mail boxes a task always reads from its own mail box.

CALL RESTRICTIONS

UNOS currently *does* not support dynamic task creation and destruction. Therefore all tasks must exist before the operating system begins executing. Therefore the **create_task** function has to be called from the **create_user_tasks** function (which is in the **usertask.cpp** module) during system initialisation. This function itself is called from the main function after the **set_up_os_data_structures** function has been called. Therefore the correct order for these calls is automatically preserved.

PARAMETERS

task_name_ptr Pointer to a **NULL** terminated ascii string containing the name of the task.

priority_of_task The priority of the task to be created. A number of equates are defined in "taskdefs.h" for the priorities. These include defining the number of priorities defined in the system.

task_tick_delta Variation of the time slice tick number around the default time slice number. The default number of ticks per kernel entry is defined in the **setup_os_data_structures** function. This parameter allows this value to be altered from the default by adding the number passed in to the default value. For example, if the default value was 4 ticks per time slice and one desired this task to only run for 2 ticks before a time slice kernel entry then the **task_tick_delta** value would be: -2.

status_of_task Initial status of the created task - **task_runnable** or **task_blocked**.

q_type	Initial queue type which the created task is to placed on (e.g. semaphore queue, priority queue), denoted by the following equates: PRIORITY_Q_TYPE – place on a priority queue whose priority is passed into the routine; SEMAPHORE_Q_TYPE – place on a semaphore blocked queue, the queue number being specified by the next parameter; DONOT_Q_TYPE – don't place on any queue (used for the null task).
semaphore_num	Semaphore number if created task is to be placed on a semaphore queue.
task_stk_size	Size of the task's stack (bytes). One can specify a custom stack size or use the generic stack size TASK_STACK_SIZE .
mess_q_size	Task's mail box message queue size.
mess_size	Maximum size of mail box message for the task's mail box.
init_task	Address of the task initialisation function. This function is usually defined in the module containing the task, and is most often used to define global variables in the module related to the task being created. For example a common use is to create the semaphores associated with the task.
task	Address of the task function.
local_var_ptr	This is a pointer to an arbitrary structure. This structure can contain information which the user wishes to pass into the task that is being created. This information is then local to the task in a local sense (although it is obviously known within the function that is calling create_task).

RETURNS

The function returns a **TRUE** if the task creation has been successful, else it returns a **FALSE**.

NOTES

- (i) If the task is to be placed on a semaphore queue then obviously the semaphore must exist before the task's tcb can be placed on it. Therefore the task initialisation function must create the semaphore. The task initialisation function is called during the task creation sequence before the an attempt is made to place the task on the semaphore queue.
- (ii) The **init_task** function does not have any parameters.
- (iii) The data which the **local_var_ptr** points to has to remain static in the system - i.e. it must persist for the total duration of the software execution. One convenient method of ensuring this is to create the data on the UNOS heap via a call to the **umalloc** or **ucalloc** memory allocation routines. If one desired the data could be copied from the heap into stack based variables which the task starts, and then the heap memory could be freed via a call to **ufree**.
- (iv) **All** task function definitions have allow the passing of a **void** parameter pointer, regardless of whether they require any local variables to be defined or not.

```
int change_task_priority ( char *task_name_ptr,  
                           unsigned char new_priority );
```

CHANGE_TASK_PRIORITY DESCRIPTION

This function allows the priority of a task to be changed.

CALL RESTRICTIONS

None (can be called from an interrupt routine).

PARAMETERS

task_name_ptr A pointer to a **NULL** terminated ascii string containing the name of the task.

new_priority The new priority of the task.

RETURNS

If the change in the priority is successful then a **TRUE** is returned, else a **FALSE** is returned.

```
unsigned int rtn_task_priority ( char *task_name_ptr );
```

RTN_TASK_PRIORITY DESCRIPTION

This function returns the current static priority of a task.

CALL RESTRICTIONS

None

PARAMETERS

task_name_ptr A pointer to a **NULL** terminated ascii string containing the name of the task.

RETURNS

Priority of the task. If the **task_name_ptr** parameter is illegal then **0xffff** is returned for the priority.

```
void start_tasks ( void ( *null_task ) ( void* ) );
```

START_TASKS DESCRIPTION

This function starts the operating system running. When this routine is called it is assumed that all the tasks have been created. The first task which begins execution is the **null_task**.

CALL RESTRICTIONS

This function should only be called from the main function after all the tasks have been created and all the hardware and software entities in the system have been initialised. It should not be called from any other place.

PARAMETERS

null_task This is the address of the **null_task** in the system. Note that the **null_task** has to be defined to accept a **void*** parameter, even though it does not use the parameter.

RETURNS

Nothing.

```
unsigned int rtn_current_task_num ( void );
```

RTN_CURRENT_TASK_NUM DESCRIPTION

This function returns the task number of the currently executing task.

CALL RESTRICTIONS

None

PARAMETERS

None

RETURNS

Task number of the currently executing task.

```
char *rtn_current_task_name_ptr ( void );
```

RTN_CURRENT_TASK_NAME_PTR DESCRIPTION

This function returns a pointer to the character string which contains the task name of the currently executing task.

CALL RESTRICTIONS

None

PARAMETERS

None

RETURNS

Task name pointer of the currently executing task.

```
int chg_task_tick_delta ( char *task_name_ptr, int new_tick_delta );
```

CHG_TASK_TICK_DELTA DESCRIPTION

This function allows the time slice number for a particular task to be varied around the default value established during task creation. This then allows the facility that tasks of the same priority can be given different amounts of execution time before they are switched out due to a time slice interrupt - i.e. effectively another level of priority control.

When a task is created it is given a default value for the number of ticks before a time slice (this is defined in the **setup_os_data_structures** function. This value can then be increased or decreased by the value passed into this function by passing in positive or negative values respectively. If one passes in a negative value larger then the default value then the tick value is set to one.

CALL RESTRICTIONS

None

PARAMETERS

task_name_ptr A pointer to a **NULL** terminated ascii string which contains the tasks name.

new_tick_delta An integer which represents the value which will be added to the default task tick number.

RETURNS

If the change is successful then **TRUE** is returned, else a **FALSE** is returned.

Task Scheduling Management Functions

```
void preemptive_schedule ( void );
```

PREEMPTIVE_SCHEDULE DESCRIPTION

This function requests that the kernel determine whether there are higher priority tasks which are runnable. If there is are higher priority runnable tasks then a task switch will occur to the highest priority one of these tasks. If there is a runnable task of the same priority as the calling task then this task *will not* be run, and the function will return with the calling task still the current task.

CALL RESTRICTIONS

This function should not be called from the middle of an interrupt routine, since it can result in a task switch which may lead to corruption of data associated with the interrupt routine. If it is called from an interrupt routine then the call should be at the end of the interrupt routine.

PARAMETERS

None

RETURNS

Nothing

```
void reschedule ( void );
```

RESCHEDULE DESCRIPTION

This function requests that the kernel reschedule another runnable task. This function differs from **preemptive_schedule** in that it always causes a task switch to occur, even if the task switched to is lower priority than the calling task.

CALL RESTRICTIONS

Care needs to be taken when calling from an interrupt routine that the call is located at the end of the interrupt routine.

PARAMETERS

None

RETURNS

Nothing

```
void start_time_slice ( void );
```

START_TIME_SLICE DESCRIPTION

This function allows time slice based scheduling to occur.

CALL RESTRICTIONS

None

PARAMETERS

None

RETURNS

Nothing

```
void stop_time_slice ( void );
```

STOP_TIME_SLICE DESCRIPTION

This function disables time slice based scheduling.

CALL RESTRICTIONS

None

PARAMETERS

None

RETURNS

Nothing

```
void chg_base_ticks_per_time_slice (int new_base_ticks_per_time_slice );
```

CHG_BASE_TICKS_PER_TIME_SLICE DESCRIPTION

This function changes the default number of ticks that a task executes for before a time slice entry into the kernel. This change is global -i.e. applies to all the tasks in the system. Some existing tasks may have negative task tick deltas which are larger than the new value for the default tick. In this case the time slice tick is adjusted to one for these tasks.

CALL RESTRICTIONS

None

PARAMETERS

new_base_ticks_per_time_slice The new default value of the ticks per time slice.

RETURNS

Nothing

Time Management Functions

```
timer_struct* create_timer ( void );
```

CREATE_TIMER DESCRIPTION

This function allocates a timer data structure on the UNOS memory heap and then places this timer structure into the linked list of inactive timers on the inactive timer queue.

CALL RESTRICTIONS

None

PARAMETERS

None

RETURNS

Returns a pointer to the timer data structure.

```
timer_struct* start_timer ( unsigned char timer_type,
                           unsigned long init_time_count,
                           void ( * timeout_handler ) ( void*),
                           void* data_ptr );
```

START_TIMER DESCRIPTION

This function starts a timer that is on the queue of inactive timers. Once the timer has been started then it is on the queue of active timers. The timer system supports two main types of timers – repetitive and single shot. A repetitive timer automatically resets when the timer times out. This type of timer is useful for generating accurate execution of functions and tasks. A single shot timer executes the time out action once and needs to be manually restarted to carry out another timing action.

CALL RESTRICTIONS

None

PARAMETERS

timer_type Specifies the type of timer – repetitive (use **REPETITIVE** definition from **GENERAL.H**) or single shot (use **SINGLE_SHOT** from **GENERAL.H**).

init_time_count This is the time value for the timer in system clock ticks.

timeout_handler Pointer to the function that is executed upon timeout of the timer. The function which is executed upon timeout of the timer (**timeout_handler**) must have a **void*** parameter.

data_ptr A pointer to an arbitrary data structure that can be used by the **timeout_handler**.

RETURNS

If the start timer command is successful then a pointer to the now active timer structure is returned. If the start of the timer is unsuccessful (due to a timer not being available) then a **NULL_PTR** pointer is returned.

OTHER RESTRICTIONS

It should be noted that the **timeout_handler** routine in the above parameter list is called at the kernel level. This means that there are restrictions on what this routine should do. For example, the routine should not carry out lengthy operations because interrupts are disabled whilst these are being carried out.

The normal thing to do with timer handler routines is to signal that some task is to run (using the **usignal** kernel call). The **usignal** is designed to work correctly in this situation, and there is no possibility of a problem occurring.

```
timer_struct* reset_timer ( timer_struct* timer_ptr );
```

RESET_TIMER DESCRIPTION

This function resets the time count value in an active timer back to its initial count value.

CALL RESTRICTIONS

None

PARAMETERS

timer_ptr Pointer to the timer structure of the timer that one wishes to reset.

RETURNS

A pointer to the timer structure which has been reset. If the reset is unsuccessful a **NULL_PTR** pointer is returned.

```
timer_struct* stop_timer ( timer_struct* timer_ptr );
```

STOP_TIMER DESCRIPTION

This function stops an active timer and then removes it from the active timer queue and places it into the inactive timer queue.

CALL RESTRICTIONS

None

PARAMETERS

timer_ptr Pointer to the timer structure that one wishes to stop.

RETURNS

Pointer to the timer structure of the timer that has been placed into the inactive timer queue. If the stop operation has not been carried out a **NULL_PTR** pointer is returned.

Intertask Communication and Synchronization Functions

```
SEMAPHORE create_semaphore ( unsigned int semaphore_value );
```

CREATE_SEMAPHORE DESCRIPTION

This function dynamically creates a semaphore. The semaphore is initialised with a value passed to it, and the semaphore multiplier value is set to 1. These values can be changed by calling the **init_semaphore** function.

CALL RESTRICTIONS

This function should not be called from an interrupt routine because the creating task's name is stored as part of the semaphore structure. Since an interrupt routine is usually unconnected to any task, it is not sensible in most circumstances to create semaphores here.

PARAMETERS

Initial value of the semaphore.

RETURNS

An unsigned integer representing the semaphore number. This type has been given the **#define** definition of **SEMAPHORE** in an effort to make the code more readable. If the semaphore creation is unsuccessful then **0xffff** is returned.

```
void init_semaphore ( SEMAPHORE sema_num, unsigned int sema_value,  
                     unsigned int multiplier_value );
```

INIT_SEMAPHORE DESCRIPTION

This function allows the default values of the semaphore structure to be modified, namely the semaphore value and the multiplier value. The multiplier value is the number of signals that effectively occur when a single **usignal** call is carried out (normally 1 for a conventional semaphore). For example if the semaphore multiplier value is 5 and if there were 5 tasks blocked on the semaphore then a single call to **usignal** would result in all 5 tasks becoming unblocked. If there were only 3 tasks blocked then the 3 tasks would be unblocked and the semaphore value would be left with a value of 2.

CALL RESTRICTION

In a strict sense there are no restrictions, however generally this routine would be called just after creating a semipro.

PARAMETERS

sema_num	The number of the semaphore which is to have its values modified.
sema_value	The new value of the semaphore count value.
multiplier_value	The value which is used to multiply the effect of a single usignal .

RETURNS

Nothing

```
void wait ( SEMAPHORE semaphore_num );
```

WAIT DESCRIPTION

This function implements the “wait” synchronization primitive. If as a result of a wait the calling task becomes blocked on a shared resource then a task switch will occur to another runnable task. See the “CAVEAT” section below for notes about the usage of this primitive.

CALL RESTRICTIONS

Because a **wait** can result in a task switch care must be taken if it is used in an interrupt routine. The **wait** statement should be placed at the end of the interrupt routine. In general it is not a good idea to use a **wait** inside an interrupt.

PARAMETERS

semaphore_num Number of the semaphore upon which one wishes to “wait”.

RETURNS

Nothing.

CAVEAT

The wait primitive is not used for critical section protection in the UNOS system. Technically it can still be used, since the “wait” operates on a full implementation of a semaphore. However, the semaphore primitives do not handle priority inversion at all, therefore there is likely going to be priority inversion problems in any real-time implementation above the trivial level. The “lock” primitive should be used for critical section protection. Semaphores should only be used for task synchronization and resource counting.

```
int timed_wait ( SEMAPHORE sema_num, unsigned long timeout_value );
```

TIMED_WAIT DESCRIPTION

This function carries out a “wait” which has a user definable duration. If the calling task becomes blocked on the semaphore and if it is not released within the user nominated time then an effective signal occurs on the semaphore resulting in the calling task becoming unblocked.

CALL RESTRICTIONS

As with the **wait** function it is generally not a good practice to use the **timed_wait** function in an interrupt routine. If it must be used ensure that it is one of the last statements in the routine.

PARAMETERS

sema_num	The semaphore number upon which the “wait” is to occur.
timeout_value	A count representing the number of tick interrupts before a timeout will occur. If the routine is called with a timeout value of zero then the routine returns immediately indicating that the return is due to a timeout.

RETURNS

- A “zero” return code indicates that the task has become unblocked as a result of a “**usignal**”.
- A “one” return code indicates that the task has become unblocked as a result of a time out.
- A “two” return code indicates that the return has occurred because there are no more timers available to carry out the **timed_wait** operation.

CAVEAT

The **timed_wait** primitive is not used for critical section protection in the UNOS system. Technically it can still be used, since the “wait” operates on a full implementation of a semaphore. However, the semaphore primitives do not handle priority inversion at all, therefore there is likely going to be priority inversion problems in any real-time implementation above the trivial level. The “lock” primitive should be used for critical section protection. Semaphores should only be used for task synchronization and resource counting.

```
void usignal ( SEMAPHORE semaphore_num );
```

USIGNAL DESCRIPTION

This function implements the “signal” synchronization primitive. If no task is waiting on the semaphore that the “signal” is carried out on, then the value of the semaphore is incremented by the value in the **semaphore_multiplier** component of the semaphore structure. If a task is blocked on the semaphore then the “signal” will unblock the task. If the number of tasks blocked is up to the value contained in the **semaphore_multiplier** location, then a single call to **usignal** will unblock all of them. If there are less tasks blocked than the value in **semaphore_multiplier** then the semaphore value will be set at the difference between the number in **semaphore_multiplier** and the number of blocked tasks. Read the “CAVEAT” section below for issues related to the use of **usignal**.

CALL RESTRICTIONS

The **usignal** function can be called from interrupt and non-interrupt routines. If called from an interrupt routine then it is important that the call to **usignal** be placed after the manipulation of all global data structures, since the “signal” may cause a task switch and the possibility of re-entry into the interrupt routine.

PARAMETERS

semaphore_num This is the number of the semaphore that the “signal” operation has to be carried out on.

RETURNS

Nothing.

CAVEAT

The **usignal** primitive is not used for critical section protection in the UNOS system. Technically it can still be used, since the “usignal” operates on a full implementation of a semaphore. However, the semaphore primitives do not handle priority inversion at all, therefore there is likely going to be priority inversion problems in any real-time implementation above the rival level. The “lock” primitive should be used for critical section protection. Semaphores should only be used for task synchronization and resource counting.

```
unsigned int return_semaphore_value ( SEMAPHORE sema_num );
```

RETURN_SEMAPHORE_VALUE DESCRIPTION

This function returns the value of the semaphore count variable. The semaphore implementation in UNOS only supports semaphore values which are zero or positive.

CALL RESTRICTIONS

None

PARAMETERS

sema_num This is the number of the semaphore that one wants to get the semaphore value from.

RETURNS

The value of the semaphore count. If the semaphore number passed in is illegal then the return value is **0xffff**.

LOCK create_lock (void);

CREATE_LOCK DESCRIPTION

This function is called to dynamically create a “lock” primitive. The “lock” primitive is used for critical section protection in the UNOS operating system. This primitive is designed so that priority inversion is eliminated when low and high priority share common resources.

Lock primitives (unlike semaphores) do not have a value – they are simply locked or unlocked. In addition there are no central kernel queues associated with locks.

CALL RESTRICTIONS

None.

PARAMETERS

None.

RETURNS

A **LOCK** primitive. The **LOCK #define** is actually a pointer to a lock structure.

```
void lock (LOCK l);
```

LOCK DESCRIPTION

This function is called prior to entering a critical section of code. If the shared resource is not being used by any other task then the calling task will lock the resource preventing any other task from accessing it until it unlocks the resource. If the shared resource has already been locked by another task then the current task will become blocked on the resource and the scheduler will be called to find the eventually holder of the resource and run it so that the resource may eventually be freed to the current task. This makes sense because the currently executing task is obviously the highest priority runnable task in the system, and therefore a task of the same priority or lower priority must be holding the resource. Therefore this task must run at the priority of the calling task so that the resource will be released and the calling task can continue. Situations involving transitive blocking can also be handled by the system.

CALL RESTRICTIONS

Don't use this function in interrupt routines as a number of tasks could run as the result of a lock operation. This can prevent the task running at the time of the interrupt from running for a long period. Furthermore there is the possibility of corruption and system crash as the task running at the time of the interrupt could be the task that is holding the lock. Therefore the kernel would attempt to run it again and this would mean that the interrupt lock call made in the interrupt routine would run again (since this is running in the context of the task running at the time of the interrupt) and hence there would be multiple recursions into the kernel until the stack overflows and the system crashes.

PARAMETERS

A **LOCK** structure for a lock that has previously been created.

RETURNS

Nothing.

```
void unlock (LOCK l);
```

UNLOCK DESCRIPTION

This function is called to release a resource that has been protected by a critical section. If the currently running task has been blocking a high priority task that wished to use the protected resource, then this call can result in a task switch to that higher priority task. Otherwise the lock state is set to unlocked and the calling task continues.

CALL RESTRICTIONS

Should not be used in interrupt routines, since the original **lock** primitive should not be used in these routines (see the **lock** description).

PARAMETERS

A pointer to a lock structure – i.e. a **LOCK**.

RETURNS

Nothing.

```
LOCK destroy_lock (LOCK l);
```

DESTROY_LOCK DESCRIPTION

This function is called to deallocate the storage previously allocated on the heap for a lock structure.

CALL RESTRICTIONS

None.

PARAMETERS

Pointer to a lock structure that has previously been created.

RETURNS

The **NULL_PTR**.

```
int send_mess ( unsigned char* mess_ptr, unsigned int mess_lgth,  
               char *mess_addr_ptr, int block_action )
```

SEND_MESS DESCRIPTION

This function sends messages consisting of a string of bytes to a task mail box. The action depends on the value of the **block_action** parameter. If there is room for the message in the mail box then a return from the function will occur as soon as the bytes of the message are transferred to the mail box. If the mail box is full then the calling task will become blocked until a message space (also known as an envelope) becomes available if **block_action** = **BLOCK_SM**, else if **block_action** = **NON_BLOCK_SM** control is returned to the caller with the code **FULL_MBX**. If the message string is too long for the envelope in the mail box then the function returns an error code to the caller.

Note that interrupts are disabled for considerable periods if one attempts to send very long messages via this mechanism. If it is still desirable to send the messages via mail boxes then break them into pieces.

CALL RESTRICTIONS

In interrupt routines only use the non-blocking form of the routine to prevent blockage. In general one cannot think of sensible ways of using this in interrupt routines.

PARAMETERS

- | | |
|----------------------|--|
| mess_ptr | A pointer to the first byte of the message to be sent. |
| mess_lgth | The length of the message to be sent in bytes. |
| mess_addr_ptr | The address of the task name string for the recipient task. This is effectively the address of the recipient task. |
| block_action | The value here determines whether the caller should be blocked on the mail box if it is full. If the parameter is BLOCK_SM then the task will become blocked on a full mail box, and if the parameter is NO_BLOCK_SM then control is returned immediately to the calling task. |

RETURNS

- If the message has been successfully placed in the mail box then a **TRUE** is returned.
- A **FALSE** will be returned if the message is too large to fit into the mail box envelope or if the task address is illegal.
- **FULL_MBX** is returned if called with **NO_BLOCK_SM** and the mail box is full.

```
int SEND_MESS(unsigned char* mess_ptr, unsigned int mess_lgth,  
              char* mess_addr_ptr)
```

SEND_MESS MACRO DESCRIPTION

This macro is a convenient way of bundling the normal **send_mess** function. It allows the user to use the old **send_mess** syntax (i.e. leaving off the **block_action** parameter). This macro sets this parameter to **BLOCK_SM**, therefore it behaves in the manner of the old version of **send_mess** – that is the calling task is blocked on a full mail box.

This function sends messages consisting of a string of bytes to a task mail box. If there is room for the message in the mail box then a return from the function will occur as soon as the bytes of the message are transferred to the mail box. If the mail box is full then the calling task will become blocked until a message space (also known as a envelope) becomes available. If the message string is too long for the envelope in the mail box then the function returns an error code to the caller.

Note that interrupts are disabled for considerable periods if one attempts to send very long messages via this mechanism. If it is still desirable to send the messages via mail boxes then break them into pieces.

CALL RESTRICTIONS

Don't use this function in interrupt routines because of the potential for blockage.

PARAMETERS

mess_ptr	A pointer to the first byte of the message to be sent.
mess_lgth	The length of the message to be sent in bytes.
mess_addr_ptr	The address of the task name string for the recipient task. This is effectively the address of the recipient task.

RETURNS

- If the message has been successfully placed in the mail box then a **TRUE** is returned.
- A **FALSE** will be returned if the message is too large to fit into the mail box envelope or if the task address is illegal.

```
int SEND_MESS_NB(unsigned char* mess_ptr, unsigned int mess_lgth,  
                 char* mess_addr_ptr)
```

SEND_MESS_NB MACRO DESCRIPTION

This macro is a convenient way of bundling the normal **send_mess** function. It allows the user to use the old **send_mess** syntax (i.e. leaving off the **block_action** parameter). This macro sets this parameter to **NON_BLOCK_SM**, therefore it does not block on a full mail box.

This function sends messages consisting of a string of bytes to a task mail box. If there is room for the message in the mail box then a return from the function will occur as soon as the bytes of the message are transferred to the mail box. If the mail box is full then control is returned to the caller with the code **FULL_MBX**. If the message string is too long for the envelope in the mail box then the function returns an error code to the caller.

Note that interrupts are disabled for considerable periods if one attempts to send very long messages via this mechanism. If it is still desirable to send the messages via mail boxes then break them into pieces.

CALL RESTRICTIONS

In general one cannot think of sensible ways of using this function in interrupt routines, therefore it is best not to use it.

PARAMETERS

mess_ptr	A pointer to the first byte of the message to be sent.
mess_lgth	The length of the message to be sent in bytes.
mess_addr_ptr	The address of the task name string for the recipient task. This is effectively the address of the recipient task.

RETURNS

- If the message has been successfully placed in the mail box then a **TRUE** is returned.
- A **FALSE** will be returned if the message is too large to fit into the mail box envelope or if the task address is illegal.
- **FULL_MBX** is returned if called with **NO_BLOCK_SM** and the mail box is full.


```
int send_qik_mess ( unsigned char* mess_ptr, unsigned int mess_lgth,  
                    char * mess_addr_ptr );
```

SEND_QIK_MESS DESCRIPTION

This function sends priority messages to a task via the mail box mechanism. If there are other messages in front of the “qik message” then it goes to the front of the queue. If the mail box is full there is a special location in the mail box for a priority message and it is placed there. This is read before any of the other messages in the mail box. If the special envelope for the priority message is full then the function returns immediately with a code indicating this.

CALL RESTRICTIONS

Since blockage can occur this routine should not be used in interrupt routines, except perhaps at the end.

PARAMETERS

mess_ptr Pointer to the beginning of the message consisting of a string of bytes which is to be sent to the mail box.

mess_lgth Length of the message in bytes.

mess_addr_ptr The address of the task name string for the recipient task. This is effectively the address of the recipient task.

RETURNS

- 0->message too big to fit into the envelope.
- 1->message successfully sent.
- 2->special quick mail box envelope full.

```
char *rcv_mess ( unsigned char* mess_ptr, unsigned int* mess_lgth,  
                unsigned long time_limit );
```

RCV_MESS DESCRIPTION

This function receives a message from the mail box associated with the calling task. The message is placed into a byte array whose address is passed to the routine. If there is no message in the mail box then the calling task will become blocked until a message arrives. An important feature of the routine is that the user can set up a maximum blockage time, after which a return from the function call will occur regardless of whether a message has been received or not. If the timeout period has not expired the return code from the function is the address of the task which sent the message. This address is in terms of a pointer to a character string which is the name of the task.

CALL RESTRICTIONS

Since this function can cause blockage of the calling task care must be taken if used in an interrupt routine. In general it is not recommended that it be used in interrupt routines.

PARAMETERS

mess_ptr Pointer to a byte array which has sufficient storage to accept the maximum length message that can be sent through the mail box.

mess_lgth The length of the message received.

time_limit The maximum number of kernel tick routine entries that the calling task can be blocked on a message arrival. If the **time_limit** variable is zero then the blockage time is infinite.

RETURNS

In normal operation the task returns a pointer to the **NULL** terminated string that contains the name of the task. If the timeout period has expired before the arrival of a message then the **NULL_PTR** pointer is returned. If a timer was not available for the time limit function then a pointer of value **0xffff:0x000f** is returned.

```
unsigned int size_mbx ( char *mbx_addr_ptr );
```

SIZE_MBX DESCRIPTION

This function returns the size of the message queue, otherwise known as the number of envelopes, for a particular mail box.

CALL RESTRICTIONS

None

PARAMETERS

mbx_addr_ptr Pointer to the **NULL** terminated string that represents the name of a task. This is the mail box address of the task.

RETURNS

The size of the mail box message queue. If the mail box address is illegal then a zero is returned.

```
unsigned int size_mbx_mess ( char *mbx_addr_ptr );
```

SIZE_MBX_MESS DESCRIPTION

This function returns the size of the envelopes in a particular mail box. An envelope is an individual message holder which has a maximum length defined at mail box creation time.

CALL RESTRICTIONS

None

PARAMETERS

mbx_addr_ptr Pointer to the **NULL** terminated string which is the name of the task that owns the mail box.

RETURNS

The maximum size of a mail box message.

```
unsigned int free_mbx ( char *mbx_addr_ptr );
```

FREE_MBX DESCRIPTION

This function returns the number of *free* message slots (i.e. free envelopes) in a mail box.

CALL RESTRICTIONS

None

PARAMETERS

mbx_addr_ptr Pointer to the **NULL** terminated string which is the name of the task that owns the mail box.

RETURNS

The number of free message slots in the mail box. If the mail box address is illegal then **0xffff** is returned.

```
unsigned int used_mbx ( char *mbx_addr_ptr );
```

USED_MBX DESCRIPTION

This function returns the number of *used* message slots (i.e. envelopes) in a mail box.

CALL RESTRICTIONS

None

PARAMETERS

mbx_addr_ptr Pointer to the **NULL** terminated string which is the name of the task that owns the mail box.

RETURNS

The number of used message slots in the mail box. If the mail box address is illegal then a **0xffff**.

```
void flush_mbx ( void );
```

FLUSH_MBX DESCRIPTION

The function flushes or clears all the messages stored in the mail box owned by the calling task. If the mail box was full at the time the flush is executed then there is the possibility that other tasks wishing to write to the mail box could be blocked – all such tasks are released.

CALL RESTRICTIONS

Since there is a possibility of a task switch occurring when the flush function is executed then use of the function from within an interrupt routine is not recommended.

PARAMETERS

None

RETURNS

Nothing.

Memory Management Functions

```
char huge* umalloc ( unsigned long num_bytes );
```

UMALLOC DESCRIPTION

This function implements a dynamic memory allocation strategy which is similar to that offered by the standard 'C' **malloc** routine. The routine allocates a user defined number of bytes of storage on from the UNOS heap and passes a pointer to the start of this block of memory back to the caller.

CALL RESTRICTIONS

None

PARAMETERS

num_bytes Number of bytes of storage to be allocated from the UNOS heap.

RETURNS

If the memory allocation has been successful then a pointer to the start of the block of memory is returned. On 8086 machines the pointer to the memory block is "normalized" so that the full segment space (with the exception of up to 15 bytes) of 64KBytes is available in the allocated block without getting offset wrap around. If unsuccessful then a **NULL_PTR** pointer is returned.


```
char huge* ucalloc ( unsigned long num_obj,  
                    unsigned long size_of_obj );
```

UCALLOC DESCRIPTION

This function implements the same functionality as the **calloc** routine from standard 'C' except that the memory is allocated from the UNOS heap. The function returns a pointer to an amount of memory which can store a user specified number of a specified data object.

CALL RESTRICTIONS

None

PARAMETERS

num_obj	Number of objects that memory should be allocated for.
size_of_object	The size of each object in bytes.

RETURNS

Pointer to the beginning of the area of memory allocated. On 8086 machines the pointer to the memory block is "normalized" so that the full segment space (with the exception of up to 15 bytes) of 64KBytes is available in the allocated block without getting offset wrap around. If there is a problem allocating the memory a **NULL** pointer is returned.

```
void ufree ( char huge* blk_ptr );
```

UFREE DESCRIPTION

This function allows memory dynamically allocated on the UNOS memory heap to be freed for other uses. In order to minimise heap fragmentation the function concatenates adjacent blocks of free memory with the newly freed block.

CALL RESTRICTIONS

None

PARAMETERS

blk_ptr	Pointer to a block of memory previously allocated from the UNOS heap.
----------------	---

RETURNS

Nothing

```
unsigned long ret_free_mem ( void );
```

RET_FREE_MEM DESCRIPTION

This function returns the amount of free memory on the UNOS heap. Note that this is the total amount of free memory and does not reflect the size of the largest block of free memory.

CALL RESTRICTIONS

None

PARAMETERS

None

RETURNS

The total amount of free memory on the UNOS heap in bytes.

Miscellaneous Functions

```
char return_interrupt_status ( void );
```

RETURN_INTERRUPT_STATUS DESCRIPTION

This function returns the status of the interrupt flag, so that the status of the interrupts can be determined (i.e. enabled or disabled).

CALL RESTRICTIONS

None

PARAMETERS

None

RETURNS

0->interrupts disabled.
1->interrupts enabled.

Introduction

This Appendix describes the keyboard handling system from a users perspective. The DOS keyboard handling system cannot be used very successfully in UNOS because UNOS is a multi-tasking operating system, and the BIOS/MS-DOS based keyboard routines are not reentrant. Therefore, to remove any problems that may arise due to the execution of BIOS routines it was decided to write a stand-alone keyboard handling system. This involved intercepting the interrupt vector generated by the keyboard controller and then decoding the scan codes received from the keyboard controller.

Organisation and Use

The keyboard handling system is organised as shown in Figure D-1. Notice that it has two tasks – one task has the responsibility of decoding the scan codes to ASCII code, and the other task then distributes these code to the user tasks that should receive keyboard input. Multiple tasks can simultaneously receive characters from the keyboard.

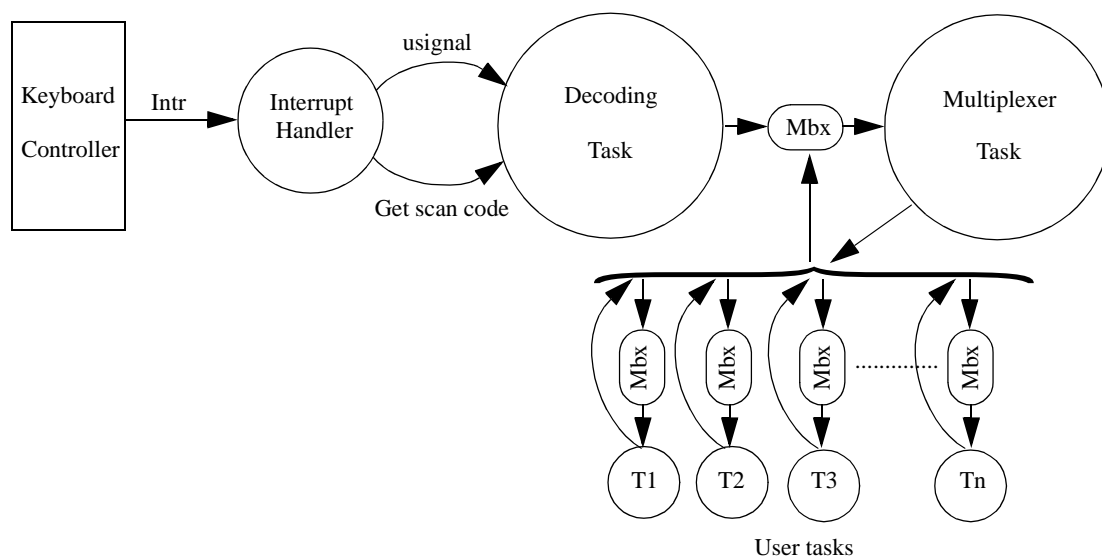


Figure D-1: Block diagram of the structure of the keyboard decoding system

In order for a user task to connect the keyboard it must send a message to the multiplexer task requesting that it be connected to the keyboard. The multiplexer task keeps track of the tasks that have sent connection requests and then sends any characters it receives from the decoder task to the connected tasks.

Appendix D — *THE KEYBOARD SYSTEM*

In addition to being able to connect to the keyboard, user tasks can also disconnect themselves as well. Both these actions can be carried out via a very simple protocol which is shown in Table D-1. These characters are sent by a user task to the mail box of the multiplexer task. An example of a connection to the keyboard would be:

```
SEND_MESS_NB(CONNECT_KB, 1, pchKeyboardMultiPlexerTask);
```

where **pchKeyboardMultiPlexerTask** is the name of the multiplexer task.

Table D-1: Keyboard Multiplexer Protocol^a

Character	Action
C (defined as CONNECT_KB)	Connect sending task to the keyboard.
D (defined as DISCONNECT_KB)	Disconnect sending task from the keyboard.
X (defined as DISCONNECT_ALL_KB)	Disconnect all tasks from the keyboard.

a. The #define definitions used in this table are in the general.h header file.

INTRODUCTION

The software for the serial handler used in UNOS is contained in three files - **SERIAL.C**, **SERINIT.C** and **SERINT.C**. In addition there are header files with the same name but having the “.H” extension. The primary function of each of these files is as follows:-

- **SERIAL.C** - This file contains the main functioning code for the tasks associated with the serial channels. This is a rather large file due to the fact that three different types of uarts are to be supported in the current implementation of the drivers - the Intel 8251, the Intel 8274 and the National NS16450 (the uart used in the IBM PC). **Currently only the NS16450 is functional.** However there is considerable code in the file for the other uarts - this code coming initially from the Pipeline Crack Investigation Vehicle project. This module would not normally be altered by a user.
- **SERINIT.C** - As the name of the file implies this is the file that contains the user configurable code which will create the serial channels and the function in the module is called during UNOS initialization. The body of the module normally contains one or more calls to the **create_serial_channel** function.
- **SERINT.C** - This file contains the interrupt routines and two functions which initialize the serial interrupt routine data base and then link a particular interrupt number and buffer with an interrupt routine. This function is not normally modified by the user. However in the current implementation only 20 interrupt routines are supported. If more than 20 were required then the number can be easily increased by editing this module.

One of the primary design objectives of the UNOS serial handlers was to allow a number of serial channels to be set up by the user by a simple call to a procedure. All the other details should be hidden from the user. With certain limitations the UNOS serial handlers achieve this. The main limitation (in the current implementation) is that the serial channels are simple asynchronous channels with bidirectional control S/control Q protocol controlling the flow of data into and out of the channel. Within this limitation the user can program such things as the number of stop bits, the number of data bits, parity on or off, the size of the interrupt buffers etc.

USER INTERFACE

The primary user interface to the serial handler is the function **create_serial_channel**. This routine passes in a large number of parameters which allow the characteristics of the particular serial channel to be customized. A call to this function always sets up the total number of serial channels for the particular uart being used. For example for an Intel 8251 the receive and transmit channels would be set up. One cannot just set up the transmitter channel alone, for example. Similarly for the Intel 8274 (which is a dual uart) two complete channels are established - i.e. two transmit/receive channels.

Associated with each channel set up are two tasks - a task associated with the transmit section

of the channel, and a task associated with the receive section of the channel. These tasks are normal UNOS tasks, therefore their formation in turn means that a number of mail boxes are created, which also implies that semaphores are also created (associated with the mail boxes).

Another side effect of the serial channel formation is the creation of interrupt routines. The number of interrupt routines created depends on the type of uart. For example the Intel 8251 has a separate interrupt line for the transmitter and the receiver. This allows the hardware priorities of the transmitter to be individually defined. On the other hand the Intel 8274 and the NS16450 only have a single interrupt line for all their internal interrupt sources. In this case the source of the interrupt internally has to be determined by a single interrupt routine and the handling routine for the particular interrupt source is then called. All of these details are hidden from the user, however the number of interrupt routines used is useful to know in cases where there is a possibility of exceeding the default number of interrupt routines.

The details of the serial channel task creation function appear towards the end of this document.

SETTING UP AND USING THE SERIAL CHANNELS

Before calling `serial_init` the `serial_init` function in the **SERINT.C** module has to be called. This function initialises the data structures required for the automated setup of the serial interrupt routines. The calling convention is `serial_init();`.

In order to output something to a serial channel one simply sends the data to be output to the mail box for the transmit task for the particular serial channel. It is up to the user to ensure that the mail box is of sufficient size to accept all messages that will be sent to it.

In order to receive characters from a serial channel one has to “connect” the receive task output for a particular channel to some receiving task. This is achieved by the receiving task sending a message to the mail box for the particular serial channel. The message does not matter - it can be anything, a single character for example. The relevant affect of the message is that the return address is tagged to it, and this is how the serial receive task knows what mail box to send received characters to. It is intended in the future that a single serial channel can be connected to multiple tasks by the same mechanism. This feature is *not* present in the Cook Islands version of the software.

The header files

The other major files that are part of the serial driver software package are the header files. From time to time these may have to be modified to accommodate additions to the serial driver module. The contents of these header files are largely self explanatory and will not be discussed in detail in this document.

The header files are:-

- (i) **SERIAL.H** – The header file for the **SERIAL.C** module. This file should not normally have to be altered by the user. There are “`UART_TYPE`” in this file for the various uarts that the software supports. It is intended to support the NS16450, Intel 8251A and the Intel 8274 uarts, however at the time of writing this document *only the NS16450 uart is supported*. However considerable support is already present for the other uarts.
- (ii) **SERINT.H** – The header file for the **SERINT.C** interrupt routine module. This file only contains the prototype for the function which sets up the interrupt routines, and consequently would not normally be modified by a user.
- (iii) **SERINIT.H** – The header file for the **SERINIT.C** initialization module. This file only

SETTING UP AND USING THE SERIAL CHANNELS

contains the prototype for the
normally be modified by the user.

function and would not

USER INTERFACE DESCRIPTION

```
int create_serial_channel (char uart_type,
                          unsigned int baud_rate,
                          float stop_bits, char char_lgth,
                          char parity,
                          unsigned int command_reg,
                          unsigned int data_reg,
                          unsigned char division_ratio,
                          int alarm_num_base,
                          unsigned char rx_intr_num,
                          unsigned char tx_intr_num,
                          unsigned char tx_enable_disable_bit,
                          unsigned char intr_ctrl_reg_offset,
                          unsigned int rx_int_buf_size,
                          unsigned int tx_int_buf_size,
                          unsigned int buf_low_lim,
                          unsigned int buf_upper_lim,
                          unsigned int mbx_q_size,
                          unsigned int mbx_mess_size,
                          char num_8259s,
                          unsigned int addr_8259_mast,
                          unsigned int addr_8259_slave,
                          char* ch1_rx_name_ptr,
                          char* ch1_tx_name_ptr,
                          char* ch2_rx_name_ptr,
                          char* ch2_tx_name_ptr,
                          unsigned char rx_task_pri,
                          unsigned char tx_task_pri);
```

CREATE_SERIAL DESCRIPTION

This function is called to create a serial channel for UNOS. The extensive parameter list allows all the normal parameters associated with an asynchronous serial channel to be customized.

Upon return from this function the complete serial channel has been created including the tasks for the receive and transmit and the associated interrupt routines. It should be noted that this function does not allow the transmitter or receiver for a channel alone to be set up – the complete receive and transmit channel is established. If the uart is a dual uart then two complete receive/transmitter channels are established.

CALL RESTRICTIONS

The normal place to call this function is during system initialisation – i.e. before UNOS begins execution. This would normally be necessary because the hardware of the system is also initialised during this phase and consequently the interrupts routines would need to be set up to handle the interrupts being generated by the hardware.

PARAMETERS

uart_type One of the “ ” in the SERIAL.H file is passed in to indicate the type of uart that the serial channel is being set up for. *Currently the only type supported is the NS16450.*

baud_rate A number representing a baud rate. The NS16450 is capable of supporting a variety of baud rates – it has an on board divider and baud rate genera-

tor, therefore the baud rate clock is worked out within the creation routine based on the clock frequency driving the NS16450.

stop_bits The supported numbers are 1, 1.5 and 2 stop bits.

char_lgth	The supported lengths for the characters are 5, 6, 7, 8 bits.
------------------	---

parity Support for , and .
These values are defined in the header file SERIAL.H

command_reg This is the I/O address of the command register for the uart. In the case of uarts that have more registers than a command register and a status register this is the base address for all the registers. The other registers are accessed by adding offsets of the other registers to this value.

data_reg This is the I/O address from which received characters are read and transmitted characters are written.

divisor_ratio Some uarts have an internal divider (eg. the Intel 8251). This is the value which is programmed into this internal divider.

alarm_num_base Most uarts can generate a number of errors (eg. overrun errors, framing errors etc.), and this number is the base alarm number in the UNOS error handling system. All the other errors are consecutive offsets from this number. The offsets are defined in the **ALARMNUM.H** header file.

rx_intr_num	The interrupt number used by the uart receiver.
--------------------	---

tx_intr_num	The interrupt number used by the uart transmitter.
--------------------	--

tx_enable_disable_bit	A byte with the bit which controls the disabling and enabling of interrupts set.
------------------------------	--

intr_ctrl_reg_offset The offset of the interrupt control register from the command register base address.

rx_int_buf_size	The size of the receive interrupt routine circular buffer in bytes.
------------------------	---

tx_int_buf_size	The size of the transmit interrupt routine circular buffer in bytes.
------------------------	--

buf_low_lim The lower limit that must be reached on the content of the receive interrupt buffer for the issue of a control Q after the receiver has issued a control S.

buf_upper_lim The upper limit on the amount of data in the receive buffer before a control S is sent by the receive interrupt routine to the external sending device.

mbx_q_size	The number of messages that can be stored in the transmit tasks mail box.
-------------------	---

mbx_mess_size The maximum size of a message that can be stored in one of the mail box message holding structures (known as envelopes in UNOS terminology).

num_8259s This parameter contains the number of 8259 interrupt controllers used by the uarts. For example there may be zero 8259 if the uart is directly connected to the processors interrupt line, there may be a single 8259 in a non-cascaded interrupt structure, or there may be 2 8259s when the uart is connected to a slave 8259. The difference in these numbers determines how many 8259 “end of interrupt” commands are sent to the interrupt control hardware by the interrupt routines.

addr_8259_mast The base register I/O address of the 8259 master interrupt controller (if present).

addr_8259_slave The base register I/O address of the 8259 slave interrupt controller (if present).

ch1_rx_name_ptr A pointer to a **NULL** terminated string which contains the name to be given to the received task for channel 1.

ch1_tx_name_ptr A pointer to a **NULL** terminated string which contains the name to be given to the transmit task for channel 1.

ch2_rx_name_ptr A pointer to a **NULL** terminated string which contains the name to be given to the received task for channel 2 of a dual uart.

ch2_tx_name_ptr A pointer to a **NULL** terminated string which contains the name to be given to the transmit task for channel 2 of a dual uart.

rx_task_pri This is the priority of the receive buffer interface task.

tx_task_pri This is the priority of the transmit buffer interface task.

RETURNS

The function returns if the serial channel creation has been successful, else it returns a **FALSE**.

Introduction

In order to simplify the decoding of complex input, a command decoder has been written. The decoder is table driven, this approach allowing reliable decoding of very complex commands. The input to the command decoder can be user command input or can be protocols used between computers, or even between tasks on the same computer. The routines are written to be reentrant so they can be used to decode commands from multiple threads of execution.

Tasks wishing to use the command decoder do so by calling a special command decoder function which decodes the messages arriving in the calling tasks mail box. In order to allow a single task to receive messages from multiple sources the return address is used to essentially select which set of command tables will be used to decode the current message. This means that a single task can simultaneously decode messages from multiple tasks.

The decoder has been written so that it can decode messages of an arbitrary length – i.e. from a single character up.

Basic Data Structures

Much of the functionality of the command decoder can be gleaned from an understanding of the data structures used by it.

The data structure which is passed into the decoder when it is to decode a message is called the **addr_struct** and has the following form:

```
char huge* rtn_addr
char huge** valid_addr_tab_ptr
dec_input_struct** addr_vector_ptr
unsigned char* mess_ptr
unsigned int mess_lgth
void* info_struct_ptr
```

The function of each component is:-

- **rtn_addr** - This location contains the address of the task name which sent the current message. In the UNOS system this is effectively the task name.
- **valid_addr_tab_ptr** (valid address table pointer) - This location contains a pointer to a table of valid return addresses in the current context.
- **addr_vector_ptr** - This location contains a pointer to the table of pointers to the structures that will be used to decode the message from the particular sending task.
- **mess_ptr** - Pointer to the message which is to be decoded.
- **mess_lgth** - The length of the message pointed to by the mess_ptr location above.
- **info_struct_ptr** - A pointer to an arbitrary data structure that can contain information that will be passed into the handling routines via the **dec_input_struct**

info_struct_ptr component. This is designed to allow virtually completely reentrant tasks to be written – i.e. a reentrant task can use the *same* decoding tables, but the information passed to the decoder handling routines (see later for a discussion of what these are) can be different for the different tasks. The different information structure can be passed to the task when it is created via the last component of the **create_task** parameter list.

This structure serves to tell the decoder where the decode tables are which decide whether the current return address is a valid one in this context. The decoding is table based with the **valid_addr_tab_ptr** pointing to a table containing one or more pointers to character strings which represent the names of valid tasks that can be sending messages in the current context. As this table is searched an index is incremented down a corresponding array of structures called the **dec_input_struct**, each of which have the following fields:-

```
char char_dec_mode
unsigned char data_byte
unsigned char *valid_data_table_ptr
void ( **vector_table_ptr ) ( dec_input_struct * )
unsigned char *bit_table_ptr
void* info_struct_ptr
```

Each of the above fields can be changed dynamically during program execution. The function of the field is now described:-

- **char_dec_mode** (character decode mode) - This location contains a '0' or a '1' indicating whether the decoder should operate in ascii or binary mode. This has significance in relation to the way the character decoding is carried out. This is discussed in detail the following section on the **dec_input_struct**. 0=>ascii decoding; 1=>binary decoding.
- **data_byte** - This is the location which is used to feed the message string to the character decoder byte by byte.
- **valid_data_table_ptr** - This pointer points to a table containing the next valid character in a message.
- **vector_table_ptr** - This pointer points to a table of handling routines, one of which is activated depending on a matching character being found in the table pointed to by the **valid_data_table_ptr**. The pointed to routines are passed a pointer to the current **dec_input_struct** (i.e. this structure) so that the **valid_data_table_ptr** and **vector_table_ptr** locations can be modified to point to the next set of tables for the next character.
- **bit_table_ptr** - This pointer points to an array of bits which is used to aid in the decoding of binary data. The precise use of this table is explained in the following sections.
- **info_struct_ptr** - This pointer can point to any data structure which may be used in one of the handling routines. The contents of this pointer are initialised in the **decode_mbx_mess** function from the data stored in the **addr_struct.info_struct_ptr** variable.

These data structures function in the following manner when a message is to be decoded. The **addr_struct** is filled out with the appropriate pointers to the message, message length,

return address etc. and then the **decode_mess** routine is called with a pointer to this structure passed as a parameter. The **valid_addr_tab_ptr** and the **addr_vector_ptr** both point to the tables which provide the mapping between the return address and the command tables which will be used for the decoding. This table would normally be static in nature, being set up at task initialisation. However, it is also possible to have alternate mappings which could change relative to the context. A case where this might be useful is if at some stage a certain task should no longer talk to the receiving task. Therefore it would seem logical to change the mapping table so that this certain task is no longer in it. If the task attempted to sent something to the receiver then an error message would be generated.

If the return task number appears in the mapping to the decode tables then the structure pointed to is passed to the character decoder with the first byte of the message in the **data_byte** location. This character decoder is then called repeatedly until the message is exhausted. The **valid_data_table_ptr** and the **vector_table_ptr** locations are updated with each character processed by the handling routines. When the message is exhausted these two locations are left pointing to the set of tables which will process the next message. The exact format of the decoder tables will be discussed in detail in the next section.

The above description is by necessity a little complex. However, Figure F-1 allows one to visualise the relationship between the various tables.

Decoder Tables - How to use them

The user will have to manipulate two main table groups – the return address to decoder structure mapping table and the command decoder tables. As already mentioned above the mapping between the return address and the decoder structure is usually static and can be set up before a task is going to decode any input. These mapping tables have the following format:-

return addr 1	Decoder struc 1
return addr 2	Decoder struc 2
.	.
.	.
.	.
return addr n	Decoder struc n
return addr n+1	Decoder struc n+1
.	.
.	.
NULL_PTR	Term Dec. Struc

The return addresses are pointed to by the **valid_addr_tab_ptr** and the decoder structures are pointed to by the **addr_vector_ptr** components of the **addr_struct**. The **return addr** values listed in the table do not have to be in any particular order (e.g. do not have to be in numerical order). Consequently they also do not have to be continuous – values can be omitted. The decoder will search through this table looking for the value of the return address stored in the **return_addr** component of the **addr_struct** passed in. As an index moves down through the address table a matching index moves through the decoder structure table. If a match is found with the return address then the decoder structure becomes the one pointed to by the matching index. The return address at the end of the return address table is 0x0000:0x0000 – i.e. the **NULL_PTR** for a x86 processor. This address pointer is not used for any task name pointer (since it is an address residing in the interrupt vector area of the processor). If this address is reached in the search then it means that the return address passed into the

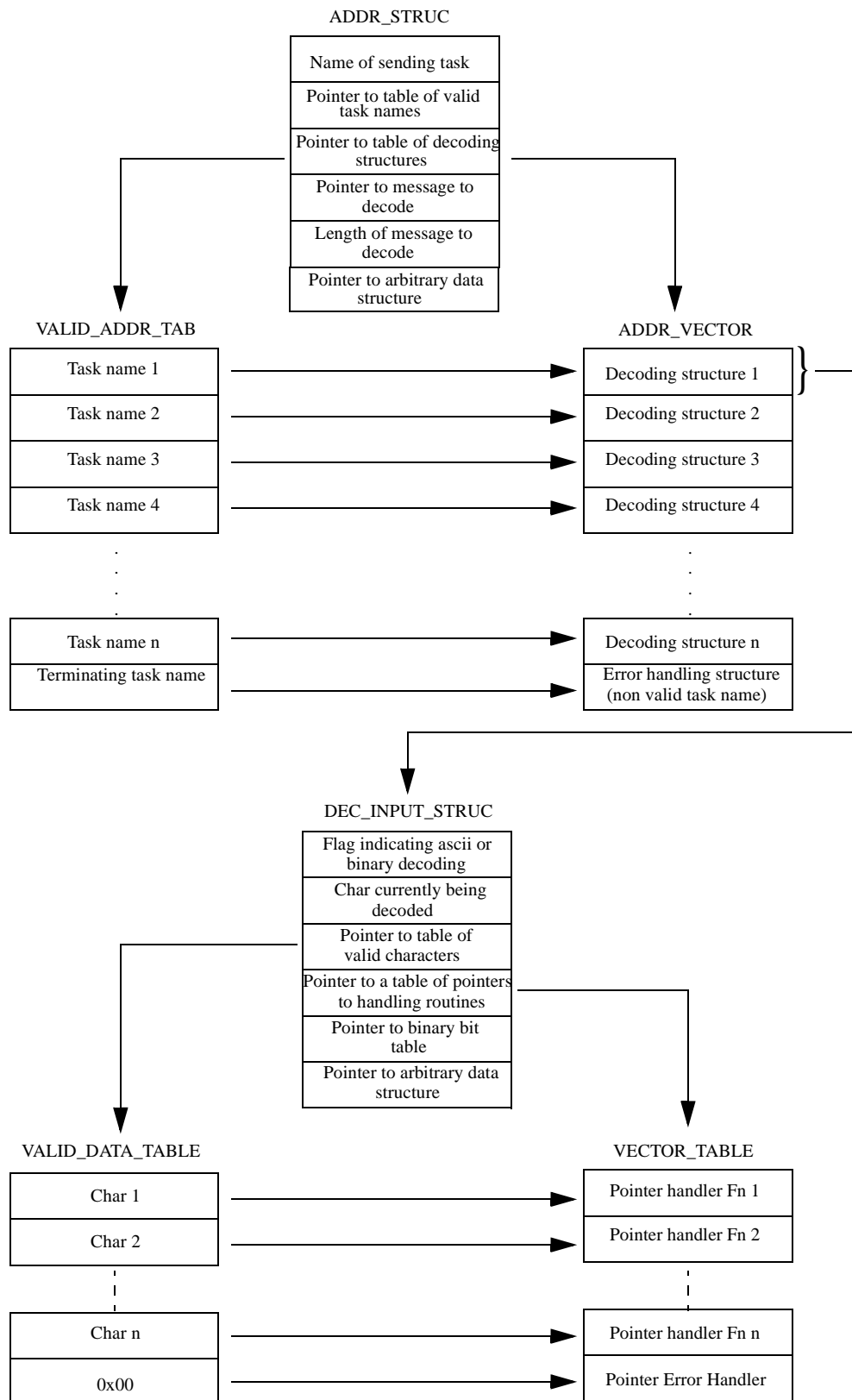


Figure F-1: Schematic of decoding table relationships for ascii character decoding.

decoder is not a legal return address. In this case the decoder structure which is being pointed to by the matching index is used to start processing the message. Generally the first character of the message to be processed will cause an error routine to be executed. Another alternative is that the erroneous message is printed out along with the error message. It is up to the user how they deal with this situation.

Given that the return address is a legal one then the second phase of the decoding task is to check that the message is a legal one in the current context by checking it character by character. The **dec_input_struct** which is connected to the valid return address contains all the necessary information to carry out this task. The basic components of this structure have already been described above. The decoding technique used for character decoding is similar to that described for the address decoding except that it is more flexible.

The main components of the **dec_input_struct** are the tables pointed to by the **valid_data_table_ptr** and the **vector_table_ptr**. These two contain the valid characters and their handling routine pointers respectively, for the current context. The precise format of the tables is as shown below:-

<u>VALID DATA TABLE</u>	<u>VECTOR TABLE</u>
valid char 1	handler ptr 1
valid char 2	handler ptr 2
.	.
.	.
.	.
valid char n	handler ptr n
valid char n+1	handler ptr n+1
.	.
.	.
0xff	error handler ptr

How the data in these tables is decoded depends on the mode of the character decoder. For the sake of the explanation I shall assume that only one character is to be decoded. If the mode in the **dec_input_struct** is **ascii** then the decode table may be compressed by using the high order bit of the valid characters to signify data ranges. This mechanism is best understood by an example.

Consider the case where the next character in a particular context can be any character from A-Z. Furthermore all these characters will invoke the same handling routine. If data ranges were not allowed then there would have to be 27 entries in the valid data table (for the 26 characters plus the terminating character) as well a 27 handling routine vectors. This type of situation occurs quite frequently and would result in a lot of wasted space for command tables, but more importantly a lot of wasted time typing in repetitive tables. To cater for this situation valid data ranges can be defined so that such a table would only have a length of 3 entries. The beginning of an address range is signified by the use of the 8th bit of the **ascii** bytes. If a valid data character with the high bit set is encountered then this character is taken to be the inclusive beginning of a valid data range. The following character is taken to be the inclusive end of the valid data range. In this way arbitrary ranges of numerically consecutive **ascii** characters can be compressed to one entry. Normally the decoder works its way through the valid data table checking whether the **data_byte** component of the **dec_input_struct** is in the table. As each byte is checked and the index moves to the next byte a corresponding index moves through the handler pointer table. If the character is found then the handler for that character is known. In the case of data ranges there is only one handler pointer for the entire range. When a range is detected the indexing into the handler pointer table is altered appropriately. Therefore

data ranges also serve to compress the number of handler table components required. Data ranges can be freely mixed with no range values in the one valid data table.

If a **data_byte** character is not in a valid data table then the comparison index will eventually reach the terminating character which is defined as the 0xff character. This character can be safely used since 0x7f cannot be part of a valid data range when one is confined to the ascii character set. Therefore 0xff is unique.

Usually ascii decoding occurs when decoding input from a terminal or when decoding ascii data sent between tasks. In these cases the 0xff table component has been reached due to the fact that the **data_byte** character is not valid. Therefore the handler associated with the 0xff entry is an error handler which can carry out any required user defined action (e.g. write an error message to a screen, call an error handler). This mechanism allows context sensitive error messages to be generated in a simple fashion.

Decoding of binary data is more complicated mainly due to the implementation of valid data ranges. Also the lack of a unique termination character causes a problem. These difficulties are overcome by having a third table known as the **bit_table**. This table consists of a collection of bits which indicate whether the currently indexed valid data can be considered as the beginning of a valid range or a termination. As the index is moved through the valid data table and the handling table (as described in the ascii decoder case) a third index is moved through an array of bits. If for a particular value the bit indexed is a zero then the valid data entry is not considered to be part of a range but is treated as a single data item. If the corresponding bit table value is a one then there are two possibilities depending on what the next bit is. If the next bit is a zero then the corresponding valid data value is the second part of an inclusive data range (the first part being the data associated with the set bit). If the next bit is a one then the end of the table has been reached. The bit table comparisons for a particular valid data table index are carried out before the data comparison. Therefore there does not need to be a valid data table value for the table termination bit table values, but there does need to be a handler pointer in the handler table. The bits stored in the bit table are from the least significant bit to the most significant bit in each byte of the table (i.e. the lsb in the first byte corresponds to the first valid data byte, the 9th bit which is the lsb in the second byte corresponds to the 9th valid data byte if there are no ranges etc.). The triple table nature makes the user configuration of binary decoding more complex, with particular care needing to be taken with the structure of the **bit_table**.

Example

As an example the following shows how one would configure the address and character decoder tables to decode messages arriving from tasks 0 and 3. For simplicity the messages are only one character in length and ascii. The valid messages from task 0 are 'A' and 'B' and from task 3 are 'C' and 'D'. In the receiving task one must firstly set up the address to decode structure mapping tables (i.e. the address table).

```
char* valid_addr [ 3 ] =  
    { pchTask0_Name, pchTask3_Name, NULL_PTR};  
dec_input_struct* addr_vec [ 3 ] = { &dec_0, &dec_3, &decterm };
```

The **dec_0**, **dec_3** and **decterm** entries in the **addr_vec** table are the character decoder structures which contain the pointers to the character decode tables. For this example the character tables are set up as follows:-

```
unsigned char vdt_0_1 [ 3 ] = { 'A', 'B', 0xff };  
void ( *vt_0_1 [ 3 ] ) ( dec_input_struct * ) =  
    { a_handler, b_handler, err_handler_0 };
```

```
unsigned char vdt_3_1 [ 3 ] = { 'C', 'D', 0xff };
void ( *vt_3_1 [ 3 ] ) ( dec_input_struct * ) =
    { c_handler, d_handler, err_handler_3 };
unsigned char vdt_term [ 1 ] = { 0xff };
void ( *vt_term [ 1 ] ) ( dec_input_struct * ) =
    { illegal_addr_handler };
```

The **a_handler**, **b_handler** etc. are the addresses of the functions which will handle the corresponding entry in the **vdt**.

As mentioned above in order to locate these tables two structures also have to be set up - the **addr_struct** and the **dec_input_struct**. The address structure which is declared as **addr_map** has to have three of its components set up as follows:-

```
addr_map.valid_addr_tab_ptr = &valid_addr;
addr_map.addr_vector = &addr_vec;
addr_map.info_struct_ptr = NULL; /* no data being passed to handlers */
```

Similarly 3 decoder structures have to be set up corresponding to **dec_0**, **dec_3** and **decterm**. The relevant **dec_0**, **dec_3** and **decterm** components are:-

```
dec_0.char_dec_mode = 0;
dec_0.valid_data_table_ptr = &vdt_0_1;
dec_0.vector_table_ptr = &vt_0_1

dec_3.char_dec_mode = 0;
dec_3.valid_data_table_ptr = &vdt_3_1;
dec_3.vector_table_ptr = &vt_3_1;

decterm.char_dec_mode = 0;
decterm.valid_data_table_ptr = &vdt_term;
decterm.vector_table_ptr = &vt_term;
```

The only remaining thing to do is to write the various handling routines.

There is one user interface routine to the decoding system and this is described below.

```
char* decode_mbx_mess ( addr_struct *addr_struct_ptr,
                        unsigned long time_limit );
```

This routine would probably be the normal interface with the command decoder. It will wait for *a maximum time of* **time_limit** for a message to arrive in the mail box of the calling task. The message will then be decoded in the manner described above. If there is no timer available then the routine returns 0xffff:f, if a timeout occurs before a message is received then 0x0000:0 is returned. Under normal conditions the task name of the sending task is returned. The parameters passed to the function are:-

- **addr_struct_ptr** - This is a pointer to an appropriately initialised **addr_struct**.
- **time_limit** - The maximum time that the decode routine will wait for a message to arrive in the tasks mail box before returning. The value passed in the number of system clock ticks the calling task will wait. If a zero is passed then this means the task will wait an infinite time if necessary for a message to arrive.

Features

The UNOS error handling system was first designed for the Cook Islands Project. This project involved the development of a real-time tracking control system for a satellite earth station. This earth station was installed by AOTC in the Cook Islands, north of New Zealand.

The alarm handling system has been designed to be flexible, yet not difficult to use. The following are the key features of the system:

- versatile - able to handle most alarm handling situations
- local definition of alarm types and associated messages. The main alarm structure is dynamically allocated.
- alarm diagnosis information automatically stored – the time and date at which an alarm occurred is stored in the alarm structure and is available for observation by the user using the alarm monitor; the task that was executing at the time of the alarm is also stored. This is particularly important in enabling the tracing of alarms in reentrant code.
- stores information which allows precise determination of the sequence of alarm events.
- alarm numbers are associated with each alarm type.
- allows latching and automatic clearing alarms.
- keeps count of the cumulative number of alarms (both reset and unreset alarms) from a designated date and time. Also keeps a count of the unreset alarms.
- allows the user to set up a general handling routine for a particular alarm type.
- Allows the user to nominate a parameter of any type to be passed to the general handling routine for an alarm.
- allows the user to define a **NULL** terminated text string to describe each alarm type.
- provides an mechanism for consequential suppression of alarms.

The remainder of this document is organised as follows. Section 2 describes the central data structure used in the alarm handling system. From a users perspective this information may seem redundant, however it is included to give a more complete understanding of the alarm handling system. Section 3 describes the programmers interface to the alarm system. Finally section 4 gives some hints as to how some of the features may be used.

Data structures used in the Alarm System

An array and a structure constitute the major data structures of the alarm handling system. The array is used to connect the alarm number of a particular alarm to the relevant alarm structure. The alarm structure itself contains all the relevant information for a particular alarm. Each data structure shall now be considered in detail.

The Alarm Number Array

The alarm number array is an array of 256 pointers which are used to allow the rapid location of the dynamically allocated alarm structure for a particular alarm number. The pointer at a particular index into the array would contain the address of the alarm structure associated with the alarm number. If there are more than 256 alarm numbers then the index into the array is formed as follows:-

$$\text{index into array} = \text{alarm number} \bmod 256$$

This means that there will be more than one alarm structure associated with a single index into the array. The particular structure is found by searching through a linked list of alarm structures, the first of which is pointed to by the content of the relevant alarm number array entry.

The Alarm Structure.

The alarm structure is the main data structure used in the alarm system. It contains all the information related to any particular alarm. The **typedef** for the data structure is shown below:

```
typedef
    struct stAlarmStruc {
        int inAlarmNum;
        char chAutoClearFlag;
        unsigned long ulgSequenceNum;
        tstTimeDate stTimeDateLastAlarm;
        char *pchTaskLastAlarm;
        tstTimeDate stAccumStrtTimeDate;
        unsigned long ulgAccumAlarmNum;
        tstTimeDate stUnresetAlarmTimeDate;
        unsigned long ulgAccumUnresetAlarm;
        void (*pfvdAlarmHandler)(void *);
        void (*pfvdAlarmClearHandler)(void);
        struct stAlarmStruc *pstNextAlarmStruc;
        int inConseqSuppressFlag;
        unsigned int uinSuppressNumAlarms;
        tstConseqSuppressData *pstConseqSuppressData;
        char *pchAlarmDescrip;
    } tstAlarm;
```

The meaning of each of the components is as follows:

inAlarmNum – The alarm number that the structure is used to represent.

chAutoClearFlag – This flag is used to indicate whether or not the alarm is a latching type. If it is a latching alarm then the alarm will remain until it is cleared by some other internal or external mechanism. If the alarm is not latching then it automatically clears itself after the alarm handling function is carried out. **FALSE** \Rightarrow latching alarm; **TRUE** \Rightarrow auto-clearing alarm type.

ulgSequenceNum – This location contains the value of the alarm system sequence counter at the time that the alarm was processed. The alarm system sequence counter is incremented for each alarm that is processed. The purpose of the counter is to allow the determination of the sequence of alarm occurrences that occur within one clock tick.

stTimeDateLastAlarm – The contents of this structure contain the time and date of the last alarm on the alarm number associated with the structure.

pchTaskLastAlarm – This location contains the pointer to the task name of the task that

was executing at the time the last alarm was generated.

stAccumStrtTimeDate – This location contains the time and date from which the contents of the **ulgAccumAlarmNum** have been accumulated.

ulgAccumAlarmNum – This location contains the total number of alarms of the type associated with a particular alarm structure, that have occurred since a particular date and time. Note that this number contains alarms which have been reset as well as those which have not been reset.

stUnresetAlarmTimeDate – This structure contains the time and date of the first unreset alarm.

ulgAccumUnresetAlarm – This location contains the total number of unreset alarms. The time and date of the first unreset alarm is contained in the **stUnresetAlarmTimeDate** structure as described above.

pfvdAlarmHandler – This location contains a pointer to the alarm handling function. The function should be defined to accept a pointer to some argument of unknown type. If there is no handling function for the alarm type then this location would contain a **NULL** pointer.

pfvdAlarmClearHandler – This location contains a pointer to an alarm clear handling routine. The alarm clear function does not accept an argument. If there is no handling function then set this location to a **NULL** pointer.

pstNextAlarmStruc – This is a pointer to the next alarm structure in a linked list of alarm structures.

inConseqSuppressFlag – This location is a flag. If **TRUE** then the alarm associated with this structure is subject to consequential suppression. That is if an alarm occurs of this type then it is ignored whilst this flag is set, subject to the value in **uinSuppressNumAlarms** (see below).

uinSuppressNumAlarms – This location contains the number of alarms that should be suppressed as a result of a request of consequential suppression. Upon receiving an alarm with **inConseqSuppressFlag** set the value of this location is decremented. When the value drops to zero then the **inConseqSuppressFlag** is reset to **FALSE**, and alarm processing returns to normal. If **inConseqSuppressFlag** is **TRUE** and the value of **uinNumConseqAlarms** is zero then this means that the time of consequential suppression is infinite. In this case the suppression must be removed by some external action.

pstConseqSuppressData – This location contains a pointer to a structure which contains the information required for consequential suppression.

pchAlarmDescrip – This location contains a pointer to a **NULL** terminated character array which contains a descriptive message for the particular alarm.

Ancillary Data Structures

As mentioned in the above there two other data structures which are used in conjunction with the main alarm structure, namely the **tstTimeDate** and **tstConseqSuppressData** structure types. These (and their subcomponents) are discussed in detail below. Firstly the **tstTimeDate** structure:

```
typedef
    struct {
        unsigned char uchHours;
```

```
    unsigned char uchMins;  
    unsigned char uchSecs;  
    unsigned int uinMilliSecs;  
    unsigned char uchDay;  
    unsigned char uchMonth;  
    unsigned int uinYear;  
} tstTimeDate;
```

where the components are obvious from the names. Note that the year is stored in its complete form (e.g. 1992).

The consequential suppression data structure is defined as:

```
typedef  
    struct {  
        unsigned int uinNumAlarmToSuppr;  
        tstSupprSpecificData *pstSupprSpecificData;  
    } tstConseqSuppressData;
```

where:

uinNumAlarmToSuppr – Contains the number of separate alarm numbers to suppress (e.g. if Alarm 3, Alarm 20 and Alarm 35 are to be suppressed then the value would be 3).

pstSupprSpecificData– Contains a pointer to an array of structures of the type shown below, which itself contains data specific to each individual alarm number which is to be suppressed.

```
typedef  
    struct {  
        int inAlarmNum;  
        unsigned int uinSuppressNumAlarms;  
    } tstSupprSpecificData;
```

where:

inAlarmNum – Contains the alarm number for which consequential suppression is to apply.

uinSuppressNumAlarms – Contains the number of subsequent alarms to be consequentially suppressed for a particular alarm number.

Low Level User Interface to the Alarm System

The users interface to the alarm handling system is via the following functions.

```
int finCreateAlarm(int inAlarmNum, char chAutoClearFlag,  
    void (*pfvdAlarmHandler)(void *),  
    void (*fvdAlarmClearHandler)(void),  
    tstConseqSuppressData *pstConseqSuppressData,  
    char *pchAlarmDescrip)
```

finCreateAlarm Description

This function is called by the user to create an alarm. The above alarm data structure is created dynamically and the appropriate sections of the structure are filled in based on the parameters passed in. This function is usually called during the initialisation phase.

Parameters

- inAlarmNum** – alarm number to create.
- chAutoClearFlag** – a **FALSE**⇒ alarm type; a **TRUE**⇒auto-clearing alarm type.
- *pfvdAlarmHandler** – this is a pointer to the function which is called when an alarm is signalled. If there is no alarm handler then store **NULL** here.
- *pfvdAlarmClearHandler** – this is a pointer to the function which is called when an alarm is cleared. If there is no alarm clear handler then store a **NULL** is this location.
- *pstConseqSuppressData** – this is a pointer to a user supplied structure of type **tstConseqSuppressData**. This structure contains the alarm numbers and number of alarms that should be consequentially suppressed.
- pchAlarmDescrip** – this is a pointer to a user supplied string of characters which contains the alarm description.

Returns

- the alarm number created if the creation is successful.
- a *minus* 1 if the alarm number has previously been allocated.
- a *minus* 2 if the is a problem allocating memory for the alarm structure.

```
unsigned long fulgAlarm(int inAlarmNum, void* pvdAlarmParam)
```

fulgAlarm Description

This function is called by the user to signal that an alarm event has occurred.

Parameters

inAlarmNum – the number of the alarm.

pvdAlarmParam – a pointer to an arbitrary data structure that can be used to pass parameters to the alarm handling routine. If there are no parameters then this parameter is **NULL**.

Returns

- the number of unset or outstanding alarms on the particular alarm number.

```
unsigned long fulgRetNumUnresetAlarms(int inAlarmNum)
```

fulgRetNumUnresetAlarms Description

This function returns the number of unreset alarms on alarm number **inAlarmNum**. If the alarm number passed into the routine is not a legal one then zero is returned.

Parameter

inAlarmNum – The number of the alarm.

Returns

Returns the number of unreset alarms.

```
unsigned long fulgResetAlarm(int inAlarmNum,  
                             unsigned int ulgNumAlarmEvents)
```

fulgResetAlarm Description

This function allows the resetting of a number of alarm events for a particular alarm number.

Parameters

inAlarmNum – number of the alarm upon which the events have to be reset.

uinNumAlarmEvents – number of alarm events to reset.

Returns

- number of alarm events reset.

```
void fvdStopConseqSuppr(int inAlarmNum)
```

fvdStopConseqSuppr Description

This function clears the consequential suppression flag in the alarm data structure. This disables consequential suppression until another alarm condition occurs.

Parameter

inAlarmNum – the alarm number to disable the consequential suppression on

Returns

- nothing

```
unsigned long fulgResetAllAlarmsClearOnce (int inAlarmNum)
```

fulgResetAllAlarmsClearOnce Description

This function resets all set alarms for a particular alarm number, but only executes the alarm clear routine (if there is one) once.

Parameter

inAlarmNum – the alarm number for which the alarms have to be reset.

Returns

- the number of alarms reset.

```
unsigned long fulgRetCumulAlarms (int inAlarmNum);
```

fulgRetCumulAlarms Description

This function returns the cumulative number of alarms for a particular alarm since the last reset of this value in the alarm data structure.

Parameters

inAlarmNum – The alarm number for which the cumulative number of alarms have to be returned.

Returns

- The cumulative number of alarms for a particular alarm number.

```
int finRetCumulStartTimeDate (int inAlarmNum,  
                             tstTimeDate *pstTempTimeDate)
```

finRetCumulStartTimeDate Description

This function returns the start time and date of the cumulative count of alarms for a particular alarm number. The time and date is returned via the pass by address parameter which points to a **tstTimeDate** structure.

Parameters

inAlarmNum – The alarm number for which the start time and date information has to be retrieved.

pstTempTimeDate – pointer to a **tstTimeDate** structure which is used to store the start time date for the accumulation of alarm numbers.

Returns

TRUE means that the cumulative count for the alarm is >0, therefore the value passed back via the reference parameter **pstTempTimeDate** is valid.

FALSE means either that the cumulative alarm count is =0 (which means that there is no valid start time and date for the accumulation of the alarms since one has not occurred) or the alarm number specified is not a legal alarm number.

```
int finRetFirstUnresetAlarmTimeDate (int inAlarmNum,  
                                     tstTimeDate *pstTempTimeDate)
```

finRetFirstUnresetAlarmTimeDate Description

This function returns the time and date of the first unreset alarm for a particular alarm number.

Parameters

- inAlarmNum** – the alarm number for which the information is being sought.
- pstTempTimeDate** – pointer to a **tstTimeDate** structure which is used to store the time and date of the first unreset alarm.

Returns

- **TRUE** if the alarm structure exists and there is an unreset alarm.
- **FALSE** if either the alarm does not exist or the number of unreset alarms is zero.

```
int finRetLastUnresetAlarmTimeDate (int inAlarmNum,  
                                     tstTimeDate *pstTempTimeDate)
```

finRetLastUnrestAlarmTimeDate Description

This function returns the time and date of the last unreset alarm for a particular alarm number.

Parameters

- inAlarmNum** – the alarm number for which the information is being sought.
- pstTempTimeDate** – pointer to the time date structure used to store the time and date of the last unreset alarm.

Returns

- **TRUE** if a valid time and date has been found.
- **FALSE** if the alarm number specified is illegal, or there are no unreset alarms and no cumulative alarms.
- two (2) if there are reset alarms.

```
int finClearCumulAlarms (int inAlarmNum)
```

finClearCumulAlarms Description

This function clears the cumulative number of alarms for a particular alarm number.

Parameters

inAlarmNum – the alarm number for which the cumulative alarm count has to be cleared.

Returns

- **TRUE** if the operation is successful.
- **FALSE** if the operation is not successful.

Some Notes About Usage

- (i) **pvdAlarmParam** – This pointer parameter to the alarm handler routine can be used to pass a pointer to some arbitrary data structure or even to another function that will be executed from the error handler.
- (ii) Consequential suppression – One can use this facility to suppress the alarm that is initiating the consequential suppression. This would be useful if the occurrence of a particular alarm condition would lead to a large or indeterminate number of subsequent alarms of the same type.

If one does not wish to use consequential suppression then pass a **NULL** pointer when creating the alarm data structure.

CONDITIONS OF USE OF THE UNOS KERNEL

The University of Newcastle Operating System (UNOS) kernel listing has been provided mainly for educational use. If the kernel is used for commercial use it is requested that the user inform the author of this software of their intention to use the kernel. The kernel is provided *as is* for these applications, and the author offers no warranty.

MISCELLANEOUS INFORMATION

The version of UNOS supplied in this listing is suitable for compilation on IBM-PC or compatible machines using the Borland C++ compiler, version 2.0 or above. The software should be compiled using the “huge” memory model. Since the software is compatible with C++ it should be compiled as a C++ program. This affords one better type checking, and allows the software to be linked with user written C++ modules. The UNOS kernel itself does not contain any C++ code (it is all written as C code), but there may be the occasional set of C++ comments (i.e. ‘//’).

The version of the kernel that appears in this listing is the latest one – UNOS-V2.0. The significant changes in this version as compared to Version1.5a were that LOCK primitives were introduced so that a full priority inheritance system could be implemented, and secondly a mechanism using kernel request queues was developed so that interrupts could be enabled in the kernel thereby lowering interrupt latency.

FULL KERNEL LISTING

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <math.h>
#include <alloc.h>
#include ".\globalh\unos.h"
#include ".\globalh\general.h"
#include ".\globalh\fpv.h"
#include ".\globalh\unosasm.h"

/*****
/*
/*
/*          University of Newcastle Operating System          */
/*
/*          ( UNOS )                                          */
/*
/*          Version 2.0-BETA                                  */
/*
/*          KERNEL MODULE                                     */
/*
/*          by                                                */
/*
/*          Robert Betz                                       */
/*          Department of Electrical and Computer Engineering */
/*          University of Newcastle                           */
/*
/*          ( Copyright 1989,90,91,92,93,94,95,99 )           */
/*
/*
/*
*****/

/*=====*/

/*****

HISTORY

UNOS had its genesis in a consulting project carried out by R.E. Betz in
1988-1990. This project involved the construction of a vehicle for a local
non-destructive testing company that could go down underground pipelines looking
for cracks in the pipes using ultrasonics. This version of UNOS was rather
primitive and was written entirely in PL/M, a systems level programming
language championed by Intel.

The pipeline project had limited success (mainly due to limitations in the
signal processing used in the system). The system test results were good enough
to attract further funding from the American Gas Association and this project
entered a new stage undertaken by the industrial partner. The
University of Newcastle had no further significant role in the project.

The software for the system was well structured and written, this mainly being
due to the use of an operating system in the design, as opposed to a looping
executive of some description. The usefulness of the operating system kernel,
which we intimately understood, motivated further development of UNOS.

The first 'C' version of UNOS was originally created by carrying out a hand
translation of the PL/M version written for the Pipeline Project. This version
was functionally identical to the PL/M version. Some changes had to be made to
```

Appendix H — *UNOS-V2.0 LISTING*

the task creation to account for the new language.

The first enhancement carried out was the rewriting of the task creation section to make it considerably simpler - i.e. just call the `create_task` routine passing in the correct parameters. In addition the user modified sections of UNOS were put into a separate module so that the operating system itself does not normally have to be recompiled.

After this initial enhancement then the development of a Version 1.5 of UNOS began in earnest. The Version 1.5 features added are listed below :

- greatly improved task creation procedure - dynamic allocation of most operating system data structures.
- operating system need not be recompiled to include changes to task number, priority number etc.
- separation of the time slice period from the clock period.
- made the operating system pre-emptive
- allow an arbitrary number of priorities
- include general purpose timers which can be used by user tasks and operating system functions.
- implementation of dynamic priorities for tasks so that semaphore blockage will be fair.
- addition of mailbox intertask communications.
- dynamic memory management module.

Version 1.5 of UNOS was used extensively by the Universities consulting company TUNRA in a range of commercial satellite tracking products. It has also been used by a local Newcastle mining instrumentation company in a range of monitoring products.

16/12/93 - fixed the remaining memory bug in the `ufree` routine.

26/12/94 - updated this version to include the `np-signal` routine - this is a non-preemptive signal routine which can be called at any point within an interrupt routine without a preemptive schedule occurring. If a task is unblocked then it is moved to the appropriate priority queue. At the end of the interrupt routine the preemptive schedule routine can be called to allow the reschedule to occur if necessary.

26/12/94 - began to implement the full priority inheritance system which included the addition of the new lock and unlock primitives. First step was to remove the old dynamic priority stuff.

4/1/95 - added the full priority inheritance and carried out preliminary testing which was successful. This involved adding the `LOCK` and `UNLOCK` primitives and their associated creation and destruction routines. The scheduler underwent a major change to support the new priority inheritance system. The old dynamic priorities of UNOS 1.5 have been taken out altogether.

The `create_semaphore` routine has had a change to its parameter list, so that the value that a semaphore has to be set to is passed to the semaphore at creation time.

31/1/95 - added new send message macros in `UNOS.H` to allow messages to be sent to mail boxes without being blocked if the mail box is full. The macros are:

```
SEND_MESS(message ptr, message length, mail box ptr);  
This is the normal send_mess as before
```

```
SEND_MESS_NB(message ptr, message length, mail box ptr);  
This is the non blocking form of the send message routine. If the mail box  
is full it returns FULL_MBX.
```

26/12/95 - changed the `kbdrv` module so that the UNOS termination sequence is now `Ctrl-Alt-DEL` or `Ctrl-Alt-INSERT` (both of these keys being on the keypad). This was added so that UNOS can be correctly terminated when running from a DOS shell under Windows 3.1x or Windows 95.

8/2/99 - began adding the keyboard multiplexer task so that multiple tasks can connect to the keyboard dynamically, and also disconnect from the keyboard dynamically. REB

18/2/99 - finished adding the new multiplexed keyboard driver setup. Tested and appears to be working correctly. REB

18/2/99 - changed the `_signal` routine name to `usignal` (for Unos signal). REB

26/2/99 - began working on including the feature to allow interrupts to be enabled inside the kernel. This feature is designed to minimise the interrupt latency. Because many interrupt routines call a `usignal` to start up a processing task, this feature implies that we must handle the issue of kernel reentrancy. The short answer to this problem is that we cannot allow reentrancy, but at the same time we have to allow interrupts to be enabled.

The solution was to modify the kernel itself and all its interface routines. These were modified so if a request is made for a kernel service the request is queued onto a linked list of queued requests, known as the `kcb` queue. A check is then made on the `kernel_executing` flag, and if it is not set (meaning that the kernel was not executing prior to the current interrupt) then the kernel is entered as normal. The state of the calling task is saved, the `kernel_executing` flag is set and the interrupts are enabled. The request queue is then accessed and a `kcb` is retrieved. This request is then carried out. If any interrupts occur whilst this is in progress, and these interrupts result in a call to the kernel (most likely a `usignal` call or a tick call), then the request is queued and not carried out immediately. This scheme prevents reentrancy into the kernel, and at the same time orders the requests that are arriving.

The `kcb` request queue is accessed in a loop until there are no requests left on the queue.

Whilst the queueing is carried out for all calls for kernel services, not all the routines are suitable for use from within an interrupt routine. The main kernel routines that would be used from within interrupt routines would be:

```
usignal
np_signal
tick interrupt
preemptive_schedule
```

One of the techniques of entering the kernel is via the tick interrupt. During the first part of the tick interrupt routine the `dec_timers` routine is called. As the name implies this routine has the job of decrementing the timers in the system. If one of the timers times out then a timeout handling routine is called. This may result in a `usignal` operation. This `usignal` is handled in exactly the same fashion as those generated outside the kernel. The timer generated `usignal` also relies on the use of two flags - `sig_entry_from_dec_timers` and `possible_preemption` (used in the kernel proper), to allow the correct actions to be undertaken under this type of signal condition. It should be noted that interrupts are disabled whilst the decrementing of the timers is carried out. This is to prevent any possibility of anything upsetting the precision of any of the timers routines, and in addition would make sure that there is no possibility of recursive entry into the kernel routine. This strategy should not have a dramatic effect on the interrupt latency since the `dec_timer` routine is designed to execute as fast as possible, and any timeout routines that do more than a `usignal` should be very short and quick (else the timing precision of the system will be upset anyhow).

Note the the `timed_wait` semaphore is a special use of a timer. The `timed_wait_handler` (which is called if a `timed_wait` times out) effectively does a similar operation to a `usignal`. However, since it is directly manipulating the `tcb` queues it is important this is executed with interrupts disabled protection.

Appendix H — *UNOS-V2.0 LISTING*

In order to carry out this change a number of routines will have been changed, namely:

```
kernel
tick
usignal
preemptive_schedule
timer_handler
timed_wait_handler
reschedule
change_task_priority
semaphore_queue_handler
place_semaphore_queue
wait_handler
```

New functions added were: queue_kcb, dequeue_kcb, usignal_handler, change_task_priority_handler.

The variable 'kernel_executing' has been added to indicate that the kernel code is currently being executed by the processor. The local_kcb_ptr variable was added to point to the kcb structure that is used in the kernel to indicate the operation to be carried out. A component called prreemption_status has been added to the kcb_structure to allow correct operation of the np_signal usignal variant.

As of the 6/4/99 the system has undergone preliminary testing and appears to work. However, undoubtedly under heavier use some problems will come to light. For this reason this version is currently known as UNOS 2.0 BETA.

REB: 6/4/99.

-----< END HISTORY >-----

WISH LIST

Things to add to UNOS for version 2.1:

- (i) Allow interrupts to be enabled during mail box message exchange.
- (ii) Fix up the multiple definitions of the 8259 addresses etc - in UNOS.C, hwpc.h (twice).
- (iii) Full dynamic task creation.

*****/

```

/*****/
/*                                          */
/*                                          */
/*              GLOBAL VARIABLES          */
/*                                          */
/*                                          */
/*                                          */
/*****/
```

FULL KERNEL LISTING

```
/*----- 8259 INTERRUPT CONTROLLER RELATED DEFINITIONS -----*/

#define I8259_INT_CTRL_ADD 0x20
#define I8259_EOI 0x20


/*
definitions of indexes into the central table
*/

#define BGN_PRIORITY_Q_1 1
#define CURRENT_TASK 0


/*
number of the highest priority in the system - currently always 1
*/

#define HIGH_PRIORITY 1


/*----- Mail box hash table size -----*/
/* Note that this number is set up to be a prime number */
#define HASH_TABLE_SIZE 241


/*----- MAXIMUM NUMBER OF TASKS IN THE SYSTEM -----*/
/* The variable defined below contains the maximum number of tasks that can be
created in the system. This value is initialised from a define in the
main program. The maximum number of tasks is significant since it defines
the amount of resources allocated to some crucial system tables.
*/
unsigned int max_num_of_tasks = 0;


/*----- NUMBER OF TASKS IN THE SYSTEM -----*/
/* This location is used to store the number of tasks in the system. This
location is incremented each time a task is created in the system, so that
after all the tasks have been created it will contain the total number of
system tasks. This total number of tasks has to be less than the value
contained in the max_num_of_tasks variable contained in the operating system.
*/
static unsigned int num_of_tasks = 0;


/*----- NUMBER OF SEMAPHORES IN THE SYSTEM -----*/
/* This location is used to store the total number of semaphores in the
system. As with the num_of_tasks above this has been implemented as
a variable so that the UNOS module does not have to be recompiled if
the number of semaphores changes.
*/

static unsigned int num_semaphores;


/* the following variable is the index into the semaphore array
indicating the next semaphore number to allocate. If there are no
semaphores to allocate then the semaphore number is set to zero. This
number cannot be one for a semaphore which can be allocated because
there must always be at least one task in a operational system,
therefore semaphores 0 and 1 must be allocated to the mailbox
associated with that task.
*/
unsigned int next_semaphore = 0;


/*----- NUMBER OF PRIORITY LEVELS -----*/
/* This location is used to store the number of priority levels in the
```

Appendix H — *UNOS-V2.0 LISTING*

```
system. this is stored in a variable so that UNOS does not have to be
recompiled if the number of priorities is changed. This number
is fundamental in that it establishes the basic structure of the
central table of the system. The maximum number of priorities is 256,
the minimum is 1 ( the null task ).
*/

static unsigned char num_of_priorities = 1;

/*----- START SEMAPHORE CENTRAL TABLE ENTRIES -----*/
/* This location stores the central table index where the semaphore central
table entries begin. This number is calculated from the number
of priorities that the system supports.
*/

static unsigned int bgn_semaphore_central_table;

/*----- BASE NUMBER OF TICKS PER TIME SLICE -----*/
/* This location contains the number of clock ticks that should occur
before a time slice entry to the kernel should occur if the sub-priority
is zero
*/

static unsigned int base_ticks_per_time_slice;

/*----- VARIABLES TO STORE TIME QUEUE INDEXES INTO CENT TABLE -----*/

static unsigned char bgn_active_time_q;
static unsigned char end_active_time_q;
static unsigned char bgn_inactive_time_q;
static unsigned char end_inactive_time_q;

/*----- INTERRUPT NUMBER FOR KERNEL ENTRY -----*/

unsigned char kernel_entry;

/*----- TASK START FLAG -----*/
/* This flag is used to indicate that the tasks are all ready to start.
The flag is tested inside the create_task routine when the first
schedules of the starting tasks occur. If set then a call to each
particular task is carried out in the create_task routine.
*/

static unsigned char start_flag;

/*----- KERNE EXECUTING FLAG -----*/
/* This flag is set to true at the beginning of the kernel. It is required
because the kernel reenables interrupts and if one of the these interrupt
routines calls a usignal then the usignal routine needs to be able to detect
that the kernel was executing at the time of kernel entry.
*/

static char kernel_executing = FALSE;

/*----- REAL TIME CLOCK VARIABLE -----*/
/* The real time clock variable is used by the operating system to keep
track of time. Upon every entry to the tick routine the value stored in
this variable is incremented. The value stored in this variable finds a
number of uses in UNOS. The kernel itself uses the times stored here to
check whether certain critical tasks are executed within user specified
```

time bins. User tasks can directly access the time stored here to carry out user defined timing functions.

Note that the value linearly increases until it wraps around. The effects of wrap around must be handled by the user tasks which use the timer.

*/

```
static unsigned long rt_clock = 0;
```

/*----- TIME SLICE FLAG -----*/

/* The time slice flag is used by the tick routine to determine whether to allow a time slice entry into the kernel. If true then a time slice entry is allowed, if FALSE then a time slice entry is not allowed. The value of the flag is manipulated by the start_time_slice and stop_time_slice routines.

*/

```
static unsigned char time_slice_flag = FALSE;
```

/*----- CURRENT PRIORITY -----*/

/* This variable is used to store the current priority of the task that the scheduler wishes to run. It should be noted that because of the priority inheritance system being used this value may not be equal to the priority of the task actually running. This occurs when the task that should be running is not allowed to run because another low priority task has a lock. Therefore the scheduler would choose this task to run, even though the task that should be running is the higher priority task. Therefore the current_priority location will contain the priority of the higher priority task, even though the low priority task is running.

*/

```
static unsigned char current_priority;
```

/*----- SIGNAL ENTRY FROM TIMERS FLAG -----*/

/* This flag is set if a timer has expired and is going to execute a timeout handler. One of the most common uses of a timeout handler is to signal another task that it is supposed to do something. This is usually achieved by the handling routine doing a signal to the appropriate semaphore that the responding task is currently blocked on. A signal normally could cause a pre-emptive task switch. However for timing reasons it is undesirable for this to occur from the dec_timers routine (this is where the timeout function handler is called). Therefore the signal routines need to be modified if called from the dec_timers routine. This is the purpose of this flag and another defined below - possible_preemption. This flag is checked in the signal routine. If set then the signal routine will place the request for a signal operation onto the kernel's signal queue. Therefore the signal operation operates as though it has occurred from an interrupt routine called from the kernel. Hence the signal operation will be delayed until the kernel is called. The signal handler sets a flag called possible_preemption if a signal operation occurs.

*/

```
static char sig_entry_from_dec_timers = FALSE;
```

/*----- POSSIBLE PRE-EMPTION FLAG -----*/

/* This flag is related to that defined above so read the comments for it. This flag is checked in the tick routine. If set then it indicates that there is the possibility of a pre-emptive task switch due to a timeout of one of the timers.

*/

```
static char possible_preemption = FALSE;
```

Appendix H — *UNOS-V2.0 LISTING*

```
/*----- PREEMPTION_ALLOWED -----*/
/* This variable is used as a flag to decide whether a preemptive entry to the
kernel is to occur when a signal occurs. The flag is set to false in the
np_signal routine, just before the usignal routine is called, and then set to
true immediately upon return from that call.
*/
static char preemption_allowed = TRUE;


/*----- MAIL BOX TABLE POINTER -----*/
/* This pointer points to a table of pointers. There are num_of_tasks
pointers in the table which when initialise by the create_mbx routine
point to the mail box structures. The index into the array of pointers
pointed to by this pointer is the task number with which the mail box is
associated. This table is required to allow the OS to find the mail box
memory address given the task number with which communication is
required.
*/

static mbx **mbx_table;


/*----- MAIL EXCHANGE HASH TABLE -----*/
/* this table contains pointers to taskname_map structures. The array of
pointers is indexed into by a hash value derived from the address of the
array containing the name of the task concerned.
*/

static taskname_map **hash_table;


/*----- MEMORY ALLOCATION RELATED STRUCTURES AND LOCATIONS -----*/
/* The following two locations have to be initialised by the user to define
the system memory pool. This pool of memory is the memory from which all
dynamically allocated data structures will be allocated.
*/
/* the following pointer points to the beginning of the memory pool */

char huge *mem_pool_ptr = NULL_PTR;

/* the following location is used to store the size of the memory pool
which can be dynamically allocated at run time.
*/

unsigned long mem_pool_size = 0;

/* the following location is used to store the remaining total memory in the
memory pool. Note that this does not mean that any one memory block will
be large enough for an allocation - if the memory fragmentation is bad then
memory may not be able to be allocated even though the total memory is large
enough.
NB. The remaining memory is stored in header sized units, where the header
is that used by the memory allocation routines. To get the actual memory
in bytes then multiply the value stored here by the sizeof ( blk_header ).
*/

unsigned long rem_memory = 0;


/*----- Initial Empty Free Block Header -----*/
```

```
/* The following is the initial memory header required to initiate the
formation of the free list.
*/

static blk_header start_header;

/* the last_blk_alloc is used to keep track of the last header block
accessed in the free list of memory blocks. This position is the starting
point for searches through the queue of free memory blocks.
*/
static blk_header huge *last_blk_alloc = NULL_PTR;      /* last allocated block */


/*****
/*
/*      Task Control Block ( tcb ) Related Definitions      */
/*
/*
/*****

/*----- pointer to the array of pointers to the task control blocks -----*/

static task_control_block** tcb;


/*----- Task status masks -----*/

/*   Note that the bits in the tcb are defined as follows:
        bit 0 - 1 => task runnable
                0 => task blocked on a semaphore.
        bit 1 -> 7 - reserved
*/

/*=====*/


/*****
/*
/*      Semaphore Related Definitions      */
/*
/*
/*****

/*----- Semaphore type definitions -----*/

/*----- Queueing types used in the semaphore queue manipulation -----*/
/*----- routines -----*/

#define DEQUEUE 0      /* Remove tcb from semaphore queue */
#define QUEUE 1        /* Add tcb to semaphore queue */

/*----- Semaphore data structure array -----*/
```

Appendix H — *UNOS-V2.0 LISTING*

```
static semaphore_struct **semaphore;

/*=====*/

/*****
/*
/*      Kernel Control Block ( kcb ) Related Definitions
/*
/*
*****/

/*----- TOP_KCB_QUEUE_PTR -----*/

/* This variable stores a pointer to the top of the queue of pending usignal
operations. This queue essentially stores a number of kcbs for usignals
that have occurred whilst the kernel was executing. If the pointer is a NULL_PTR
then the queue is empty. The queue is arranged as a linked list.
*/

static kcb_structure* top_kcb_queue_ptr = NULL_PTR;

/*----- END_KCB_QUEUE_PTR -----*/

/* Similarly to the previous pointer this pointer points to the end of the
queue of pending usignal kcbs. This pointer is required to enable kcbs to be
quickly added to the end of the usignal kcb queue. If there are no kcbs in the
queue then its value is a NULL_PTR.
*/

static kcb_structure* end_kcb_queue_ptr = NULL_PTR;

/*----- Kernel entry type definitions -----*/

#define WAIT_ENTRY 0
#define USIGNAL_ENTRY 1
#define RESCHEDULE_ENTRY 2
#define TICK_ENTRY 3
#define CREATE_TASK_STK_ENTRY 4
#define PREEMPTIVE_ENTRY 5
#define LOCK_ENTRY 6
#define UNLOCK_ENTRY 7
#define CHANGE_TASK_PRIORITY 8

/* UNCOMMENT THE FOLLOWING LINE IF ONE WISHES TO INCLUDE DEBUGGING STUFF IN
THE KERNEL
*/
/* #define DEBUG */
#ifndef DEBUG

/*****
/*
/*      UNOS DEBUGGING STUFF
/*
*****/

/* Below is defined a structure which is used to store the previous task
execution history. This structure is filled out during each kernel
```


entry. Currently the structure can only be interrogated by using a debugger, however the longer term plan is to allow the future UNOS monitor to interrogate the structure.

The components of the structure are:-

char *taskname_ptr	contains the name of the pointer to the previous task that was executing.
int priority	priority of the task.
unsigned long clock_cnt	contains the value of the UNOS real-time clock at the time of the task switch.
char *reason_ptr	contains a pointer to a short message that indicates the reason for the last task switch.
unsigned int semaphore_num	The semaphore number upon which a wait or signal is occurring. Only valid number if this is the entry condition.
char *nxt_taskname_ptr	contains the name of the task that executed after the stored task.
int nxt_priority	the static priority of the next task.
unsigned long count_num	a simple counter that allows one to see the order of the task switches stored in the circular array of these structures.

Upon entry to the kernel a structure of the above form is filled out for temporary storage. If upon exit from the kernel it is still the same task then nothing is stored in the circular buffer of structures. If on the other hand the task is different then the `nxt_taskname_ptr` part of the structure is updated and the structure is stored in the circular array.

*/

```
typedef
struct
{
    char *taskname_ptr;
    int priority;
    unsigned long clock_cnt;
    char *reason_ptr;
    unsigned int semaphore_num;
    char *nxt_taskname_ptr;
    int nxt_priority;
    unsigned long count_num;
} kernel_debug;

/* Now define the temporary structure used to hold the info upon kernel
entry
*/
static kernel_debug temp_kernel_store;

/* Now define the array of kernel_debug structures which holds a dozen or so
of the previous tasks that have executed in the system.
*/
#define SIZE_DEBUG_BUFFER 12
static kernel_debug kernel_debug_info [SIZE_DEBUG_BUFFER];

/* The following variable is used to point to the next location in the
array of kernel_debug structures that information is to be stored.
*/
static int kernel_debug_head = 0;
```

Appendix H — UNOS-V2.0 LISTING

```
/* Task switch count value */
static unsigned long count_num = 0;

/* The messages that are used for the task switch reasons */
static char tick_entry [] = "Tick entry";
static char wait_entry [] = "Wait entry";
static char usignal_entry [] = "Usignal entry";
static char reschedule_entry [] = "Reschedule entry";
static char preemptive_entry [] = "Preemptive task switch";
static char create_task_entry [] = "Create task entry";
static char lock_entry [] = "Lock entry";
static char unlock_entry [] = "Unlock entry";
static char change_task_priority_entry [] = "Change task priority entry";

#endif /* DEBUG */

/*****
/*
/*          CENTRAL TABLE DEFINITION          */
/*
/*          *****/
static void **central_table;

/*----- FUNCTION PROTOTYPE DEFINITIONS -----*/

static void add_queue ( task_control_block *tcb_ptr,
                        unsigned int cent_tab_index );

static void remove_top_queue ( unsigned int cent_tab_index );

static void remove_queue ( task_control_block *tcb_ptr,
                           unsigned int cent_tab_index );

static void place_priority_queue ( task_control_block *tcb_ptr );

static void semaphore_queue_handler ( int queue_op,
                                      kcb_structure* local_kcb_ptr );

static void place_semaphore_queue ( unsigned int semaphore_num );

static void scheduler ( void );

static void wait_handler ( kcb_structure* local_kcb_ptr );

static void reschedule_handler ( void );

static void time_slice_handler ( void );

static void add_timer_active_q ( timer_struct* timer_ptr,
                                unsigned char timer_type,
                                unsigned long init_time_count,
                                void ( *timeout_handler )(void*),
                                void *data_ptr );

static void add_timer_inactive_q ( timer_struct *timer_ptr );

static timer_struct* remove_timer_inactive_q ( void );

static timer_struct* remove_timer_active_q ( timer_struct* timer_ptr );

static void dec_timers ( void );

static void timed_wait_handler ( void* handler_ptr );
```

```
static mbx** alloc_mbx_table ( unsigned int num_of_tasks );

static mbx* create_mbx ( unsigned int task_num, char mbx_type, unsigned int
                        num_mess, unsigned int mess_size, unsigned int
                        spce_avail_sema, unsigned int mess_avail_sema );

static blk_header huge* morecore ( void );

static void preemptive_schedule_handler ( void );

static unsigned int mail_exchange ( char *mess_addr_ptr );

static unsigned int hash_taskname_ptr ( char *task_name_ptr );

static taskname_map **alloc_mail_exch_table ( unsigned int size_of_table );

static void queue_kcb (kcb_structure* local_kcb_ptr);

static kcb_structure* dequeue_kcb (void);

static void usignal_handler (kcb_structure* local_kcb_ptr);

static void change_task_priority_handler (kcb_structure* local_kcb_ptr);


/*****
/*
/*          OPERATING SYSTEM ATOMS          */
/*
/*          */
*****/

/*
=====
|
| queue_kcb
|
| This function places the kcb passed to it onto the kcb queue maintained by
| the kernel. The new kcb is always placed on the end of the queue.
|
| Parameters:   pointer to a kcb_structure
|
| Returns:      nothing
|
=====
*/

static void queue_kcb (kcb_structure* local_kcb_ptr)
{
    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );

    if (end_kcb_queue_ptr != NULL_PTR)
    {
        /* The queue has something in it
        */
        end_kcb_queue_ptr->kcb_ptr = local_kcb_ptr;
        local_kcb_ptr->kcb_ptr = NULL_PTR;
    }
}
```

Appendix H — *UNOS-V2.0 LISTING*

```
        end_kcb_queue_ptr = local_kcb_ptr;
    }/* if */
    else
    {
        /* The queue is currently empty so place the kcb at the
        beginning of the queue and set up the queue pointers
        appropriately.
        */
        local_kcb_ptr->kcb_ptr = NULL_PTR;
        top_kcb_queue_ptr = local_kcb_ptr;
        end_kcb_queue_ptr = local_kcb_ptr;
    } /* else */

    if ( int_status )
        enable ( );

}/*queue_kcb*/


/*-----*/


/*
=====
|
| dequeue_kcb
|
| This routine retrieves a KCB from the kernel kcb queue. Note that the
| interrupts are disabled during this operation as the queue variables are
| global and are manipulated by the kernel interface routines.
|
| Parameters:    none
|
| Returns:      pointer to the kcb removed, or
|               NULL_PTR if the queue is empty
|
=====
*/

static kcb_structure* dequeue_kcb (void)
{
    kcb_structure* local_kcb_ptr;

    /* NOTE: These operations constitute a critical section as the variables
    associated with the queue handling are also manipulated in the
    other kernel entry routines, and this routine can run concurrently with
    the kernel. Therefore interrupts have to be disabled around this short
    section of code.
    */
    disable();
    if (top_kcb_queue_ptr == NULL_PTR)
    {
        /* have reached the end of the kcb queue */
        end_kcb_queue_ptr = NULL_PTR;
        return NULL_PTR;
    }
    /*if*/
    else
    {
        /* else there is another kcb on the queue so get it and put it
        into kcb_ptr
        */
        local_kcb_ptr = top_kcb_queue_ptr;
        top_kcb_queue_ptr = top_kcb_queue_ptr->kcb_ptr;
    }
}
```

```
/* reenable the interrupts for another loop into the main section
of the kernel - only if the system has been started.
*/
if (start_flag)
{
    enable();
}/*if*/

return local_kcb_ptr;

}/*else*/
}/*dequeue_kcb*/

/*-----*/

/*
=====
|
| add_queue
|
| This is a general purpose queue maintenance routine which will add a tcb to
| the end of the appropriate queue.
|
| The parameters passed to the routine are:
| (i) a pointer to the tcb to be placed in a queue.
| (ii) the index into the central_table which contains the pointer to the
| beginning of the particular queue.
|
|=====
*/

static void add_queue ( task_control_block* tcb_ptr,
                      unsigned int cent_tab_index )
{
    task_control_block* end_q_ptr;

    /* update the queue information in the tcb */
    tcb_ptr->cent_tab_index = cent_tab_index;

    cent_tab_index++;
    end_q_ptr = ( task_control_block* ) central_table [ cent_tab_index ];

    if ( end_q_ptr == NULL_PTR )
    {
        /* The queue is at the moment empty. This implies that the
        bgn_q_ptr is also a NULL_PTR. Therefore the bgn_q_ptr and the
        end_q_ptr should be made equal.
        */
        ( task_control_block* ) central_table [ cent_tab_index - 1 ] = tcb_ptr;
        ( task_control_block* ) central_table [ cent_tab_index ] = tcb_ptr;
        tcb_ptr->prev_task_ptr = NULL_PTR;
        tcb_ptr->next_task_ptr = NULL_PTR;
    } /* if */
    else
    {
        /* already other tcb in the queue link the previous end of the queue
        to the new tcb
        */
        end_q_ptr->next_task_ptr = tcb_ptr;
        tcb_ptr->prev_task_ptr = end_q_ptr;
    }
}
```

Appendix H — UNOS-V2.0 LISTING

```
tcb_ptr->next_task_ptr = NULL_PTR;
/* update the central_table end of queue pointer */
( task_control_block* ) central_table [ cent_tab_index ] = tcb_ptr;
} /* else */
} /* end of add_queue */

/*-----*/

/*
=====
|
| add_top_queue
|
| This function adds the tcb passed in onto the top of the priority queue
| for that particular task.
|
| The parameters passed to the routine are:
| (i) a pointer to the tcb to be placed in a queue.
| (ii) the index into the central_table which contains the pointer to the
| beginning of the particular queue.
|
=====
*/

static void add_top_queue ( task_control_block* tcb_ptr,
                           unsigned int cent_tab_index )
{
    if (central_table [cent_tab_index] == NULL_PTR)
    {
        /* Nothing in the queue at the moment */
        (task_control_block*)central_table [cent_tab_index] = tcb_ptr;
        (task_control_block*)central_table [++cent_tab_index] = tcb_ptr;
        tcb_ptr->next_task_ptr = NULL_PTR;
        tcb_ptr->prev_task_ptr =NULL_PTR;
    } /* if */
    else
    {
        /* Something in the queue so put at the beginning */
        ((task_control_block*)central_table[cent_tab_index])->prev_task_ptr =
                                                    tcb_ptr;

        tcb_ptr->next_task_ptr =
            (task_control_block*)central_table[cent_tab_index];
        tcb_ptr->prev_task_ptr = NULL_PTR;
        (task_control_block*)central_table[cent_tab_index] = tcb_ptr;
    } /* else */
} /* end of add_top_queue */

/*-----*/

/*
=====
```

```

| remove_top_queue
|
| This is a simpler queue maintenance routine which only allows tcb's to be
| removed from the top of a queue. This is included for speed reasons since
| on some occasions the more complex remove_queue routine is not needed.
|
| Parameters: - index into the central table location which contains
|              the pointer to the top of the queue in question.
| Entry via : - multiple places in the kernel
|
|=====
*/

static void remove_top_queue ( unsigned int cent_tab_index )
{
    task_control_block* tcb_ptr;

    tcb_ptr = ( task_control_block* ) central_table [ cent_tab_index ];

    if ( tcb_ptr != NULL_PTR )
    {
        /* something to remove so do it */
        tcb_ptr->q_type = DONOT_Q_TYPE;
        tcb_ptr->cent_tab_index = 0;
        ( task_control_block* ) central_table [ cent_tab_index ] =
            tcb_ptr->next_task_ptr;
        if ( central_table [ cent_tab_index ] == NULL_PTR )
            central_table [ cent_tab_index + 1 ] = NULL_PTR; /* queue empty */
        else
        {
            tcb_ptr =
                ( task_control_block* ) central_table [ cent_tab_index ];
            tcb_ptr->prev_task_ptr = NULL_PTR;
        } /* else */
    } /* if */
} /* end of remove_top_queue */

/*-----*/

/*
|=====
|
| remove_queue
|
| This is a general purpose queue maintenance routine. It removes the tcb
| passed in from the queue passed in. The tcb does not have to be at the
| beginning or end of a queue, but can also be removed from the middle.
|
| The parameters passed in are defined as follows:
| (i) pointer to the task control block to be removed from the queue.
| (ii) index into the central_table to the position which contains the
|      pointer to the top of the particular queue.
|
|=====
*/

static void remove_queue ( task_control_block* tcb_ptr,
                          unsigned int cent_tab_index )

```

Appendix H — UNOS-V2.0 LISTING

```
{
    task_control_block* prev_tcb_ptr;

    /* Note: prev_tcb_ptr = 0 if the tcb to be removed is at the beginning
    of a queue. This routine is tricky because one must account for the
    the case where the queue only contains one entry - therefore the
    central_table begin and end pointers should end up with zero in them.
    */
    prev_tcb_ptr = tcb_ptr->prev_task_ptr;

    if ( tcb_ptr ==
        ( task_control_block* ) central_table [ cent_tab_index ] )
    {
        /* removing first entry in the queue - update beginning of queue
        pointer
        */
        ( task_control_block* ) central_table [ cent_tab_index ] =
            tcb_ptr->next_task_ptr;
        if ( central_table [ cent_tab_index ] != NULL_PTR )
            ( ( task_control_block* )central_table [ cent_tab_index ] )
                ->prev_task_ptr = NULL_PTR;
    } /* if */

    if ( tcb_ptr ==
        ( task_control_block* ) central_table [ cent_tab_index + 1 ] )
    {
        /* removing last entry in queue. */
        ( task_control_block* ) central_table [ cent_tab_index + 1 ] =
            prev_tcb_ptr;

        if ( prev_tcb_ptr != NULL_PTR )
            /*
            not the only entry in the queue.
            */
            prev_tcb_ptr->next_task_ptr = NULL_PTR;
    } /* if */
    else
    {
        /* removing an entry somewhere in the middle of a queue so update
        the pointers in the tcbs appropriately.
        */
        if ( prev_tcb_ptr != NULL_PTR )
        {
            prev_tcb_ptr->next_task_ptr = tcb_ptr->next_task_ptr;
            tcb_ptr = tcb_ptr->next_task_ptr;
            tcb_ptr->prev_task_ptr = prev_tcb_ptr;
        } /* if */
    } /* else */

    /* update the queue related components of the tcb */
    tcb_ptr->q_type = DONOT_Q_TYPE;
    tcb_ptr->cent_tab_index = 0;
} /* end of remove_queue */
```

```
/*-----*/
```

```
/*
=====
|
| place_priority_queue
|
| This procedure accepts a parameter of type pointer which points to the tcb
```



```
| which is to be placed on the appropriate priority queue of runnable tasks.
| The task is placed on the end of the priority queue. The status in the
| task control block is made runnable. Note that in the case of a task coming
| from the time queue it should always be runnable, however tasks from the
| blocked queues have to be made runnable.
|
| Parameters :- pointer to the tcb to be queued.
|
|=====
*/

static void place_priority_queue ( task_control_block* tcb_ptr )
{
    /* firstly change the status of the task to runnable. */
    tcb_ptr->task_status = TASK_RUNNABLE;

    /* update the queue related information in the tcb */
    tcb_ptr->q_type = PRIORITY_Q_TYPE;

    /* now link the tcb into the correct priority queue. */
    add_queue ( tcb_ptr, ( ( tcb_ptr->priority ) - 1 ) << 1 ) + 1 );
} /* end of place_priority_queue */

/*-----*/

/*
|=====
|
| timed_wait_handler
|
| This function is entered if a timed wait has timed out. The function carries
| out a usignal on the semaphore that has timed out. This causes a usignal kcb
| to be queued for kernel processing. The timeout_flag is set in
| the offending tcb so that the timed_wait routine can determine if the tcb
| has been released due to a signal or a time out. In order that the kernel
| is always called the possible_preemption flag is set.
|
| Parameters : - pointer to the task control block to be remove froma queue
|
| Entry via : - dec_timers
|
|=====
*/

static void timed_wait_handler ( void* handler_ptr )
{
    task_control_block* tcb_ptr;

    tcb_ptr = ( task_control_block* )handler_ptr;

    /* set the timeout_flag to indicate that the tcb has been put onto the
    priority queue
    */
    tcb_ptr->timeout_flag = TRUE;

    /* Now carry out a normal usignal operation on the semaphore that the
    task is blocked on. The semaphore number should be contained in the tcb
    since this task has claimed the semaphore.
    */
    usignal(tcb_ptr->sema_blocked_on);
```

Appendix H — *UNOS-V2.0 LISTING*

```
} /* end of timed_wait_handler */

/*-----*/

/*
=====
|
| semaphore_queue_handler
|
| This routine handles the queueing and dequeuing operations for all the
| semaphore queues in the system. This is the default routine which is stored
| in the semaph_queue_handler pointer in the semaphore data structure during
| system initialization.
|
| The routines can work in two possible ways depending on a compilation
| directive.
|
| For a queue operation:
| If the _PRIORITY_ORDER_SEMA_QUEUES_ is not defined then the tcb
| is added to the end of the appropriate queue. If the
| _PRIORITY_ORDER_SEMA_QUEUES_ is defined then it is added to the
| appropriate queue in priority order. With the new implementation of the
| priority inheritance/disinheritance mechanism this is really not
| necessary, since the usignal and wait primitives are generally used as
| synchronization primitives, and priority ordering is not as important
| as if the wait and usignal are used for critical section protection.
|
| If a task is to be removed from the queue it is removed from the top. If a
| task is removed from the top of the queue then a preemptive_schedule must
| occur since the current running task may now not be the highest priority one.
| If so a preemptive task switch occurs. There are two other variations to
| this basic behaviour. The first is related to entering this routine from a
| signal instigated by the expiring of a timer in the system. In this case
| one does not want a pre_emptive task switch to occur since this could cause
| a lengthy task to run when the entry to this point has occurred from the
| dec_timers routine in the tick function. This is a problem because it would
| upset the precision of the expiry of other timers if multiple timers were
| expiring at the same time as the first timer. The other is the dequeuing
| when the multiplier component of the semaphore is non zero. This is
| usually the case for a semaphore which is being used for an event. In this
| case one wants multiple tasks to be started from the one signal. Therefore
| the dequeue routine will attempt to move multiple tasks from the semaphore
| queue to the appropriate priority queues. If the number of tasks specified
| by the num_of_tasks_to_remove cannot be removed then the remainder is
| stored on the semaphore_value component of the semaphore structure.
|
| If on a semaphore dequeue operation the semaphore value is zero (for both
| single or multiple value semaphores) then the tcb which has just been
| dequeued is the new tcb which has the semaphore.
|
| A further complication occurs in this routine due to the presence of timed
| waits on semaphores. A check is made to see if the task removed from the
| queue is a timed wait task and if so then the timer associated with the
| task is stopped.
|
| The parameters passed in is the required operation. The queue to be acted on
| is pointed to by the semaphore contained in the kernel control block.
|
| A change was made to this routine when interrupts were enabled in the kernel-
| namely that the kcb structure which instigated the signal operation in the
| kernel is passed to this routine. This is a new feature. This allows the
| semaphore number that has to be acted on to be passed into the routine.
|
```

```

|
| Parameters : - operation to be performed on the queue.
|
| Entry via  : - usignal function.
|
|=====
*/

static void semaphore_queue_handler ( int queue_op,
                                     kcb_structure* local_kcb_ptr )
{
    task_control_block* tcb_ptr, * bgn_sema_q_ptr;
    unsigned int num_of_tasks_to_remove;
    unsigned int bgn_q_index;
#ifdef _PRIORITY_ORDER_SEMA_QUEUES_
    unsigned int end_q_index;
    task_control_block * tcb_ptr_1,
#endif

    if ( queue_op == QUEUE )
    {
        bgn_q_index =
            semaphore [ local_kcb_ptr->semaphore_index ]->semaph_bgn_q_index;
        bgn_sema_q_ptr = ( task_control_block* )central_table [ bgn_q_index ];
        tcb_ptr = ( task_control_block* )central_table [ CURRENT_TASK ];
        /* update the information on the queue type stored in the tcb */
        tcb_ptr->q_type = SEMAPHORE_Q_TYPE;
        tcb_ptr->task_status = TASK_BLOCKED;
        tcb_ptr->cent_tab_index = bgn_q_index;
        tcb_ptr->sema_blocked_on = local_kcb_ptr->semaphore_index;

        /* now check to see if the semaphore queue is currently empty. If so
        then place the current task at the beginning of the queue.
        */
        if ( bgn_sema_q_ptr == NULL_PTR )
        {
            ( task_control_block* )central_table [ bgn_q_index ] = tcb_ptr;
            ( task_control_block* )central_table [ bgn_q_index + 1 ] =
                                                    tcb_ptr;

            tcb_ptr->prev_task_ptr = NULL_PTR;
            tcb_ptr->next_task_ptr = NULL_PTR;
        } /* if */
        else
        {
            /* there are already other tcbs on the semaphore queue therefore
            find the correct spot on the queue to place the current task
            */
            /*-----*/
#ifdef _PRIORITY_ORDER_SEMA_QUEUES_
            /* ONLY COMPILE THIS SECTION IF ONE DESIRES TO HAVE THE SEMAPHORE QUEUES
            PRIORITY ORDERED.
            */

            tcb_ptr_1 = bgn_sema_q_ptr;

            /* now search through the queue to find the correct spot to put
            the current task. N.B. the highest priority is priority 1.
            */
            while ( ( tcb_ptr->priority >=
                    tcb_ptr_1->priority ) &&
                    ( tcb_ptr_1 != NULL_PTR ) )
                tcb_ptr_1 = tcb_ptr_1->next_task_ptr;

            /* at this stage tcb_ptr_1 is either pointing to the tcb which
            should be the one after the tcb to be placed, or alternatively
            it is pointing to the end of the queue.
            */
            if ( tcb_ptr_1 != NULL_PTR )

```

```

{
    /* the tcb has to be placed somewhere in the middle of a
    semaphore queue.
    */
    if ( tcb_ptr_1 != bgn_sema_q_ptr )
        tcb_ptr_1->prev_task_ptr->next_task_ptr = tcb_ptr;
    else
        ( task_control_block* )central_table [ bgn_q_index ] =
            tcb_ptr;
    tcb_ptr->prev_task_ptr = tcb_ptr_1->prev_task_ptr;
    tcb_ptr->next_task_ptr = tcb_ptr_1;
    tcb_ptr_1->prev_task_ptr = tcb_ptr;
} /* if */
else
{
    /* the tcb has to be placed on the end of the queue */
    end_q_index = bgn_q_index + 1;
    ( ( task_control_block* )central_table [ end_q_index ] )->
        next_task_ptr = tcb_ptr;
    tcb_ptr->prev_task_ptr = ( task_control_block* )central_table [
        end_q_index ];
    tcb_ptr->next_task_ptr = NULL_PTR;
    ( task_control_block* )central_table [ end_q_index ] =
        tcb_ptr;
} /* else */

#else
/* THIS SECTION IS COMPILED TO HAVE ORDER OF BLOCKAGE ORDERING OF THE SEMAPHORE
QUEUES - I.E. NOT PRIORITY ORDERED QUEUES.
*/
    add_queue (( task_control_block* )central_table [ CURRENT_TASK ],
        semaphore [ local_kcb_ptr->semaphore_index ]->semaph_bgn_q_index);

#endif

/*-----*/
    } /* else */
} /* if */
else { /* must be a dequeue operation that is required */
    bgn_q_index = semaphore [
        local_kcb_ptr->semaphore_index ]->semaph_bgn_q_index;

    num_of_tasks_to_remove = semaphore [ local_kcb_ptr->semaphore_index ]->
        semaphore_multiplifier;

    /* now enter the remove tcb from queue loop. This loop attempts to
    remove num_of_tasks_to_remove from the semaphore queue. If it
    cannot remove this number of tasks then the remainder is stored in
    the semaphore_value location of the semaphore structure.
    */
    do
    {
        tcb_ptr = ( task_control_block* )central_table [ bgn_q_index ];

        /* now check to see if this task is on a timed wait. If so then
        stop the timer associated with the task. The timer pointer is
        contained within the tcb structure.
        */
        if ( tcb_ptr->timer_ptr != NULL_PTR )
        {
            stop_timer ( tcb_ptr->timer_ptr );
            tcb_ptr->timer_ptr = NULL_PTR;
        } /* if */

        remove_top_queue ( bgn_q_index );
        place_priority_queue ( tcb_ptr );
        num_of_tasks_to_remove--;
    }

```

```
    } while ( ( num_of_tasks_to_remove ) &&
               central_table [ bgn_q_index ] != NULL_PTR );

    semaphore [ local_kcb_ptr->semaphore_index ]->semaphore_value =
        num_of_tasks_to_remove;

    } /* else */
} /* end of semaphore_queue_handler */


/*-----*/


/*
=====
|
| place_semaphore_queue
|
| This routine places the tcb pointed to by the central_table(CURRENT_TASK)
| on the blocked queue indicated by the semaphore passed in.
|
| Parameter
|     - semaphore array index to semaphore in question
=====
*/

static void place_semaphore_queue ( unsigned int semaphore_num )
{
    kcb_structure local_kcb;
    void ( *handler_semaphore_q ) ( int queue_op, kcb_structure* );

    local_kcb.semaphore_index = semaphore_num;
    handler_semaphore_q = semaphore [ semaphore_num ]->semaph_queue_handler;
    ( *handler_semaphore_q ) ( QUEUE, &local_kcb );
} /* end place_semaphore_queue */


/*-----*/


/*
=====
|
| usignal_handler
|
| This routine is called from the kernel routine to carry out the handling
| of all matters related to signal operations.
|
| Parameters:  pointer to the kcb
|
| Returns:     nothing
|
=====
*/

static void usignal_handler (kcb_structure* local_kcb_ptr)
{
    void ( *handler_semaphore_q ) ( int queue_op, kcb_structure* );

    if ( ( task_control_block* )central_table [ semaphore [
```

Appendix H — UNOS-V2.0 LISTING

```
local_kcb_ptr->semaphore_index ]->semaph_bgn_q_index ] != NULL_PTR )
{
    /* There is a task on the blocked queue for the semaphore so start
    the process of dequeuing it.
    */

    handler_semaphore_q = semaphore [ local_kcb_ptr->semaphore_index
                                     ]->semaph_queue_handler;
    ( *handler_semaphore_q ) ( DEQUEUE, local_kcb_ptr );

    /* now carry out a preemptive schedule to see if there is a new
    task to run at this stage. If the entry was via the np_signal routine
    then the preemption_status flag will be clear. There is also a check to
    see if the entry was from the dec_timers routine - this is indicated
    by the possible_preemption flag.
    */
    if (local_kcb_ptr->preemption_status &&
        !local_kcb_ptr->dectime_signal_flag)
    {
        /* Preemption allowed and entry was not from the decrement timers
        routine.
        */
        preemptive_schedule_handler ( );
    } /* if */
    else if (local_kcb_ptr->dectime_signal_flag)
    {
        /* signal generated via the dec_timers routine so set a flag that
        will be used in the tick handler routine to cause a preemptive
        schedule occur if a normal time slice is not going to occur.
        */
        possible_preemption = TRUE;
    } /*else if*/
} /* if */
else
{
    /* no blocked or claiming tasks so increment the semaphore */
    semaphore [ local_kcb_ptr->semaphore_index ]->semaphore_value +=
        semaphore [ local_kcb_ptr->semaphore_index ]->semaphore_multiplier;
} /* else */

} /*usignal_handler*/
```

```
/*
=====
|
| init_tcb_and_q_it
|
| The function of this procedure is to initialize all of the elements of the
| tcb for a task except the stack pointer and to place the tcb in the correct
| queue. Note that to allow for setting up of the null task ( which is set
| up as the current task and is not queued ) there is a special q_type called
| DONOT_Q_TYPE. This is simply a return statement. Hence all the routine
| has done is to initialise the tcb for the task.
|
| In order to carry out the initialisation of the tcb memory firstly must
| be allocated for it. This is allocated from the UNOS heap. The pointer to
| the allocated memory is then placed in the tcb pointer array in the
| location corresponding to the task number.
|
| The routine returns the pointer to the tcb memory area allocated. If the
| memory allocation has been unsuccessful then the pointer returned is a
| NULL_PTR.
|
| The parameters passed into the routine are:
```

```

task_name - a pointer to a character array containing the name of the
            task.

priority_of_task - the priority of the newly created task.

tick_delta - the amount that the ticks_per_time_slice value is altered
            to change the micro priority of tasks at the same priority
            level.

status_of_task - contains the desired contents of the tcb status. Must
            be consistent with the queue that the task is placed on.

q_type - the type of queue the tcb is to be placed in - priority queue,
        semaphore queue, or the time queue. This information is required
        to invoke the correct routine for queue addition.
        Queue types -
            PRIORITY_Q_TYPE
            SEMAPHORE_Q_TYPE
            DONOT_Q_TYPE

semaphore - index into the semaphore array if tcb is to be added to the
            blocked queue of a particular semaphore.
=====
*/

static task_control_block *init_tcb_and_q_it ( char *task_name,
        unsigned char priority_of_task,
        int tick_delta,
        unsigned char status_of_task,
        unsigned char q_type,
        unsigned int semaphore_num
        )
{
    /* firstly allocate the memory for the task control block */
    if ( ( tcb [ num_of_tasks ] = ( task_control_block * )umalloc
        ( sizeof ( task_control_block ) ) ) == NULL_PTR )
    {
        return tcb [ num_of_tasks ];
    } /* if */

    tcb [ num_of_tasks ]->task_num = num_of_tasks;
    tcb [ num_of_tasks ]->task_name_ptr = task_name;
    tcb [ num_of_tasks ]->priority = priority_of_task;

    /* now set up the number of ticks before this particular task will
    enter the kernel. The number of ticks is set up as an absolute value.
    */
    if ( base_ticks_per_time_slice + tick_delta <= 0 )
    {
        tcb [ num_of_tasks ]->ticks_before_kernel_entry = 1;
    } /* if */
    else
    {
        tcb [ num_of_tasks ]->ticks_before_kernel_entry =
            base_ticks_per_time_slice + tick_delta;
    } /* else */

    tcb [ num_of_tasks ]->ticks_to_go =
        tcb [ num_of_tasks ]->ticks_before_kernel_entry;

    tcb [ num_of_tasks ]->q_type = q_type;
    tcb [ num_of_tasks ]->task_status = status_of_task;
    tcb [ num_of_tasks ]->timeout_flag = FALSE;
    tcb [ num_of_tasks ]->cent_tab_index = 0;
    tcb [ num_of_tasks ]->waiting_for = NULL_PTR;
    tcb [ num_of_tasks ]->timer_ptr = NULL_PTR;

    /* Set up these initializations for the following routines */

```

Appendix H — UNOS-V2.0 LISTING

```
( task_control_block* ) central_table [ CURRENT_TASK ] =
                                tcb [ num_of_tasks ];
/* Now place the tcb in the appropriate queue */
switch ( q_type)
{
    case 0 :
        place_priority_queue ( tcb [ num_of_tasks ] );
        break;
    case 1 :
        place_semaphore_queue ( semaphore_num );
        break;
    case 2 :
        tcb [ num_of_tasks ]->cent_tab_index = 0;
        break;
    default :
        cprintf ( "Illegal queue type\n\r" );
} /* switch */

return tcb [ num_of_tasks ];
} /* end of init_tcb_and_q_it */

/*-----*/

/*****
/*
/*          OPERATING SYSTEM MOLECULES
/*
/*
*****/

/*
=====
|
| scheduler
|
| This procedure searches through the priority queues to find the next task
| to run. The search begins with the high priority queue - the task at the
| top of the queue is selected to run. If the queue is empty or contains
| no active task then the search continues in the low priority queue. If
| there is not a task here to run then the search continues in the null queue.
| The null task is always ready to run. The task to next run is placed in the
| central_table CURRENT_TASK location.
|
| Parameters : - none
| Entry via  : - multiple paces in kernel
|
=====
*/

static void scheduler ( void )
{
    unsigned int central_table_index = BGN_PRIORITY_Q_1;
    task_control_block* tcb_ptr;
    LOCK lock_ptr;

    while ( central_table_index < ( num_of_priorities * 2 ) )
    {
        if ( central_table [ central_table_index ] != NULL_PTR )
```



```

{
    /* task in priority queue so make it the current priority */
    tcb_ptr = (task_control_block*)central_table[central_table_index];
    current_priority = tcb_ptr->priority;

    do
    {
        /* Now check to see if the task is being blocked from running by
        a lock.
        */
        lock_ptr = tcb_ptr->waiting_for;

        /* Now check to see if the task is blocked by a lock, or if the
        lock it is competing for has already been released.
        */
        if ((lock_ptr == NULL_PTR) || (!lock_ptr->locked))
        {
            /* No lock or the lock is now available so get out of the
            loop
            */
            break;
        } /* if */

        /* There is a lock blocking the task, so find out which task
        has the lock.
        */
        tcb_ptr = lock_ptr->holder;
    } while (1);

    /* At this point the tcb_ptr location contains the next task to run.
    Now locate the queue this task is in.
    */

    central_table [ CURRENT_TASK ] = tcb_ptr;
    remove_queue ( tcb_ptr,
                    ( ( tcb_ptr->priority ) - 1 ) << 1 ) + 1 );
    central_table_index = ( num_of_priorities * 2 );
} /* if */
else
{
    /* either no task in the queue so go to the next queue */
    central_table_index = central_table_index + 2;
} /* else */
} /* while */

/* At this point the CURRENT_TASK entry in the central table should contain
the next task to be dispatched. Note that the current_priority location
may not contain the same priority for this task if the highest priority
runnable task is blocked on a lock. The ultimate blocker is the task that
is running.
*/

} /* end of scheduler */

/*-----*/

/*****
*/
*/

```

Appendix H — UNOS-V2.0 LISTING

```
/*                                                    */
/*                  UNOS_2 MODULE                      */
/*                                                    */
/*                  by                                  */
/*                                                    */
/*                  Robert Betz                        */
/*      Department of Electrical Engineering and Computer Science */
/*                  University of Newcastle             */
/*                  Australia                           */
/*                                                    */
/*                  ( Copyright 1989 )                 */
/*                                                    */
/*                                                    */
/*                                                    */
/******

/*

HISTORY

This module was input on the 22/11/89

*/

/*

DESCRIPTION

This module contains the lattice routines of the operating system kernel and
the operating functions related to the timers in UNOS. The user interface
routines for the timers reside in the UNOS_3 include module.

For a complete description of the timer mechanism refer to the UNOS.DOC file.

*/

/******
/*                                                    */
/*                  THE LATTICE ROUTINES                */
/*                                                    */
/*                  I.E. KERNEL HANDLER ROUTINES        */
/*                                                    */
/******

/* These routines are called directly from the kernel to handle the source of
kernel entry.
*/

/*
=====
|
| wait_handler
|
| This handler is entered if when a semaphore value is decremented in the wait
| routine the value goes to zero. This means that the task has become blocked
| therefore another has to be scheduled. This routine puts the current task
| onto the appropriate semaphore queue and then calls the scheduler.
|
| Parameters : - pointer to the current task control block
|
| Entry via  : - kernel routine
|
```

```
|
=====
*/

static void wait_handler ( kcb_structure* local_kcb_ptr )
{

    void ( *handler_semaphore_q ) ( int queue_op, kcb_structure* );

    handler_semaphore_q = semaphore [
        local_kcb_ptr->semaphore_index ]->semaph_queue_handler;
    ( *handler_semaphore_q ) ( QUEUE, local_kcb_ptr );
    scheduler ();

} /* end of wait_handler */


/*-----*/


/*
=====
|
| reschedule_handler
|
| This procedure is entered if the current task has requested a reschedule.
| This routine differs from the pre-emptive handler in that the current
| task is not placed on the priority queue before the scheduler is called.
| Therefore the current task is not the next task to run. After the scheduler
| has got the next task to run the previous current task is placed on the
| appropriate priority queue. This implies that the next task to run could
| be lower priority then the current task.
|
| Parameters : - none
|
| Entry via  : - multiple places
|
=====
*/

static void reschedule_handler ( void )
{

    task_control_block* tcb_ptr;

    tcb_ptr = ( task_control_block* )central_table [ CURRENT_TASK ];

    scheduler ();
    place_priority_queue ( tcb_ptr );
} /* end of reschedule_handler */


/*-----*/


/*
=====
|
| time_slice_handler
|
```

Appendix H — UNOS-V2.0 LISTING

```
|
| This routine is entered if a time slice interrupt has been received. It
| simply calls the scheduler to get the next routine to run.
|
| Parameters : - none
|
| Entry via  : - kernel
|
|=====
|*/
|
static void time_slice_handler ( void )
{
    place_priority_queue (
        ( task_control_block* ) central_table [ CURRENT_TASK ] );
    scheduler ( );
} /* end of time_slice_handler */

/*-----*/

/*
|=====
|
| preemptive_schedule_handler
|
| This routine is entered if there is the possibility of a preemptive
| schedule occurring. This routine differs from the normal schedule
| routine in that it checks to see if there is a task of higher priority
| than the current task. This is achieved by checking the beginning of
| each of the queues for higher priority tasks than the current task to
| see if the first tcb pointer is not a NULL_PTR. If not a NULL_PTR then there is a
| task on the queue ready to run. The current task is then saved in the
| appropriate priority queue and the higher priority task is then made
| the current task.
|
| Note that even if there is a task of the same priority then this task will
| not be run - i.e. with a pre-emptive task switch round robin scheduling
| does not occur.
|
| Parameters : - none
| Entry via  : - kernel routine
|
|=====
|*/
|
static void preemptive_schedule_handler ( void )
{
    task_control_block* cur_tcb_ptr;

    /* see if the priority of the current task is less then the highest
    priority. If not then leave since no other task can be of higher
    priority.
    */
    cur_tcb_ptr = ( task_control_block* ) central_table [ CURRENT_TASK ];
    if ( cur_tcb_ptr->priority > 1 )
    {
        /* Task does not have the highest priority so add it to the top of its
```

```

        queue then call the scheduler
        */
        add_top_queue ( cur_tcb_ptr,
                        ( ( ( cur_tcb_ptr->priority ) - 1 ) << 1 ) + 1 );
        scheduler ();

    } /* if */
} /* end preemptive_schedule_handler */

/*-----*/

/*
=====
|
| change_task_priority_handler
|
| This function is entered to handle the change of a tasks priority. This
| is a reasonably complex process since it can involve moving a task to
| a different priority queue. Therefore there is also the possibility of a
| reschedule occurring.
|
| Parameters:   pointer to a kcb_structure
|
| Returns:      nothing
|
=====
*/

static void change_task_priority_handler (kcb_structure* local_kcb_ptr)
{
    int old_priority;
    task_control_block *cur_tcb_ptr, *tcb_ptr;

    cur_tcb_ptr = (task_control_block*)central_table[CURRENT_TASK];
    tcb_ptr = tcb[local_kcb_ptr->task_number];

    /* save the old priority for later */
    old_priority = tcb_ptr->priority;

    /* always change the priority to the new priority */
    tcb_ptr->priority = local_kcb_ptr->new_priority;

    /* Now reposition the task on its queue */
    /* Firstly check to see if the task is the currently running task or it is
    on a semaphore queue
    */
    if ((local_kcb_ptr->task_number == cur_tcb_ptr->task_num)
        || (tcb_ptr->q_type == SEMAPHORE_Q_TYPE))
    {
        /* task is current task or is on semaphore queue so do not have to do
        anymore.
        */
        return;
    } /* if */

    /* Control reaches this point if the task whose priority has been changed
    is not the current task and is on a priority queue.
    */

    /* If the task whose priority has been changed is on a priority queue, then
    remove it from the queue and replace it on the priority queue. This will

```

Appendix H — UNOS-V2.0 LISTING

```
    ensure that it is on the correct priority queue. Note that the old
    priority is used to locate the priority queue the tcb is on.
    */
    remove_queue (tcb_ptr, ( ( old_priority - 1 ) << 1 ) + 1 );
    place_priority_queue ( tcb_ptr );

    /* now check to see if the kernel needs to be entered at all */
    if ( local_kcb_ptr->new_priority >= cur_tcb_ptr->priority )
    {
        /* No need to do anything */
        return;
    } /* if */

    preemptive_schedule_handler();
}/*change_task_priority_handler*/

/*-----*/

/*
=====
|
| kernel
|
| This procedure is usually entered as a result of a software interrupt. The
| routine is set up as an interrupt procedure and this ensures that all the
| current task data is saved on the stack. The routine assumes that the kcb
| has been set up by some prologue routine before entering this routine - see
| the kcb definition and the description at the begining of the program
| for the details on this.
|
| The first function of the routine is to save the current stack offset and
| frame in the tcb of the current task. The procedure then enters a case
| statement whose selector is a component of the kernel control block. It is
| assumed by the kernel that the kernel control block has been set up by one
| of the prologue routines before entering the kernel proper.
|
| The central section of this routine consists of a while loop that continues
| processing kcb's retrieved from the kcb queue. This process continues until
| the kcb queue is empty. Note that the kcb's are dynamically allocated in the
| kernel interface routines. As each kcb is processed its memory is freed
| back to the system.
|
| After the kcb queue has been completely processed one eventually exits from
| the kernel via the dispatcher section. This section uses the information
| in the central_table[CURRENT_TASK] tcb to restore the state of the next task
| in the system.
|
| Parameters:    none
|
| Returns:      nothing
|
|=====
*/

void interrupt kernel ( void )
{
    static task_control_block* cur_tcb_ptr;
    unsigned int stkbases, stkptr, baseptr;
    kcb_structure* local_kcb_ptr;
```

```
//      char strg1 [20];
//      unsigned char priority;
//      char* base_video_ptr;
static char* video_ptr = (char*)MK_FP(0xb800, 0x0EFE);
static int i =0;

/* The following flag is set to indicate that the kernel has been entered.
This flag is always checked in the signal routines, an it set then the
kernel is no entered from the signal routine, but the signal operation is
queued.
*/
kernel_executing = TRUE;

/*****
NOW REENABLE THE INTERRUPTS - THIS IS POSSIBLE BECAUSE OF CHANGES
MADE FOR UNOS 2.0 THAT QUEUE ANY USIGNAL OPERATIONS IF THEY OCCUR WHILST
THE KERNEL IS EXECUTING.
*****/

NOTE: If one wishes the use floating point in interrupt routines then
one needs to move this enable to below the saving of the floating point
registers. This will then prevent the interrupt routine from
corrupting the floating point register saving process.

The if statement is around this to prevent the interrupts from being
enable during the taks creation process.
*/

if (start_flag)
{
    enable();
}/*if*/

/* Firstly save the stackbase and stack pointer for the task pointed to
by the central_table. For a TurboC compiler the base pointer must also
be saved since this is used to reference the variables defined in this
routine - i.e. the local variables.
!!!! Note this section of the code is not portable and would most likely
have to be implemented as an assembly language routine in most 'C'
implementations.
*/

stkbase = _SS;
stkptr = _SP;
baseptr = _BP;

cur_tcb_ptr = ( task_control_block* ) central_table [ CURRENT_TASK ];
cur_tcb_ptr->task_stkbase = stkbase;
cur_tcb_ptr->task_stkptr = stkptr;
cur_tcb_ptr->task_baseptr = baseptr;

/* save all the floating point registers in the system.
*/
save_fp_regs ( cur_tcb_ptr->fp_save_area );

/* --->>> Put enable here if one wishes to use floating point in
interrupt routines.
*/

/* Now save the task information in the kernel debugging structure.
Note that this code can be commented out (along with similar code at
the end of the kernel) with no ill effects in terms of kernel operation.
In fact it will speed up the task switching.
*/

/* The task information is saved in a temporary location, and a check
is made at the end of the kernel to see if there has been a task switch.
```

Appendix H — UNOS-V2.0 LISTING

```
    If so then this temporary information is stored in the circular buffer
    of these structures.
    */
#ifdef DEBUG
    temp_kernel_store.taskname_ptr = cur_tcb_ptr->task_name_ptr;
    temp_kernel_store.priority = cur_tcb_ptr->priority;
    temp_kernel_store.clock_cnt = rt_clock;
#endif /* DEBUG */

    /* We are now ready to execute the kernel proper.
    This involves processing the queue of kcb's. The first kcb is removed from the
    queue and processed by the kernel. This continues until all the kcb's
    have been processed. At this stage there will be a current task which is
    dispatched by the kernel.

    The queue concept is used to make sure that only one request for kernel
    services can be processed at a time. It is this structure that allows the
    interrupts to be enable whilst we are in the kernel.
    */

    if ((local_kcb_ptr = dequeue_kcb()) != NULL_PTR);
    {
        do
        {
            /* Now enter a case statement which uses the entry_type component
            of the kcb as the selector to begin execution of the correct
            handling routine.
            */

            switch ( local_kcb_ptr->entry_type )
            {
                case WAIT_ENTRY :
#ifdef DEBUG
                    temp_kernel_store.reason_ptr = wait_entry;
                    temp_kernel_store.semaphore_num =
                        local_kcb_ptr->semaphore_index;
#endif

                    wait_handler ( local_kcb_ptr );
                    break;

                case USIGNAL_ENTRY :
                    /* Only enter here if a task is to be unblocked via
                    the semaphore
                    */
#ifdef DEBUG
                    temp_kernel_store.reason_ptr = usignal_entry;
                    temp_kernel_store.semaphore_num =
                        local_kcb_ptr->semaphore_index;
#endif

                    /* Call the handler to carry out the necessary operations
                    */
                    usignal_handler(local_kcb_ptr);

                    break;

                case RESCHEDULE_ENTRY :
#ifdef DEBUG
                    temp_kernel_store.reason_ptr = reschedule_entry;
#endif

                    reschedule_handler ( );
                    break;

                case TICK_ENTRY :
                    /* This routine is entered if the kernel has been entered
                    via the tick interrupt. At this stage the only processing
                    that has been carried out by the tick interrupt routine
                    is the decrementing of timers (if any). This could have
```



```

        resulted in a task being moved from a blocked
        queue to a priority queue, and therefore there is the
        possibility of preemption being required.
        */

#ifdef DEBUG
        temp_kernel_store.reason_ptr = tick_entry;
#endif

        /* get the current tcb - this could be different from
        that present upon entry since one can loop in the
        kernel
        */
        cur_tcb_ptr =
        ( task_control_block* ) central_table [ CURRENT_TASK ];

        /* Now check to see if the ticks_to_go == 0 - if so then
        carry out a time slice schedule. Note that this will
        account for the case of possible preemption - if the task
        moved to the priority queue is the highest priority then
        it will be selected to run.
        */
        if (((--cur_tcb_ptr->ticks_to_go) == 0) && time_slice_flag)
        {
            time_slice_handler();
        }/*if*/
        else if (possible_preemption)
        {
            /* A timer has resulted in a task being made runnable
            so check whether a preemption is necessary.
            */
            possible_preemption = FALSE;
            preemptive_schedule_handler ( );
        }/*else if*/

        break;

case CREATE_TASK_STK_ENTRY :

#ifdef DEBUG
        temp_kernel_store.reason_ptr = create_task_entry;
#endif

        break;

case PREEMPTIVE_ENTRY :

#ifdef DEBUG
        temp_kernel_store.reason_ptr = preemptive_entry;
        temp_kernel_store.semaphore_num =
            local_kcb_ptr->semaphore_index;
#endif

        preemptive_schedule_handler ( );
        break;

case LOCK_ENTRY :

#ifdef DEBUG
        temp_kernel_store.reason_ptr = lock_entry;
#endif

        cur_tcb_ptr = (task_control_block*)central_table[CURRENT_TASK];
        cur_tcb_ptr->task_status = TASK_RUNNABLE;
        cur_tcb_ptr->q_type = PRIORITY_Q_TYPE;
        add_top_queue ( cur_tcb_ptr,
            ( ( ( cur_tcb_ptr->priority ) - 1 ) << 1 ) + 1 );
        scheduler ( );
        break;

case UNLOCK_ENTRY :

#ifdef DEBUG
        temp_kernel_store.reason_ptr = unlock_entry;
#endif

        preemptive_schedule_handler ( );

        break;

```

Appendix H — UNOS-V2.0 LISTING

```
        case CHANGE_TASK_PRIORITY :
#ifdef DEBUG
            temp_kernel_store.reason_ptr = change_task_priority_entry;
            temp_kernel_store.taskname_ptr =
                tcb[local_kcb_ptr->task_number]->task_name_ptr;
            temp_kernel_store.priority =
                tcb[local_kcb_ptr->task_number]->priority;
            temp_kernel_store.nxt_priority =
                local_kcb_ptr->new_priority;
#endif

            change_task_priority_handler(local_kcb_ptr);

            break;

        default:

            break;

    } /* switch */

    /* now free the memory allocated for the kcb */
    ufree((char huge*)local_kcb_ptr);

    } while ((local_kcb_ptr = dequeue_kcb()) != NULL_PTR);
} /*if */

/* After returning from one of the kernel entry handlers the task pointed
to by the central_table(CURRENT_TASK) is the next task to run. The
dispatcher simply reinstates the stored stack pointer and stack base by
executing the normal exit from an interrupt routine.
*/

/* NOTE: interrupts will again be disabled when we reached this point.
From now on until the task is dispatched we are eliminating any possibility
of another reentrant usignal.
*/

/* The following bit is for debugging purposes only. Allows data to be
written directly out onto the video screen.
*/

// priority = tcb [1]->priority;

// base_video_ptr = MK_FP(0xb800, 0x0000);

/* now format the info */
// sprintf (strg1, "P%d", priority);

/* Now print onto the screen */

// base_video_ptr = MK_FP(0xb800, 0x0000);
// i = 0;
// while (strg1 [i] != NULL_PTR)
// {
//     *base_video_ptr = strg1 [i];
//     i++;
//     base_video_ptr++;
//     *base_video_ptr = 07;
//     base_video_ptr++;
// } /* while */

/* Print a rotating line at the bottom right hand corner of the screen.
*/
switch(i)
{
    case 0:
        *video_ptr = '\|';
```

```

        i++;
        break;

    case 1:
        *video_ptr = '/';
        i++;
        break;

    case 2:
        *video_ptr = '-';
        i++;
        break;

    case 3:
        *video_ptr = '\\';
        i = 0;
        break;

    default:
        break;

} /*switch*/

/* Dispatcher */

cur_tcb_ptr = ( task_control_block* ) central_table [ CURRENT_TASK ];

#ifdef DEBUG
/* Now check to see if there has been a task switch - if so then
write the extra information into the temp_kernel_store structure and then
save it into the circular buffer of structures, else do nothing.
*/
if (temp_kernel_store.taskname_ptr != cur_tcb_ptr->task_name_ptr)
{
    /* Has been a task switch so save the information */
    temp_kernel_store.nxt_taskname_ptr = cur_tcb_ptr->task_name_ptr;
    temp_kernel_store.nxt_priority = cur_tcb_ptr->priority;
    temp_kernel_store.count_num = ++count_num;

    /* Now copy the data into the circular buffer structure */
    kernel_debug_info [kernel_debug_head++] = temp_kernel_store;

    if (kernel_debug_head >= SIZE_DEBUG_BUFFER)
    {
        kernel_debug_head = 0;
    } /* if */
} /* if */
#endif /* DEBUG */

kernel_executing = FALSE;

/* now restore the floating point registers */
restore_fp_regs ( cur_tcb_ptr->fp_save_area );

_SP = cur_tcb_ptr->task_stkptr;
_BP = cur_tcb_ptr->task_baseptr;
_SS = cur_tcb_ptr->task_stkbase;

} /* end of kernel */

/*-----*/

```

Appendix H — UNOS-V2.0 LISTING

```
/*
=====
|
| add_timer_active_q
|
| This function places a timer on the queue of active timers. During this
| process the timing related contents of the timer structure have to be
| initialised with the variables passed in.
|
| The active time queue is a delta queue in relation to the time count. This
| means that the period before timeout for any timer is the accumulation of
| all the time counts for the timers preceding the timer in question plus the
| time count for the timer in question. The initial time count is the total
| timeout time for the timer. Therefore when a timer is placed on the time
| queue it is placed in the correct location so that the accumulated deltas
| will give the correct total timeout time. Depending on the position that
| the timer is placed this may involve altering the next timer delta count as
| well.
|
| Parameters : - timer type - single shot or repetitive.
|               - initial time count
|               - pointer to timeout handler
|               - pointer to timeout handler data
|
| Entry via   : - task$to$sleep procedure and alarm set routines
|
=====
*/

static void add_timer_active_q ( timer_struct* timer,
                                unsigned char timer_type,
                                unsigned long init_time_count,
                                void ( *timeout_handler ) (void * ),
                                void* data_ptr )
{
    timer_struct* temp_timer;
    timer_struct* otemp_timer = NULL_PTR;
    unsigned long accumulated_time;
    char placed_timer = FALSE;

    if ( timer != NULL_PTR )
    {
        timer->status = ACTIVE;
        timer->type = timer_type;
        timer->init_time_cnt = init_time_count;
        timer->timer_handler = timeout_handler;
        timer->timer_handler_data = data_ptr;

        /* now link the timer into the correct part of the active queue. Start
        searching from the top of the queue to find the correct spot.
        */
        temp_timer = ( timer_struct* ) central_table [ bgn_active_time_q ];

        if ( temp_timer == NULL_PTR )
        {
            /* active queue is empty so put on top of queue */
            timer->prev_timer_ptr = NULL_PTR;
            timer->next_timer_ptr = NULL_PTR;
            timer->delta_time_cnt = init_time_count;
            ( timer_struct* )central_table [ bgn_active_time_q ] = timer;
            ( timer_struct* )central_table [ end_active_time_q ] = timer;
        } /* if */
        else
        {
            /* have to search queue to find correct spot */
            accumulated_time = temp_timer->delta_time_cnt;

            while ( ( temp_timer != NULL_PTR ) && ( !placed_timer ) )

```

```

{
    if ( accumulated_time >= timer->init_time_cnt )
    {
        /* place timer in current position */
        timer->delta_time_cnt = accumulated_time -
                               temp_timer->delta_time_cnt;
        timer->delta_time_cnt = timer->init_time_cnt -
                               timer->delta_time_cnt;
        timer->prev_timer_ptr = temp_timer->prev_timer_ptr;
        timer->next_timer_ptr = temp_timer;
        if ( otemp_timer != NULL_PTR )
            otemp_timer->next_timer_ptr = timer;
        temp_timer->prev_timer_ptr = timer;
        temp_timer->delta_time_cnt = accumulated_time -
                                   timer->init_time_cnt;

        placed_timer = TRUE;

        /* see if at the beginning of the active queue */
        if ( timer->prev_timer_ptr == NULL_PTR )
            ( timer_struc* )central_table [ bgn_active_time_q ]
                = timer;
    } /* if */
    else
    {
        /* update temp_timer and increment accumulated_time */
        otemp_timer = temp_timer;
        temp_timer = temp_timer->next_timer_ptr;
        if ( temp_timer != NULL_PTR )
            accumulated_time += temp_timer->delta_time_cnt;
    } /* else */
} /* while */

if ( !placed_timer )
{
    /* have reached the end of the queue so add timer here */
    timer->delta_time_cnt = timer->init_time_cnt -
                          accumulated_time;

    timer->prev_timer_ptr = otemp_timer;
    timer->next_timer_ptr = NULL_PTR;
    otemp_timer->next_timer_ptr = timer;
    ( timer_struc* ) central_table [ end_active_time_q ] = timer;
} /* if */
} /* else */
} /* if */

} /* end of add_timer_active_q */

```

/*-----*/

```

/*
=====
|
| add_timer_inactive_q
|
| This function adds a timer to the queue of inactive timers. Timers are
| generic structures therefore a timer is simply added to the end of the
| queue.
|
| Parameters : - pointer to the timer to be added to inactive queue
|
| Entry via : -
|

```

Appendix H — *UNOS-V2.0 LISTING*

```
=====
*/

static void add_timer_inactive_q ( timer_struct* timer_ptr )
{
    timer_struct* end_q_ptr;

    end_q_ptr = ( timer_struct* )central_table [ end_inactive_time_q ];

    if ( end_q_ptr == NULL_PTR )
    {
        /* the inactive time queue is currently empty */
        ( timer_struct* )central_table [ bgn_inactive_time_q ] = timer_ptr;
        ( timer_struct* )central_table [ end_inactive_time_q ] = timer_ptr;
        timer_ptr->prev_timer_ptr = NULL_PTR;
        timer_ptr->next_timer_ptr = NULL_PTR;
        timer_ptr->status = INACTIVE;
    } /* if */
    else
    {
        /* link onto the end of exisiting queue */
        end_q_ptr->next_timer_ptr = timer_ptr;
        timer_ptr->prev_timer_ptr = end_q_ptr;
        timer_ptr->next_timer_ptr = NULL_PTR;
        timer_ptr->status = INACTIVE;
        ( timer_struct* )central_table [ end_inactive_time_q ] = timer_ptr;
    } /* else */
} /* end add_timer_inactive_q */

/*-----*/

/*
=====
|
| remove_timer_inactive_q
|
| This function simply removes a timer from the top of the inactive timer
| queue.  If there is no timer in the queue then a NULL_PTR pointer is returned.
|
| Parameters : - none
|
| Entry via : -
|
=====
*/

static timer_struct* remove_timer_inactive_q ( void )
{
    timer_struct* temp_timer = NULL_PTR;

    if ( central_table [ bgn_inactive_time_q ] != NULL_PTR )
    {
        temp_timer = ( timer_struct* ) central_table [ bgn_inactive_time_q ];
        ( timer_struct* ) central_table [ bgn_inactive_time_q ] =
                                temp_timer->next_timer_ptr;
        if ( central_table [ bgn_inactive_time_q ] == NULL_PTR )
        {
            central_table [ end_inactive_time_q ] = NULL_PTR;
        } /* if */
        else
    }
}
```

```

    {
        /*--- queue is not empty so make sure that the first timer
        structure in the queue has its prev_timer_ptr set to NULL_PTR.
        ---*/
        temp_timer->next_timer_ptr->prev_timer_ptr = NULL_PTR;
    } /* else */
} /* if */

return temp_timer;
} /* end of remove_timer_inactive_q */

/*-----*/

/*
=====
|
| remove_timer_active_q
|
| This is a function which allows one to remove a timer from the queue of
| active timers. The timer can be removed from the queue at any position
| in the queue. This routine is used generally to remove timers from the
| queue before they have timed out. When a timer is removed the link
| pointers have to be updated and the delta_time_cnt of the following timer
| altered appropriately. A pointer to the timer removed is returned.
|
| Parameters : - pointer to a timer structure
|
| Entry via : - multiple places
|
=====
*/

static timer_struct* remove_timer_active_q ( timer_struct* timer_ptr )
{
    if ( ( central_table [ bgn_active_time_q ] ==
          central_table [ end_active_time_q ] ) &&
        ( timer_ptr->status == ACTIVE ) )
    {
        /* only one timer on the queue */
        timer_ptr = ( timer_struct* ) central_table [ bgn_active_time_q ];
        central_table [ bgn_active_time_q ] = NULL_PTR;
        central_table [ end_active_time_q ] = NULL_PTR;
        return timer_ptr;
    } /* if */

    if ( ( timer_struct* ) central_table [ bgn_active_time_q ] == timer_ptr )
    {
        /* timer is on the top of the queue and there is more than one entry
        in the queue
        */
        ( timer_struct* ) central_table [ bgn_active_time_q ] =
            timer_ptr->next_timer_ptr;
        timer_ptr->next_timer_ptr->delta_time_cnt += timer_ptr->delta_time_cnt;
        timer_ptr->next_timer_ptr->prev_timer_ptr = NULL_PTR;
        return timer_ptr;
    } /* if */

    if ( ( timer_struct* ) central_table [ end_active_time_q ] == timer_ptr )
    {
        /* timer is on the end of the queue */
        ( timer_struct* ) central_table [ end_active_time_q ] =

```

Appendix H — UNOS-V2.0 LISTING

```

                                timer_ptr->prev_timer_ptr;
    timer_ptr->prev_timer_ptr->next_timer_ptr = NULL_PTR;
    return timer_ptr;
} /* if */

/* timer should be somewhere in the middle of the active time queue. Check
to see if the timer is active as this will ensure that it is on the active
time queue and not on the inactive time queue.
*/
if ( timer_ptr->status == ACTIVE )
{
    timer_ptr->prev_timer_ptr->next_timer_ptr = timer_ptr->next_timer_ptr;
    timer_ptr->next_timer_ptr->prev_timer_ptr = timer_ptr->prev_timer_ptr;
    timer_ptr->next_timer_ptr->delta_time_cnt +=
                                timer_ptr->delta_time_cnt;

    return timer_ptr;
} /* if */

return NULL_PTR;
} /* end of remove_timer_active_q */

/*-----*/

/*
=====
|
| dec_timers
|
| This function is called on every tick entry to the operating system. The
| function firstly checks to see if the delta_time_cnt value of the time on
| the top of the active time queue is zero. If so then :
| - remove the timer from the time queue
| - if timer is repetitive then place back at the appropriate place in
|   in the time queue else put it onto the inactive time queue
| - call the timeout handler
| - check if the next timer on the queue has got a delta of zero. If so
| - then go back to the first point on this list and repeat.
|
| - enter at this point if all timed out timers have been removed from the
|   time queue. If there is a timer on the time queue then decrement the
|   timer and return.
|
| Parameters : - none
|
| Entry via  : - tick interrupt routine
|
|=====
*/

static void dec_timers ( )
{
    timer_struc* timer_ptr;
    void ( * func ) ( void* ptr );

    /* set the sig_entry_from_dec_timers in case the handler routine is
going to do a signal. This flag prevents the signal handler from
doing a preemptive task switch at this stage. However it is detected
in the handler for the tick routine and results in the a preemptive
call to the scheduler to see if another task should run.
*/

```



```

sig_entry_from_dec_timers = TRUE;

/* check to see if the timer at the top of the queue has timed out. */
while ( ( ( timer_struc* ) central_table [ bgn_active_time_q ] )
        ->delta_time_cnt == 0 ) &&
        ( central_table [ bgn_active_time_q ] != NULL_PTR ) )
{
    /* timer has timed out so remove timer from the top of the queue.
    Check to see if the timer is a repetitive timer or a single shot timer.
    If repetitive timer then place the timer back into the active queue,
    else place the timer in the inactive queue if single shot.
    */
    timer_ptr = ( timer_struc* ) central_table [ bgn_active_time_q ];
    timer_ptr = remove_timer_active_q ( timer_ptr );
    if ( timer_ptr->type == REPETITIVE )
    {
        add_timer_active_q ( timer_ptr, timer_ptr->type,
                             timer_ptr->init_time_cnt, timer_ptr->timer_handler,
                             timer_ptr->timer_handler_data );
    }/*if */
    else
    {
        /* single shot timer */
        add_timer_inactive_q ( timer_ptr );
    }/*else*/

    /* now execute the handling routine */
    func = timer_ptr->timer_handler;
    ( * func ) ( timer_ptr->timer_handler_data );
} /* while */

/* decrement the current delta time count if there is a timer on the
queue
*/
if ( central_table [ bgn_active_time_q ] != NULL_PTR )
    ( ( timer_struc* ) ( central_table [ bgn_active_time_q ] ) )->
        delta_time_cnt--;

/* clear the sig_entry_from_dec_timers flag as it will have already
done its job at this stage.
*/
sig_entry_from_dec_timers = FALSE;
} /* end of dec_timers */

/*-----*/

/*****
/*
/*
/*
/*          UNOS_3 MODULE
/*
/*          by
/*
/*          Robert Betz
/*          Department of Electrical Engineering and Computer Science
/*          University of Newcastle
/*          Australia
/*
/*          ( Copyright 1989 )
/*
*****/

```

Appendix H — *UNOS-V2.0 LISTING*

```
/*
/*
/*****

/*

HISTORY

This module was formed from the initial single module UNOS to allow the
interactive environment of Turbo C to be used. At a later stage this module
and the others which comprise UNOS can be formed into a single module.

*/

/*

DESCRIPTION

This module is a sub-part of UNOS. It is "included" into the UNOS_1
module to form a complete system. The module primarily contains the user
interface routines to the kernel.

*/

/*****
/*
/*          USER INTERFACE TO KERNEL          */
/*
/*
/*****

/*
=====
|
| start_timer
|
| This routine is the user interface to start a timer for whatever purpose.
| This procedure preserves the interrupt status of the calling routine meaning
| that it can be called from an interrupt routine. The routine gets an existing
| timer from the inactive time queue.
|
| Returns a pointer to the timer structure.
|
| Parameters : - timer_type - repetitive or single shot
|              - initial timer count
|              - pointer to the timeout handler
|              - pointer to the timeout handler data structure
|
| Entry via  : - multiple places
|
|=====
*/

timer_struc* start_timer ( unsigned char timer_type,
                          unsigned long init_time_count,
                          void ( * timeout_handler ) (void*),
                          void* data_ptr )
{
    char int_status;
    timer_struc* timer_ptr;
```

```

    int_status = return_interrupt_status ();

    disable ();
    timer_ptr = remove_timer_inactive_q ();
    if ( timer_ptr != NULL_PTR )
        add_timer_active_q ( timer_ptr, timer_type,
                               init_time_count,
                               timeout_handler,
                               (void*)data_ptr );

    if ( int_status )
        enable ();

    return timer_ptr;
} /* end of start_timer */

/*-----*/

/*
=====
|
| reset_timer
|
| Takes a timer currently on the time queue and resets its count to the
| initial value. This involves firstly taking the timer out of the active
| timer queue and then putting it back into the time queue. This is required
| because the reset timer will in all probability sit at a different position
| on the time queue.
|
| If the reset is successful then a pointer to the timer is returned.
|
| Parameters : - pointer to the timer to be reset
|
| Entry via  : - multiple places
|
=====
*/

timer_struc* reset_timer ( timer_struc* timer_ptr )
{
    char int_status;
    timer_struc* temp_ptr;

    int_status = return_interrupt_status ();
    disable ();
    temp_ptr = remove_timer_active_q ( timer_ptr );
    if ( temp_ptr != NULL_PTR )
        add_timer_active_q ( timer_ptr, timer_ptr->type,
                               timer_ptr->init_time_cnt,
                               timer_ptr->timer_handler,
                               timer_ptr->timer_handler_data );

    if ( int_status )
        enable ();

    return temp_ptr;
} /* end of reset_timer */

```

Appendix H — *UNOS-V2.0 LISTING*

```
/*-----*/

/*
=====
|
| stop_timer
|
| This function takes a timer off the active time queue and places it onto
| the inactive timer queue. The status of the timer is changed to inactive.
|
| Parameters : - pointer to the timer of interest
|
| Entry via : - multiple places
|
=====
*/

timer_struct* stop_timer ( timer_struct* timer_ptr )
{
    char int_status;
    timer_struct* temp_ptr;

    int_status = return_interrupt_status ();
    disable ();
    temp_ptr = remove_timer_active_q ( timer_ptr );
    if ( temp_ptr != NULL_PTR )
        add_timer_inactive_q ( timer_ptr );
    if ( int_status )
        enable ();

    return temp_ptr;
} /* end of stop_timer */

/*-----*/

/*
=====
|
| create_timer
|
| The purpose of this function is to allocate storage for a timer structure
| on the heap. If the allocate is successful then a pointer to the
| structure is returned, else a NULL_PTR pointer is returned.
|
| If the timer is created successfully then it is added to the inactive
| timer queue.
|
| Parameters : - none
|
| Entry via : - UNMAIN.C or user tasks.
|
=====
*/

timer_struct* create_timer ( void )
{
```

```

timer_struct* timer;
char int_status;

int_status = return_interrupt_status ( );
disable ( );
timer = ( timer_struct * )umalloc ( sizeof ( timer_struct ) );
if ( timer != NULL_PTR )
    add_timer_inactive_q ( timer );
if ( int_status )
    enable ( );
return timer;
} /* end of create_timer */

/*-----*/

/*
=====
|
| timed_wait
|
| The purpose of this routine is to implement a timed wait. This is
| different from the normal wait in that if the task becomes blocked it will
| only be blocked for a maximum time as determined by the wait time. The
| task can obviously become unblocked earlier than this due to a signal
| on the semaphore from another task. The means by which the wait has been
| terminated can be determined by the return code received. If the return
| code is zero then the wait has not timed out and has been terminated by
| a signal. However if the return code is not zero then some problem has
| occurred. The nature of the problem can be determined by the number of the
| return code. If the number is 1 then the terminate has occurred due to
| a timeout on the wait. If the number is 2 then the wait has been
| terminated due to the absence of a timer being available to implement the
| timed wait. It is up to the user routine to do something about these
| problems.
|
| Parameters : - semaphore number to wait on
|              - time to wait in clock ticks.
|
| Entry via  : - multiple places in the user code.
|
=====
*/

int timed_wait ( SEMAPHORE sema_num, unsigned long timeout_value )
{
    timer_struct* timer_ptr;
    task_control_block* tcb_ptr;
    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );
    tcb_ptr = ( task_control_block* )central_table [ CURRENT_TASK ];

    /* the first step is to get a timer from the inactive time queue and
    start it. It is assumed that the timer has already been created.
    */
    timer_ptr = start_timer ( SINGLE_SHOT, timeout_value, timed_wait_handler,
                             ( void* )tcb_ptr );
    if ( timer_ptr == NULL_PTR )
    {
        if ( int_status )

```

Appendix H — *UNOS-V2.0 LISTING*

```
        enable ( );
    return 2;    /* no timer available */
} /* if */

/* set up the components in the tcb with the correct data */
tcb_ptr->timer_ptr = timer_ptr;
tcb_ptr->timeout_flag = FALSE;

/* now carry out the actual wait operation by doing a normal wait */
wait ( sema_num );

/* have returned from the wait so now check to see what caused the
return. This information is contained in the timeout_flag contained
in the tcb
*/
tcb_ptr->timer_ptr = NULL_PTR;

if ( int_status )
    enable ( );
if ( tcb_ptr->timeout_flag )
    return 1;    /* semaphore timed out */
else
    return 0;    /* return due to a signal */
} /* end of timed_wait */

/*-----*/

/*
=====
|
| lock
|
| This routine is called to lock multiple access to a shared resource - i.e.
| this function implements critical section protection.
|
| If the calling task calls this function it is either going to get the
| requested resource, or access to the resource has already been locked by
| another task. Let us consider each of these scenarios"
|
| Resourced can be obtained - i.e. resource UNLOCKED:
|
| In this case the calling task sets the lock_state component of the locks
| structure to LOCKED, and sets its own waiting_for field to NULL_PTR to indicate
| that it is not waiting for a lock, and finally sets the holder component of
| the lock structure to itself.
|
| Resourced is locked - i.e. LOCKED:
|
| In this case the task has to work out the ultimate holder of the lock and then
| make this task run. Clearly this task has to have an equal or lower priority
| than the current task (since it would be running otherwise). This is achieved
| by entering a loop to look through the tree of tasks transitively blocked
| until it finds a task that is not waiting for a lock, or that is competing
| for a lock that has become unlocked. A task switch is then made to this task.
|
| NB: This routine should not be used in interrupt routines.
|
| Parameters:    - pointer to lock requested
|
| Entry:        - anywhere
|
| Returns:      - nothing
|
=====
*/
```

```

|
=====
*/

void lock (LOCK l)
{

    char int_status;
    task_control_block* cur_tcb_ptr;
    kcb_structure* local_kcb_ptr;

    int_status = return_interrupt_status ( );
    disable ( );
    cur_tcb_ptr = (task_control_block*)central_table[CURRENT_TASK];

    if (l->locked == LOCKED)
    {
        /* Some other task has locked the critical section so we have to search
        for the ultimate blocker of the current task. This is achieved by
        putting the current task on the top of its priority queue and then
        calling the scheduler. The scheduler then carries out all the
        necessary scheduling functions, including the search through the
        transitive blocking tree. A kernel entry is carried out for the
        above operations.
        */
        local_kcb_ptr = (kcb_structure*)umalloc(sizeof(kcb_structure));
        local_kcb_ptr->entry_type = LOCK_ENTRY;
        queue_kcb(local_kcb_ptr);
        cur_tcb_ptr->waiting_for = 1;
        geninterrupt ( kernel_entry );

    } /* if */

    cur_tcb_ptr->waiting_for = NULL_PTR;
    l->locked = LOCKED;
    l->holder = (task_control_block*)central_table[CURRENT_TASK];

    if (int_status)
    {
        enable ();
    } /* if */
} /* end of lock */


/*-----*/


/*
=====
|
| unlock
|
| This function is called when a holding function wishes to unlock a critical
| section. This operation changes the "blocked by" status of all tasks blocked
| by the current task. However, the lock structure does not keep track of
| all the tasks that it is blocking. Therefore the waiting_for component of
| the blocked tcb's cannot be updated. Instead this is sorted out in the
| scheduler routine as it checks both the waiting_for component of the tcb
| structure and the locked component of the lock structure.
|
| This routine checks to see whether the priority of the current task is less
| than the current priority. If so then it has effectively inherited a higher
| priority and the scheduler should be called to enable a higher priority
| task or the next task in a transitive blocking tree to run.

```

Appendix H — *UNOS-V2.0 LISTING*

```

NB: Do not use in interrupt routines.

Parameters:   - pointer to the lock to be released.

Entry via:    - anywhere

Returns:      - nothing
=====
*/

void unlock (LOCK l)
{
    char int_status;
    kcb_structure* local_kcb_ptr;

    int_status = return_interrupt_status ( );
    disable ( );

    l->locked = UNLOCKED;
    l->holder = NULL_PTR;

    if (((task_control_block*)central_table[CURRENT_TASK])->priority >
        current_priority)
    {
        /* Remember that priority 1 is the highest priority so the above
        statement means that we enter here if the current task has a priority
        that is less than the current priority (in a priority sense and not
        a numerical sense)
        */
        local_kcb_ptr =(kcb_structure*) umalloc(sizeof(kcb_structure));
        local_kcb_ptr->entry_type = UNLOCK_ENTRY;
        queue_kcb(local_kcb_ptr);
        geninterrupt ( kernel_entry );
    } /* if */

    if (int_status)
    {
        enable ( );
    } /* if */
} /* end of unlock */

/*-----*/

/*
=====
|
| wait
|
| This procedure implements the wait synchronization primitive. The value
| passed in is the index into the semaphore array. The routine checks to see if
| the semaphore value has reached zero. If so then the kernel is entered proper
| to put the task on the appropriate semaphore queue and a new task is
| scheduled.
|
| If the semaphore value has not reached zero then the value is simply
| decremented and the routine is exited. This serves to keep the overhead of
| entering the kernel proper to a minimum.
|
| Note that the routine preserves the interrupt status upon entry.
|
NB: Don't use within an interrupt routine.

```



```

|
| Parameters : - semaphore number
| Entry via  : - multiple places
|
|=====
*/

void wait ( SEMAPHORE semaphore_num )
{

    char int_status;
    task_control_block* cur_tcb_ptr;
    timer_struct *timer_ptr;
    kcb_structure* local_kcb_ptr;

    int_status = return_interrupt_status ( );
    disable ( );

    if ( semaphore [ semaphore_num ]->semaphore_value > 1 )
    {
        /* semaphore value not zero or one so decrement. */
        semaphore [ semaphore_num ]->semaphore_value--;

        /* now make sure that the routine is not being called from the
        timed_wait routine. If it has been then the timer for the task
        has to be removed from the active timer queue and placed back in the
        inactive timer queue. The tcb_ptr must be reset for the situation
        where the task is no longer claiming a timer.
        */
        cur_tcb_ptr = ( task_control_block* )central_table [ CURRENT_TASK ];
        if ( cur_tcb_ptr->timer_ptr != NULL_PTR )
        {
            /* task claiming a timer */
            timer_ptr = remove_timer_active_q ( cur_tcb_ptr->timer_ptr );
            add_timer_inactive_q ( timer_ptr );
            /* indicate that timer has not timed out */
            cur_tcb_ptr->timeout_flag = FALSE;
        } /* if */
    } /* if */
    else
    {
        if ( semaphore [ semaphore_num ]->semaphore_value == 1 )
        {
            cur_tcb_ptr = ( task_control_block* )central_table [ CURRENT_TASK ];

            semaphore [ semaphore_num ]->semaphore_value--;
            /* now make sure that the routine is not being called from the
            timed_wait routine. If it has been then the timer for the task
            has to be removed from the active timer queue and placed back in
            the inactive timer queue. The tcb_ptr must be reset for the
            situation where the task is no longer claiming a timer.
            */
            if ( cur_tcb_ptr->timer_ptr != NULL_PTR )
            {
                /* task claiming a timer */
                timer_ptr = remove_timer_active_q ( cur_tcb_ptr->timer_ptr );
                add_timer_inactive_q ( timer_ptr );
                /* indicate that timer has not timed out */
                cur_tcb_ptr->timeout_flag = FALSE;
            } /* if */
        } /* if */
        else
        {
            /* semaphore value is equal to zero so enter the kernel to put
            the task to sleep and reschedule another task
            */

            local_kcb_ptr = (kcb_structure*)umalloc(sizeof(kcb_structure));

            /* Set up the entry type. */

```

Appendix H — *UNOS-V2.0 LISTING*

```
        local_kcb_ptr->entry_type = WAIT_ENTRY;

        /* Index into semaphore table. */
        local_kcb_ptr->semaphore_index = semaphore_num;

        queue_kcb(local_kcb_ptr);

        /* Process number is irrelevant in this case. */
        geninterrupt ( kernel_entry );
    } /* else */
} /* else */

if ( int_status )
    enable ( );
} /* end of wait */

/*-----*/

/*
=====
|
| np_signal
|
| Non-preemptive signal. This routine was added to allow a signal to
| occur without the kernel proper being called. As a result no pre-emption
| of the current running task or interrupt routine occurs. However any
| tasks blocked on the semaphore will be moved from the blocked queue
| to the appropriate ready queue. If no tasks are blocked the semaphore
| value is incremented by the appropriate value.
|
| The preemption_allowed flag is set to false, and this is checked inside
| the signal routine. If false then the kernel is not called.
|
| The primary use of this routine is to allow a signal operation to be carried
| out in the middle of an interrupt routine without any possibility of a
| task switch occurring at this time. Under normal use the interrupts
| would be disabled when this routine is called.
|
| ADDED IN RESPONSE TO A SPECIAL REQUIREMENT FOR THE TRUCK TRACKING PROJECT
| DATE: 22/9/94
| PROGRAMMER: BOB BETZ
|
| CAN BE USED FROM AN INTERRUPT ROUTINE
|
| Parameters      -   semaphore number
| Entry via       -   multiple places
|
=====
*/

void np_signal ( SEMAPHORE semaphore_number)
{
    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );

    preemption_allowed = FALSE;
    usignal ( semaphore_number );
}
```

```
preemption_allowed = TRUE;

if ( int_status )
    enable ( );
} /* end of np_signal */

/*-----*/

/*
=====
|
| usignal
|
| This procedure implements the signal ( or send ) synchronization primitive.
| If no task is waiting or has claimed the semaphore then regardless of the
| value of the semaphore its value is simply incremented by the value in
| the semaphore_multiplier.
|
| Only when necessary is the kernel proper entered - i.e., when a task
| switch is required. Note the use of the possible_preemption flag which is
| used to delay a preemptive kernel call in the case of the signal routine
| being entered from the dec_timers routine.
|
| Note that the state of the interrupts is preserved by this routine.
|
| The name of this routine is usignal (Unos signal) so that it will
| not conflict with the signal name used in standard C.
|
| CAN BE USED FROM AN INTERRUPT ROUTINE
|
| Parameters : - semaphore number
| Entry via  : - multiple places
|
=====
*/

void usignal ( SEMAPHORE semaphore_num )
{
    char int_status;
    kcb_structure* local_kcb_ptr;

    int_status = return_interrupt_status ( );
    disable ( );

    possible_preemption = FALSE;

    /* Queue the signal operation for the kernel queue.

    This is achieved by allocating a structure for a kcb from dynamic memory,
    initialising it appropriately and then placing it on the kernel queue,
    which is a linked list of these structures.
    */

    local_kcb_ptr = (kcb_structure*)umalloc(sizeof(kcb_structure));

    /*
    if (local_kcb_ptr == NULL_PTR)
    {
        // an error condition has occurred - don't know what to do
        //about this
    }
    */
}
```

Appendix H — UNOS-V2.0 LISTING

```
    }
    */

    local_kcb_ptr->entry_type = USIGNAL_ENTRY;
    local_kcb_ptr->semaphore_index = semaphore_num;
    local_kcb_ptr->preemption_status = preemption_allowed;

    /* if the entry was from the dec_timers routine then set the
    possible_preemption flag so that a preemptive schedule will be carried out
    for this invocation of the tick routine. Note that the kernel may not
    be entered on every invocation of the tick routine depending on
    the numbers of ticks before kernel entry is set.
    */
    local_kcb_ptr->dectime_signal_flag = sig_entry_from_dec_timers;

    /* Now place the allocated kcb structure onto the kcb queue.
    */

    queue_kcb (local_kcb_ptr);

    /* Now decide whether the kernel should be entered or not. If the
    kernel_executing flag is set then the kernel should not be entered. The
    queued operation will be carried out in due course by the kernel.
    */

    if (!kernel_executing)
    {
        /* The kernel is not executing so enter the kernel to carry out the
        signal operations.
        */
        geninterrupt(kernel_entry);
    }/*if*/

    if ( int_status )
        enable ( );
} /* end of usignal */


/*-----*/

/*
=====
|
| rtn_current_task_num
|
| This function can be called from any task and it returns the task number of the
| currently executing task.
|
| Parameters : - none
|
| Entry via : - any currently executing task
|
=====
*/

unsigned int rtn_current_task_num ( void )
{
    unsigned int cur_tsk_num;
    task_control_block* cur_tcb_ptr;
    char int_status;
```

```

    int_status = return_interrupt_status ( );
    disable ( );
    cur_tcb_ptr = (task_control_block*)central_table [ CURRENT_TASK ];
    cur_tsk_num = cur_tcb_ptr->task_num;
    if ( int_status )
        enable ( );

    return cur_tsk_num;
} /* end of rtn_current_task_num */


/*-----*/

/*
=====
|
| rtn_current_task_name_ptr
|
| This function can be called from any task and it returns the task name pointer
| of the currently executing task.
|
| Parameters : - none
|
| Entry via : - any currently executing task
|
=====
*/

char *rtn_current_task_name_ptr ( void )
{
    char *cur_tsk_name_ptr;
    task_control_block* cur_tcb_ptr;
    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );
    cur_tcb_ptr = (task_control_block*)central_table [ CURRENT_TASK ];
    cur_tsk_name_ptr = cur_tcb_ptr->task_name_ptr;
    if ( int_status )
        enable ( );

    return cur_tsk_name_ptr;
} /* end of rtn_current_task_name_ptr */


/*-----*/

/*
=====
|
| reschedule
|
=====
*/

```

Appendix H — *UNOS-V2.0 LISTING*

```
| This routine sets up the kcb for an entry to the kernel which will cause a
| software reschedule to another task. This software reschedule is exactly
| the same as if a time slice induced reschedule had occurred except for one
| difference. The task which requested the reschedule will not run after
| leaving the kernel even if it is still the highest priority task in the
| system. If this is not the desired behaviour then see preemptive_schedule
| below.
|
| Note that the interrupt status of the calling task is preserved.
|
| Parameters : - none
| Entry via  : - multiple places
|
|=====
|*/

void reschedule ( void )
{

    char int_status;
    kcb_structure* local_kcb_ptr;

    int_status = return_interrupt_status ( );
    disable ( );
    local_kcb_ptr = (kcb_structure*)umalloc(sizeof(kcb_structure));
    local_kcb_ptr->entry_type = RESCHEDULE_ENTRY;
    queue_kcb(local_kcb_ptr);

    geninterrupt ( kernel_entry );
    if ( int_status )
        enable ( );

} /* end of reschedule */

/*-----*/

/*
|=====
| preemptive_schedule
|
| This routine is entered if the user wishes to carry out a preemptive
| schedule. It sets up the kcb for this type of entry and then enters
| the kernel.
|
| A preemptive_schedule differs from a reschedule in that it may or may
| not cause a reschedule, whereas a reschedule always runs the scheduler
| and will never return with the same task running even if it is the only
| task in the system which is runnable ( except for the nulltask ). A
| preemptive reschedule only occurs if there is a higher priority task
| than the current task running.
|
| Note that the interrupt status of the calling task is preserved allowing
| this routine to be called from within interrupt routines.
|
| Also note that the routine detects if it is being called from an interrupt
| routine that has been invoked whilst the kernel was executing. If this is the
| case then the call to the preemptive schedule is queued so be run when the
| kernel is finished executing.
|
| CAN BE USED FROM AN INTERRUPT ROUTINE
|
```

```
| Parameters : - none
| Entry via  : - multiple places
|
=====
*/

void preemptive_schedule ( void )
{

    char int_status;
    kcb_structure* local_kcb_ptr;

    int_status = return_interrupt_status ( );
    disable ( );

    local_kcb_ptr = (kcb_structure*)umalloc(sizeof(kcb_structure));
    local_kcb_ptr->entry_type = PREEMPTIVE_ENTRY;
    queue_kcb(local_kcb_ptr);

    /* If the kernel is not executing then enter the kernel
    */
    if (!kernel_executing)
    {
        geninterrupt(kernel_entry);
    }/*if*/

    if ( int_status )
        enable ( );
} /* end of preemptive_schedule */


/*-----*/


/*
=====
|
| stop_time_slice
|
| This routine disables the time slice entry. This is achieved by setting a
| flag so that the operating system proper is not entered after the time slice
| number of clock ticks. The clock tick routine still runs.
|
| Parameters : - none
|
| Entry via : - multiple places
|
=====
*/

void stop_time_slice ( void )
{

    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );
    time_slice_flag = FALSE;
    if ( int_status )
        enable ( );
} /* end of stop_time_slice */
```

Appendix H — *UNOS-V2.0 LISTING*

```
/*-----*/

/*
=====
|
| start_time_slice
|
| This routine enables the time slice interrupt. This is done by setting
| the time_slice_flag to true. This flag is checked in the tick routine to
| see whether a time slice shall be carried out after the enter_kernel_value
| is reached.
|
| Parameters : - none
|
| Entry via : - multiple places
|
=====
*/

void start_time_slice ( void )
{
    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );
    time_slice_flag = TRUE;
    if ( int_status )
        enable ( );
} /* end of start_time_slice */

/*-----*/

/*
=====
|
| tick
|
| The tick routine provides the basic timing for UNOS. This routine is
| entered upon an interrupt from the main system timer. It should be noted
| that the tick routine entry can be at a different frequency to the kernel
| time slice entry. This is done so that the time resolution of the system
| timers is independent of the time slice frequency. For example, it is
| feasible to have a time resolution of say 1 msec for sleeping tasks and a
| time slice of 20 msec. There the user has independent control of timing
| accuracy and the operating system overhead.
|
| Upon the tick interrupt the tick routine is entered. The first function
| carried out by the routine is to send an end of interrupt to the 8259
| interrupt controller. This is done here so that the exit path from the
| routine becomes irrelevant. The next task is to increment the UNOS real
| time clock variable ( rt_clock ) which is implemented as a unsigned double
| word binary number. This variable is used by the kernel for task time bin
| testing and can be used by user tasks to determine elapsed time.
|
=====
*/
```



```
| The dec_timers routine is called. This routine decrements the time
| values in the timer data structures. If a timer goes to zero then the
| timer handler ( whose address is stored in the timer structure ) is
| executed. If the timer type is repetitive then it is also immediately
| put back onto the timer queue. The timer handler routine would
| generally execute a signal to cause make a task become runnable.
| If the timer handler is urgent then it can be handled directly in the
| interrupt routine.
|
| After the dec_timers routine the TICK_ENTRY request is put into the kernels
| kcb queue. The kernel is then entered if the kernel was not executing at the
| time of the tick interrupt.
|
| From the above discussion it can be deduced that a timer handler which
| does a signal could cause a pre-emptive task switch. This would be the
| situation if measures were not taken to prevent this from occurring. Two
| flags are used to prevent a pre-emptive task switch from occurring from
| the timer handler level - the sig_entry_from_dec_timers flag indicates
| to the signal routines that they are being entered from a timer handler.
| The kernel is entered from the usignal routine, but special processing
| takes place which does not result in a preemptive task switch at this stage.
| If the kernel is running at the time this occurs then the kcb queuing will
| take care of the order of these processes.
|
| WARNING :
| Although the user kernel interface routines can be called from interrupt
| routines with the interrupt status preserved, it is not sensible to call
| some of them. For example, if a 'wait' is executed from within an interrupt
| routine then the task which was being executed at the time that the
| interrupt occurred would become blocked, even though it most probably has
| nothing to do with the wait semaphore causing the blockage. Furthermore, if
| the routine executing at the time of the interrupt is the routine which
| 'signals' on the semaphore that the interrupt routine waits on then a
| deadlock situation occurs.
|
| THE TICK INTERRUPT CAN OCCUR WHILST THE KERNEL IS EXECUTING. IT IS DESIGNED
| TO HANDLE THIS SITUATION CORRECTLY.
|
| Parameters : - none
|
| Entry via  : - hardware timer interrupt
|
|=====
|*/
void interrupt tick ( void )
{
    kcb_structure* local_kcb_ptr;

    /* end of interrupt for the 8259 interrupt controller */
    outportb ( I8259_INT_CTRL_ADD, I8259_EOI );

    /* increment the real time clock counter */
    rt_clock++;

    /* decrement the counts for system timers and execute handler if a
    timeout has occurred
    */
    dec_timers ( );

    /* now create a kcb_ptr to enter the kernel to carry out the tick handling
    operation.

    Note that this approach allows the tick interrupt to occur whilst we are
    already inside the kernel. If the kernel is executing at the time of the
    tick interrupt then the tick operation is scheduled on the kcb queue
    BUT THE KERNEL IS NOT ENTERED.

    It is not necessary to enter the kernel because we are already in the
```

Appendix H — *UNOS-V2.0 LISTING*

```
kernel. Therefore the queued tick operation will be carried out in due
course as the kernel kcb queue is emptied.
*/

local_kcb_ptr = (kcb_structure*)umalloc(sizeof(kcb_structure));

local_kcb_ptr->entry_type = TICK_ENTRY;

/* Now queue the kernel call request */

queue_kcb(local_kcb_ptr);

/* Now check to see if we enter the kernel now or if we are already in
the kernel (i.e. kernel_executing = TRUE).

If the kernel is executing then the queued kernel request will be dealt with
in due course by the kernel itself.
*/

if (!kernel_executing)
{
    geninterrupt(kernel_entry);
} /*if*/
} /* ticks */

/*-----*/

/*
=====
|
| rtn_task_priority
|
| This function returns the priority of the task whose name is passed into
| the routine. If the task name is illegal then the routine returns 0xffff
| as the task priority.
|
| Parameters : - pointer to the task name
|
| Entry via  : - any user task
|
=====
*/

unsigned int rtn_task_priority ( char *task_name_ptr )
{
    unsigned int task_num, temp_task_priority;
    char int_status;

    /* firstly corrolate the task name with the task number */
    task_num = mail_exchange ( task_name_ptr );

    /* now check to see if the task name was a legal one */
    if ( task_num >= num_of_tasks )
    {
        /* task name was illegal so return an illegal priority */
        return 0xffff;
    } /* if */

    int_status = return_interrupt_status ( );
    disable ( );
```

```

/* now get the task priority */
temp_task_priority = tcb [ task_num ]->priority;

if ( int_status )
{
    enable ( );
} /* if */

return temp_task_priority;
} /* rtn_task_priority */


/*-----*/


/*
=====
|
| change_task_priority
|
| This procedure allows a task to change the priority of any task (including
| itself).
|
| It should be noted that if this routine carries out a kernel entry, then
| the scheduler will be called. Therefore the calling task may not be running
| after exit from the kernel.
|
| If the change in the task priority is successful then a TRUE is returned,
| else a FALSE is returned.
|
| THIS FUNCTION CAN BE CALLED FROM AN INTERRUPT ROUTINE.
|
| Parameters : - task number, new priority
|
| Entry via  : - multiple places
|
=====
*/

int change_task_priority ( char *task_name_ptr,
                          unsigned char new_priority )
{
    char int_status;
    unsigned char task_num;
    kcb_structure* local_kcb_ptr;

    /* now make the connection between the task name and the task number. */
    task_num = mail_exchange ( task_name_ptr );

    /* firstly check to see if the task number and the priority are legal
    values.
    */
    if ( ( task_num >= num_of_tasks ) || ( new_priority < 1 ) ||
        ( new_priority > num_of_priorities ) )
    {
        /* normally call an error handler here */
        return FALSE;
    } /* if */

    int_status = return_interrupt_status ( );
    disable ( );

```

Appendix H — UNOS-V2.0 LISTING

```
local_kcb_ptr = (kcb_structure*)umalloc(sizeof(kcb_structure));
local_kcb_ptr->entry_type = CHANGE_TASK_PRIORITY;
local_kcb_ptr->new_priority = new_priority;
local_kcb_ptr->task_number = task_num;
queue_kcb(local_kcb_ptr);
if(!kernel_executing)
{
    geninterrupt(kernel_entry);
}/*if*/

if ( int_status )
    enable ( );

return TRUE;
} /* end of change_task_priority */

/*-----*/

/*
=====
|
| alloc_tcb_table
|
| This function returns a pointer to an array of pointers to the tcb's. The
| array allocated is of a size to accommodate the maximum number of tasks
| allowed in the system. The values of the pointers are all initialised to
| have NULL_PTR values.
|
| Parameters : - maximum number of tasks
|
| Entry via  : - setup_os_data_structures
|
=====
*/

static task_control_block **alloc_tcb_table ( unsigned int max_num_of_tsks )
{
    int i;

    tcb = ( task_control_block ** )ucalloc ( max_num_of_tsks,
                                              sizeof ( task_control_block * ) );

    if ( tcb != NULL_PTR )
    {
        for ( i = 0; i < max_num_of_tsks; i++ )
        {
            tcb [ i ] = NULL_PTR;
        } /* for */
    } /* if */

    return tcb;
} /* end of alloc_tcb_table */

/*-----*/
```

```

/*
=====
|
| alloc_semaphore_array
|
| This function allocates an array of pointers to the semaphore array. This
| array of pointers is used to store the addresses of the semaphore
| structures themselves. Initially the array is initialised with NULL_PTR
| values in all the locations.
|
| The size of this array determines the maximum number of semaphores that can
| be allocated by user tasks. The advantage of having a table to store the
| location of all semaphores is that a operating system performance monitor
| can easily locate all the semaphores.
|
| The function returns a pointer to the semaphore array. If the pointer
| returned is a NULL_PTR then the required memory could not be allocated.
|
| Parameters : - number of semaphores to allocate.
|
| Entry via   : - setup_os_data_structures
|
=====
*/

static semaphore_struc **alloc_semaphore_array ( unsigned int max_num_semaph )
{
    unsigned int i;

    num_semaphores = max_num_semaph;
    semaphore = ( semaphore_struc ** )ucalloc ( max_num_semaph,
                                                sizeof ( semaphore_struc * ) );

    if ( semaphore == NULL_PTR )
    {
        return semaphore;
    } /* if */

    /* now initialise the semaphore pointer array */
    for ( i = 0; i < num_semaphores; i++ )
    {
        semaphore [ i ] = NULL_PTR;
    } /* if */

    return semaphore;
} /* end of alloc_semaphore array */


/*-----*/


/*
=====
|
| create_lock
|
| This routine is called to create a lock primitive. These primitives do not
| have any associated queue, and consequently they are simply created as
| dynamic entities on the stack.
|
=====

```

Appendix H — UNOS-V2.0 LISTING

```

Locks are a form of critical section primitives that are required for
a complete priority inheritance system. They should be used for all
critical section protection in place of the normal semaphores, which should
only be used for synchronization and resource counting uses.

This routine preserves the interrupt status of the caller.

Parameters:   - none

Entry:        - anywhere

Returns:      - pointer to the lock structure.
=====
*/

LOCK create_lock ( void )
{
    char int_status;
    LOCK temp_lock;

    int_status = return_interrupt_status ( );
    disable ( );

    /* Now allocate the storage for the lock */
    temp_lock = (LOCK)ucalloc (1, sizeof (lock_type));
    temp_lock->locked = UNLOCKED;
    temp_lock->holder = NULL_PTR;

    if (int_status)
    {
        enable ();
    } /* if */

    return (temp_lock);
} /* end of create_lock */

/*-----*/

/*
=====
|
| destroy_lock
|
| This function is called if the user wishes to deallocate the storage for a
| lock.
|
| Parameters:   - pointer to the lock
|
| Entry via:    - anywhere
|
| Returns:      - a NULL_PTR to store in the lock pointer
|
|=====
*/

LOCK destroy_lock (LOCK l)
{
    ufree((char*)l);

```

```

        return (NULL_PTR);
    } /* end of destroy_lock */

```

```

/*-----*/

```

```

/*
=====
|
| init_semaphore
|
| This function allows any semaphore to be custom initialised. The values that
| it is to be initialised to are passed into the function.
|
| Parameters : - number of semaphore to be initialised
|               - initial semaphore value
|               - semaphore multiplier value for the semaphore.
|
=====
*/

```

```

void init_semaphore ( SEMAPHORE sema_num, unsigned int sema_value,
                     unsigned int multiplier_value )
{
    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );
    semaphore [ sema_num ]->semaphore_value = sema_value;
    semaphore [ sema_num ]->semaphore_multiplier = multiplier_value;
    if (int_status)
    {
        enable ( );
    }
} /* end of init_semaphore */

```

```

/*-----*/

```

```

/*
=====
|
| create_semaphore
|
| This function as the name implies creates a semaphore for the caller. What
| in actual fact that is returned is an index number into the semaphore
| pointer array for the new semaphore. A pool of semaphores is created during
| system initialisation, and if this pool is exhausted then an illegal
| semaphore number is returned - namely a 0xffff.
|
| The semaphores whose index is returned have been initialised at the time
| they were created to have an initial value of semaphore_value, and the
| multiplier is set to 1. If these values are later found to not be suitable
| then they can be modified in the users task to the appropriate values using

```

Appendix H — *UNOS-V2.0 LISTING*

```
| the init_semaphore function.
|
| Parameters : - value the semaphore is to be set to.
|
| Entry via  : - user tasks requiring semaphores.
|
|=====
*/

SEMAPHORE create_semaphore ( unsigned int semaphore_value )
{
    char int_status;
    unsigned int return_semaphore_num;
    semaphore_struct *sema_ptr;

    int_status = return_interrupt_status ( );
    disable ( );

    /* check to see if the total number of semaphores has not been violated */
    if ( next_semaphore != 0xffff )
    {
        /* allocate the memory for the semaphore structure */
        sema_ptr = ( semaphore_struct * )umalloc (
                                sizeof ( semaphore_struct ) );

        if ( sema_ptr != NULL_PTR )
        {
            return_semaphore_num = next_semaphore;

            /* assign the pointer to the semaphore pointer array */
            semaphore [ next_semaphore ] = sema_ptr;

            /* initialise the semaphore structure */
            sema_ptr->creating_taskname_ptr = tcb [
                rtn_current_task_num ( ) ]->task_name_ptr;
            sema_ptr->semaphore_value = semaphore_value;
            sema_ptr->semaphore_multiplier = 1;
            sema_ptr->semaph_bgn_q_index = bgm_semaphore_central_table;
            sema_ptr->semaph_queue_handler = semaphore_queue_handler;
            bgm_semaphore_central_table += 2;

            next_semaphore++;
            if ( next_semaphore >= num_semaphores )
            {
                /* have run out of semaphores */
                next_semaphore = 0xffff;
            } /* if */
        } /* if */
    }
    else
    {
        /* cannot allocate the memory for the semaphore */
        return_semaphore_num = next_semaphore = 0xffff;
    } /* else */
} /* if */
else
{
    /* have run out of semaphores */
    return_semaphore_num = next_semaphore;
} /* else */

if (int_status)
{
    enable ( );
} /* if */

return return_semaphore_num;
} /* end of create_semaphore */
```



```

/*-----*/

/*
=====
|
| alloc_central_table
|
| This function allocates memory from the heap for the central table. If the
| allocation is successful then the function passes back a pointer to the
| table, else it passes back a NULL_PTR. If the central table can be allocated then
| it is initialised to zero.
|
| In addition to the central table storage being allocated the indexes into
| the central table are also defined.
|
| Parameters : - number of tasks
|               - number of semaphores
|
| Entry via  : - setup_os_data_structures
|
=====
*/

static void** alloc_central_table ( unsigned char number_priorities,
                                   unsigned int num_semaph )
{
    int i;

    num_of_priorities = number_priorities;

    /* calculate the beginning point for the semaphore central table entries.
    Every entry in the central table consists of two pointers ( except for the
    current task pointer ). The first pointer points to the beginning of a
    queue and the second to the end of a queue. The pointers appear in the
    central table from index zero as :
        the pointer to the current task
        sets of pointers to the various priority queues
        sets of pointers to the time queue for task control blocks
        sets of pointers to the active timer queue
        sets of pointers to the inactive timer queue

    In the following calculation for the index to the beginning of the
    active time queue based on adding 1 for the current task pointer
    and then 2 times the number of semaphores, since these occur in
    pairs.

    */

    bgn_active_time_q = 1 + num_of_priorities * 2;
    end_active_time_q = bgn_active_time_q + 1;
    bgn_inactive_time_q = 3 + num_of_priorities * 2;
    end_inactive_time_q = bgn_inactive_time_q + 1;
    bgn_semaphore_central_table = 5 + num_of_priorities * 2;

    ( void** ) central_table = ( void ** )ucalloc (
        bgn_semaphore_central_table + 2 * num_semaph,
        sizeof ( void * ) );
    if ( central_table == NULL_PTR )
        return ( void ** ) central_table;

    /* initialise the central table to NULL_PTR */
    for ( i = 0; i < ( bgn_semaphore_central_table + num_semaph * 2 ); i++ )

```

Appendix H — *UNOS-V2.0 LISTING*

```
        central_table [ i ] = NULL_PTR;

        return ( void** ) central_table;
} /* end of alloc_central_table */


/*-----*/

/*
=====
|
| return_start_flag
|
| This function returns the status of the start_flag. If this flag is
| clear then the operating system has not started. If the flag is set
| then the operating system has started. The function is used mainly to
| aid in the initialisation of reentrant tasks. When the task is entered
| for the first time and the flag is not set then it is known that this
| entry is for initialisation purposes. Therefore local variables can be
| initialised (since the normal runtime stack is being used). Upon the
| second entry into the routine the flag is again checked. This time it
| will be set indicating that entry has occurred as a consequence of a normal
| task switch. In this case the local variables are not initialised.
|
| Parameters:  -   none
|
| Returns:    -   nothing.
|
=====
*/

char return_start_flag (void)
{
        return (start_flag);
} /* end of return_start_flag */


/*-----*/

/*
=====
|
| create_task
|
| This function creates and initialises the tcb and creates a ready to run
| stack for the task name passed in. The stack is dynamically allocated on the
| UNOS heap. After the task is created the flag start_flag determines whether
| the task is entered or not. This flag is set by the start processes routine.
|
| The routine also provides a mechanism to allow the global variables for
| a task to be initialised before any tasks in the system start to run. This
| mechanism is ideal to set up any semaphores that are to be used by a task.
|
| When the task is created the routine also creates the mailbox associated
```

```

| with the task. Since the size of the various mailbox components are passed
| into the routine then the mail box for individual tasks can be
| customized. Implementing the mail boxes in this section of code also
| allows the name of the task to be placed into the mail exchange so that
| the address for mail box messages is effectively the task name as far as
| other user tasks are concerned.
|
| Parameters : - pointer to the name of the task
|               - priority of task created
|               - tick delta for the task
|               - status of the task - runnable, blocked, suspended, active.
|               - queue type which task is to placed on
|               - semaphore number if task to be placed on a semaphore queue.
|               - size of the task stacks
|               - mail box message queue size
|               - maximum size of mail box message
|               - address of the task initialisation function
|               - address of the task function
|               - pointer to an unknown variable that is used to pass
|                 initialisation data to a task. The primary use is for
|                 setting up local variables in reentrant tasks.
|
| Returns :    - TRUE if task creation successful
|               - FALSE if task creation unsuccessful
|
| Entry via   : - main module
|
|=====
|*/
|
|BOOL create_task (  char *task_name_ptr, unsigned char priority_of_task,
|                   int task_tick_delta, unsigned char status_of_task,
|                   unsigned char q_type, unsigned int semaphore_num,
|                   unsigned int task_stk_size, unsigned int mess_q_size,
|                   unsigned int mess_size, void ( *init_task ) ( void ),
|                   void ( *task ) ( void* ), void* local_var_ptr )
|
|{
|
|    static char huge* stack_location;
|    static void ( *task_ptr ) ( void* );
|    static void* temp_local_var_ptr;
|    static unsigned int stackbase;
|    static unsigned int stackptr;
|    static unsigned int baseptr;
|    unsigned int semal, sema2;
|    mbx* mbx_ptr;
|    taskname_map* taskname_map_ptr;
|    taskname_map* nxt_ptr;
|    taskname_map* old_nxt_ptr;
|    unsigned int hash_table_index;
|    kcb_structure* local_kcb_ptr;
|
|    /* upon entry to this routine the num_of_tasks variable can be used as
|       an index into the tcb table to the position of the next task to be
|       created.
|    */
|
|    /* check to see that we are not attempting to create too many tasks. */
|    if ( num_of_tasks >= max_num_of_tasks )
|    {
|        return FALSE;
|    } /* if */
|
|    if ( init_tcb_and_q_it ( task_name_ptr, priority_of_task, task_tick_delta,
|                           status_of_task, q_type, semaphore_num ) == NULL_PTR )
|    {
|        return FALSE;
|    } /* if */
|
|    /* if the task stack size is zero then we don't want to form a stack

```

Appendix H — UNOS-V2.0 LISTING

```
for the task. This condition only occurs if the task is the null task -
this task uses the stack setup by the compiler. Note that the initialisation
is also bypassed since the null task usually does not use any resources
or variables. Note that a mail box is not created for the null task.
*/
if ( task_stk_size == 0 )
{
    central_table [ CURRENT_TASK ] = ( void *)tcb [ num_of_tasks ];
    num_of_tasks++;
    return TRUE;
} /* if */

/* now create a mail box for the task */
/* firstly get the two semaphores required and customize*/
sema1 = create_semaphore ( mess_q_size );
sema2 = create_semaphore ( 0 );

if ( ( sema1 == 0xffff ) || ( sema2 == 0xffff ) )
{
    /* have run out of semaphores so exit */
    return FALSE;
} /* if */

mbx_ptr = create_mbx ( num_of_tasks, FIFO_TYPE, mess_q_size, mess_size,
                      sema1, sema2 );

if ( mbx_ptr == NULL_PTR )
{
    /* cannot allocate the mail box */
    return FALSE;
} /* if */

/* place the mail box pointer into the mail box table */
mbx_table [ num_of_tasks ] = mbx_ptr;

/* now allocate the memory for the taskname_map structure associated with
this task.
*/
taskname_map_ptr = ( taskname_map * )umalloc
                    ( sizeof ( taskname_map ) );
if ( taskname_map_ptr == NULL_PTR )
{
    return FALSE;
} /* if */

/* set up the taskname_map structure with the appropriate data */
taskname_map_ptr->task_name_ptr = task_name_ptr;
taskname_map_ptr->task_number = num_of_tasks;
taskname_map_ptr->nxt_taskname_map = NULL_PTR;

/* now set up the mail exchange table so that this mail box is associated
with the currently created task. In order to do this the task_name pointer
has to be hashed to index into the hash_table.
*/
hash_table_index = hash_taskname_ptr ( task_name_ptr );

/* now place the taskname_map into the hash table at the index
position.
*/
nxt_ptr = hash_table [ hash_table_index ];

if ( nxt_ptr == NULL_PTR )
{
    /* the index position is empty so place the taskname_map structure
address into the hash table itself.
*/
    hash_table [ hash_table_index ] = taskname_map_ptr;
} /* if */
else
```

```

{
    /* already a taskname_map structure in the has table so have to chain
    the new entry onto the end of the taskname_map structures which hash
    to this location.
    */
    while ( nxt_ptr != NULL_PTR )
    {
        old_nxt_ptr = nxt_ptr;
        nxt_ptr = nxt_ptr->nxt_taskname_map;
    } /* while */

    /* have now found the end of the chained list so store the new
    taskname_map structure into the table.
    */
    old_nxt_ptr->nxt_taskname_map = taskname_map_ptr;
} /* else */

/* now create the stack for this task. Firstly save the stack base and
stack pointer for the working stack.
*/
stackbase = _SS;
stackptr = _SP;
baseptr = _BP;

/* store the task to be executed address in a global variable. This is done
because the variables passed into the routine are stored on the working
stack and referenced by the BP register. When the new stack is set up
a new BP register value will be established. Therefore upon start up
of the tasks control will return to the line after the geninterrupt in
this function with the BP register containing the value for the new stack.
Therefore the task variable used in the call to the task to be started
will be incorrect since the software will be indexing into the new stack
which does not contain the task address. If one decides not to create a
BP register value for the next stack then upon start of a task when
control enters this routine just after the geninterrupt the BP value will
be that of the working stack. Therefore the value of task picked up
for all tasks will be that left on the stack from the last task created -
i.e. the null task. Hence one gets multiple versions of the null task
running.

The approach taken to get around these problems is to store the task
address in a global and then transfer it to the appropriate place on
the new stack so that a new BP register value will find it correctly.
This approach ensures that each new stack carries with it the address
of the task to be executed.
*/
task_ptr = task;
temp_local_var_ptr = local_var_ptr;

central_table [ CURRENT_TASK ] = tcb [ num_of_tasks ];

/* now create the new stack. The stack is created with an offset of 0x1b
so that the task address can be placed on the new stack in the same
relative location to the base pointer as the working stack. Currently
this relative location is bp+0x18. The stack is allocated from the
dynamic memory heap.
*/
if ( (stack_location = umalloc ( task_stk_size ) ) == NULL_PTR )
{
    return FALSE;
} /* if */

/* Now allocate the memory for the kcb */
if ((local_kcb_ptr =
    (kcb_structure*)umalloc(sizeof(kcb_structure))) == NULL_PTR)
{
    return FALSE;
}/*if*/
local_kcb_ptr->entry_type = CREATE_TASK_STK_ENTRY;

```

Appendix H — *UNOS-V2.0 LISTING*

```
queue_kcb(local_kcb_ptr);

/* !!!!!!! NOTE !!!!!!!

THE FOLLOWING STACK OFFSET CALCULATIONS ARE COMPILER DEPENDENT. THE
PRECISE NUMBERS CHANGE DEPENDING ON THE VERSION OF TURBOc COMPILER USED.
*/
_SP = FP_OFF ( stack_location ) + task_stk_size - 1 - 0x21;
_SS = FP_SEG ( stack_location );

/* now create the new task BP register value and store the task address in
the correct relative location.
*/
// This line was added because the next line was not producing code for
// some reason.
asm mov bp, sp
// For some reason the line below was not producing any code.
// _BP = _SP;
_SP = _BP - 0x14;
task = task_ptr; /* place task address on new stack */
local_var_ptr = temp_local_var_ptr;

geninterrupt ( kernel_entry );

/* now check if the task is to be started */
if ( start_flag )
{
    enable ();
    ( *task ) (local_var_ptr);
} /* if */

/* restore the working stack */
_SP = stackptr;
_SS = stackbase;
_BP = baseptr;

/* Now check if there is a task initialisation function that has to be
run.
*/
if (init_task != NULL_PTR)
{
    (init_task)();
} /* if */

/* Task creation successful so increment the number of tasks created
in the system.
*/
num_of_tasks++;

return TRUE;
} /* end of create_task */

/*-----*/

/*
=====
|
| start_tasks
|
| This function starts the processes in the system running. This is achieved
```

```
| by setting the start_flag and starting the time slice interrupts. The
| last function of the routine is to call the null task. Note that the
| routine uses the task setup by the compiler.
|
| Parameters : - address of the null task
|
| Entry via  : - main routine
|
=====
*/

void start_tasks ( void ( *null_task ) ( void * ) )
{
    start_flag = 1;
    ( *null_task ) ( NULL_PTR );
} /* end of start_tasks */

/*-----*/

/*
=====
|
| set_kernel_interrupt_num
|
| Sets the interrupt number for the software interrupt to enter the kernel
|
| Parameters : - kernel interrupt number
|
| Entry via  : - unmain
|
=====
*/

void set_kernel_interrupt_num ( unsigned char kernel_interrupt_num )
{
    kernel_entry = kernel_interrupt_num;
} /* end of set_kernel_interrupt_num */

/*-----*/

/*
=====
|
| chg_task_tick_delta
|
| The purpose of this function is to change the tick delta for the particular
| task whose address is passed into the routine. The tick delta is then
| added to the base tick time slice frequency to decide how many ticks the
| particular task will run for if not preempted. Therefore the tick delta
| mechanism is effectively a secondary priority mechanism, in that tasks of
| the same priority can be allocated different amounts of the processor
|
```

Appendix H — *UNOS-V2.0 LISTING*

```
| time.
|
| If the change has been successfully completed then the routine returns
| a TRUE else it returns a FALSE.
|
| Parameters : - pointer to the tasks name
|               - the new tick delta value
|
| Entry via  : - any user task
|
|=====
|*/

int chg_task_tick_delta ( char *task_name_ptr, int new_tick_delta )
{

    unsigned int task_num, int_status;

    /* now make the connection between the task name and the task number. */
    task_num = mail_exchange ( task_name_ptr );

    /* firstly check to see if the task number is a legal one */
    if ( task_num >= num_of_tasks )
    {
        /* normally call an error handler here */
        return FALSE;
    } /* if */

    int_status = return_interrupt_status ( );
    disable ( );

    /* now set up the number of ticks before this particular task will
    enter the kernel. The number of ticks is set up as an absolute value.
    */
    if ( base_ticks_per_time_slice + new_tick_delta <= 0 )
    {
        tcb [ task_num ]->ticks_before_kernel_entry = 1;
    } /* if */
    else
    {
        tcb [ task_num ]->ticks_before_kernel_entry =
            base_ticks_per_time_slice + new_tick_delta;
    } /* else */

    tcb [ task_num ]->ticks_to_go =
        tcb [ task_num ]->ticks_before_kernel_entry;

    if ( int_status )
    {
        enable ( );
    } /* if */

    return TRUE;
} /* chg_task_tick_delta */

/*-----*/

/*
|=====
|
| chg_base_ticks_per_time_slice
|
|
```



```
| As this functions name implies its purpose is to initialise the base
| time slice frequency used in the operating system. This time slice
| frequency is expressed in terms of the number of clock ticks which
| occur before a time slice entry to the operating system occurs. Note that
| this base time slice frequency is not necessarily the time slice frequency
| of a particular task, since the tick_delta can alter this base frequency
| on an individual task basis.
|
| Parameters : - unsigned int containing the number of ticks
|
| Entry via  : - multiple places
|
|=====
*/

void chg_base_ticks_per_time_slice ( int new_base_ticks_per_time_slice )
{
    int i;
    int tick_delta;
    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );

    /* have to go through the tcb array and alter all the
    ticks_before_kernel_entry values to conform to the new base
    tick frequency. If the altered value becomes less than or equal to
    zero then the value is set to 1.
    The current value of the tick_delta for a particular task is calculated
    before the base tick value is changed to its new value.
    */
    new_base_ticks_per_time_slice = abs ( new_base_ticks_per_time_slice );

    for ( i = 0; i < num_of_tasks; i++ )
    {
        tick_delta = tcb [ i ]->ticks_before_kernel_entry -
                     base_ticks_per_time_slice;
        if ( new_base_ticks_per_time_slice + tick_delta <= 0 )
        {
            tcb [ i ]->ticks_before_kernel_entry = 1;
        } /* if */
        else
        {
            tcb [ i ]->ticks_before_kernel_entry =
                new_base_ticks_per_time_slice + tick_delta;
        } /* else */
    } /* for */

    base_ticks_per_time_slice = ( unsigned int )new_base_ticks_per_time_slice;

    if (int_status)
    {
        enable ( );
    } /* if */
} /* end of chg_base_ticks_per_time_slice */

/*-----*/

/*
=====
```

Appendix H — UNOS-V2.0 LISTING

```

| init_start_flag
|
| This routine initialises the start flag to zero.
|
=====
*/

void init_start_flag ( void )
{
    start_flag = 0;
} /* end of init_start_flag */

/*-----*/

/*
=====
| return_semaphore_value
|
| This function returns the current value of the semaphore whose number is
| passed to the function. The interrupt status of the calling task is
| preserved by the routine.
|
| Function returns the semaphore value. If the semaphore number passed in
| is illegal then the returned value is 0xffff.
|
| Parameters : - semaphore number whose value is to be returned.
|
| Entry via  : - multiple places
|
=====
*/

unsigned int return_semaphore_value ( unsigned int sema_num )
{
    int int_status;
    unsigned int temp = 0xffff;

    int_status = return_interrupt_status ( );
    disable ( );

    /* check if the semaphore number is legal */
    if (sema_num < next_semaphore)
    {
        temp = semaphore [ sema_num ]->semaphore_value;
    } /* if */

    if ( int_status )
        enable ( );

    return ( temp );
} /* end of return_semaphore_value */

/*-----*/

```

```

/*
=====
|
|  setup_os_data_structures
|
|  This function sets up all the major data structures in the UNOS system.
|  All these data structures are created dynamically on the heap. This will
|  help leave the option open to extend UNOS so that tasks can be dynamically
|  created (and destroyed) at run time. Obviously this option would only be
|  of use in a system where tasks can be loaded from disk. Since the current
|  version of UNOS is targeted from ROM based embedded applications these
|  features are currently not implemented.
|
|  The data structures created in this routine are:-
|
|      central table
|      tcbs
|      semaphores
|      mail box table
|      mail box structures
|
|  If there are no problems in the creation of these structures then the
|  routine returns TRUE, else it returns FALSE.
|
|  Parameters : - interrupt number for the kernel entry
|                - number of ticks before time slice entry of kernel
|                - number of priority levels
|                - maximum number of semaphores
|                - number of tasks
|                - pointer to the memory pool
|                - size of the memory pool
|
|  Entry via : - software_initialise in the main module
|
=====
*/

char setup_os_data_structures ( unsigned char kernel_interrupt_num,
                                int clock_ticks_kernel_ent,
                                unsigned int num_of_priorities,
                                unsigned int max_num_semaphores,
                                unsigned int maximum_num_of_tasks,
                                char huge* ptr_to_mem_pool,
                                unsigned long memory_pool_size )
{
    semaphore_struc **sema_ptr;

    mem_pool_size = memory_pool_size;
    mem_pool_ptr = ptr_to_mem_pool;
    max_num_of_tasks = maximum_num_of_tasks;

    /* make sure that the clock ticks to enter kernel value is a
    positive number, and assign it to the global base value. This value
    is then used to set up the values in the tcb.
    */
    base_ticks_per_time_slice = abs ( clock_ticks_kernel_ent );

    set_kernel_interrupt_num ( kernel_interrupt_num );
    init_start_flag ( );
    alloc_central_table ( num_of_priorities, max_num_semaphores );
    tcb = alloc_tcb_table ( max_num_of_tasks );
    sema_ptr = alloc_semaphore_array ( max_num_semaphores );
    mbx_table = alloc_mbx_table ( max_num_of_tasks );
    hash_table = alloc_mail_exch_table ( HASH_TABLE_SIZE );

```

Appendix H — UNOS-V2.0 LISTING

```
        if ( (mbx_table != NULL_PTR) && (tcb != NULL_PTR) && (hash_table != NULL_PTR) &&
                (sema_ptr != NULL_PTR) )
        {
            return TRUE;
        } /* if */
        else
        {
            return FALSE;
        } /* else */
    } /* end of setup_os_data_structures */

/*-----*/

/*****
/*
/*
/*
/*
/*          UNOS_4 MODULE
/*
/*          by
/*
/*          Robert Betz
/*          Department of Electrical Engineering and Computer Science
/*          University of Newcastle
/*          Australia
/*
/*          ( Copyright 1989, 1990 )
/*
/*
/*
/*
*****/

/*
HISTORY

Began writing this module on the 18th January 1990.

*/

/*

BRIEF DESCRIPTION

This module is included into the UNOS operating system. Its function is to
provide mail box communications for the operating system. A full description
of the operation of the mail box facility can be found in the UNOS.ASC (ascii
documentation file for UNOS) or in the printed documentation (which is a
laser printer output of this file).
*/

/*

DATA STRUCTURES

The main data structures used by the mail box system are defined at the top of
the UNOS_1.C module of the operating system.
*/
```

```
/*-----*/
```

```
/*
=====
|
| alloc_mail_exch_table
|
| The purpose of this function is to allocate an array of pointers to the
| the taskname_struc structures. This table forms the centre of the mail
| exchange mechanism used in UNOS. This mechanism allows messages to be
| sent to a task simply by using the array name of the array containing the
| tasks name. This name is then "hashed" to form an index into this table
| which contains pointers to taskname_map structures which contain the
| information to connect the task name to the task number. This task number
| is then used in the lower levels of the mail system to index into the
| mail box array. The function returns a pointer to the array of pointers.
| If the allocation fails then a NULL_PTR pointer is returned. If the
| allocation occurs correctly then the table is initialised with NULL_PTR's
| throughout.
|
| Parameters      :-  size of the hash table
|
| Entry via      :-  setup_os_data_structures
|
=====
*/
```

```
static taskname_map **alloc_mail_exch_table ( unsigned int size_of_table )
{
    taskname_map **mail_exch_ptr;
    int i;

    mail_exch_ptr = ( taskname_map ** )ucalloc ( size_of_table,
                                                sizeof ( taskname_map * ) );

    if ( mail_exch_ptr != NULL_PTR )
    {
        for ( i = 0; i < size_of_table; i++ )
        {
            mail_exch_ptr [ i ] = NULL_PTR;
        } /* for */
    } /* if */
    return mail_exch_ptr;
} /* end of alloc_mail_exch_table */
```

```
/*-----*/
```

```
/*
=====
|
| hash_taskname_ptr
|
| This function accepts a pointer to an array of characters and hashes the
| pointer address to an index within the range of zero to hash_table_size-1.
| The hashing is carried out by converting the 8086 segmented address into
| a linear address (this is obviously implementation specific) and then
```

Appendix H — *UNOS-V2.0 LISTING*

```
| carrying out a mod hash function on this address, with the size of the
| table being a prime number.
|
| Parameters      :- pointer to the character array defining the name of the
|                   task.
|
| Entry           :- create_task function.
|
|=====
|*/

static unsigned int hash_taskname_ptr ( char *task_name_ptr )
{

    unsigned long int offset;
    unsigned long int segment;
    unsigned long int lin_addr;

    /* !!!! NOT THAT THIS IS A NON-PORTABLE SECTION OF THE CODE. THE REFERENCES
    TO FP_OFF AND FP_SEG AND THE LINEAR ADDRESS CALCULATIONS ARE A REQUIREMENT
    FOR THE 80X86 SERIES ARCHITECTURE.
    */

    offset = FP_OFF ( task_name_ptr );
    segment = FP_SEG ( task_name_ptr );

    /* now convert to a linear address */
    lin_addr = segment * 16 + offset;

    /* now carry out the hash operation */

    return ( (unsigned int)(lin_addr % HASH_TABLE_SIZE) );
} /* end of hash_taskname_ptr */

/*-----*/

/*
|=====
|
| alloc_mbx_table
|
| The function of this routine is to allocate storage for an array of pointers
| to the mail boxes in the system. This array is created on the heap at
| initialisation time. The array is initialised with NULL_PTR pointers which are
| set up to the correct values in the create_mbx routine which itself is
| called from the create_task routine. The array components point to the mail
| boxes themselves. The purpose of the table is to allow the kernel to
| physically locate where a mail box is in memory using only the task number
| associated with the mail box. The task number effectively forms the index
| into this table.
|
| The function returns a pointer to the area of memory which has been
| allocated for the table. If the pointer is a NULL_PTR then the allocation has
| not been successful.
|
| Parameters : - number of tasks in the system.
|
| Entry via : - create_unos_data_struct function
|
|=====
```

```

*/

static mbx** alloc_mbx_table ( unsigned int num_of_tasks )
{
    mbx** mbx_table;
    int i;

    mbx_table = ( mbx** )ucalloc ( num_of_tasks, sizeof ( mbx* ) );
    if ( mbx_table != NULL_PTR )
    {
        for ( i = 0; i < num_of_tasks; i++ )
            mbx_table [ i ] = NULL_PTR;
    } /* if */

    return mbx_table;
} /* end of alloc_mbx_table */

/*-----*/

/*
=====
|
| create_mbx
|
| The function of this routine is to allocate and initialise a mail box.
| The routine allocates storage on the heap for a mail box and then
| initialises its contents based on the parameters passed in.
|
| The function returns the address of the memory allocated for the mail box.
| If there has been a problem allocating memory then a NULL_PTR pointer is
| returned.
|
| Parameters : - task number for which the mbx is to be created
|               - type of mail box - fifo or priority (only fifo currently
|                   implemented)
|               - number of messages in the mail box
|               - size of each message
|               - space available semaphore
|               - message available semaphore
|
| Entry via : - create_task routine
|
=====
*/

static mbx* create_mbx ( unsigned int task_num, char mbx_type, unsigned int
                        num_mess, unsigned int mess_size, unsigned int
                        spce_avail_sema, unsigned int mess_avail_sema )
{
    mbx* mbx_ptr;
    char allocation_problem = FALSE;
    unsigned int i;

    mbx_ptr = ( mbx* )umalloc ( sizeof ( mbx ) );
    if ( mbx_ptr != NULL_PTR )
    {
        mbx_ptr->mess_addr = task_num;
        mbx_ptr->mbx_type = mbx_type;
        mbx_ptr->q_size = num_mess;
    }

```

Appendix H — UNOS-V2.0 LISTING

```
mbx_ptr->mess_size = mess_size;
mbx_ptr->spce_avail_sema = spce_avail_sema;
mbx_ptr->mess_avail_sema = mess_avail_sema;
mbx_ptr->free = num_mess;
mbx_ptr->used = 0;
mbx_ptr->get_ptr = 0;
mbx_ptr->put_ptr = 0;
mbx_ptr->qik_mess_flag = FALSE;
/* now allocate the quick message envelope */
mbx_ptr->qik_mess_ptr = ( envelope* )umalloc ( sizeof ( envelope ) );
/* now allocate the message queue envelopes */
mbx_ptr->mess_q_ptr =( envelope* )ucalloc ( num_mess,
                                         sizeof ( envelope ) );

/* check to see if the allocation for the envelopes has been
successful
*/
if ( ( mbx_ptr->qik_mess_ptr == NULL_PTR ) ||
    ( mbx_ptr->mess_q_ptr == NULL_PTR ) )
    return ( mbx* )NULL_PTR;

/* now allocate the message buffers for each of the envelope
structures
*/
mbx_ptr->qik_mess_ptr->message_ptr = ( char* )ucalloc ( mess_size,
                                                       sizeof ( char ) );

for ( i = 0; i < num_mess; i++ )
{
    mbx_ptr->mess_q_ptr [ i ].message_ptr = ( char* )ucalloc (
                                                mess_size, sizeof ( char ) );
    if ( mbx_ptr->mess_q_ptr [ i ].message_ptr == NULL_PTR )
        allocation_problem = TRUE;
} /* for */

/* now check if the allocation of the message buffers for each of
the envelopes has been successful
*/
if ( ( mbx_ptr->qik_mess_ptr->message_ptr == NULL_PTR ) ||
    allocation_problem )
    return ( mbx* )NULL_PTR;
else
    return mbx_ptr;
} /* if */
else
{
    cprintf ( "Mail Box allocation problem\n" );
    return ( mbx* )NULL_PTR;
} /* else */
} /* end create_mbx */

/*-----*/

/*
=====
|
| mail_exchange
|
| The purpose of this function is to match the pointer to a task name to the
| task number. This matching operation is required in the mail system to
| allow the address of a task to be independent of its task number. The
| mapping from the pointer value to the task number is implemented using
| a chained hashing algorithm. The hash table is initialised during the
```



```

| task creation process. The hash table itself contains pointers to linked
| lists of taskname_map structures. The index into the table is calculated
| using a simple (but effective) modulus based hashing scheme.
|
| If the pointer address hashes to a table entry which has no taskname_map
| structure pointer then one is attempting to send data to an undefined
| task. This is indicated by a hash_table entry being NULL_PTR. The routine
| in this case returns the task number as the contents of num_of_tasks
| (which is actually one more than the number of the last task number).
|
| Parameter :- pointer to a character string.
|
| Entry via :- send_mess function
|
=====
*/

static unsigned int mail_exchange ( char *mess_addr_ptr )
{
    unsigned int hash_table_index;
    taskname_map *taskname_struct_ptr;
    unsigned int task_number = 0xffff;

    /* firstly hash the pointer */
    hash_table_index = hash_taskname_ptr ( mess_addr_ptr );

    /* now look into the hash table */

    if ( hash_table [ hash_table_index ] != NULL_PTR )
    {
        taskname_struct_ptr = hash_table [ hash_table_index ];
        do
        {
            if ( taskname_struct_ptr->task_name_ptr ==
                                     mess_addr_ptr )
            {
                task_number = taskname_struct_ptr->task_number;
                taskname_struct_ptr = NULL_PTR;
            } /* if */
            else
            {
                taskname_struct_ptr = taskname_struct_ptr->nxt_taskname_map;
            } /* else */
        } while ( taskname_struct_ptr != NULL_PTR );
    } /* if */

    /* now check to see if a legal task number has been found */
    if ( task_number == 0xffff )
    {
        /* illegal task number found */
        return num_of_tasks;
    } /* if */
    else
    {
        return task_number;
    } /* else */
} /* end of mail_exchange */

/*-----*/

/*

```

Appendix H — UNOS-V2.0 LISTING

```
=====
|
| send_mess
|
| This function puts a message into a mail box for a particular task. The
| can be called in two modes - blocking mode and non-blocking mode. In blocking
| mode if the mail box is full then the calling task is blocked on a semaphore.
| In non-blocking mode, if the mail box is full then control is returned to
| the calling task, along with a return code to indicate that the message was
| not sent (FULL_MBX).
|
| In order to make the address of the message independent of task numbering
| message addresses consist of a pointer to a string describing the task. The
| connection between this pointer and the task number is created at task
| creation time. In order to make it fast to obtain the task number from
| the string pointer a hash table is used. This mapping is carried out by the
| mail exchange which implements a chained hashing algorithm.
|
| NOTE : this routine currently does not handle priority mail boxes.
|
| Note that interrupts are disabled for a large part of this routine in order
| to make the message indivisible.
|
| Parameters : - pointer to the message to be sent
|               - length of the message to be sent
|               - address of the receiver of the message. This is a pointer to
|                 a character string that contains the name of the task. The
|                 pointer value is actually used as address of the task and
|                 the contents of the character string.
|               - a number indicating whether the caller should be blocked or
|                 not on the mail box semaphore. A zero indicates that the
|                 caller should be blocked (BLOCK_SM), a one indicates that
|                 if the mail box is full then control should be returned to
|                 the caller (NO_BLOCK_SM).
|
| Returns:      - a TRUE if the message is successfully sent
|               - a FALSE if the message is too big for the mail box, or an
|                 attempt is made to send a message to a non-existent task.
|               - a FULL_MBX is returned if the mailbox is full and the routine
|                 is called with the block_action parameter set to NO_BLOCK_SM.
|
| Entry via :   - multiple places
|
|=====
*/

int send_mess ( unsigned char* mess_ptr, unsigned int mess_lgth,
                char *mess_addr_ptr, int block_action )
{
    unsigned int mess_addr, sender_task_num;
    char *sender_addr_ptr;
    char int_status;
    unsigned int i;

    /* firstly call the mail exchange to establish which task number is
    being addressed, and then check to see if the task that the message
    is being sent to actually exists. If it doesn't then return with a FALSE.
    */
    if ( ( ( mess_addr = mail_exchange ( mess_addr_ptr ) ) >= num_of_tasks))
    {
        return FALSE;    /* non-existent task being addressed */
    } /* if */

    /* now check whether the message will fit in the message buffer */
    if ( mess_lgth > mbx_table [ mess_addr ]->mess_size )
        return FALSE;    /* does not fit in the message buffer */

    int_status = return_interrupt_status ( );
    disable ( );
}
```

```

/* Now check whether this should block or not */
if (block_action == NO_BLOCK_SM)
{
    /* No blocking allowed so check to see if the mail box has space */
    if ( mbx_table [ mess_addr ]->spce_avail_sema == 0 )
    {
        /* Will get blocked so return with a return code */
        if (int_status)
        {
            enable ();
        } /* if */

        return (FULL_MBX);
    } /* if */
} /* if */

/* Enter at this point if the call is a blocking call, or the calling task
will not get blocked.
*/
wait ( mbx_table [ mess_addr ]->spce_avail_sema );

sender_task_num = rtn_current_task_num ( );

/* now get the name of the sending task by looking in the tcb of
the task.
*/
sender_addr_ptr = tcb [ sender_task_num ]->task_name_ptr;

/* copy the message into the correct mail box envelope */
for ( i = 0; i < mess_lgth; i++ )
    mbx_table [ mess_addr ]->mess_q_ptr [
        mbx_table [ mess_addr ]->put_ptr ].message_ptr [ i ] =
                                                mess_ptr [ i ];

/* now complete the rest of the envelope structure */
mbx_table [ mess_addr ]->mess_q_ptr [ mbx_table [ mess_addr ]->put_ptr ]
    .mess_lgth = mess_lgth;
mbx_table [ mess_addr ]->mess_q_ptr [ mbx_table [ mess_addr ]->put_ptr ]
    .rtn_addr_ptr = sender_addr_ptr;
mbx_table [ mess_addr ]->mess_q_ptr [ mbx_table [ mess_addr ]->put_ptr ]
    .sender_pri = tcb [ sender_task_num ]->priority;

/* now update the mail box accounting locations */
mbx_table [ mess_addr ]->put_ptr++;
if ( mbx_table [ mess_addr ]->put_ptr >= mbx_table [ mess_addr ]->q_size )
    mbx_table [ mess_addr ]->put_ptr = 0;
mbx_table [ mess_addr ]->free--;
mbx_table [ mess_addr ]->used++;

if ( int_status )
    enable ( );

usignal ( mbx_table [ mess_addr ]->mess_avail_sema );

return TRUE;    /* indicate that message successfully sent */
} /* end of send_mess */

/*-----*/

/*

```

Appendix H — UNOS-V2.0 LISTING

```
=====
|
| send_qik_mess
|
| This function sends a quick message to a mail box for a particular task.
| A quick or express message is different from a normal message in that
| it should be the next message read from the mail box regardless of the
| other messages that may be stored in the mail box. This is generally
| achieved by putting the quick message just prior to the next message to
| be read from the buffer and then adjusting the get_ptr appropriately.
| If the message queue is full however this cannot be done, and in fact the
| task generating the message could become blocked (depending on how this
| situation was handled). For this reason the mail box contains a special
| envelope to store quick messages. If the situation arises where the special
| mail box is full also then the routine returns immediately with a return
| code equal to 2 which indicates this condition. A return code of 0 (usually
| corresponding to a FALSE) indicates that the message is too big for the
| size of the message buffers. A return code of 1 (usually corresponding to
| a TRUE) indicates a successful message sent.
|
| Parameters : - pointer to the message to be sent
|               - message length
|               - address where message is to be sent - a pointer to the
|               task name.
|
| Entry via : - multiple places
|
|=====
*/

int send_qik_mess ( unsigned char* mess_ptr, unsigned int mess_lgth,
                   char * mess_addr_ptr )
{
    char int_status;
    unsigned int mess_addr, i;

    /* firstly call the mail exchange to establish which task number is
    being addressed, and then check to see if the task that the message
    is being sent to actually exists. If it doesn't then return with a FALSE.
    */
    if ( ( ( mess_addr = mail_exchange ( mess_addr_ptr ) ) >= num_of_tasks))
    {
        return FALSE;    /* non-existent task being addressed */
    } /* if */

    /* check if the message will fit in the message buffer */
    if ( mess_lgth > mbx_table [ mess_addr ]->mess_size )
    {
        return FALSE;    /* message too big for the buffer */
    } /* if */

    int_status = return_interrupt_status ( );
    disable ( );
    /* now check to see if there is room in the normal message queue for the
    express message
    */
    if ( mbx_table [ mess_addr ]->free == 0 )
    {
        /* enter here if there is no room in the normal message buffer so
        now check to see if there is room in the special qik message buffer.
        If there is no room here then return with the appropriate return
        code.
        */
        if ( mbx_table [ mess_addr ]->qik_mess_flag )
        {
            /* qik message envelope full so reset interrupts and return to
            the calling program
            */
            if ( int_status )

```

```

        enable ( );
    return 2;
} /* if */
else
{
    /* there is room in the qik message envelope */
    for ( i = 0; i < mess_lgth; i++ )
        mbx_table [ mess_addr ]->qik_mess_ptr->message_ptr [ i ] =
            mess_ptr [ i ];

    mbx_table [ mess_addr ]->qik_mess_ptr->rtn_addr_ptr =
        tcb [ rtn_current_task_num ( ) ]->task_name_ptr;
    mbx_table [ mess_addr ]->qik_mess_ptr->sender_pri =
        tcb [ mess_addr ]->priority;
    mbx_table [ mess_addr ]->qik_mess_ptr->mess_lgth = mess_lgth;
    mbx_table [ mess_addr ]->qik_mess_flag = TRUE;
} /* else */
} /* if */
else
{
    /* there are free locations in the message queue so put the qik
    message at the head of this queue. If the message queue is empty
    then put the message in it at the put_ptr location
    */

    if ( mbx_table [ mess_addr ]->used == 0 )
    {
        /* now write the message into the envelope in the envelope queue */
        for ( i = 0; i < mess_lgth; i++ )
            mbx_table [ mess_addr ]->mess_q_ptr [ mbx_table [
                mess_addr ]->put_ptr ].message_ptr [ i ] = mess_ptr [ i ];

        /* now update the other components of the envelope */
        mbx_table [ mess_addr ]->mess_q_ptr [
            mbx_table [ mess_addr ]->put_ptr ].rtn_addr_ptr =
            tcb [ rtn_current_task_num ( ) ]->task_name_ptr;
        mbx_table [ mess_addr ]->mess_q_ptr [
            mbx_table [ mess_addr ]->put_ptr ].sender_pri =
            tcb [ mess_addr ]->priority;
        mbx_table [ mess_addr ]->mess_q_ptr [
            mbx_table [ mess_addr ]->put_ptr ].mess_lgth = mess_lgth;

        mbx_table [ mess_addr ]->put_ptr++;
        if ( mbx_table [ mess_addr ]->put_ptr >= mbx_table [
            mess_addr ]->q_size )
            mbx_table [ mess_addr ]->put_ptr = 0;
    } /* if */
    else
    {
        if ( mbx_table [ mess_addr ]->get_ptr == 0 )
            mbx_table [ mess_addr ]->get_ptr = mbx_table [ mess_addr ]->
                q_size - 1;
        else
            mbx_table [ mess_addr ]->get_ptr--;

        /* now write the message into the envelope in the envelope queue */
        for ( i = 0; i < mess_lgth; i++ )
            mbx_table [ mess_addr ]->mess_q_ptr [ mbx_table [ mess_addr ]->
                get_ptr ].message_ptr [ i ] = mess_ptr [ i ];

        /* now update the other components of the envelope */
        mbx_table [ mess_addr ]->mess_q_ptr [
            mbx_table [ mess_addr ]->get_ptr ].rtn_addr_ptr =
            tcb [ rtn_current_task_num ( ) ]->task_name_ptr;
        mbx_table [ mess_addr ]->mess_q_ptr [
            mbx_table [ mess_addr ]->get_ptr ].sender_pri =
            tcb [ mess_addr ]->priority;
        mbx_table [ mess_addr ]->mess_q_ptr [
            mbx_table [ mess_addr ]->get_ptr ].mess_lgth = mess_lgth;
    } /* else */
}

```

Appendix H — *UNOS-V2.0 LISTING*

```
        /* now update the used and free locations */
        mbx_table [ mess_addr ]->free--;
        mbx_table [ mess_addr ]->used++;
    } /* else */

    if ( int_status )
        enable ( );

    /* now signal that data is available in the mail box */
    usignal ( mbx_table [ mess_addr ]->mess_avail_sema );

    return TRUE;
} /* end of send_qik_mess */


/*-----*/

/*
=====
|
| rcv_mess
|
| This routine receives a message from a mail box associated with a particular
| task. There is no need to specify the mail box since the mail box from
| which the message is read from is that associated with the task.
| One important feature of this routine is that the receiving task can
| indicate how long it will wait for the message by passing a time limit to the
| routine. If a value of zero is passed for the time limit then the time limit
| is deemed to be infinite.
|
| The function returns the address of the sending task if the timeout period
| has not expired. Note that this address is returned as a pointer to the
| character string which names the task. If the timeout period has expired
| without a message being received it will return NULL_PTR. If the return is
| due to the fact that a timer is not available for the timed_wait operation
| then the return code is 0xffff:000f.
|
| Parameters : - pointer to the memory area where the message will be stored
|               - pointer to the integer where the message length will be
|                 stored.
|               - time limit value.
|
| Entry via : - multiple places
|
=====
*/

char *rcv_mess ( unsigned char* mess_ptr, unsigned int* mess_lgth,
                 unsigned long time_limit )
{
    unsigned int mbx_addr;
    char *rtn_addr_ptr;
    int wait_result = 0;
    char int_status;
    unsigned int i;
    unsigned int *mess_lgth_ptr;
    char *qik_mess_ptr;
    char *mbx_mess_ptr;

    mbx_addr = rtn_current_task_num ( );
```

```

/* firstly check what type of wait has to be carried out */
if ( time_limit == 0 )
    wait ( mbx_table [ mbx_addr ]->mess_avail_sema );
else
    wait_result = timed_wait ( mbx_table [ mbx_addr ]->
                                mess_avail_sema, time_limit );

/* now check the value of the wait_result. If zero then the wait has
been terminated by a signal condition or no wait has occurred. If the
wait result is non-zero then some error condition has occurred. the type
of error condition is determined by the value:- 1=>wait terminated due
to a timeout on the semaphore; 2=>a timer was not available for the
timed_wait.
*/
if ( wait_result )
{
    if ( wait_result == 1 )
        return NULL_PTR; /* timeout occurred */
    return ( (char*)MK_FP ( 0xffff, 0x000f ) ); /* no timer available */
} /* if */

int_status = return_interrupt_status ( );
disable ( );

/* enter this section of the code if there is a message to receive. Now
check to see if the message is in the qik message envelope. If so then
retrieve it from there, else get the message from the message envelope
queue.
*/
if ( mbx_table [ mbx_addr ]->qik_mess_flag )
{
    /* message in the qik envelope */
    /* now setup pointers to the appropriate variables so that the
    compiler will be forced to produce efficient code.
    */
    mess_lgth_ptr = &mbx_table [ mbx_addr ]->qik_mess_ptr->mess_lgth;
    qik_mess_ptr = mbx_table [ mbx_addr ]->qik_mess_ptr->message_ptr;

    for ( i = 0; i < *mess_lgth; i++ )
    {
        mess_ptr [ i ] = *(qik_mess_ptr + i);
    } /* for */

    *mess_lgth = *mess_lgth_ptr;
    mbx_table [ mbx_addr ]->qik_mess_flag = FALSE;
    rtn_addr_ptr = mbx_table [ mbx_addr ]->qik_mess_ptr->rtn_addr_ptr;
} /* if */
else
{
    /* message must be in the normal message queue so retrieve from here
    */
    /* firstly assign the key variables to pointers to force the compiler
    to produce efficient code.
    */
    mess_lgth_ptr = &mbx_table [ mbx_addr ]->mess_q_ptr [
        mbx_table [ mbx_addr ]->get_ptr ].mess_lgth;
    mbx_mess_ptr = mbx_table [ mbx_addr ]->mess_q_ptr [
        mbx_table [ mbx_addr ]->get_ptr ].message_ptr;
    for ( i = 0; i < *mess_lgth_ptr; i++ )
    {
        mess_ptr [ i ] = *(mbx_mess_ptr + i);
    } /* for */

    *mess_lgth = mbx_table [ mbx_addr ]->mess_q_ptr [
        mbx_table [ mbx_addr ]->get_ptr ].mess_lgth;
    rtn_addr_ptr = mbx_table [ mbx_addr ]->mess_q_ptr [
        mbx_table [ mbx_addr ]->get_ptr ].rtn_addr_ptr;
    mbx_table [ mbx_addr ]->get_ptr++;
    if ( mbx_table [ mbx_addr ]->get_ptr >= mbx_table [ mbx_addr ]->
        q_size )

```

Appendix H — *UNOS-V2.0 LISTING*

```
    {
        mbx_table [ mbx_addr ]->get_ptr = 0;
    } /* if */

    mbx_table [ mbx_addr ]->free++;
    mbx_table [ mbx_addr ]->used--;
} /* else */

/* now signal that space is available in the mail box */
usignal ( mbx_table [ mbx_addr ]->spce_avail_sema );

if ( int_status )
    enable ( );

return rtn_addr_ptr;
} /* end of rcv_mess */

/*-----*/

/*
=====
|
| size_mbx
|
| This function returns the size of the message queue for a particular mail
| box. If one tries to look at an illegal mail box then a zero is returned,
| else the size of the mail box is returned.
|
| Parameters : - mail box address - which is the task name with which it is
|               associated.
|
| Entry via : - multiple places
|
=====
*/

unsigned int size_mbx ( char *mbx_addr_ptr )
{
    unsigned int mbx_num;
    unsigned int mbx_size;

    /* carry out the mapping between the mail box name (which is the task
    name) and the mail box number.
    */
    if ( ( mbx_num = mail_exchange ( mbx_addr_ptr ) ) >= num_of_tasks )
    {
        /* trying to look at an illegal mail box */
        mbx_size = 0;
    } /* if */
    else
    {
        /* address OK */
        mbx_size = mbx_table [ mbx_num ]->q_size;
    } /* else */
    return mbx_size;
} /* end of size_mbx */
```



```
/*-----*/

/*
=====
|
| size_mbx_mess
|
| This function returns the maximum size of a message which can be sent to a
| mail box (i.e. the size of each of the message slots in the message queue.
| If the mail box address is illegal then the routine returns a value of
| 0, else the length of the messages is returned.
|
| Parameters : - address of the mail box (i.e. task name with which it is
|               associated) whose size is to be determined.
|
| Entry via : - multiple places
|
=====
*/

unsigned int size_mbx_mess ( char *mbx_addr_ptr )
{
    unsigned int mbx_num;
    unsigned int size_mbx_mess;

    /* carry out the mapping between the mail box name (which is the task
    name) and the mail box number.
    */
    if ( ( mbx_num = mail_exchange ( mbx_addr_ptr ) ) >= num_of_tasks )
    {
        /* trying to look at an illegal mail box */
        size_mbx_mess = 0;
    } /* if */
    else
    {
        /* address OK */
        size_mbx_mess = mbx_table [ mbx_num ]->mess_size;
    } /* else */
    return size_mbx_mess;
} /* end of size_mbx_mess */
```

```
/*-----*/
```

```
/*
=====
|
| free_mbx
|
| This function returns the number of free message slots in a mail box. If
| the mail box number is illegal then the routine returns a 0xffff value, else
| it returns the number of free locations.
|
| Parameters : - address of the mail box (i.e. task name pointer with which
|               it is associated) whose free space is to be determined.
|
=====
```

Appendix H — *UNOS-V2.0 LISTING*

```
|
| Entry via : - multiple places
|
=====
*/

unsigned int free_mbx ( char *mbx_addr_ptr )
{
    char int_status;
    unsigned int mbx_num;
    unsigned int free;

    int_status = return_interrupt_status ( );
    disable ( );
    /* carry out the mapping between the mail box name (which is the task
    name) and the mail box number.
    */
    if ( ( mbx_num = mail_exchange ( mbx_addr_ptr ) ) >= num_of_tasks )
    {
        /* trying to look at an illegal mail box */
        free = 0xffff;
    } /* if */
    else
    {
        /* address OK */
        free = mbx_table [ mbx_num ]->free;
    } /* else */
    if ( int_status )
        enable ( );
    return free;
} /* end of free_mbx */

/*-----*/

/*
=====
|
| used_mbx
|
| This function returns the number of used message slots in a mail box. If the
| mail box number is illegal then the number returned is 0xffff.
|
| Parameters : - address of the mail box (i.e. the task nmae pointer with
|               which the mail box is associated) whose used space is to
|               be determined.
|
| Entry via : - multiple places
|
=====
*/

unsigned int used_mbx ( char *mbx_addr_ptr )
{
    char int_status;
    unsigned int mbx_num;
    unsigned int used;

    int_status = return_interrupt_status ( );
    disable ( );
```

```

/* carry out the mapping between the mail box name (which is the task
name) and the mail box number.
*/
if ( ( mbx_num = mail_exchange ( mbx_addr_ptr ) ) >= num_of_tasks )
{
    /* trying to look at an illegal mail box */
    used = 0xffff;
} /* if */
else
{
    /* address OK */
    used = mbx_table [ mbx_num ]->used;
} /* else */
if ( int_status )
    enable ( );
return used;
} /* end of used_mbx */

/*-----*/

/*
=====
|
| flush_mbx
|
| This function as the name implies flushes the contents of the mail box
| which is owned by the calling function. Flushing involves setting the
| get and put pointers back to zero, resetting the used and free locations,
| and clearing the qik_mess_flag to the states which exist when the mail
| box was first created. If the mail box is full at the time that this
| command is executed then it is possible that there are blocked tasks
| waiting to access the mail box. Therefore if these are not released
| then these tasks would remain blocked (even if the semaphore value has
| been reset). Therefore under this condition any blocked tasks are removed
| from the appropriate semaphore queue. The semaphore values for the mail
| box are reset to the initial values.
|
| The flush command can only be executed from the task with which the mail
| box is associated to prevent the situation where another task could
| flush the mail box whilst the task to which it belongs is left blocked
| on a semaphore waiting to receive a message.
|
| Parameters : - none
|
| Entry via : - multiple places
|
|=====
*/

void flush_mbx ( void )
{
    unsigned int mbx_addr;
    char int_status;

    mbx_addr = rtn_current_task_num ( );

    int_status = return_interrupt_status ( );
    disable ( );
    mbx_table [ mbx_addr ]->get_ptr = 0;
    mbx_table [ mbx_addr ]->put_ptr = 0;

```

Appendix H — UNOS-V2.0 LISTING

```
mbx_table [ mbx_addr ]->free = mbx_table [ mbx_addr ]->q_size;
mbx_table [ mbx_addr ]->used = 0;
mbx_table [ mbx_addr ]->qik_mess_flag = FALSE;

/* now check to see if there are any tasks blocked on the spce_avail_sema
for the mail box.
*/
while ( semaphore [ mbx_table [ mbx_addr ]->spce_avail_sema ]
        ->semaphore_value != mbx_table [ mbx_addr ]->q_size )
{
    /* must be tasks blocked on the spce_avail_sema so remove the task
    from the semaphore queue and place on the appropriate priority
    queue.
    */
    usignal ( mbx_table [ mbx_addr ]->spce_avail_sema );
} /* while */

/* now reset the mess_avail_sema value */
semaphore [ mbx_table [ mbx_addr ]->mess_avail_sema ]->semaphore_value = 0;

if ( int_status )
    enable ( );
} /* end of flush_mbx */

/*-----*/

/*****
/*
/*
/*          UNOS_5 MODULE
/*
/*          by
/*
/*          Robert Betz
/*    Department of Electrical Engineering and Computer Science
/*          University of Newcastle
/*          Australia
/*
/*          ( Copyright 1989, 1990 )
/*
/*
/*
*****/

/*
HISTORY

Began typing this module on the 30th January 1990.

*/

/*
DESCRIPTION

The function of this module is to allow the allocation and freeing of memory
from a pseudo heap in the UNOS system. The ualloc and ufree routines
essentially follow those routines described in Kernighan and Ritchie pages
173 - 177. The main change is that the morecore routine does not call UNIX
operating system memory allocation routines but allocates initially the whole
pool size to the free block. The pool size and location have to be set up
```

by the user and depend on the target system. For example if UNOS is being executed on a MS-DOS platform then the pool can be formed by gaining an allocation from MS-DOS at system initialisation. In more embedded environments the pool size and location have to be explicitly set variables.

For a more complete description of the allocation system refer to the UNOS documentation or the ascii version of this documentation distributed with this software.

Changes which need to be made to this routine in order to make it work correctly with a segmented architecture such as the 8086 are outline in the accompanying documentation.

*/

/*-----*/

/*

```
=====
|
|  umalloc
|
|  This routine allocates the number of bytes requested from the free memory
|  block list. The routine is basically a copy of the routine in the
|  Kernighan and Ritchie book pages 175. This routine takes the requested
|  number of bytes and rounds it to a proper number of header-sized units.
|  The actual block that will allocated contains one more unit which is
|  reserved for the header itself. The pointer returned from the routine is
|  to the free memory area and not to the header. Functionally this routine
|  behaves the same as the malloc routine in the 'C' language - hence the
|  name.
|
|  Initially when the routine a is executed a free list of memory blocks will not
|  exist. In this circumstance the morecore routine is called which returns a
|  pointer to more free memory. In an UNIX environment this routine would
|  call the operating system to be allocated more memory. This is the way
|  that the Kernighan and Ritchie implementation works. However in an
|  embedded application this does not make sense and the morecore routine
|  simply allocates the entire memory pool size to the initial free block
|  of memory. I have retained the original Kernighan/Ritchie model in case
|  UNOS is extended at a future stage to be a full fledged disk base os in
|  its own right.
|
|  If a block of free memory is found and it is exactly the correct size then
|  the block is removed from the linked list of free memory blocks. If the
|  block of free memory is larger than the requested block then the remainder
|  of the free block after the requested amount of memory is removed is
|  returned to the list of free memory blocks.
|
|  If the memory allocation has been successful then a pointer to the memory
|  block is returned. If unsuccessful then a NULL_PTR pointer is returned.
|
|  Parameters : - number of bytes to be allocated
|
|  Entry via  : - multiple places
|
|=====
*/
```

```
char huge* umalloc ( unsigned long num_bytes )
{
    blk_header huge *ptr1, huge *ptr2;
    char int_status;
    unsigned long blksize_units;
    unsigned int cur_seg, cur_offset, norm_seg, norm_offset;
```

Appendix H — UNOS-V2.0 LISTING

```
blk_header huge *norm_start_header_ptr;

int_status = return_interrupt_status ( );
disable ( );

/* round the number of bytes so that it is an integral number of
header sized blocks. This is done to maintain the correct byte
alignment as forced by the union header structure. Note that the
basic allocation unit then becomes sizeof ( blk_header ) units.
It is this value which is stored in the blk_size variable of the
header structure.
*/
blksize_units = 1 + ( num_bytes + sizeof ( blk_header ) - 1 ) /
                  sizeof ( blk_header );

if ( ( ptr1 = last_blk_alloc ) == NULL_PTR )
{
    /* enter here if currently no free list so set up a dummy start
header known as start_header
    */
    cur_seg = FP_SEG ( &start_header );
    cur_offset = FP_OFF ( &start_header );
    norm_seg = (unsigned int) (( ( unsigned long )cur_seg * 16 + cur_offset)
                               / 16);
    norm_offset = (unsigned int) (( ( unsigned long )cur_seg * 16 +
                                   cur_offset ) - norm_seg * 16 );
    norm_start_header_ptr = (header_struct huge*)MK_FP ( norm_seg, norm_offset );
    start_header.header.next_blk_ptr = last_blk_alloc = ptr1 =
                                   norm_start_header_ptr;
    start_header.header.blk_size = 0;
} /* if */

/* now start searching through the linked list of block headers
searching for a block of free RAM >= the rounded number of bytes
requested.
*/
for ( ptr2 = ptr1->header.next_blk_ptr; ; ptr1 = ptr2, ptr2 = ptr2->
      header.next_blk_ptr )
{
    if ( ptr2->header.blk_size >= blksize_units )
    {
        /* the free block currently pointed to is big enough. Now
check to see if it is exactly the correct size or not.
        */
        if ( ptr2->header.blk_size == blksize_units )
            /* exactly the right size so update the pointer from the
previous free block to point to the block after the current
block
            */
            ptr1->header.next_blk_ptr = ptr2->header.next_blk_ptr;
        else
        {
            /* the block to be allocated is not the exact size of the
free area therefore take the required storage from the end
of the current free block.
            */
            /* change the size of the free memory block to reflect the
new size
            */
            ptr2->header.blk_size -= blksize_units;

            /* now update the pointer ptr2 to point to the top blksize_units
which have been taken from the top of the free area. This
is the pointer which will be used in the return statement.
            */
            ptr2 += ptr2->header.blk_size;

            /* now update the header information for the allocated
block of memory
            */

```

```

        ptr2->header.blk_size = blksize_units;
    } /* if */

    last_blk_alloc = ptr1;
    rem_memory -= blksize_units;

    if ( int_status )
        enable ( );

    /* now return the pointer to the memory in the allocated block */
    return ( ( char huge * ) ( ptr2 + 1 ) );
} /* if */

/* enter if the area in the currently pointed to free block is not
large enough. If ptr2 is pointing to the last allocated location
then the end of the free list has been reached and more memory
has to be allocated. In the UNOS case this routine should only
be called when the free list to be firstly created. The morecore
routine passes back the entire memory pool to the free list.
*/
if ( ptr2 == last_blk_alloc )
    /* wrapped around free list */
    if ( ( ptr2 = ( blk_header* )morecore ( ) ) == NULL_PTR )
    {
        if ( int_status )
            enable ( );
        return NULL_PTR;    /* no free core */
    } /* if */
} /* for */
} /* end of umalloc */

```

/*-----*/

```

/*
=====
|
| ret_free_mem ( )
| This function returns the amount of free memory in the heap maintained
| by this memory management software. The value returned is in bytes.
|
| Parameters      : - none
|
| Entry via       : - Multiple places in user code.
|
=====
*/

```

```

unsigned long ret_free_mem ( void )
{
    return ( rem_memory * sizeof ( blk_header ) );
} /* end of ret_free_mem() */

```

/*-----*/

/*

Appendix H — *UNOS-V2.0 LISTING*

```
=====
|
| morecore
|
| In the original Kernighan and Ritchie book this routine had the job of
| returning some more memory from the UNIX operating system to be dynamically
| allocated by the allocate routine. In UNOS it obviously does not do this
| since it is not operating in a UNIX environment. This routine on the first
| call returns the total pool size. On subsequent calls it will return a NULL_PTR
| to indicate that no more memory is available. In order to put the free
| memory into the free memory block list the ufree function is called with
| a pointer to the new memory area.
|
| Parameters : - none
|
| Entry via  : - umalloc function in this module
|
|=====
*/

static blk_header huge * morecore ( void )
{
    blk_header huge * new_core;

    if ( mem_pool_size > 0 )
    {
        new_core = ( blk_header huge *) mem_pool_ptr;
        new_core->header.blk_size = mem_pool_size / sizeof ( blk_header )
                                - 1;

        mem_pool_size = 0;
        ufree ( ( char huge * ) ( new_core + 1 ) );

        /* now place the new core onto the list of free memory blocks. In this
        process the last_blk_alloc is set
        */

        return ( last_blk_alloc );
    } /* if */

    else
        return NULL_PTR;          /* no memory available */
} /* end of morecore */

/*-----*/

/*
=====
|
| ufree
|
| This function scans the free list starting at last_alloc_blk looking for
| the place to insert the block which is to be added to the free list. The
| location to insert the free block is determined based on the fact that the
| blocks are stored in order from the low address to the high address. This
| ordering is maintained in order to make it simple to concatenate the block
| to be added with consecutive adjacent blocks. This is designed to help
| prevent memory fragmentation.
|
| Parameters : - pointer to the memory area to be added to the free list
|
| Entry via  : - multiple places
|
```



```
|
=====
*/

void ufree ( char huge* blk_ptr )
{

    blk_header huge *ptr1, huge *ptr2;
    unsigned long blk_size;
    char int_status;

    int_status = return_interrupt_status ( );
    disable ( );

    /* firstly make sure that the pointer is pointing to the block header
    of the free block to be added to the list
    */
    ptr2 = ( blk_header huge * )blk_ptr - 1;

    /* Now store the size of the block
    */
    blk_size = ptr2->header.blk_size;

    /* now search for the correct place to put the block in the free list.
    The following 'for' statement searches through the free list starting at
    the last_alloc_blk for the correct position to put the block
    */
    for ( ptr1 = last_blk_alloc; !( ptr2 > ptr1 && ptr2 <
        ptr1->header.nxt_blk_ptr ); ptr1 = ptr1->header.nxt_blk_ptr )
        /* now check to see if the new block has to be placed at the beginning
        or the end of the list. If so then exit this for loop, else continue to
        find the correct location.
        */
        if ( ptr1 >= ptr1->header.nxt_blk_ptr && ( ptr2 > ptr1 ||
            ptr2 < ptr1->header.nxt_blk_ptr ) )
            break;          /* has to be placed at one end or the other */

    /* now check to see if the block should be merged with the block
    above the block to be added.
    */
    if ( ( ptr2 + ptr2->header.blk_size ) == ptr1->header.nxt_blk_ptr )
    {
        /* join to the block above and amend the pointers in the added
        block to point to the free block next above the block previously
        above the block added.
        */
        ptr2->header.blk_size += ptr1->header.nxt_blk_ptr->header.blk_size;
        ptr2->header.nxt_blk_ptr = ptr1->header.nxt_blk_ptr->
            header.nxt_blk_ptr;
    } /* if */
    else
        /* the new block is being put between two existing blocks and
        cannot be concatenated with the block above it.
        */
        ptr2->header.nxt_blk_ptr = ptr1->header.nxt_blk_ptr;

    /* now check to see if the new block has to be merged with the block
    immediately below it.
    */
    if ( ( ptr1 + ptr1->header.blk_size ) == ptr2 )
    {
        /* join with the block below and then update the pointers of the
        block below to reflect the size of the new super block.
        */
        ptr1->header.blk_size += ptr2->header.blk_size;
        ptr1->header.nxt_blk_ptr = ptr2->header.nxt_blk_ptr;
    } /* if */
    else
        /* block is not adjacent so simply update the pointer in the block
        below to point to the new block.

```

```
/***** END OF FILE *****/
```

BIBLIOGRAPHY

- [1] A.M. Lister, *Fundamentals of Operating Systems, Fourth Edition*, Macmillan Computer Science Series.
- [2] Alan C. Shaw, *Logical Design of Operating Systems*, Prentice Hall.
- [3] H.M. Deitel, *Operating Systems*, Addison–Wesley, 1990.
- [4] William Stallings, *Operating Systems, Second Edition*, Prentice Hall International Editions, 1995, ISBN 0-13-180977-6.
- [5] Andrew S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice Hall International Editions, 1987, ISBN 0-13-637331-3.
- [6] *386 DX Microprocessor Programmer's Reference Manual*, Intel Cooperation.
- [7] *80386 System Software Writer's Guide*, Intel Cooperation.
- [8] *iAPX 286 Operating Systems Writer's Guide*, Intel Cooperation.
- [9] L. Sha, R. Rajkumar and J.P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-time Synchronization", *IEEE Trans. on Computers*, Vol. 39, pp 1175-1185, Sept 1990.
- [10] Alan A. Bertossi and Andrea Fusiello, "Rate-Monotonic Scheduling for Hard Real-Time Systems", *tech report*, Dipartimento di Matematica, Università di Trento, Italy, June 1996.
- [11] Mark H. Klein, John P. Lehoczky and Ragunathan Rajkumar, "Rate-Monotonic Analysis for Real-Time Industrial Computing", *IEEE Computer*, Vol. 27, pp 24-33, Jan 1994.
- [12] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a hard real-time environment", *Journal of the ACM*, Vol. 20, pp 46-61, Jan 1973.
- [13] M. Joseph and P. Pandya, "Finding Response Times in a Real-Time System", *The Computer Journal*, Vol. 29, pp 390-395, Oct. 1986.
- [14] K. Tindell, A. Burns and A.J. Wellings, "An Extendible Approach For Analysing Fixed-priority Hard Real-Time Tasks", *Real-Time Systems*, Vol. 6, pp 133-151, March 1994.
- [15] J.Y.-T. Leung and J. Whitehead, "On Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks", *Performance Evaluation*, Vol. 2, pp 237-250, Dec 1982.
- [16] J.P. Lehoczky, "Real-time Resource Management Techniques", in J.J Marciniak (editor), *Encyclopedia of Software Engineering*, pp 1011-1020, John Wiley and Sons, 1994.

