

THE NUCLEUS OF A COMPUTER.

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science
in
Computer Science
in the
University of Canterbury
by
E. L. Thompson

University of Canterbury
1980

"We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time."

"Little Gidding"

T.S.Eliot

in Collected Poems:1909-1962

(London,1963)

"Is there a thing of which it is said,
'See, this is new' ?
It has been already,
in the ages before us."

Ecclesiastes 1:10

CONTENTS

CHAPTER	PAGE
ABSTRACT	1
1. INTRODUCTION	2
2. CONCEPTS AND STRUCTURES OF COMPUTERS	4
2.1 The Hardware Configuration.	4
2.1.1 Input/Output Control.	5
2.1.2 Multi-Processor Considerations.	5
2.1.3 Defining a Control Structure.	6
2.2 Supervisors and the Users	7
2.2.1 The Extended Supervisor	7
2.2.2 Operating System Structures	8
2.3 Multiple Users and Multiprocessing.10
2.3.1 Process Swap Facilities10
2.3.2 Address Space Separation.11
2.3.3 Multiprocessing11
2.3.4 Synchronisation Primitives.12
2.3.5 Virtual Machine Environments.13
2.4 Conclusion.14
3. ADDRESSING SCHEMES16
3.1 Memory Addressing Schemes16
3.1.1 Base-Limit Register16
3.1.2 Instruction and Data Banks.17
3.1.3 Paged Memory.18
3.1.4 Segmentation.18
3.1.5 Paged-Segments.19
3.2 Peripheral and Processor Addressing20
3.2.1 Physical Address and Device Codes20
3.2.2 Memory Peripheral Addressing.20
3.2.3 Peripheral Type Detection21
3.2.4 Processor Addressing.21
3.3 Disk Storage Addressing21
3.3.1 File Structuring.22
3.4 Capability Addressing22
3.4.1 Capability as Absolute Address.23
3.4.2 Capability Integrity.23

CHAPTER	PAGE
3.5	Conclusion 24
4.	THE NUCLEUS CONCEPT 25
4.1	The Nucleus Requirements 25
4.2	An Addressing Scheme 26
4.2.1	Physical Capabilities 29
4.2.2	Storage Addressing. 34
4.2.3	Processor Addressing. 34
4.2.4	Peripheral Addressing 35
4.2.5	Function Addressing 35
4.3	Functional Characteristics 35
4.3.1	Access Flags. 35
4.3.2	System Reconfiguration. 37
4.3.3	Function Facilities 38
4.3.4	Flexibility of Nucleus Addressing Structure 39
4.4	Conclusion. 40
5.	THE NUCLEUS AND COMPUTING CONCEPTS. 41
5.1	Processors. 41
5.1.1	Process Swap. 41
5.1.2	Overview of Processor Functions . 42
5.1.3	Implementation of Processor Functions 42
5.1.4	Usage of Processor Segment and Functions 44
5.2	Peripherals. 45
5.2.1	Peripheral Controller 45
5.2.2	Peripheral Control Segment. . . . 46
5.2.3	Peripheral Requests 46
5.2.4	Peripheral Functions. 46
5.2.5	Alternative Peripheral Functions. 48
5.2.6	Peripheral Management and Interaction 50
5.3	Processes. 50
5.3.1	The Process Control Segment . . . 50
5.3.2	The Link and Priority Calls . . . 52
5.3.3	The Stack 52
5.3.4	The Fault Vector. 55

CHAPTER	PAGE
5.3.5	Timing Facilities57
5.3.6	Process Functions57
5.3.7	Storage Segment References.58
5.3.8	Processor Interaction58
5.3.9	File Creation and Deletion.62
5.4	Conclusion.62
6.	NUCLEUS IMPLEMENTATION63
6.1	Implementation on Existing Computers.63
6.1.1	The Existing Computer used for Implementation63
6.1.2	Implementing the Nucleus on the DG Eclipse.65
6.1.3	Problems with Implementation on Existing Computers67
6.2	Simulation of the Nucleus67
6.2.1	Choice of a Simulation Technique. . .67
6.2.2	Nucleus Simulation Using Burroughs Extended Algol69
6.2.3	Address Translation Simulation. . . .70
6.2.4	Problems with Simulation.71
6.3	Implementing in Hardware/Firmware71
6.4	Conclusion.72
7.	CONCLUSION73
	APPENDICES75
	ACKNOWLEDGEMENTS81
	BIBLIOGRAPHY82

LIST OF FIGURES

FIGURE		PAGE
4.1	The Capability Structure	27
4.2	Memory and Permanent Storage Access Structure	30
4.3	Processor Access Structure.	31
4.4	Peripheral Access Structure (Removable Storage)	32
4.5	Function Capability Relationship. .	33
5.1	Request Processor Function.	43
5.2	Release Processor Function.	43
5.3	Peripheral Request Function	47
5.4	Peripheral Completion Function. . .	47
5.5	Alternative Peripheral Request Function.	49
5.6	Alternative Peripheral Completion Function.	49
5.7	Process Control Segment	51
5.8	The Stack Push Operation.	53
5.9	The Stack Pop Operation	54
5.10	Process Creation Function	56
5.11	Process Initiation Function	56
5.12	Process Termination Function. . . .	56
5.13	Allocate Segment Function	59
5.14	Release Segment Function.	59
5.15	Processor Call Function	60
5.16	Processor Return Function	60
5.17	Chain to Processor Function	60
5.18	File Creation Function.	61
5.19	File Deletion Function.	61
6.1	Simulator Structure.	68

LIST OF TABLES

TABLE		PAGE
4.1	Capability Types and Structure . . .	28
4.2	Access Flags Applicable for Capability Types	36

ABSTRACT.

This thesis examines computer architecture and operating system developments to determine a set of requirements applicable to all computer systems. The requirements obtained by this examination are used to define the concept of a nucleus for a computer system.

A nucleus structure, based around an addressing scheme and a set of functions, is presented. This structure does not include processor and peripheral designs, but concentrates on the interaction between these components. The proposed addressing scheme is based on capability addressing, with the structure recognizing processors, peripherals, processes and files as addressable units.

The functions proposed assist in scheduling of processes for processors and peripherals. These functions are also referenced through the addressing scheme which assists in the creation of virtual machine environments.

Implementation techniques for the proposed nucleus are also examined in the context of validating the nucleus proposal. These implementation techniques are also discussed as final implementation methods.

CHAPTER I

INTRODUCTION

Since the development of the first stored program computers, the development of hardware technology has enabled the construction of physically smaller computers with greater processing power and storage capacity. Greater complexity in circuit composition has facilitated the construction of new capabilities in the hardware for lower cost and increased performance. In more recent developments of computers, the structure of both the software and hardware have been major considerations in the design of computer systems. With increased capabilities being available in the hardware a greater range of structures have been examined and implemented.

The emphasis on the software structure is displayed by the number of papers published in the area of software engineering. These include the concepts of modular programming (Parnas 1972) and structured programming (Dijkstra et al 1972), and programming languages designed with software structuring as a major consideration. eg Pascal (Wirth 1971), Algol (van Wijngaarden et al. 1975) and Modula (Wirth 1977).

Structuring of the hardware is illustrated by the number of different architectural approaches. For example the Data Structure Architectures (Giloi and Berg 1978), High-level Language architectures (Chu 1975), Data flow architectures (Dennis and Misunas 1975) and Flexible structure architectures such as the Burroughs B1700 (Wilner 1972).

For the construction of a computer system, either hardware or software, objectives are defined by the design team. These form a set of essential design requirements which must be met for the planned system. The contention of this thesis is that regardless of the final structure of any computer system, there is a core set of requirements which must be included, either implicitly or explicitly, in the design objectives. That is, there exists a core set of requirements which are a subset of the design objectives for all computer systems.

In addition to this hypothesis, it is contended that a central component or nucleus can be constructed to satisfy the core set of requirements. This nucleus would be independent of the external structure of the processors and peripherals used on the system, although the structure of the nucleus influences the characteristics of the other components of the computer system.

The nucleus, although not necessarily a recognisable component (no card or set of cards in a particular hardware box), presents to the remainder of the system a set of common facilities around which the various system components can be structured. In terms of existing terminology, the nucleus conforms more to a virtual machine monitor than to the kernel of an operating system, yet is more entwined in the hardware of the system than software. The software for a nucleus based system will be influenced by the facilities provided through the nucleus.

This thesis develops the nucleus concept, through the examination of existing hardware and software structures in chapter two.

Chapter three examines addressing schemes currently being used and those being studied in current research.

Chapter four presents the requirements for the nucleus and an addressing scheme developed for a nucleus based system.

Chapter five examines the structural effects of the nucleus on processors, processes and peripherals. Also presented in this chapter are some functions, which can be used to manipulate, synchronise and manage these components, provided through the nucleus.

Chapter six discusses implementation techniques and some of the problems which occur. This chapter presents the investigations of the author as the nucleus structure and concept was developed.

Chapter seven concludes the thesis by reviewing the major points and the nucleus design.

CHAPTER II

CONCEPTS AND STRUCTURES OF COMPUTERS

Von Neumann and his colleagues formalised the design objectives for the construction of the stored program computers in the design of the IAS computer (Von Neumann et al 1946). The resultant register based computer design has remained a central structure in computer development, although some current designs are utilizing different structural approaches (Dennis and Misunas 1975).

This chapter examines developments in the design of computer hardware and software structures in the context of finding a set of basic requirements for a computer.

2.1 THE HARDWARE CONFIGURATION

Unlike the architecture of buildings, which is revealed through the physical structure of the building, the architecture of the computer is not visible in the physical structure. The computer's physical componentry does not give any indication of its function within the computer system. The physical cabinets do by their naming give an indication of functions required in a computer.

Three basic components are necessary in the construction of a computer. These are a storage system, a processor and an input/output medium. Since the construction of the first stored program computers the interaction and communication between these components has been an important consideration in their design. The storage system comprises any medium on which the computer can randomly store and retrieve information (disk and main memory), and input/output media include any medium used for entering or outputting information (cards, printers, visual display units, and magnetic tapes). This classification distinguishes the type of interaction possible rather than the physical relationship between the medium and the rest of the computer system.

The storage system, and in particular the main memory requires rapid access, since the information present in the storage system controls the operation of the processor. As a consequence of the close relationship between the processor and memory, the control of access to this portion of the storage medium has been the responsibility of the processor. The addressing scheme for storage media has been, as a consequence, the centre of a considerable number of research projects. These will be examined in the next chapter.

2.1.1 INPUT/OUTPUT CONTROL

The processor, in single processor environments, is also involved in the control of the input/output components of the system. In the first stored program computers (Bell and Newell 1971), this meant that the processor initiated the required function on a peripheral and then waited for its completion. This proved to be a limiting factor on the speed of the processor and consequently with the introduction of interrupts (Rosen 1969), it became possible to initiate the required operation on a peripheral and then to continue processing other available information. The peripheral controller then issued an interrupt to the processor, when it required attention or when the requested operation was complete.

The addition of input/output channels or processors (particularly those which execute input/output control programs (IBM 360)) moved some of the control functions from the central processor but still required the processor to initiate the input/output control program. The software executing on the central processor is responsible for the scheduling of work for the system.

Under the interrupt structure, the input/output processors act as slaves to the central processor. By the appropriate structuring of software, it is possible to construct systems which appear to be driven by the interrupts from the peripherals, but which are, in fact, controlled by the central processor. The use of the central processor as a resource in a similar manner to the use of input/output processors is possible. The CDC 6600 (CDC 60372600), which is an example of this type of structure, does not use interrupts but instead relies on a monitor processor which scans fixed locations in memory to detect when input/output operations are required and when these have been completed. The monitor is responsible for scheduling new work to the central processor and also the input/output operations to the appropriate peripheral processor.

2.1.2 MULTI-PROCESSOR CONSIDERATIONS

With the addition of more central processing units the responsibility for control of the system, under the interrupt structure, requires a selection method as to which processor should handle the interrupts as they arrive. The direct attachment of input/output processors and peripherals to a particular central processor, as used in the hierarchical structure of the Data General Eclipse M-600 (DG 014-000092), eliminates the need for a selection mechanism since each peripheral and

input/output processor can only communicate with the processor to which it is attached.

In contrast, an interrupt on the Burroughs B6700 is handled by the first processor which is available (Burroughs 1058633). The use of individual process stacks with a shared base, which contains control information, enables a single control program to supervise the operations of the system without concern for which processor on the system it is utilising. This seems to eliminate problems associated with the handling of input/output completion interrupts while still providing flexibility in processor scheduling. However the software structure used on the Burroughs is restricted because of the shared stack base which must be used to implement the operating system.

Under the CDC 6600 structure, the problem of which processor handles the completion of the input/output operation is eliminated, since the monitor will detect the change of state and carry out the appropriate scheduling operations. The utilisation of the peripherals and central processors may be lessened because a central processor or peripheral may wait for the monitor to detect the completion of, or a request for, an input/output operation.

2.1.3 DEFINING A CONTROL STRUCTURE

Both the interrupt and the monitor approach to handling input/output operations can be regarded as the first level of scheduling in the system. The interrupt implies that the currently scheduled task is suspended while tasks associated with the completion of the input/output operation and the scheduling of a new input/output operation are performed. This immediate scheduling of input/output routines implies that there is a dependence on the availability of a central processor for the successful completion of the input/output operation.

The monitor approach removes this dependence and suggests that the peripherals are associated with intelligent controllers. Scheduling operations are contained in the function of the monitor processor. The two level scheduling can be eliminated by the use of a monitor approach, since the central processor does not schedule new work based on the completion information.

The defining of control structures for input/output operations is an important design consideration since it affects the scheduling techniques used. Control structures for processor interaction and synchronisation are also important. The protection of shared resources

and the communication of control information between processors is required.

Communication of control information is performed using the same mechanisms as those for input/output operations. Shared resource protection is obtained by using synchronisation primitives. Multiprocessing utilizes similar primitives, so these will be discussed in section 2.3.

2.2 SUPERVISORS AND THE USERS

Since the introduction of the stored program computer, the utilization of the hardware has been an important consideration in their use, mainly due to the cost of the computer. In an endeavour to improve the hardware utilization, supervisor routines were developed. Initially, these were created to ensure a constant flow of work through the machine by attempting to eliminate the delays which occurred while the operator loaded the card deck for the next job step.

Owing to problems of user input/output routines reading past end-of-file markers, the supervisor routines were extended to include standard input/output facilities. As no method was available to enforce the use of the standard input/output routines, hardware modifications were made to create a supervisor and user mode. By restricting the use of input/output operations to the supervisor mode, the user was forced to use the standard input/output routines. The addition of address space separation of the supervisor routines from that of the user program eliminated the possibility of the user overwriting the supervisor routines. The use of an addressing scheme, which recognises the separation of address spaces, assists in satisfying this requirement.

2.2.1 THE EXTENDED SUPERVISOR

With the development of a supervisor mode and address space, it was now possible to provide specialised file structures and new functions to the user programmer. As the user programmer could no longer manipulate the storage media and the input/output peripherals directly, he could be given the feeling that he was working with a sophisticated computer system (a virtual system). Rosen in his paper on supervisory systems history says that the objective of supervisory system design is to provide a more ideal machine interface on a machine that is not ideal (Rosen 1969).

The creation of an extended supervisor (operating system) places its own requirements on the design requirements. A calling mechanism must

be provided to enable user programs to request the functions of the supervisor. Supervisor call functions (functions for calling operating system routines) are dependent on the addressing structure and the technique used to determine the mode of operation. The principal operations of a supervisor call function are the changing of the mode (not always necessary since some functions involve access to control data structures and consequently do not require the mode change) and the changing of the address space so the supervisor routines can access the control data structures.

The address change requirements also apply for procedure call within user programs particularly where structured languages such as Algol are used. The Burroughs B6700 uses the procedure call functions and its stack addressing structure to implement the call to operating system routines. The addition of a mode bit to the program status word for the routine enables the mode change to be accomplished in the procedure call, while the address space change mechanism is that used for normal procedure calls. In this approach the operating system becomes a set of procedures called by the user software.

The actions required for the processing of interrupts also involves the mode change and the address space change. This is also simulated in a procedure call on the Burroughs B6700 and consequently aids in the structuring of the procedure based operating system. The operating system executes as though it were part of the currently executing user program, although it cannot reference the user portion of the stack and the user data areas without using the stack vector.

2.2.2 OPERATING SYSTEM STRUCTURES

With the move from the supervisor routines, which ensured the flow of work through the system, to the operating system with its additional functions to ease the programmer's task there has been an increase in the structuring techniques used in the creation of operating systems. A common operating system structure is based around an interrupt system using a fixed location in memory for the address of the interrupt handling routine and another location to save the address where the interrupted program was executing. The supervisor call mechanism in this type of system is implemented in the same manner as the interrupts. Internally the operating system structure is based on routines which process the interrupt information or, for supervisor calls, process the parameters passed via the registers of the hardware. Data structures

within the operating system are used to maintain information on the system and to direct the flow of control within the supervisor. The IBM Disk operating system is an example of this type of structure.

The procedure based structure of the Burroughs B6700 eliminates the requirement to decode the user request since the user references the required processing routine as a procedure.

The use of interacting processes within the operating system have been implemented for the various management functions. Requests to the various management functions are made through message queues with the requesting process waiting for the response message packet.

With interaction being required between the management processes, the organisation of the processes into a hierarchy was used to assist in the design, "the verification of the logical soundness of the design and the correctness of its implementation." (Dijkstra 1968). In this structure the central level or kernel is responsible for scheduling, including the processes responsible for interrupt handling. The next level involves the management of program segments, while outer levels handle functions related to input/output and user processes.

An alternative hierarchical structure (Gagliardi 1975) is based on a central storage subsystem. The lowest level function in the storage subsystem is a processor manager with the outer levels concentrating on storage management. Management functions related to input/output (communication devices and unit record devices) and user processes are outside the central subsystem and are split into three subsystems (communications, spooling, and computational) at the same level in the hierarchy.

The Honeywell Level 6 has implemented a set of 64 interrupt priority levels (Honeywell AS 22). These levels can be associated with hardware devices but can also be controlled using a processor instruction. Each interrupt level has an associated pointer (stored in a fixed storage location) to an interrupt save area where the status for the process for the interrupt level is stored. Traps associated with a process are directed to the process using a trap save area pointer contained in the interrupt save area. The highest priority active level has control until a higher level is activated or until it relinquishes the active status. This hardware structure enables management process, in the hierarchical structure, to be directly associated with the peripheral for which it was responsible, without the use of a kernel to channel the interrupts to the appropriate management process.

Process scheduling and the separation of address spaces of the user and supervisor are important requirements in the construction of a computer system.

2.3 MULTIPLE USERS AND MULTIPROCESSING

The introduction of multiple user processes presented new requirements to the design of the computer. A significant requirement was the necessity to ensure that no one user can become permanently locked in execution on the processor. The solution involves some form of timer interrupt and program swap mechanism.

Timer interrupt facilities are usually provided through the same interrupt mechanism as that used for input/output. The type of interrupt provided varies from a simple regular frequency interrupt to alarm type timers. Software designed to use the timer facilities is dependent on the type of timer provided in the system. For a regular frequency interrupt timer, it is necessary for software counters to be maintained for the timing functions required.

2.3.1 PROCESS SWAP FACILITIES

With timer facilities available the problem of swapping user processes must be considered. On many machines the software is responsible for saving the necessary information and restoring the information when the process is to be resumed. Some machines provide hardware program swap facilities. The stack swap instruction of the Burroughs B6700 implements a process swap, since each process in the system is defined by a stack. The top portion of this stack is unique to the owner process while the base (operating system) portion and possibly intermediate portions are shared with other processes. Information on the status of the process is maintained in the stack and a descriptor for the stack stored in a vector whose descriptor is in the stack base.

In contrast, the LEV instruction of the Honeywell Level 6, when used to activate or deactivate interrupt levels, can cause the hardware to perform a process swap. An external interrupt to a higher interrupt level will also perform the process swap operation. The process swap operation causes the status of the currently active interrupt level to be stored in the interrupt save area for that level and the information in the interrupt save area of the interrupt level being activated to be retrieved. The process is, under this structure, managed by the hardware

with very little awareness of scheduling being required by the programmer.

The Burroughs B6700 mechanism, like those of the CDC 6600 and VAX 11/780, makes no scheduling decisions. They provide a mechanism to save and restore process environments. Scheduling decisions are regarded as being related to the software being implemented. The Honeywell Level 6 uses a priority based mechanism implemented in the hardware in its process swap mechanism and therefore eliminates the need for software scheduling algorithms. The Honeywell Level 64 (Atkinson 1974) has a hardware/firmware "Despatching Mechanism" which utilises a "process control block" to perform the systems scheduling and process swap operations.

The process swap and scheduling operations are an important component of any multiprocessing system.

2.3.2 ADDRESS SPACE SEPARATION

The separation of the supervisor address space from the user address space does not place the same pressure on the design of an addressing scheme as that of separating the user address spaces. Where the supervisor/user address separation is designed to ensure that the user cannot destroy information contained within the supervisor, the separation of user address spaces must protect users from each other. It may also provide for the sharing of program code and some data areas. Further extensions to the addressing scheme may be required to support multiple activity (multitasking or multiprocessing) within user programs. Addressing schemes are examined in Chapter 3.

2.3.3 MULTIPROCESSING

Introducing multiprocessing into a system not only places requirements on the addressing scheme but also on the control structures used. The use of multiprocessing implies independent control information while still sharing the address space of the parent process. The parent process may also require facilities to check on the status of its child processes and possibly to control the child processes.

The Burroughs B6700 provides many facilities for multiprocessing. These include facilities for initiating the child process and for controlling the process. The initiation facilities enable processes to be initiated as either a co-routine, an independent task (similar to initiation of a new program) or an asynchronous process. In all cases the created process has its own control information created, but each has a different relationship with its creating process.

Through the task attributes associated with each process it is possible for the parent to monitor resource utilisation of the child process and to control the operation of the child process. In addition, through event and interrupt facilities it is possible to cause changes in the processing pattern of the child process, and for the parent to be informed of changes in status of the child process.

2.3.4 SYNCHRONISATION PRIMITIVES

With multiprocessing in the system the necessity to have synchronisation primitives to ensure the integrity of shared data structures and to enable communication between processes presents another design requirement. Many synchronisation primitives have been developed. The simplest of these in operation are the test and set operator, the disabled spin locks and the enabled suspend locks. The test and set operation works on the basis of setting on a bit in memory while checking to see whether the bit was already set. The implementation of additional facilities within the operator are dependent on the machines on which they are implemented. Using the test and set operator it is possible to ensure that two processes do not modify the same data area or synchronise the operations of two processes. The disabled spin locks are used for implementing data integrity. They work by causing the processor to be disabled when another processor already possesses the lock. Because they disable the processor, their use is restricted to multiprocessor environments and when the synchronisation desired only requires short wait periods.

Enabled suspend locks are used to suspend the process while enabling the processor to continue servicing another process. The operating system gains control and queues the requesting process until the lock is released.

A further synchronisation primitive is the semaphore (Dijkstra 1968). The semaphore is a variable consisting of a count and a process queue. Two operators are used on this variable. They are a signal operator and a wait operator. The signal operator increases the value of the semaphore counter by one and if the counter had a negative value before the operation a process on the queue is enabled to continue. The wait operator decrements the value of the semaphore and if the resultant counter value is negative the process issuing the wait operation is queued. A positive counter value indicates the number of resources available while a negative value indicates the number of processes queued.

It should be recognised that some form of scheduling is assumed in the use of the semaphore synchronisation primitive and that only one process is released from the queue when the signal operator is issued.

The event mechanisms of the Burroughs B6700 provide a greater range of facilities. The event variable has two status fields and associated process queues. The status fields indicate the happened and available states of the event. Processes can wait for the event to happen or attach an enabled interrupt to the happened status of the event. When the event is caused (event happened status is set to happened), all enabled attached interrupts will have their associated code executed and all processes waiting for the event to happen are placed in a ready to execute state. The available status is used to provide a mutual exclusion facility. A process can procure the event (If the event is available then the process proceeds and the event becomes not available. If the event is not available the process is queued on the event.) and ensure that it has sole access to the object being protected by the event. The process releases the event when it has finished and one of the processes waiting to procure the event is given the event. Also, on the release of the event the happened status is caused and all waiting processes are initiated.

The use of message buffers as a synchronisation method has been suggested (Brinch Hansen 1973). This is based on the principle that a process will wait to receive a message from another process when it wishes to synchronise with another process. It is possible, using message buffers, to provide mutual exclusion, and operating systems have been implemented around the passing of request messages between processes.

Many primitives and synchronisation facilities have been created. Only those considered to be used extensively in existing systems have been examined.

2.3.5 VIRTUAL MACHINE ENVIRONMENTS

Multiprogramming and multiprocessing environments enable users to compete for resources which the operating system presents. The concept of virtual machines endeavours to present a machine level interface for many users. The major problems in the implementation of virtual machine environments are address translation and input/output operations.

IBM's VM/370 accomplishes the implementation of virtual machine environments to the extent that it is possible to execute any of the IBM operating systems including VM/370 under the control of VM/370.

The virtual machines created by VM/370 require an operating system to enable users to utilise the environment without having to code interrupt handlers and other facilities. Facilities have been added to VM/370 to enable operating systems, such as IBM's DOS/VS, to utilise the memory management facilities of VM/370 and consequently enable the removal of the address translation problems.

The provision of higher level functions in the virtual machine interface indicates that although conceptually it is desirable to provide the user with a duplicate of the hardware facilities, it can degrade the performance of the system for a particular user environment. At the same time, it must be recognised that by using the virtual machine concept it is possible to separate different operating environments (for example, batch processing separated from transaction processing).

Although not presented as a virtual machine concept, the Burroughs B1700, in presenting virtual processors (S-machines), does not implement virtual machine environments for user programs. In contrast to VM/370 and other virtual machine monitors, the Burroughs B1700 presents a different machine interface to each user. This interface is intended to match the requirements of the user and the programming language he is using. The operating system executes on one of the S-machines (the micro code of the B1700 ensures that the correct machine interface is presented) and appears to be part of the machine interface presented to each user program. This means that each virtual processor (S-machine) does not require an operating system to execute within it to enable the user program to execute.

ICL's VME/B and VME/K operating systems are presented as being designed around the virtual machine monitor, which presents common facilities for the function processors (specialised operating systems for batch, transaction processing and other environments). This implementation method removes the problems associated with normal virtual machine implementations while still enabling the flexibility of using specialised facilities for the different operating environments.

2.4 CONCLUSION

In the design of a computer system, there are core elements which must be considered carefully. These elements form the nucleus for the computer.

The recurring element is the addressing scheme which is examined

in more detail in Chapter 3. The structure of the addressing scheme influences the operation of every facility in the system. Other elements exposed in this chapter include the control structures for processor interaction, process and task management, peripheral interaction mechanisms and considerations for machine environments.

CHAPTER III

ADDRESSING SCHEMES

Addressing schemes were an early development in the history of computing. With the development of the Manchester Mark I (Lavington 1978), the concept of a "one-level store" emerged. Many different addressing schemes have been developed since the implementation of the Manchester Mark I.

This chapter examines addressing schemes which have been developed, and discusses the facilities available in the schemes.

3.1 MEMORY ADDRESSING SCHEMES

The first memory referencing schemes, which were designed to suit the physical characteristics of the storage medium (for example, delay lines), treated the storage as a linear array with the program capable of referencing all the memory cells. With the introduction of supervisor routines and the requirement to protect these routines from modification by user programs, the limitations of using the linear addressing technique were first realised. The use of a restricted region, to restrict user access to the area containing the supervisor routines, was implemented. Using the restricted region concept, the memory is still addressed as a linear array, but the user's reference into the supervisor routine area is restricted to reading and executing.

Restricting the user's access into this area to one instruction, the supervisor call, ensures that the user enters and executes the code of the supervisor routines from the correct start location. The IBM/360 architecture utilises storage keys to implement the concept of restricted regions. As implemented on the IBM/360 multiple user environments are possible, however execution of another user's code is not restricted. The DOS operating system uses the storage keys to implement multiple user partitions.

3.1.1 BASE-LIMIT REGISTER

Restricted regions do not provide all the protection necessary for multiple user environments. The base-limit register approach (Burroughs B3700) enables multiple user environments where memory space is available. By the use of a virtual storage mechanism as

implemented in IBM's DOS/VS operating system, the memory restriction of the Burroughs B3700 on the number of users could be eliminated.

In the base-limit register scheme, each user program is allocated a contiguous block of memory. The low end address of the memory block is stored in the base register and all memory references of the user are relative to the address in the base register. The limit register indicates the size of the memory block and the users memory references are checked against this limit to ensure that his memory reference is within the allocated block. It is not possible for the user to reference any memory locations outside the block allocated to him.

The CDC6600 uses a control point concept, which is a slightly modified base-limit register scheme. It is designed to assist in program swap operations, as each program active in the system is known by its control point.

The base-limit register scheme, even with the use of some virtual storage scheme, is restricted either by the physical memory size or the size of the virtual storage pool. Also, the nature of the base-limit scheme requires the loading of a user's program into the storage. DOS/VS, with its implementation of virtual storage, loads programs into the storage. This program load can cause unnecessary paging operations and degradation in the performance of the system.

3.1.2 INSTRUCTION AND DATA BANKS

To overcome the restrictions of memory size in their multiprogramming system, the Univac 1100 series uses the concept of instruction and data banks. These are storage blocks defined by a base-limit method, except that the limits defined for each bank are defined in terms of lower and upper relative address limits. The user address space is still regarded as a linearly addressable array, but it is possible for some portions of the address space to be invalid. Banks of the user program do not have to be contiguous and the user can select which banks are available for addressing. These banks may even have overlapping address spaces but access to the banks, in such a situation, has restrictions to ensure successful instruction execution, since the overlapping of address spaces does not imply common contents.

This structure provides the user with a larger address space, however the system requires that all banks, currently being referenced, be in memory.

3.1.3 PAGED MEMORY

Another addressing scheme is based on fixed length pages. The user references his address space as a linear array but the hardware uses a page table to translate the user's address to a physical address. A page fault interrupt is generated when the user address attempts to reference a page which is not in his address space or which is currently not in memory. By handling the interrupt, the operating system can bring the requested page into the physical address space, modify the page table and restore the interrupted program so that it re-executes the instruction on which the page fault occurred. This structure enables a greater number of users to share the available physical address space since only those pages currently required by each user need to reside in the memory.

The exact method of implementing the address translation can restrict the number of users and the flexibility of this scheme. For example, the DG Eclipse S130 utilising DG's MAP1 memory map processor is best suited for a two partition (foreground/background) implementation.

3.1.4 SEGMENTATION

User sharing of code and data is restricted in the paged memory scheme. The Univac 1100 banks allow for user sharing, however each user must reference the bank as the same portion of the address space.

Segmentation schemes enable greater flexibility in the sharing of code and data, as well as removing the linear nature of the user address space. Segments are variable size, linearly addressed arrays of memory words defined by a descriptor which defines the access rights and the address limits.

On the Burroughs B6700, the segment descriptors are contained in the stack of the process. The stack of the Burroughs B6700 is designed for multiple levels (activation records) to support the semantic requirements of block structured languages such as Algol. In the stack base (level 0 of the stack) are the descriptors for the operating system code and data segments. The second level in the stack (level 1) contains the segment directory for the user program's code segments. Any user data segments required have their descriptors in the stack level associated with the block in which they are declared. The segment, in this structure, is addressed by a level number/displacement from level base combination which is also used to address variables in the stack. With the exception of the code segments which must be referenced through the lower two levels of the stack, the segment descriptors may be placed

anywhere in the stack. More than one descriptor can indicate the same segment and as a consequence the sharing of segments is implemented. Resultant management problems occur in the implementation of memory management.

In contrast, the Honeywell Level 64 uses a segment table structure for the segment descriptors. Each process has access to its own segment table, a segment table for the process group of which it is a member and a public segment table. There is only one descriptor for each segment and each descriptor is referenced through one of the segment tables by a segment identifier. Any shared segments have their descriptors in either the process group segment table or the public segment table.

As segments are variable length there is a possibility that some portion of the available real memory is not used. This is certainly not the case where the memory size is an exact multiple of the page size.

3.1.5 PAGED-SEGMENTS

In an endeavour to reduce the amount of real memory not used, paged-segments have been used. This structure divides the segments into a variable number of fixed length pages. By treating the real memory as a set of fixed length pages, the task of the memory management functions is simplified although the address translation mechanism is more complex. The segment descriptor points to a page table for the segment and the segment descriptor is maintained in a segment table in most implementations of this structure.

The ICL 2900 series utilises the paged-segments structure. In their implementation each process has its own segment table which translates the process's segment number to a system segment number. This number indexes the system segment table which contains pointers to the page tables for the segments. To increase the performance of the address translation mechanism the processor contains a set of current page registers. These registers contain the process identifier, process segment number, page number, access rights and real address for the most currently referenced pages and are scanned at the same time as the reference through the address translation tables. The process number is an integral part of a segment's address.

In this structure, in a similar manner to the paged structure, the user can be unaware of the paged nature of the memory. The paged-segmentation scheme provides for the user the address structuring required while allowing for the benefits of a paged structure for memory management.

3.2 PERIPHERAL AND PROCESSOR ADDRESSING

Peripherals and processors have, in most hardware designs, been treated as special cases when it comes to addressing and manipulation. This approach has caused the inclusion of special instructions to manipulate processors and peripherals. This does not mean that special operators are not required for peripheral and processor manipulation, but that for the purpose of status monitoring or passing of control data, instructions have been added to the machines repertoire because this information cannot be addressed through the memory addressing scheme.

3.2.1 PHYSICAL ADDRESS AND DEVICE CODES

The peripheral addressing scheme used on the IBM/370 system represents the physical structure of the system. Each peripheral is addressed by three four bit digits. The first of these represents the channel to which the peripheral is attached and the other two digits indicate the unit number. Some restrictions are placed on the addresses because of the physical requirements of the system for the placement of peripherals and their controllers.

In contrast, the six bit device codes used on the Data General Eclipse S/130 are preassigned to the peripherals. This removes the requirement for peripheral addresses to be generated into the operating system although the peripherals attached to the system must still be generated into the operating system so that it is aware of what is attached.

3.2.2 MEMORY PERIPHERAL ADDRESSING

The PDP-11 family of processors treats peripheral status and control words as memory locations. Consequently no special instructions are implemented for the manipulation of the peripherals. Since the address relocation scheme is based on 4K word pages and the peripheral addresses are in the top 4K of the 128K word address space, the protection against user access is implemented through the address translation mechanism.

This style of addressing for peripherals eliminates the requirement for specialised instructions to read the peripheral's status or to initialise control words. The normal instructions of the processor can be used to test the status and in the case of the PDP-11 family initiate the input/output operations.

3.2.3 PERIPHERAL TYPE DETECTION

A feature of the Burroughs computers is the capability of the software to examine the input/output paths to determine what peripherals are attached. This approach provides additional flexibility in the reconfiguration of the system both at initial program load and when problems occur. Burroughs operating systems are designed to configure their environment based on the information gained by examining the input/output paths and options entered by the operator.

3.2.4 PROCESSOR ADDRESSING

Processor addressing has only been required since multiple processor systems have been implemented. In this environment the context of addressing has been to enable a program to determine which processor it is currently utilising and to indicate which processor an interprocessor interrupt is intended for. This philosophy is applicable since the interrupt routine can determine from locations in memory the reason for the interrupt. The IBM/370 system provides control and status reading functions to enable the monitoring of one processor by another.

The requirement for processor addressing is of limited importance, even in a multiprocessor environment, since each processor is responsible for the management of the resources allocated to it. The hierarchical relation of host processor to input/output processor requires only that the central processor can address individual input/output processors. This can easily be accomplished through the peripheral addressing structure.

It is only when co-operating processors, working outside any form of hierarchy, wish to interact that processor addressing is required. Since these interaction requirements tend to be of an exception type then the interrupt mechanism provides an easy implementation technique.

Scheduling requirements tend not to require processor addressability since each processor is either responsible for the selection of its own work or is a slave of a master processor.

3.3 DISK STORAGE ADDRESSING

The addressing of information on disk storage mediums has become almost standard for all manufacturer's equipment. Some differences are apparent in terminology although the final implementation is very similar. Positioning on a disk storage medium is determined more by the physical

characteristics of the device than the information being stored. The three qualities which form a disk address are primarily the cylinder, surface and sector. The cylinder selects a set of recording track on a vertical basis across all available recording surfaces. The surface selects a recording surface while the sector selects a particular block upon the circular track.

3.3.1 FILE STRUCTURING

A user regards a file as a set of records which are accessed by some mechanism. Within the physical structuring of the file these records may be grouped together to form a block. The block is the unit of transfer between the storage medium and main memory. In some file management systems the file can have a grouping of blocks into an area with more than one area composing the complete file.

The size of the block is usually designed to be a multiple number of records as well as disk sectors. This provides for a reduction in the number of disk accesses to retrieve the data while not leaving unused disk space. The area is a physical grouping rather than a programmer requirement. The area is a set of contiguous sectors, surfaces and cylinders which the file management system has allocated to the file.

3.4 CAPABILITY ADDRESSING

Since the design of the Manchester Mark I computer (Lavington 1978), implementation of one-level store addressing techniques has been researched and developed. Many schemes, such as codewords and descriptors (Iliffe 1972), segment and page tables, and symbolic naming (Gordon 1973) have been used and researched.

The capability (Fabry 1974) was developed to act "like a ticket authorizing the use of some object". It is a generalisation of addressing and protection schemes such as codewords and descriptors. The scheme has been extended to include all systems objects (memory, processors, input/output devices etc.)(Dennis and Van Horn 1966) and to enable explicit manipulation of access control by non-systems programs. "The idea is that a capability is a special kind of address for an object, that these addresses can be created only by the system, and that, in order to use any object, one must address it via one of these addresses" (Fabry 1974).

The protection facilitated by the use of capabilities has created considerable interest. However, the advantages of using capabilities as

a basic component of the address of every object is developed by Fabry.

3.4.1 CAPABILITY AS ABSOLUTE ADDRESS

The capability acts as an absolute address for an object and is context independent in its interpretation. In effect, the capability is the only method for referencing an object in the computer system.

A capability is constructed from access rights and a unique number which is generated at the creation of the object. The unique number is never re-used in the system even though the object may become non-existent.

The capability addresses a virtual object which is mapped onto a physical object by appropriate address translation mechanisms such as the hash tables suggested by Fabry or associative registers similar to the ICL 2900 Series address translation mechanism.

In many systems this translation is used in relation to memory segments only, however the extension to all objects of the computer is possible (Dennis and Van Horn 1966).

3.4.2 CAPABILITY INTEGRITY

As capabilities are generated by the system, it is essential that the user is not capable of modifying or generating his own capabilities. Two approaches have been used to preserve the integrity of the representation of capabilities.

The partition approach used in the Cambridge University CAP system (Needham et al 1977) has the advantage that it is easier to implement. In this approach there are capability segments and registers and data segments and registers. It is not possible for capabilities to be stored in data registers or segments. By restricting the operations possible on capability registers and segments it is possible to ensure their integrity.

Since data structures often require both data and addresses to be stored in the same structure, the partition approach requires two segments with appropriate capabilities to represent such structures.

The alternative approach is to use the tagged word architecture used on the Burroughs B6700 and the Basic Language Machine (Iliffe 1972). The advantages (Feustal 1973) enable the same preservation of integrity while enabling greater flexibility in the use of the capability.

3.5 CONCLUSION

Addressing requirements and the structures used to meet these requirements are examined in this chapter. The structures examined include those for memory, peripherals, processors and disk storage devices. In addition, the capability addressing scheme was discussed.

Capability addressing can provide the facilities to address program segments, file records, peripherals and processors. It is on capability addressing that the nucleus addressing scheme of this thesis is based.

CHAPTER IV

THE NUCLEUS CONCEPT

The examination of computing structures and concepts has revealed that there is a set of requirements which need to be satisfied during the design of a computer. In meeting these requirements the nucleus of a computer is established.

The nucleus of a computer defines the structural characteristics and the manipulative functions required for controlling the nucleus. The processing characteristics of the computer are not established in the nucleus although the structure and functions of the nucleus may affect the operational characteristics of processing functions.

In this chapter the requirements for the nucleus are discussed and a nucleus structure is presented.

4.1 THE NUCLEUS REQUIREMENTS

The design requirements which the examination of computing structures revealed are primarily associated with the addressing structure. These requirements are those which are used to assist in defining the nucleus of a computer. Not all the requirements listed below are essential for all computer systems since some systems are designed for specialised purposes (Ozkarahan et al 1975). The objective of this thesis is to demonstrate that there are basic requirements for the design of a computer and that these can be met through the design of a central structure, the nucleus, which will form the basis of a more generalised computing structure.

The requirements derived from the examination are :-

- 1) an addressing structure which facilitates software structuring including data structuring,
- 2) an addressing structure which facilitates protection in a multiprogramming environment,
- 3) an addressing structure which enables sharing of resources (memory segments, peripherals, etc),
- 4) an addressing scheme which facilitates flexible address space changes,
- 5) an addressing scheme and control functions to enable virtual machine environments to be established without excessive overheads,
- 6) an addressing scheme which enables flexibility in hardware configuration,

- 7) control structures which enable flexibility in the use of the processors and peripherals, and
- 8) a structure to enable the use of the distributed processing capabilities of intelligent peripherals, terminals and networks.

The nucleus of a computer is more than an addressing structure with control functions for peripheral and processor use. The nucleus also includes facilities to assist in the creation of software which will use the computer. These functions are summarised in the additional objectives.

The additional objectives for the design of the nucleus are :-

- 1) the recognition of the process as a manageable entity,
- 2) facilities to assist process scheduling,
- 3) the relating of both internal and external interrupts to the appropriate handling process,
- 4) the elimination of the duplication of tables for the control of resources within the computer system, and
- 5) to enable the construction of a variety of operating system structures on the same basic hardware structure.

The nucleus of a computer is composed of :-

- 1) the addressing scheme,
- 2) the processor and peripheral interaction mechanisms (input/output structures and interprocessor interrupts), and
- 3) the operating system kernel (task scheduling and interrupt handling mechanisms).

The structure of the latter two components can be assisted by the structure of the addressing scheme. This is shown in the development and implementation of the nucleus scheme presented here.

4.2 AN ADDRESSING SCHEME

The techniques used for the proposed nucleus addressing scheme are based on the principles of the capability addressing scheme examined in section 3.3. The capability is the access ticket for any object within the system and it can only be created and maintained by the system.

Nucleus capabilities are designed so that it is possible to detect, from the capability, the type of object being addressed. In this manner additional security can be maintained over the capabilities. For example,

FIGURE 4.1 THE CAPABILITY STRUCTURE

Tag	Type	Access	Identifying Number	Length	Displacement
-----	------	--------	--------------------	--------	--------------

Tag :- Distinguishes the capability from data and other items in the system (2 bits).

Type :- Indicates the type of object addressed via the capability (3 bits).

Access :- Controls the use of the object addressed via the capability as well as what can be done with the capability (4 bits).

Identifying Number :- Unique Number within type. This number is structured depending on the type (Note).

Length :- Specifies the length in words of data which the capability addresses (Note).

Displacement :- Used where the object allows for portions to be addressed. For example storage pages (Note).

Note :- The number of bits required in the capability for these fields is dependent on the type of the capability.

TABLE 4.1 CAPABILITY TYPES AND STRUCTURE

TYPE		FIELD SIZE IN BITS		
Code	Name	Identifying Number	Length	Displacement
0	Physical	2	10	10
1	Functional	9	-	-
2	Storage	10	12	12
3	Processor Type	3	3	3
4	Peripheral	10	-	10
5	Processor	6	-	10
6	Process	18 (Note 1)	15	15 (Note 3)
7	File	18 (Note 1)	15	15 (Note 3)

Note 1) Process id 10 bits 1024 processes
 Segment id 8 bits 256 segments

2) The Page Size is 1024 words.

3) Displacement

5 bits for Page identification

10 bits for Word address

the storing of a capability on a removable storage medium (magnetic tape or disk pack) can be done provided the capability is addressing an object within the storage medium.

The structure of the capability for the nucleus is dependent on the type of object being addressed. To distinguish the capability from other objects which can be stored in data structures the tagged architecture approach is used, the basic tag being extended by the use of a type field as used by Illiffe (1972) in his Basic Language Machine. As the tags and types that are relevant to the operation of the nucleus are of importance in this discussion, the extension of this technique to other areas, other than object addressing, can be obtained from Illiffe's work.

The capability (Figure 4.1) consists of the identifying tag, the type field, a unique identifying number and the access key. Included in the capability are a size or length field and a displacement. These fields assist in restricting the addressability of objects. Their use is discussed later in this section.

Each capability type addresses objects with their own characteristics. The address requirements for each capability type are given in Table 4.1. The capabilities presented are based on a 32 bit word length with the capability being either a double or single word. A word size of 32 bits was chosen because it consisted of four 8 bit characters. Using modifications to the capability structure it is possible to address byte or bit arrays with reasonable ease.

Since capabilities can only be created or destroyed by the nucleus, it is essential that all items addressable through the nucleus be made available to the task first initiated. During the process of loading this task capabilities for the storage the task will use must also be initialised.

4.2.1 PHYSICAL CAPABILITIES

The physical capabilities provide the mechanism for all objects to be addressed by the initiation task. System initialization is dependent on the processor structures used. These capabilities enable the task to reference the storage capabilities (Figure 4.2), the function capabilities (Figure 4.5), the processor type capabilities (Figure 4.3) and the peripheral capabilities (Figure 4.4). The physical capability used to address an object's capability is the only physical address usable for the object. It is a physical address since its

FIGURE 4.2 MEMORY AND PERMANENT STORAGE ACCESS STRUCTURE

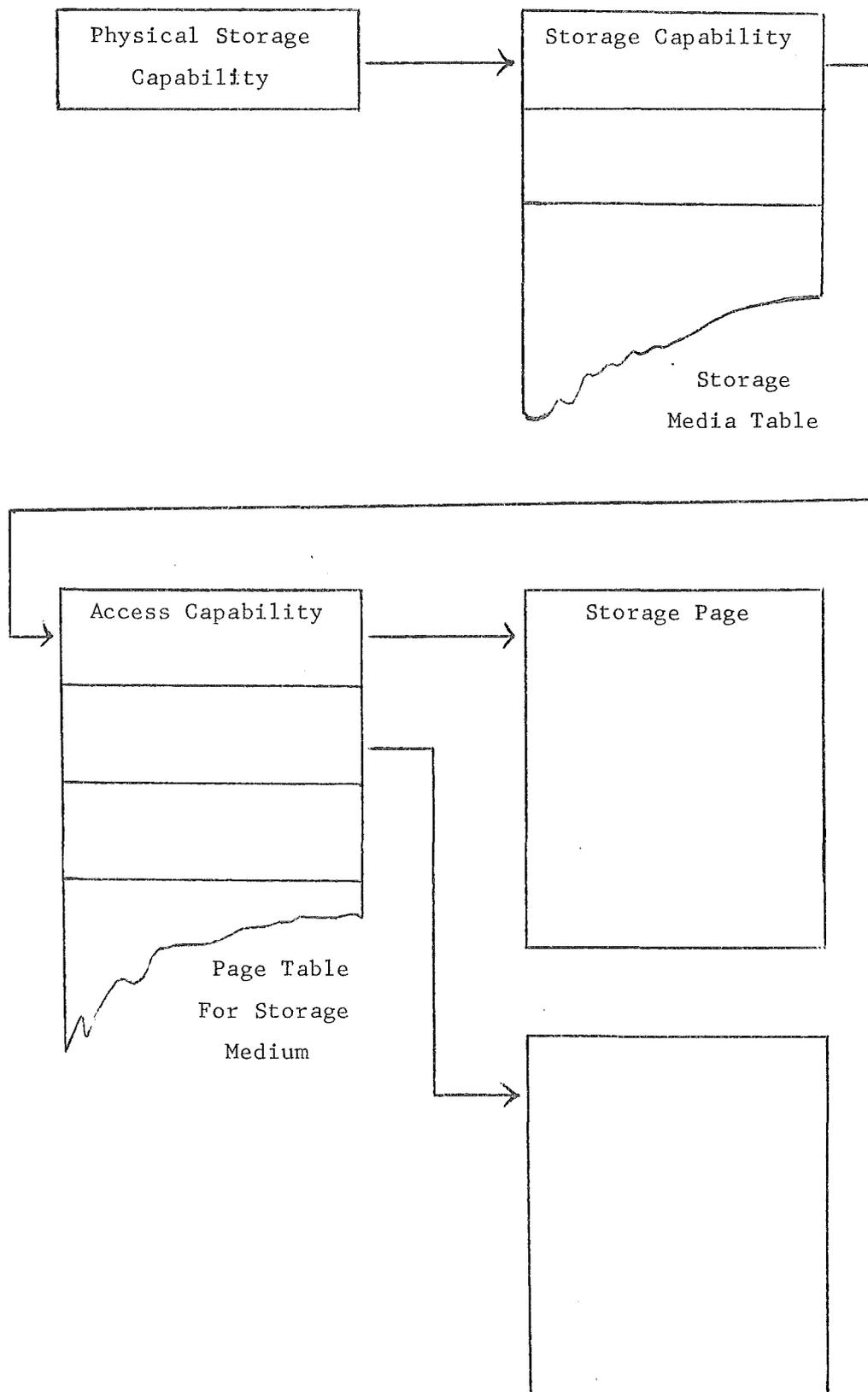


FIGURE 4.3 PROCESSOR ACCESS STRUCTURE

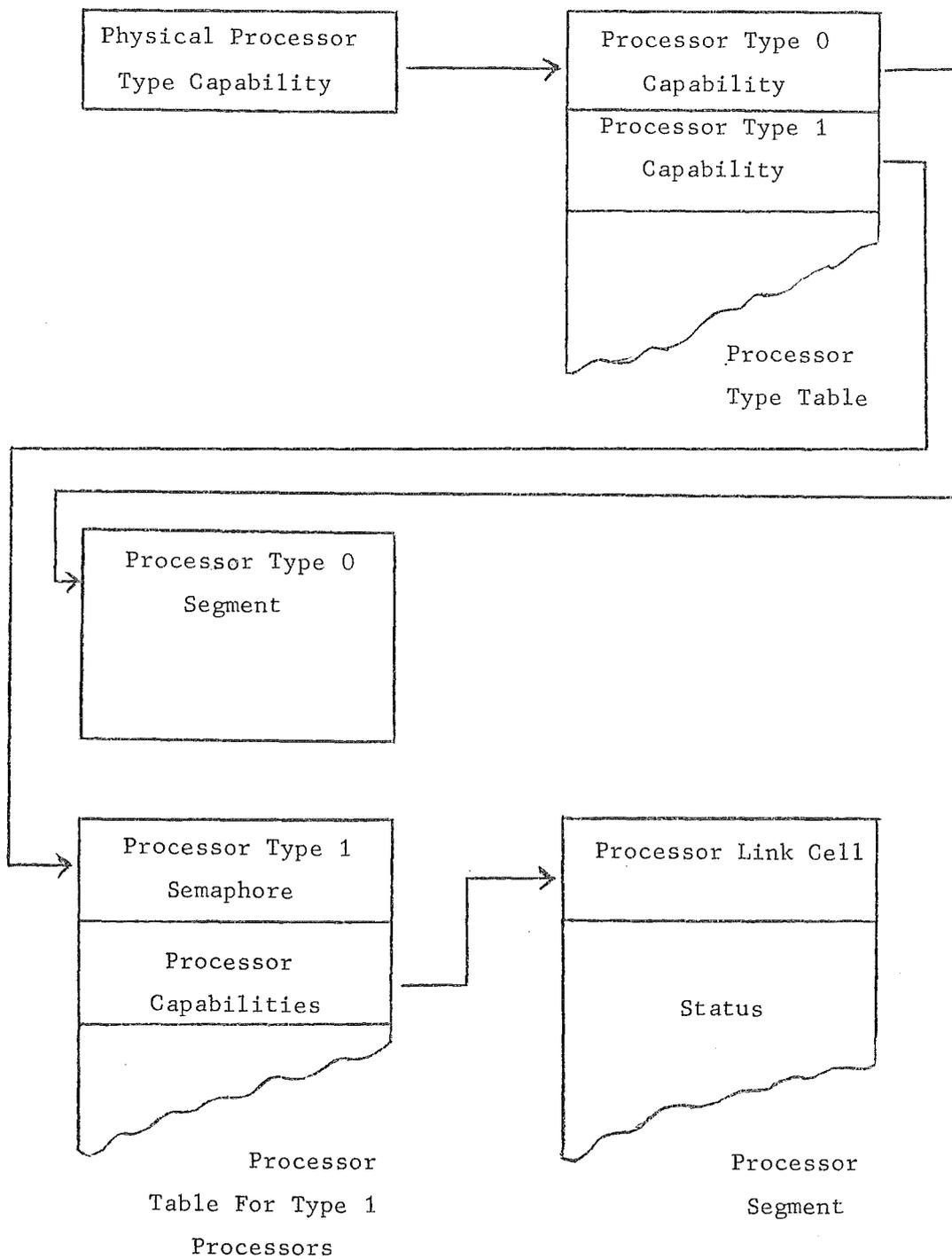


FIGURE 4.4 PERIPHERAL ACCESS STRUCTURE (REMOVABLE STORAGE)

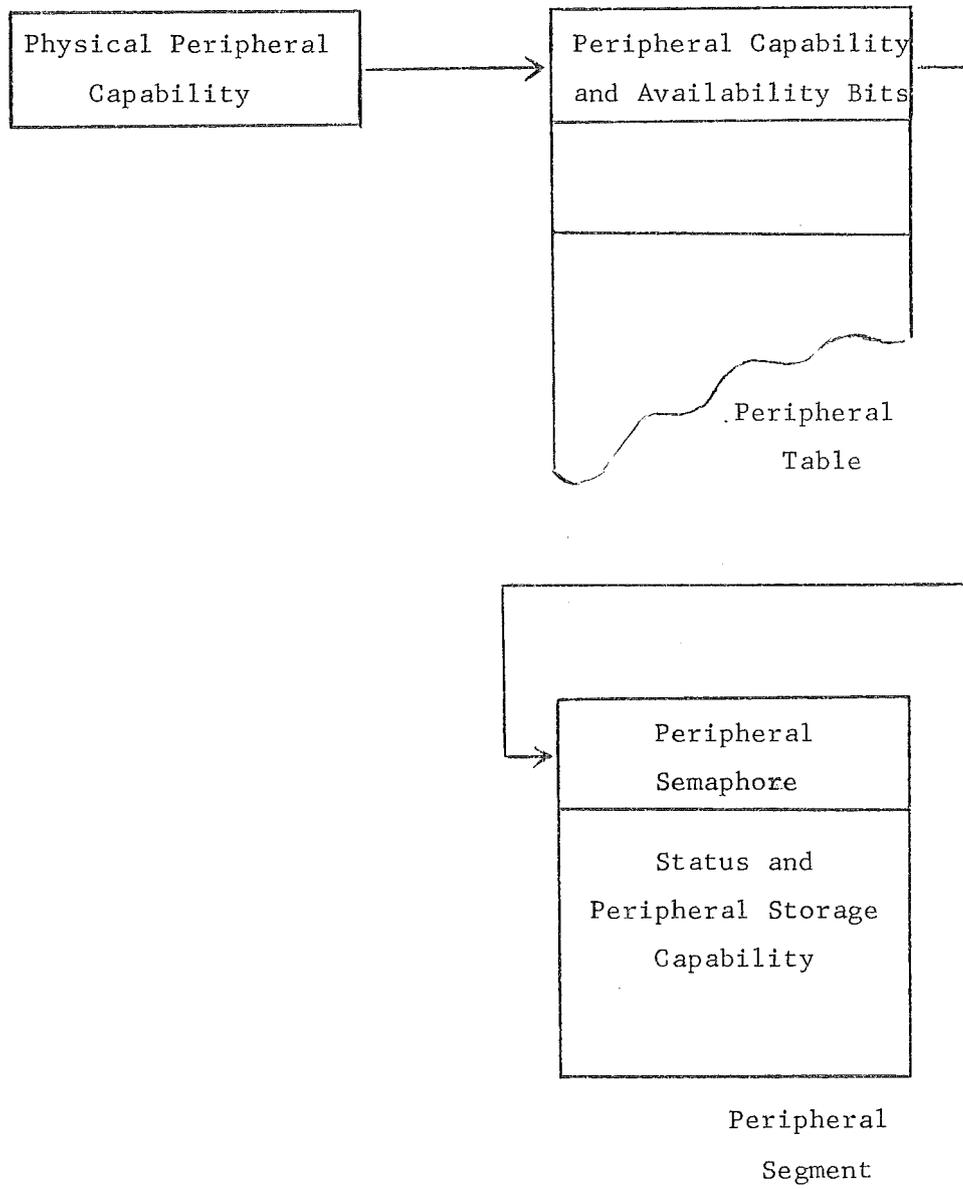
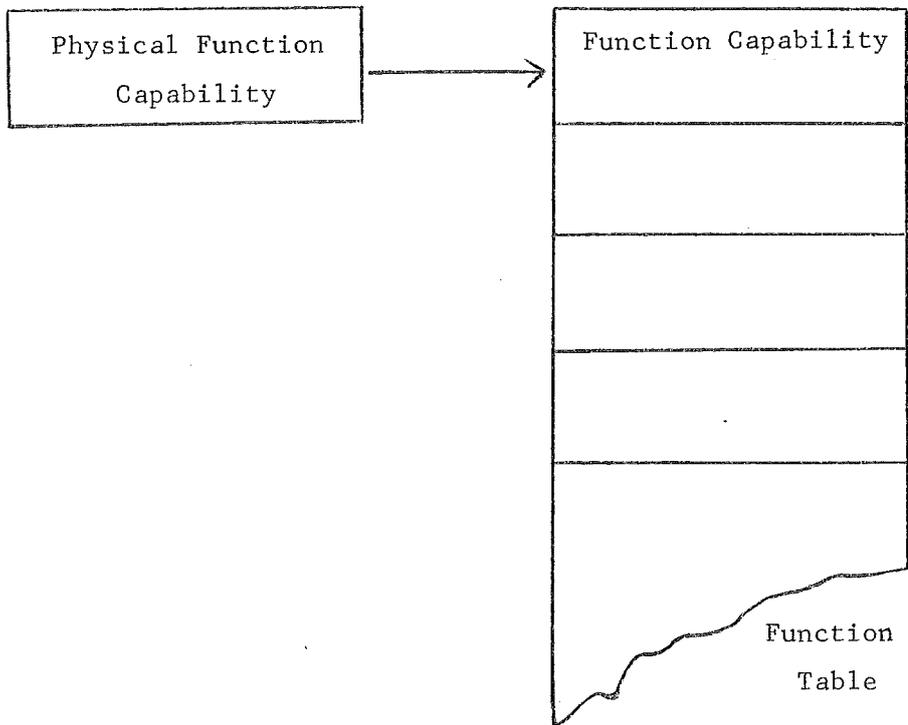


FIGURE 4.5 FUNCTION CAPABILITY RELATIONSHIP



interpretation is a function of the hardware rather than an associative addressing function (that is, addressing through a match with a memory cell (Hanlan 1966)) or table translation function.

The length field of the physical capabilities indicates the number of capabilities for the category addressed through that capability. For example, the length field of the physical capability for peripherals indicates the number of peripherals which may be attached to the system.

4.2.2 STORAGE ADDRESSING

The physical storage capability defines the media table, a set of capabilities which define fixed storage media available to the system. The capabilities of the storage media table are addresses for the page tables of the media available to the system. The media table defines the hierarchy of the media for the translation of the process and file virtual capabilities.

The page table entries contain the type, access, identifying number and the page identification portion of the displacement for the file or process whose page is stored in the associated storage page. The table entries form the associative lookup registers for the address translation logic. The movement of pages from one level of the storage hierarchy to another can be performed either by hardware facilities or software routines depending on the design of the processing system and the type of storage media involved.

By using the Null tag for entries in the media table and page tables it is possible to indicate the unavailability of a storage media or storage page to the software.

4.2.3 PROCESSOR ADDRESSING

The physical processor type capability defines the processor type table, a set of capabilities which define the type of processors available on the system. The processor type capabilities are addresses for the processor type. The control segment contains a semaphore call used for the nucleus scheduling (Refer to Chapter 5.3) and the processor capabilities. The processor control segment, referenced via the processor capability, contains a link cell and the status information for the processor. This structure would primarily be used for monitoring and controlling processors in the system.

4.2.4 PERIPHERAL ADDRESSING

Peripheral equipment represents a dynamic resource in the system. The peripheral table referenced through the physical peripheral capability contains capabilities and availability information for peripherals associated with the system. Each peripheral has an associated control segment containing a semaphore cell for requesting use of the peripheral (Refer to Chapter 5.2), status information and a storage capability for referencing the data which the peripheral may store or retrieve for the system.

4.2.5 FUNCTION ADDRESSING

The final physical capability, the physical function capability, references a function table. This table consists of function capabilities for the control functions provided in the nucleus (Refer to Chapter 5).

The address translation algorithms are shown in Appendix 1. As the algorithms demonstrate the storage media table, processor type table, processor type segments and function table are not physically identifiable objects but are part of the addressing mechanism.

4.3 FUNCTIONAL CHARACTERISTICS

The addressing scheme provides, through its structure, facilities which assist in the structuring of the system components. The facilities are provided by the recognition in the addressing scheme of the structure that is in the system components (hardware and software). The recognition then enables the system to use the structure to advantage.

The separation of address spaces by means of capability possession enables the swapping of address spaces to be based on the use of a codeword (the capability for segment zero, the control segment, for the process). By passing to a processor the codeword for a process, the address space for the processor operations is defined.

4.3.1 ACCESS FLAGS

The use of the access flags provides the mechanism to restrict the use by a process of the objects being addressed. The access flags supported by the nucleus are Read, Write, Execute and Store. Table 4.2 lists the access flags which are valid for each capability type.

The Read and Write access flags indicate that the object being addressed by the capability can be read or written by the process holding the capability. The Read access flag being on indicates that

TABLE 4.2 ACCESS FLAGS APPLICABLE FOR CAPABILITY TYPES

CAPABILITY TYPE	ACCESS FLAGS
Physical	R/S
Functional	E/S
Storage	R/W/S
Processor type	R/E/S
Peripheral	R/W/E/S
Processor	R/E/S
Process	R/W/E/S
File	R/W/E/S
R	Read
W	Write
E	Execute
S	Store

NOTE :- The access flags, which are not listed as applicable for a capability type, are always in the most restricted state. That is, if the write access flag is not listed it is not possible to write to the addressed object.

the process is not able to examine the contents of the addressed object. The Write access flag being on indicates that the process cannot alter the contents of the addressed object.

A process segment would use the Write access flag to indicate that the segment was to be read only. If both the Read and Write access flags are on the process possessing the capability can only reference the information in the capability. In combination with the Execute access flag the Read and Write access flags can be used to enforce an execute only status for a segment.

The Execute access flag indicates whether a file or process segment contains executable code or whether the owner process of a functional, processor type, peripheral or processor capability may request use of the object addressed. The Execute access flag being on enables execution of file or process segments or the request to use a function, peripheral or processor. The use of the Execute access flag for functions, processors, processor types and peripherals enables the creation of monitor tasks which can examine the status of these components but which cannot interfere in the use of the component.

The Store access flag restricts the use of the capability. If the Store access flag is on then the process owning the capability can use the objects it addresses depending on the other three access flags but it cannot copy the capability or give the capability to another process. This access flag enables a supervisor process to give a process a capability and know that the process will not pass the capability on to other processes. The supervisor process then knows that it has complete control over the allocation of the capability to other processes.

The access flags combined with the capability addressing principles (only the nucleus can create new capabilities and processes can only address objects for which they hold a capability) provide the mechanisms for security and protection. A process can only use objects if it holds the capability for the object and even then its use of the object is controlled by the access given to it by the supervisor process. Each process in the system can be given its own unique address space or be allowed to share part of its address space with another process.

4.3.2 SYSTEM RECONFIGURATION

Through the table structure of the nucleus addressing scheme the dynamic alteration of the system configuration is possible. The use of the Null tag word in the storage media and page tables has already

been mentioned as a technique to indicate the unavailability of a storage medium or page. The contents of the word could easily be used to indicate status information about the unavailable medium or page. This implies that some logic exists at the media table level which can set the entry to the null tag word should the storage media for some reason not be available to the system. This does not imply dynamic reconfiguration of the storage system when a storage component is removed because for the successful execution of the system it would be necessary to remove from the medium any process or file segments which may be stored and which may be necessary for the successful execution of programs in the system. The fact that the addressing structure enables storage media to be removed and faulty storage pages to be tagged enables the system to continue without having major system reorganisation problems. The software can be written so as to know what storage it has available by examining the capabilities and the storage page tables.

System reconfiguration is made possible through the table structure. Any component which is removed can have its table entry changed to the null word. By the inclusion of a facility to provide system separation, as is available for Univac 1100 series (UP 8094), even greater dynamic reconfiguration is possible and for a multiprocessor system, it is feasible to run the system as either a single system or as multiple systems. Reconfiguration of the system was not considered a design objective, however it is a function which could be provided.

Peripheral devices are an area where dynamic reconfiguration, removal and restoring of the device or storage associated with the peripheral is required. The nucleus addressing structure enables this dynamic peripheral movement. Chapter 5.2 examines the structure with respect to peripheral handling.

4.3.3 FUNCTION FACILITIES

The inclusion of the functional capability provides a mechanism for the inclusion of specialised management functions. Chapter 5 presents a set of functions for scheduling and peripheral management which are accessed through these capabilities. The inclusion of the functional capabilities provides for easy extension of facilities within the nucleus and additional flexibility in constructing software to use the system.

If the functional facilities of the nucleus were provided through

an instruction type interface, restrictions placed on their use would have to be based on a supervisor/user mode status flag. The functional capability removes the requirement for privileged instructions and supervisor/user mode, since access is restricted to the processes which hold the appropriate capability. In addition, it is possible to reference the functions provided by the nucleus through a procedure call mechanism and thereby remove any difficulties associated with implementing function call and return facilities.

A further advantage is that supervisory routines can give either the capability for a nucleus function or for the supervisor's own processing routine to a dependent process. This enables virtual machine monitors to be created for the nucleus without the problems of simulating supervisor/user mode and handling of privileged instructions. Since a process executing on a nucleus based system can only hold virtual addresses for storage, unless it has been given the physical storage capability, and its knowledge of addressable objects is restricted to the capabilities it possesses, the storage and peripheral mapping problems associated with implementing virtual machine environments are also eliminated.

4.3.4 FLEXIBILITY OF NUCLEUS ADDRESSING STRUCTURE

Because any capability is the address of an object (possibly with restricted access), it is possible to replace one capability by another provided the same access rights are available through the replacement capability, and the structure or operation of the addressed object is maintained. For example, a function capability can be interchanged with a file or process capability of a segment containing a procedure, suitable for replacing the function. Extending the concept of replacing capabilities (replacement of use rather than replacement within its position in the addressing structure), the apparent duplication of the addressing structure by using process capabilities is feasible.

The co-existence of supervisors and virtual machine monitors can be implemented through duplicating the address structure. Precise implementation of the co-existing supervisors would depend on the planned objectives. The techniques used can involve simple management routines to alter the real address tables, shared resources such as peripherals and processors (this sharing is assisted by the functions presented in Chapter 5), and the dedication of resources to a particular supervisor or virtual machine monitor. Resource dedication is achieved by placing

the capability for that resource in the duplicated addressing structure.

The addressing scheme presented does not limit the software implementations which can be created to execute on a nucleus based system. Any restrictions on software structure would be caused by processor architectures used in the system. The design of the processors for the system will affect the facilities made available to the system user.

4.4 CONCLUSION

In this chapter the requirements for the design of a nucleus have been presented. An addressing scheme which aims to satisfy these requirements has been presented and some of the functional characteristics of this scheme have been examined.

The scheme meets the requirements which relate to the addressing structure. To provide the facilities for the remainder of the requirements it is necessary to examine the functions provided by the nucleus.

CHAPTER V

THE NUCLEUS AND COMPUTING CONCEPTS

An addressing scheme for the nucleus has been specified and the functional characteristics examined. In this chapter the computing concepts concerned with processor, peripheral and process structure and management are examined. Functions to assist in these management responsibilities are presented.

5.1 PROCESSORS

In most current computers the processor is responsible for the management of the entire system. The address translation logic forms a part of its structure and consequently can restrict the processing capabilities of the processor. The inclusion of the addressing logic as part of the processor can cause problems in a multiprogramming environment where swapping of address spaces is required. This would imply a change in the control information for the address translation logic which would not be required for the nucleus scheme.

The nucleus addressing scheme eliminates the need to have the address translation logic as a part of the processor. The addressed objects do the address recognition required and therefore eliminate the requirement for translation logic. Capabilities are therefore treated as physical addresses by the processor.

5.1.1 PROCESS SWAP

The process swap operation establishes the address space for the processor by passing the capability for the process control segment. The control segment contains the information necessary to establish the address space and to initialise the processor control information.

The processor requirements to perform a process swap are to have the process currently in control to release control and for another process to have requested control. Control information for the processor must be available for the activation of the requesting process. The control information is provided by the requesting process on requesting the use of a processor. For implementation convenience and also to attempt to keep the structure tidy, the supply and initialisation of the processor with the control information has been made a part of the process orientated functions (Refer to Section 5.3).

The request and release functions for the processors are examined without consideration for the process call requirements. The flow through the functions will appear as the process would execute the function.

5.1.2 OVERVIEW OF PROCESSOR FUNCTIONS

The request for processor function has no effect on the processor from which the function is performed. This enables a process to initiate another process without loss of control. If the process is actually wishing to utilise another processor it would follow the request by a release processor function.

The release processor function causes a change of state for the processor. The processor relinquishes the current process and continues, possibly after a wait for an available process, with a new process in control. From the processor viewpoint the release processor function is actually the process swap instruction. However, for the processor to continue processing without waiting a process must have requested the use of a processor of the processor's type and be waiting for a processor to become available.

If no process is waiting for a processor of the processor's type then the processor will enter a wait state. Reactivation of the processor is initiated on a processor request of the processor's type. The processor, although only being involved in the issuing of a processor release function, is dependent on a processor request function for completion of the operation.

5.1.3 IMPLEMENTATION OF PROCESSOR FUNCTIONS

The internal operation of the processor request and release functions are based on the operation of semaphores (Dijkstra 1968). The request operation (Figure 5.1) is based on the P (wait) operator and the release operation (Figure 5.2) is based on the V (Signal) operator.

The processor type segment (Figure 4.3) has a semaphore cell located in the segment. This cell contains the counter (value) and a link. The request processor function decrements the counter and queues the requesting process, by inserting the process in the list headed by the link portion of the semaphore cell, if the resulting value of the counter is negative. The requesting process is allocated a processor if the resulting value of the counter is zero or positive.

The value of the counter indicates either the number of available

processors if the value is positive or the number of processes requesting use of a processor of the semaphore cell's processor type if the value is negative. The processor list is managed as a first-in-first-out queue which ensures a more even loading on the processors.

The process list order determines the scheduling order and for this reason a simple priority queue is to be used. This approach allows simple priority alteration to be implemented in the process oriented functions (Refer to Section 5.3).

5.1.4 USAGE OF PROCESSOR SEGMENT AND FUNCTIONS

The processor segment (Figure 4.3) contains a link cell for the processor list linkage, status information for system management routines and control information for the processor. The link cell doubles as the process identification field for the currently active process. From the link cell it is possible to detect whether the processor is active or in a wait state.

In the normal processing cycle of the system, the processor capabilities would be available only to the process active on the processor. Requests for processor utilisation are made using the processor type capabilities. The processor request and release functions are subfunctions for the process related functions. Software written to execute on a nucleus based system would not use these functions directly unless its own process management routines are being used.

To allow full flexibility in the use of a nucleus based system the two processor functions have been kept independent. It would have been possible to cause a processor release to occur as the last operation in the processor request but this would have restricted the use of the function. In a later section of this chapter, the combining of the processor functions is shown in their use in the process related functions.

The processor request and release functions provide considerable flexibility and enable other functions to be implemented with ease. The priority based queuing of processes does not place any restrictions on the type of supervisor scheduling to be used since it is always possible for the supervisor to place capabilities for its own scheduling routines in the process code where calls to the nucleus process oriented functions would normally be used. For example, the supervisor routine to handle process swapping could save the capability of the current process in its own scheduling queue, then perform a request processor function for the process which it wishes to schedule and then

perform a release processor function to cause the current task to be suspended.

The next two sections of this chapter illustrate the use of the processor functions in peripheral functions and process functions. No attempt has been made to suggest the type of processing which might be provided through the processors since the interaction with surrounding hardware components is the major interest of the nucleus. This implies that the processor would not provide operators which enabled interaction with other components other than those of the nucleus functions and addressing scheme.

5.2 PERIPHERALS

The management of peripherals is a major task in any operating system. Many of the functions are independent of the operation of the peripheral. Routines which manage peripherals are responsible for the maintenance of peripheral status flags, signalling peripheral errors, issuing commands to the peripheral and rescheduling tasks or setting completion of operation flags for the requested operation.

5.2.1 PERIPHERAL CONTROLLER

The addressing scheme enables peripheral status and error flags to be addressed via the peripheral capability. The peripheral controller can directly manipulate the status and error flags. By extending the status segment and the facilities of the peripheral controller, the manipulation of peripheral information such as the size of sectors for disk packs and peripheral identifiers can be included in the peripheral status segment. The peripheral controller could also be actively involved in the management of the peripheral. For example, for a disk pack the controller could take over the responsibility for the allocation of storage space.

The peripheral controller in this structure becomes more of a specialised processor; a processor responsible for the operations of the associated peripherals. The operations provided by the controller would be those required to manipulate and control the peripheral. The operations being requested by the requesting process are specified through the parameters of the peripheral request function.

The peripheral controller differs from a processor in that the peripheral controller performs specialised requests connected with the operation of the peripheral, while a processor executes the operations

of the process as a continuous stream. Peripheral requests are directed to a particular peripheral since data to be stored or retrieved is only available on a particular peripheral. Processor requests are directed to the processor type since the particular processor on which the process executes is, within processor type, irrelevant to the successful execution of the process.

5.2.2 PERIPHERAL CONTROL SEGMENT

The peripheral control segment (Figure 4.4) has a peripheral request semaphore which is used to queue requests for the peripheral. The peripheral is indicated to be available if the semaphore value is positive. As only one peripheral is associated with the semaphore no queue for available peripherals is required. It should be recognised that one peripheral controller may be responsible for more than one peripheral and therefore may have multiple peripheral segments associated with it.

5.2.3 PERIPHERAL REQUESTS

A process which uses the peripheral issues a request to the peripheral and releases the processor on which it is executing. On completion of the requested peripheral operation the process is placed back on the processor type request queue. The processor request is issued by the peripheral on completion of the requested operation. The process appears to enter a wait state while the request is queued and then processed.

The peripheral accepts requests queued on the peripheral request semaphore, processes the requests and reinitiates the process on the processor type from which the process made the request. If there are no requests then the peripheral waits for a request.

5.2.4 PERIPHERAL FUNCTIONS

To implement the peripheral interaction, two functions are provided. These are the request peripheral operation function (Figure 5.3) used by the process and the peripheral completion function (Figure 5.4) used by the peripheral controller. These functions also utilise the P and V operators for semaphores.

The request peripheral operations function places the processor information, required at completion of peripheral operation for reestablishing the process, on the process's stack (Refer to Section 5.3). The processor release function is then executed. The peripheral

FIGURE 5.3 PERIPHERAL REQUEST FUNCTION

```

PROCEDURE requestperipheraloperation = (
    CAPABILITY peripheral,process,requestinfo ) VOID:
BEGIN
    save(processorinfo(processorid(process)));
    releaseprocessor(processorid(process));
    push(requestinfo);
    IF counter(peripheral) -= 1 < 0 THEN
        enqueueprocess(process,peripheral)
    ELSE
        initiate(peripheral,process);
END;

```

FIGURE 5.4 PERIPHERAL COMPLETION FUNCTION

```

PROCEDURE peripheralrequestcompletion = (
    CAPABILITY process ) INTEGER:
BEGIN
    pop;
    IF counter(peripheral(process)) += 1 > 0 THEN
        deactivate(peripheral(process))
    ELSE
        initiate(peripheral,dequeueprocess(peripheral(process)));
        requestprocessor(type(processorinfo(process)),process);
        requestresult;
END;

```

request information is placed on the process's stack and the counter of the semaphore is decremented. If the resulting counter value is negative then the process is placed on the peripheral's request queue, otherwise the requested peripheral is initiated to execute the process's request.

The peripheral completion function removes the request information from the process's stack. The request processor function is performed and the counter of the peripheral's semaphore is incremented. If the resulting counter value is positive then the peripheral is deactivated (waits for a request), otherwise the next peripheral request on the queue is processed.

The method used for queuing peripheral requests will vary depending on the peripheral type. For example, to a printer the requests would be queued in a first-in-first-out technique, while for a moving head disk the requests may be queued according to the cylinder or track address from which the information is to be used. The peripheral semaphore cell can be used to indicate the queuing technique to be utilised.

Peripheral device scheduling is a part of these functions and therefore eliminates the requirement for the software supervisor to be involved in the scheduling.

5.2.5 ALTERNATIVE PERIPHERAL FUNCTIONS

The alternative request and completion functions add the process to the request queue regardless of the availability of the peripheral and remove the process from the request queue on completion of the request. Under this structure the peripheral controller can become more involved in the request queue management and process selection. Note that the completion function does not perform a peripheral initiation since it is assumed that the peripheral already has control and is selecting the next request.

In the alternative structure the peripheral controller is more characteristic of a process which accepts requests from other processes, performs the requested operation and then reinitiates the requesting process. The requesting process remains on the request queue until the operation it requested is completed. The initial structure has the requesting process becoming the controlling process on the peripheral controller. Consequently, the peripheral controller resembles a processor since it requires a controlling process to be active before it performs any operations.

FIGURE 5.5 ALTERNATIVE PERIPHERAL REQUEST FUNCTION

```

PROCEDURE requestperipheraloperation = (
    CAPABILITY peripheral, process, requestinfo ) INTEGER:
BEGIN
    save(processorinfo(processorid(process)));
    releaseprocessor(processorid(process));
    push(requestinfo);
    queueprocess(process, peripheral);
    IF counter(peripheral) -= 1  $\geq$  0 THEN
        initiate(peripheral);
    END;

```

FIGURE 5.6 ALTERNATIVE PERIPHERAL COMPLETION FUNCTION

```

PROCEDURE peripheralrequestcompletion = (
    CAPABILITY peripheral ) VOID:
BEGIN
    pop;
    requestprocessor(type(processorinfo(processid(peripheral))),
                    processid(peripheral));
    dequeueprocess(peripheral);
    IF counter(peripheral) += 1  $>$  0 THEN
        deactivate(peripheral);
    END;

```

5.2.6 PERIPHERAL MANAGEMENT AND INTERACTION

Some peripheral management tasks do not require a requesting process. For example, handling volume labels on removable disk packs. These operations would be performed by the controller regardless of the requests on the queue.

The requesting process for both structures would interact with the peripheral through what appears to be a standard procedure call, the request function acting as the procedure call and the peripheral completion function acting as the procedure return. Any result is returned and request parameters are handled in the same manner as procedure results and parameters.

Peripheral interaction is simplified and the input/output interruptions are eliminated. The process is the dominant structure and is in control of the peripherals which it wishes to use during the period of its request.

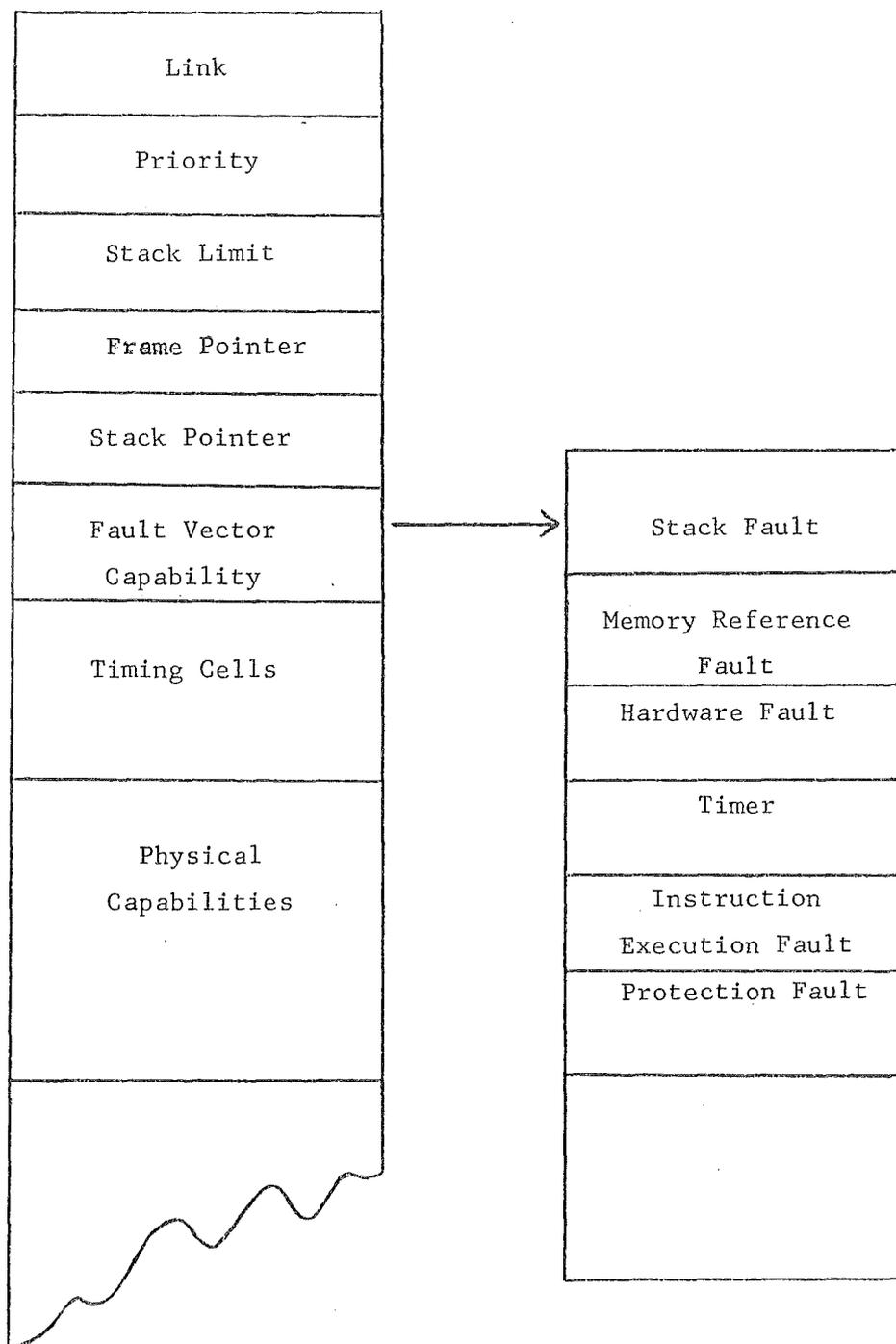
5.3 PROCESSES

The type of systems which can be implemented on a nucleus based system is dependent on the structure of the process and its control segment. The functions of processors limit the type of operations which can be performed but, since a variety of processor types can be supported, this does not limit the processing operations provided. The capabilities held by a process limit the operations which can be performed and define the address space which can be manipulated. The processor is only able to communicate with objects for which the active process possesses capabilities. The process is the controlling object in the system.

5.3.1 THE PROCESS CONTROL SEGMENT

The process control segment defines the process and the address space available for manipulation. The primary fields of the control segment (Figure 5.7) are those required by the nucleus functions and for the initial process (the process initiated when the system is started) to determine what objects are available in the system. The operating system designer can add to the control segment any fields which he considers necessary for the operation of his supervisory routines and control structures.

FIGURE 5.7 PROCESS CONTROL SEGMENT



5.3.2 THE LINK AND PRIORITY CELLS

To enable the nucleus scheduling functions to be implemented the link and priority cells are included in the process control segment. The link cell provides the storage to enable the queues to be formed. On the queuing of a request (that is, for processor usage) the link cell is used to store the capability for the next process on the request queue. When the process is in control of a processor then the link cell contains the capability of the processor.

The priority cell contains the priority used by the queuing algorithms to enable priority based scheduling to be implemented.

5.3.3 THE STACK

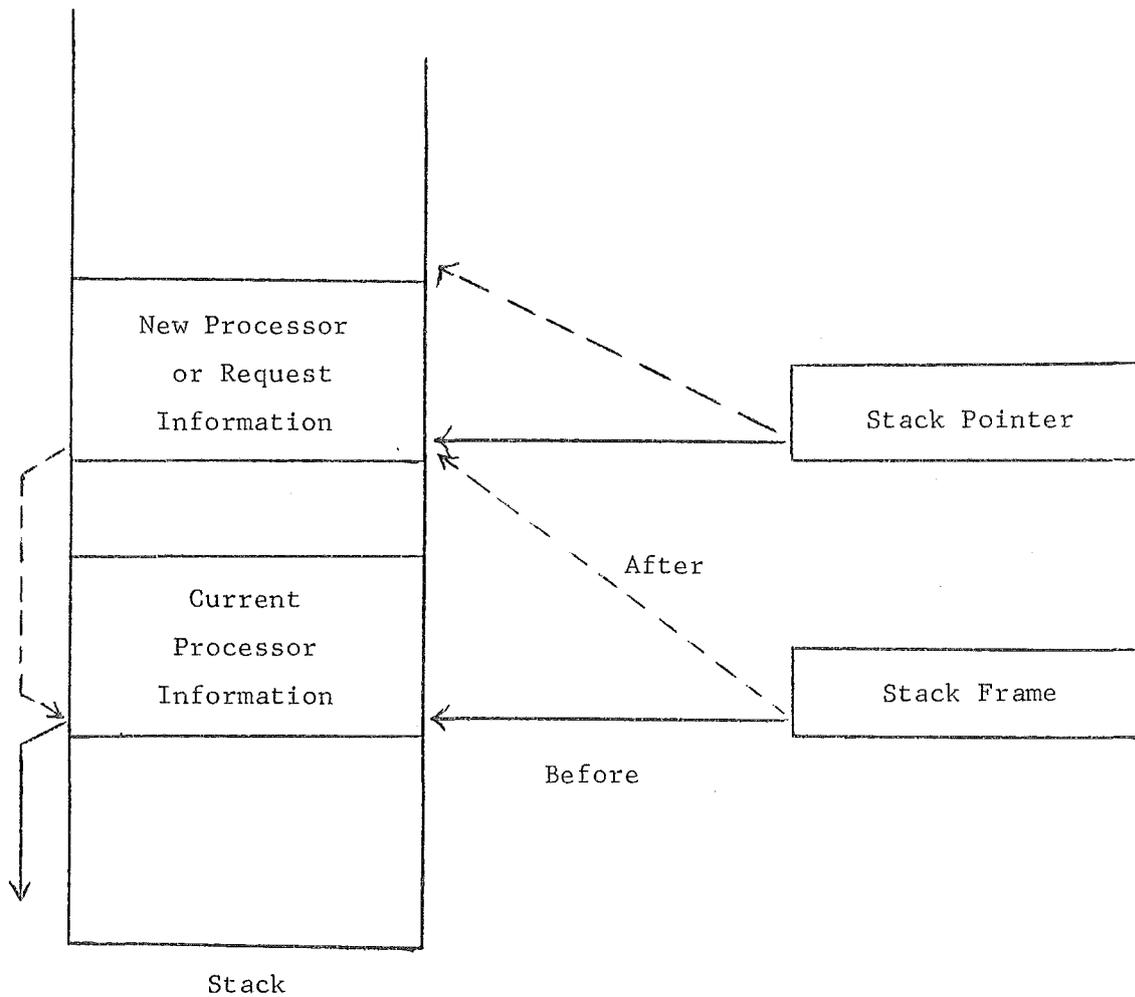
The implementation of the peripheral functions and the process functions rely on the existence of a stack for each process. The nucleus uses the stack to store processor information and peripheral request information while the process is queued, and for the saving of the information for the return to the processor type from which a request function was issued.

Three cells in the control segment provide the capabilities required to implement the stack. The stack limit cell provides the capability for the stack. This capability gives the address of the base of the stack and the stack size. Consequently the stack's available space is defined. The stack pointer cell indicates the current position of the top of the stack while the frame pointer points to the base of the portion of the stack which is applicable for the current processor or peripheral request information.

The stack is maintained by three stack operations, these being the push and pop operations and a save operation. The push operation (Figure 5.8) is used to place a new processor or request information on the stack and to adjust the stack pointer and frame pointers ready to issue the request. The pop operation (Figure 5.9) is used to remove the current process frame from the stack and return the stack and frame pointers to their status before the previous push operation. The save operation places the processor information at the position indicated by the frame pointer. The position of the stack and frame pointers is unchanged.

These stack operations are designed for use with the nucleus functions. This does not restrict the use of stack to the nucleus use only. However, if processor functions are used to manipulate the stack

FIGURE 5.8 THE STACK PUSH OPERATION

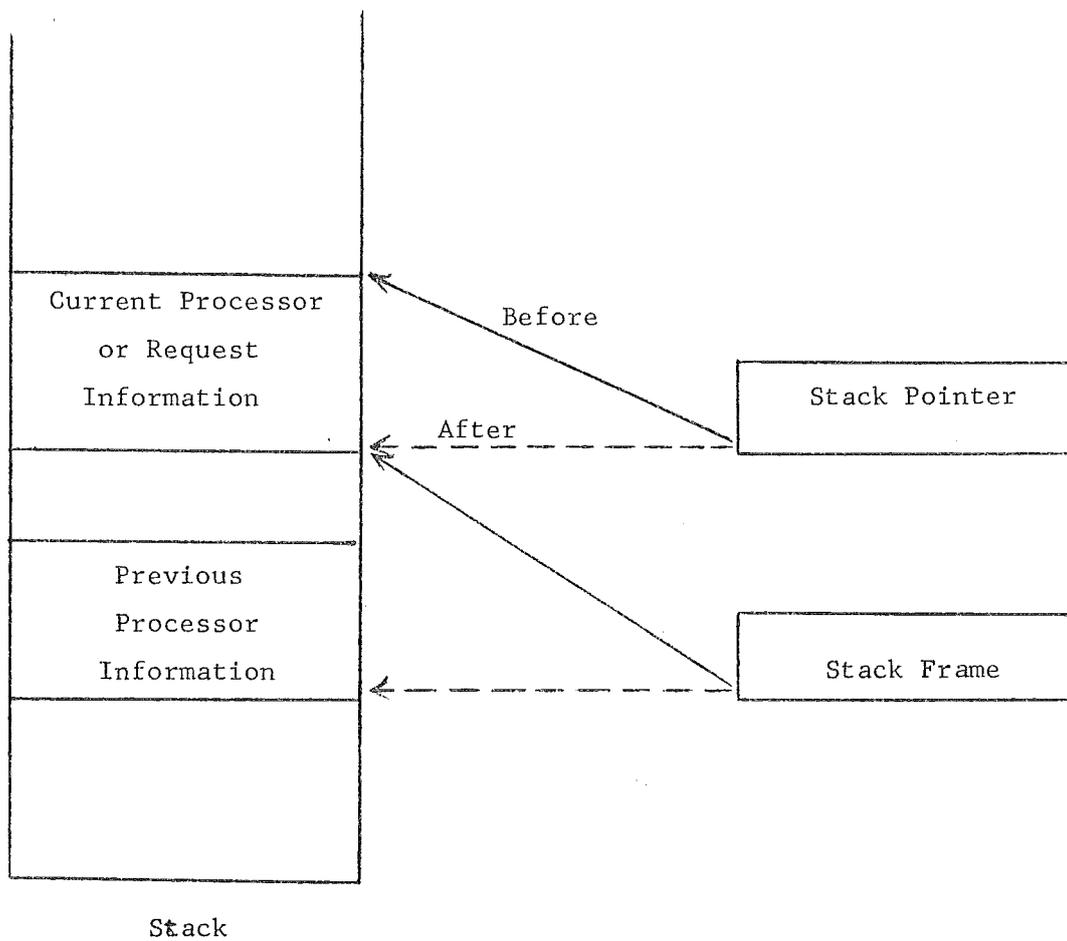


```

PROCEDURE push = ( CAPABILITY info ) VOID:
  BEGIN stackcell(stackpointer): = stackframe
    stackframe: = stackpointer;
    stackpointer+: = length(info)+1;
    save(info);
  END;

```

FIGURE 5.9 THE STACK POP OPERATION



```

PROCEDURE pop = () VOID:
  BEGIN
    stackpointer := stackframe;
    stackframe := stackcell(stackframe);
  END;

```

then they should not alter the contents of the frame pointer since doing so would destroy the operation of the nucleus stack operations.

5.3.4 THE FAULT VECTOR

The fault vector contains the capabilities for routines to handle abnormal conditions which occur during process execution. These faults relate to the operation of the process and not to any external operations. The fault routines may be those provided by the supervisory program. The nucleus does not provide functions to handle the faults, although a timer related function is provided to enable the priority based scheduling to time slices to processes.

The faults provided for in the fault vector are for stack faults, memory reference faults, hardware faults, instruction execution faults and protection faults. The stack faults relate to stack overflow and underflow conditions. Stack underflow is based on the stack frame pointer for stack operations executed by processors and on the stack limit for the stack push and pop operations described in the previous section.

Memory reference faults occur when a process or file capability is found on a storage medium (that is, a disk) on which the requested reference operation cannot be performed. This fault implies that a transfer operation is required to bring the referenced object to a storage medium capable of handling the requested operation. Although the storage structure suggests that all storage components are directly addressable, some media cannot be referenced for retrieval and update of data within the storage pages. When these pages require to be updated they must be moved within the storage hierarchy. This movement can be accomplished either by hardware facilities, in which case the memory reference fault would not occur (Cache-extended concept), or software routines (Disk or Drum to high speed memory). The memory reference fault capability provides the mechanism for the implementation of the software routines.

The hardware fault provides a mechanism for the software to be notified of hardware failures. The process currently involved in referencing or executing on the failing hardware receives the fault notification.

Instruction execution fault notifies exception conditions which occur during processor instruction execution. Examples of this would be integer overflow and exponent underflow which may occur on arithmetic operations.

Protection faults are caused by attempts to make unauthorised access to objects. For example, an attempt to write to an object for which the process does not hold a capability with a write access flag.

Other fault capabilities can be added to the vector as necessary.

5.3.5 TIMING FACILITIES

The nucleus is designed to provide timing facilities as a part of its structure. The timing cells of the process control segment provide the interface with the nucleus. These timing cells are used to accumulate effective times used on components of the system and for allocating time slices or setting maximum processing time limits.

The nucleus provides a timing function which forces the process executing on the processor to release control and to be placed on the processor type request queue. Although this does not suit all requirements it provides a simple default function should the user wish to place the capability for this function in the timer cell of the fault vector. The user's own timer routine should be used to meet his own needs. Time of day, timed waits, and time related interruptions can be implemented by providing functions for these facilities.

5.3.6 PROCESS FUNCTIONS

The nucleus addressing scheme and facilities are based on the nucleus recognising the process as an object in the system. The requirements of the addressing scheme enforce that the creation of processes, their initiation and termination, the allocation and release of storage segments and the creation and deletion of files be implemented through nucleus functions.

The request processor and release processor functions (Refer to Section 5.2) do not interact with the process structure. Therefore functions for calling or chaining to another processor type and for returning to a previous processor type are provided in the nucleus. These functions are created by combining other nucleus functions.

The process creation, initiation and termination functions provide for the integrity of the system. The process creation function (Figure 5.10) establishes a control segment capability for the process and allocates a process number. The process issuing the creation request is responsible for initialising the contents of the control segment. The nucleus maintains a table of process numbers and their status (Refer to Appendix 3). From this information an available process number is

FIGURE 5.10 PROCESS CREATION FUNCTION

```
PROCEDURE createprocess = ( INTEGER userlength ) CAPABILITY:
    allocatesegment(findprocessnumber,base + userlength);
```

Note: Findprocessnumber returns a number which is available, that is it has no storage segments allocated. Allocatesegment allocates the next available storage segment for the process, which for a new process is segment zero (the control segment).

FIGURE 5.11 PROCESS INITIATION FUNCTION

```
PROCEDURE initiateprocess = (
    CAPABILITY processid,processorinfo,INTEGER priority) BOOLEAN:
    IF checksegment(processid) THEN BEGIN
        push(processorinfo); setpriority(priority);
        requestprocessor(type(processorinfo),processid);
        TRUE;
    END ELSE FALSE;
```

FIGURE 5.12 PROCESS TERMINATION FUNCTION

```
PROCEDURE terminateprocess = (
    CAPABILITY processid,processor) BOOLEAN:
    IF segmentsreleased(processid) THEN BEGIN
        releasesegment(processid);
        releaseprocessnumber(processid);
        IF process(processor) = processid THEN
            releaseprocessor(processor);
        TRUE;
    END ELSE FALSE;
```

assigned.

The process initiation function (Figure 5.11) checks the contents of the control segment and uses the information contained in it to initiate the process. Process initiation is performed by using the processor request function.

The process termination function (Figure 5.12) releases the control segment and process number if all segments other than the control segment are released. The segments allocated to a process are maintained in a status table by the nucleus (Refer to Appendix 3). It is the responsibility of the process issuing the termination request to release all segments other than the control segment. It is not possible for the nucleus to keep a record of all copies of the capabilities allocated. Therefore it is essential that the process requesting termination releases segments and destroys copies of any capabilities for the segments.

5.3.7 STORAGE SEGMENT REFERENCES

Storage segments in the system are addressed by process or file capabilities. The creation of these capabilities is the responsibility of the segment allocation function (Figure 5.13). This function uses the status tables to allocate an available segment number. It allocates storage pages to the capability from amongst the storage pages with a null capability in their page table entries. If there are not enough pages available then a null capability is returned to indicate failure of the request.

The release segment function (Figure 5.14) reverses the allocation process by storing a null capability in the page table entries allocated to the segment and releasing the segment number. The allocation and release functions do not operate as part of a memory management module. They are allocating and releasing pages regardless of the medium referenced. A memory management operation may still be required to transfer the allocated segment from disk storage to accessible memory.

5.3.8 PROCESSOR INTERACTION

Three process functions are provided for interaction with processors at a level more acceptable to the process. These functions are the processor call (Figure 5.15), processor return (Figure 5.16) and the chain to processor (Figure 5.17) functions. The functions are

FIGURE 5.13 ALLOCATE SEGMENT FUNCTION

```
PROCEDURE allocatesegment = (  
    CAPABILITY id,  
    INTEGER segmentlength) CAPABILITY:  
    allocatestorage(id,nextsegmentnumber(id),segmentlength);
```

FIGURE 5.14 RELEASE SEGMENT FUNCTION

```
PROCEDURE releasesegment = (  
    CAPABILITY segmentid) VOID:  
    BEGIN  
        releasestorage(segmentid);  
        COMMENT the objective of this operation is to ensure that  
the storage object addressed via the capability is no longer  
addressable;  
        releasesegmentnumber(segmentid);  
    END;
```

FIGURE 5.15 PROCESSOR CALL FUNCTION

```

PROCEDURE callprocessor = (
    CAPABILITY calledprocesortype,processid,
        calledprocessorinfo) VOID:
BEGIN
save(processorinfo(processorid(processid)));
releaseprocessor(processorid(processid));
push(calledprocessorinfo);
requestprocessor(calledprocesortype,processid);
END;

```

FIGURE 5.16 PROCESSOR RETURN FUNCTION

```

PROCEDURE processorreturn = (
    CAPABILITY processid) VOID:
BEGIN
releaseprocessor(processorid(processid));
pop;
requestprocessor(type(processorinfo(processid)),processid);
END;

```

FIGURE 5.17 CHAIN TO PROCESSOR FUNCTION

```

PROCEDURE chainprocessor = (
    CAPABILITY chainprocesortype,processid,
        chainprocessorinfo) VOID:
BEGIN
releaseprocessor(processorid(processid));
save(chainprocessorinfo);
requestprocessor(chainprocesortype,processid);
END;

```

composed of other nucleus functions.

The processor call and processor return functions provide a call return mechanism for transferring to a new processor type and then returning to the processor type of the processor from which the process issued the call request. These functions simulate procedure calls for processor types.

The chain to processor function causes a transfer to a new processor type. Information about the current processor type is discarded. A routine to terminate a time slice would use this function to recall the same processor type.

Process management is simplified by the process functions. They provide a convenient interaction mechanism and eliminate some of the problems of process handling.

5.3.9 FILE CREATION AND DELETION

The storage allocated to a process is only addressable during the life span of the process, which is dependent on its execution path. In contrast, the file is created by a process and remains in existence until deleted by a process. The file creation function (Figure 5.18) resembles the process creation function. The file control segment has no structure forced on it by the nucleus. The nucleus, on allocating the control segment, assumes that an access semaphore is required and that file status information will be stored in this segment.

The file deletion function (Figure 5.19) releases the control segment provided that all segments allocated to the file are released. The nucleus does not provide any other file handling facilities.

5.4 CONCLUSION

The structures and functions discussed in this chapter provide the mechanism to achieve the nucleus requirements and design objectives.

Through the addressing scheme structure (Refer to Section 4.2) the facilities on which to build nucleus functions have been provided. Software systems can be structured from the facilities provided by the functions. These functions also make available the control structures which enable flexibility in the use of processors and peripherals. The process becomes the important entity through the functions of the nucleus and represents the complete power of the machine to the user.

FIGURE 5.18 FILE CREATION FUNCTION

```
PROCEDURE createfile = (  
    INTEGER userlength) CAPABILITY:  
    allocatesegment(findfilenumber, filebase + userlength);
```

FIGURE 5.19 FILE DELETION FUNCTION

```
PROCEDURE deletefile = (  
    CAPABILITY fileid) BOOLEAN:  
    IF segmentsreleased(fileid) THEN  
        releasesegment(fileid);  
        releasefilenumber(fileid);  
    TRUE;  
    END ELSE FALSE;
```

CHAPTER VI

NUCLEUS IMPLEMENTATION

The development of the nucleus was carried out by experimenting with concepts of hardware and operating system construction. Having gained an understanding of the workings of various existing hardware and software structures and determined some of the good points and some areas for improvement, it became possible to develop design objectives and requirements (Refer to Section 4.2). The nucleus facilities presented are derived from the design objectives and requirements.

Three implementation techniques for the nucleus are discussed in this chapter. Two of these techniques, the use of the resources of an existing computer and simulation, were used during the development of the nucleus. The third technique, the use of hardware and firmware, has been examined although it was not utilised during the nucleus development. This latter approach must be considered for a practical implementation.

6.1 IMPLEMENTATION ON EXISTING COMPUTERS

Since the nucleus does not define the instruction repertoire for processors or the operations performed by the peripherals of the system, the use of existing processors and peripherals eliminates the need to define these during testing of the nucleus. Implementation on existing equipment therefore seemed a logical choice.

In using existing equipment, the objective was to implement software routines which utilised existing facilities, to provide the facilities of the nucleus. This approach had been used before in the implementation of capability lists (Dennis and Van Horn 1966) and therefore appeared to provide a convenient implementation technique.

6.1.1 THE EXISTING COMPUTER USED FOR IMPLEMENTATION

The choice of computers available for implementing the nucleus was fairly restricted. Only one computer was reasonably accessible and allowed the use of the full resources, although it was not possible to modify the existing disk file structure. This machine was a DG Eclipse S130.

The addressing mechanism of the DG is based on a fixed length paging scheme utilising a translation mechanism which supports the loading of two page maps. These maps coexist, with a simple operation for selecting the active page map. This structure favours the Background/Foreground partition operation used by the RDOS operating system. The mechanism is designed to enable reasonably static page allocation with no support for demand paging mode of operation.

The instructions are based on a sixteen bit word length with the primary instructions being register-to-register transfers. Extended instructions are available to allow for a greater range of addressing. All addresses are of the form displacement plus index, with an index of zero representing absolute addressing, an index of one representing displacement relative to current program counter and indices of two and three representing the use of registers two or three as an index register. Support for a program stack is provided but no arithmetic operations can be performed on the stack elements. Some shift and instruction skip capability is combined with the arithmetic and logical instructions.

Control of peripherals, the address map and CPU related functions for handling interrupts are handled through eight input/output instructions. These are three Data In instructions, three Data Out instructions, a Skip on device busy and done flag status and a No Input/Output transfer instruction. The Data In and Data Out instructions are used to read and load device control registers. These, along with the No Input/Output transfer instruction, can also alter the values of the device busy and done flags and consequently initiate the Input/Output operation indicated by the control registers. Input/Output address maps assist in address translation for data transfers through the data channel.

Fixed locations in low memory are used for addresses of Input/Output handler, Supervisor Call handler and page fault handler and for other control information.

During operation of the system, three address spaces are established - the supervisor address space (nontranslated addresses), and the two user address spaces defined by the page maps discussed earlier. The nucleus implementation was planned to utilise the supervisor space.

6.1.2 IMPLEMENTING THE NUCLEUS ON THE DG ECLIPSE

The facilities of the addressing scheme are the major component of any nucleus implementation. As no support for demand paging was available and the page map mechanism was not suitable for the implementation of the segmentation scheme, this feature of the nucleus addressing scheme was not implemented. Any attempt to implement the full addressing scheme was impractical on the DG. Therefore only those portions related to the control segments and nucleus function requirements were implemented.

As all interactions between components in a nucleus based system revolve around the addressing scheme and the nucleus functions, the effectiveness of this implementation was immediately limited. However, the effectiveness of the nucleus functions was still able to be examined.

The DG implementation had all control segments within the supervisor address space with limited access available to the processes. Processes were restricted to single segments so that their address space was contained in a single DG page map. This was not considered to be a major influence and was accepted as a practical implementation restriction.

The processor release and request functions were implemented as documented in Section 5.2 with the exception that the processor release function passed control to a scheduling routine which adjusted the DG map for the new process and then passed control to the new process.

Peripheral interrupt handlers and interface routines were implemented based on the peripheral request/completion functions documented in Section 5.3. Because it was not possible to detect whether device access had been allocated to a process, two additional functions were created to allocate and release peripheral assignments. These functions would be accomplished by giving the capability of the peripheral to the process but, as no capabilities existed, this facility was provided.

The lack of capability addressing facilities was most evident during implementation of the process and related functions. The functions for defining, initiating and terminating processes proved easy to code but the allocation of pages was more difficult. Two things were apparent:

- 1) Since the DG used fixed locations in the mapped page zero, it was necessary to allocate page zero as well as a control segment before a process could be defined.

- 2) Under the capability scheme, storage pages are allocated from

an available pool of storage pages and a capability created which is given to the process requesting the segment. This enables a process to reference at least the control segment for the processes it has spawned. In the DG implementation the process knew the process number of spawned processes but could not access their control segments nor any of its allocated pages, if the proposed nucleus scheme was used.

Therefore page allocation was implemented so that a process allocated a page in its address space to the spawned process. This enabled the parent to have access to the child's address space and forced all memory pages to be allocated to the initial system process. Parent - child communication was possible but implementation of supervisor routines required a more flexible communication link.

With capability addressing, the capabilities for supervisor routines could be placed in the spawned process's control segment or some other addressable segment, and thereby allow the spawned process to communicate with its parent or supervisor process through these routines. Placing an address for a supervisor routine in the spawned process's address space was not appropriate since the address would only be valid if the supervisor's routine was also in the process's address space. Any control structures used by the supervisor's routine would also be forced into the process address space and therefore limit the effectiveness of the address separation which capability addressing could achieve.

To overcome this problem supervisor call and return functions were implemented. These functions were implemented by using a semaphore cell in page zero of the supervisor process and the semaphore P and V operators to implement request queuing.

Although the system created was based upon concepts presented in the design of the nucleus, the final result failed to represent the facilities of the nucleus. The failure of this attempt to use the existing features of the DG was caused by not implementing the capability addressing scheme.

As the disks used on the DG were to remain formatted for the RDOS operating system, they had to be treated as peripherals rather than as storage media. This implied that any efforts to emulate the full features of capability addressing were restricted to the real memory of the DG.

The use of existing language processors, the DG assembler and a BCPL compiler, also restricted the ability to implement nucleus facilities because of predefined usage of registers and some memory

locations. The implementation did indicate that the processor and peripheral functions were applicable in the design of the nucleus and that the addressing scheme was the most important construct in the design of any computer system.

6.1.3 PROBLEMS WITH IMPLEMENTATION ON EXISTING COMPUTERS

Although it is desirable to utilise the instruction repertoire of existing processors, the nucleus implies an alteration in the processor's interaction with other components of the system. Therefore the nucleus fits under the existing instruction repertoire modifying those instructions which interact with other system components. To utilise the facilities of existing computers to implement the nucleus implies a change in the instruction repertoire and consequently a requirement to produce new programming tools.

Therefore this approach does not provide the easy implementation path that it appeared to offer through the pre-existence of an instruction repertoire and language processors. Also some of the features available, for example the page map scheme of the DG, hinder the implementation of nucleus facilities.

6.2 SIMULATION OF THE NUCLEUS

Simulation of a structure can be a valuable tool for testing a design. For the nucleus, simulation enables the checking of the addressing scheme and its interaction with the nucleus functions. Processor and peripheral interaction can also be tested and the most appropriate relationship between the nucleus and the processors can be determined.

In constructing a simulator, more work is required than in implementing the facilities on an existing computer, since not only the structure of the nucleus, but also the processors and peripherals, must be designed and implemented. This proved to be a major consideration in testing the design of the nucleus.

6.2.1 CHOICE OF A SIMULATION TECHNIQUE

Considerable time was consumed in an effort to find the correct tools for simulating the nucleus. Initial investigations concentrated on Computer Hardware Description Languages (Chu 1965).

Many of the CHDLs examined were concerned with timing considerations or the hardware logic required to implement the computer

structure. The nucleus emphasis was on functional characteristics rather than the hardware logic. Hardware logic considerations certainly affect the final design and implementation but were not regarded as the primary objective.

Of the languages examined, Computer Structure Language (CSL) (Smith 1975) seemed the most appropriate for the simulation of the nucleus, but no implementation of the language was available and details on the language were limited. This applied to many of the CHDL's examined, so it was decided to use an available programming language.

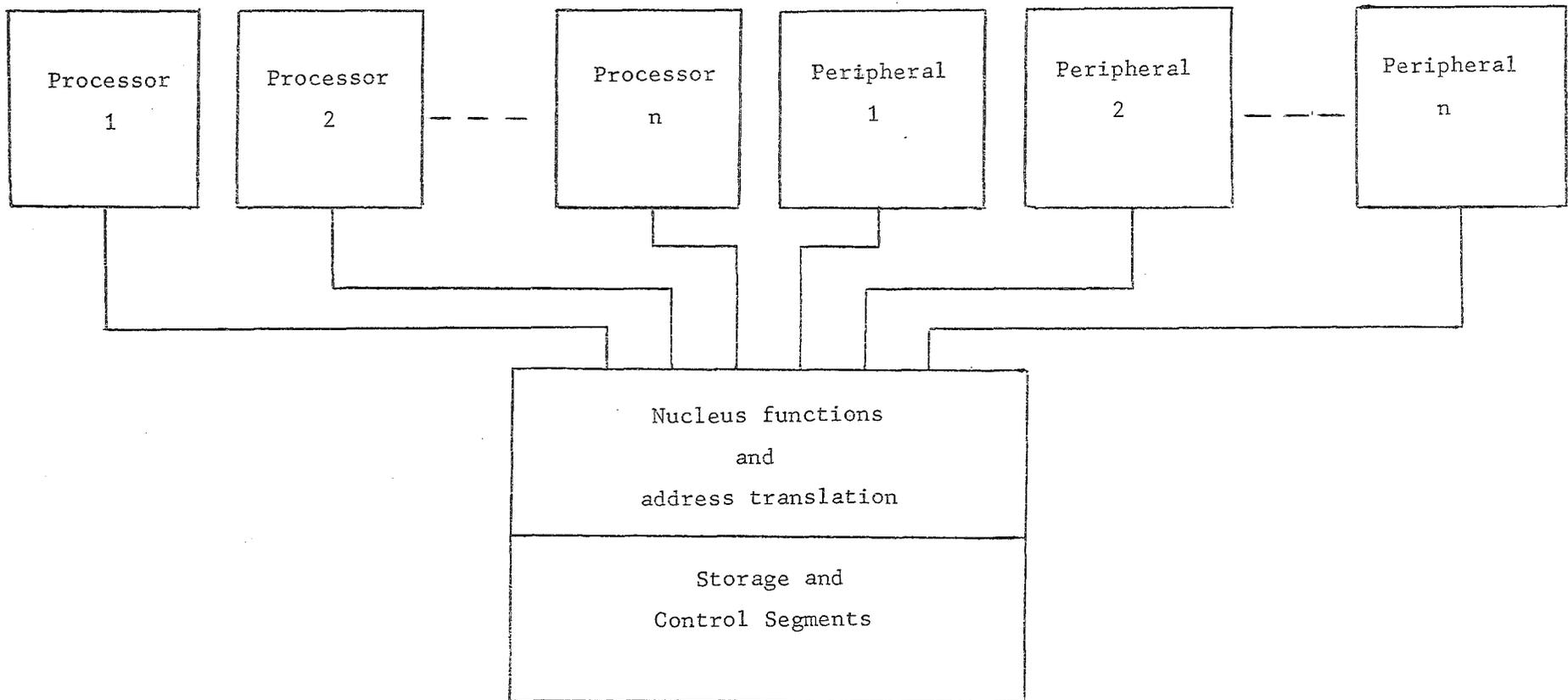
Of the languages available, only Burroughs Extended Algol provided the facilities required to simulate the nucleus. The language provided a multiprocessing facility which could be used to implement the parallel processing required for implementing processors and peripherals. Structured programming facilities of the language, such as the case statement, could be used to decode instructions and addresses. Event variables and related control functions enabled synchronisation of the processors and peripherals to be accomplished. The descriptive nature of a CHDL was not available and therefore limited the effectiveness of Burroughs Extended Algol to that of a simulator implementation tool.

6.2.2 NUCLEUS SIMULATION USING BURROUGHS EXTENDED ALGOL

Because of the block structural nature of Burroughs Extended Algol and the multitasking facilities, it was possible to provide the nucleus facilities as a common set of procedures accessible to all pseudo processors and peripherals. The processors and peripherals were simulated as asynchronous processes which were dependent on the initiator's stack. The nucleus facilities were globals in the initiators stack where they were accessible to the processor and peripheral processes. The structure is illustrated in Figure 6.1.

The use of the Burroughs multiprocessing facilities did not prove to be satisfactory as the scheduling algorithm for the pseudo processors appeared to favour one process. This apparent favouritism of one process was actually a fault in the implementation of the simulator. A processor (implemented as an Algol process) would remain active until no more processes required to use a processor of its type. This gave the appearance that only one processor of any given type was active during the testing of the processor request and release functions. To ensure that other processors were activated it was necessary to force the processors into a wait status' after the processor release function was executed and before

FIGURE 6.1 SIMULATOR STRUCTURE



a new process was allocated to the processor. An additional alteration to the simulator was to create an active processor chain. The objective of this was to control the scheduling of the simulated processors to ensure that each was given a proportionate share of the available real processor time. The simulation objective was to check the functional characteristics of the nucleus rather than the time dependent characteristics.

The emphasis on the functional characteristics of the nucleus made the implementation of the nucleus easier since implementing the nucleus functions was a direct Algol coding exercise. The only additions to the functions were the inclusion of mutual exclusion facilities to ensure the integrity of control structures such as the semaphore cells. Implementing the addressing scheme certainly did not fall into a simple coding of the algorithms.

6.2.3 ADDRESS TRANSLATION SIMULATION

As it was not practical to have Algol tasks for each addressable component, the addressing algorithms presented in Appendix 1 could not be effectively implemented. Address translation was therefore implemented by a central translation algorithm. This algorithm validated the capability and returned an index into an array. A single array for all addressable objects was used since the return of addresses or reference variables was not possible. Although a processor or peripheral is addressed via a capability, it is primarily the processor or peripheral segment which is being addressed. Therefore the single array containing the addressable objects did not limit the effectiveness of this implementation approach. Function capabilities were not included as these were called as Algol procedures.

Since the word length on the Burroughs B6700 was 48 bits, the number of processes, the segment size and the page size were restricted to enable the capability to be stored in a single word.

The approach used to implement the address translation algorithm resembles the use of segment and page tables except that the tables are related to the physical page location rather than the address used by the process. The page table entries contain the capability used by the process rather than the storage location. This implies a table scan for address translation which is the technique outlined in the algorithms except that associative tables entries are proposed.

The flexibility in the use of capabilities proved to be the most

important component of the nucleus design. The simulator was easier to program than the Eclipse implementation because the full power of the capability addressing structure was available in the simulator.

6.2.4 PROBLEMS WITH SIMULATION

The use of simulation to test the nucleus required the implementation of processors and peripherals to enable the complete checkout of nucleus facilities. This requirement presented a major problem in the use of simulation as there was no desire to design processors and peripherals for the nucleus. To enable the testing of the nucleus, processors and peripherals with a limited range of facilities were implemented. These enabled the checking of the nucleus but did not allow for software systems to be created for a nucleus based system.

The lack of a suitable description language limited the usefulness of simulation as it became easy to distort the nucleus structure to ensure performance of the simulated nucleus. Techniques used in simulating the nucleus did not resemble the techniques which would be used if a hardware/firmware implementation technique were utilised.

The feasibility of the nucleus was demonstrated by the success of the simulated nucleus but with the use of a hardware description language, it would have been possible to simulate the implementation details as they would be implemented using hardware/firmware. For example, the address translation algorithms of Appendix 1 could have been implemented and the requirements for interaction with other components developed.

6.3 IMPLEMENTING IN HARDWARE/FIRMWARE

As a practical implementation of the nucleus, neither simulation or the use of existing computer facilities are adequate. The close association of the nucleus and the addressing scheme requires hardware and firmware involvement in the implementation. Simulation has helped to establish the structure of the nucleus and to define its place in the computer system, while the implementation on existing equipment has assisted in checking the functional characteristics of its design.

Final implementation rests in the design of the hardware and firmware of the computer system, although it was not possible in the environment of this research. From the implementations attempted it was clear that the address translation mechanism must be implemented below the processors and peripherals of the system. Also, the address translation was not dependent on the current process in control of a

processor or peripheral. This implies that the control tables for the addressing mechanism remain relatively static compared with the paged system of the DG Eclipse.

The static nature of the control tables is also represented in the algorithms presented in Appendix 1. These algorithms are designed as part of each system component, the functions being the only additional components of the system.

6.4 CONCLUSION

Although the implementation of nucleus facilities on existing computers, by utilising their facilities, appeared to be a practical method for final implementation it proved to have difficulties which implied the necessity to simulate facilities of the nucleus. Simulation proved to be the most appropriate technique for checking the design approach.

Simulation and the utilisation of existing computer facilities proved to be ideal tools for testing the nucleus design. Hardware and firmware implementation is the only practical implementation approach for the nucleus.

CHAPTER VII

CONCLUSION

This thesis develops the concept of a nucleus around which the processors, peripherals and software of a computer system can be constructed. The primary component of the nucleus is the addressing scheme which meets the requirements outlined in Section 4.1. These requirements included facilities for software structuring, protection in a multiprogramming environment, sharing of resources and flexibility in address space changes. The scheme proposed in this thesis, by the use of the capability structure, provides for these requirements. Also, with the static address translation algorithms (Appendix 1) proposed the problems involved in changing address spaces is alleviated. The recognition of the process and the file as addressable units has also aided in eliminating address separation and protection problems. Protection is a natural part of the capability structure since the process must hold the capability for the object it wishes to address.

Furthermore, the extension of the capability scheme to include the addresses for hardware components (processors and peripherals) has removed the requirement for a privileged (supervisor) state and a non-privileged (problem) state. Processor and peripheral protection can also be accomplished by the allocation of capabilities. This addressability of processors and peripherals also enables the status of these components to be addressed through the normal processing structures of the system and consequently eliminates the requirement for additional control structures. In addition, the table structure of the addressing scheme provides additional flexibility in the configuration of the hardware.

The provision of functions which are accessed through the addressing scheme also ensures that the privileged and non-privileged states are not required. These have also been used to assist in process scheduling and to enable flexibility in the use of processors and peripherals while allowing the controlling process to be identified to both processors and peripherals.

The removal of privileged and non-privileged modes assists the implementation of virtual machine environments. The flexibility of the capability addressing scheme also removes many of the address translation problems which are present with the virtual machine environments.

Although the nucleus as presented in this thesis appears to meet design requirements and objectives, considerable work is still necessary to prove that the nucleus is feasible in an operational environment. As was discussed in Chapter 6, the problems of implementing the nucleus without designing accompanying processor structures limited the ability to check the nucleus design. Further requirement to prove the nucleus in a practical situation would require the creation of software to execute on the completed system. The only practical tool available for this checking of the nucleus design was simulation. The lack of description languages for the creation of the required simulator increased the difficulty of testing the nucleus.

To eliminate the requirement to construct processor simulators, an attempt was made to implement the nucleus within the privileged state of an existing processor. The encouraging aspect of this approach was that the functions of the nucleus could be tested although the failure to implement the proposed nucleus addressing scheme limited the effectiveness of this approach.

To extend the work completed would require more than the time available and more than a single researcher. The work did enable the author to expand his understanding of the design requirements for both the hardware and software. In some aspects of this research the author feels that existing concepts have been blended together to extend his understanding of the current position with regard to research in computer architecture. In some respects the quote from T.S.Eliot, at the front of this thesis, summarises the effect as the concepts of the nucleus were developed and implemented. As each new revelation provides more knowledge of the current state of the art we see more possibilities just on the horizon. Each of these provides us with more insight to what is possible and our search goes on.

So in the development of the nucleus, the possibilities of implementing portable software based on a nucleus concept seemed a definite possibility for future research. Also possibilities for future research are the problems related to ensuring the integrity of the capabilities, and the structure of processors and the software systems to utilise the nucleus facilities.

APPENDIX 1

ADDRESS TRANSLATION ALGORITHMS

The algorithms presented here are associated directly with the object in the system which is being addressed. The processor or requesting object's responsibility in the address translation process is to place the capability on the address bus. The addressed object recognises the capability and responds according to the request being made, the facilities it provides and the capability access bits in the requesting capability.

The algorithms presented do not indicate exception conditions or the non-recognition of a capability by the addressing mechanism. The algorithms present the recognition techniques for the various object categories. An Algol 68 like language has been chosen as the language in which the algorithms are written.

1. FUNCTION ADDRESS RECOGNITION

```
PAR BEGIN
```

```
  IF addressbus.type = physical AND
    addressbus.id = functionaltype AND
    addressbus.displacement = myfunctionindex THEN
    requestformycapability,
  IF addressbus.type = functional AND
    addressbus.id = myid THEN
    requestformyfunction
```

```
END;
```

2. STORAGE, PROCESS AND FILE ADDRESS RECOGNITION

(a) Storage media address recognition

```
IF addressbus.type = physical AND
  addressbus.id = storagetype AND
  addressbus.displacement = mystorageindex THEN
  requestformycapability
```

(b) Page table addressing

```

PAR BEGIN
  IF addressbus.type = storage AND
    addressbus.id = mystorageid AND
    addressbus.displacement = mypageindex THEN
    requestforoperationonmypagetableentry,
  IF addressbus.type = mypagetableentry.type AND
    addressbus.id = mypagetableentry.id AND
    addressbus.pageid = mypagetableentry.pageid THEN
    requestforoperationonmystoragepage
END;

```

3. PROCESSOR TYPE ADDRESS RECOGNITION

```

PAR BEGIN
  IF addressbus.type = physical AND
    addressbus.id = processortypetype AND
    addressbus.displacement = myprocessortypeindex THEN
    requestformycapability,
  IF addressbus.type = processortype AND
    addressbus.id = myid AND
    addressbus.displacement = semaphorecell THEN
    requestforsemaphorefunction
END;

```

4. PROCESSOR ADDRESS RECOGNITION

```

PAR BEGIN
  IF addressbus.type = processortype AND
    addressbus.id = myprocessortype AND
    addressbus.displacement = myprocessorindex THEN
    requestformycapability,
  IF addressbus.type = processor AND
    addressbus.id = myid THEN
    requestformycontrolinformation
END;

```

5. PERIPHERAL ADDRESS RECOGNITION

```
PAR BEGIN
  IF addressbus.type = physical AND
    addressbus.id = peripheraltype AND
    addressbus.displacement = myperipheralindex THEN
    requestformyperipheraltableentry,
  IF addressbus.type = peripheral AND
    addressbus.id = myperipheraltableentry.id THEN
    requesttoaccessmyperipheral
END;
```

APPENDIX 2

FUNCTIONS OF THE NUCLEUS

This appendix lists the functions available in the nucleus. A brief description of the function's usage is included.

Request Processor :- Used to queue a process for execution on a processor.

Release Processor :- Used to release control of a processor.

Request Peripheral Operation :- This is used to queue a request for a peripheral device in the system. (Equivalent operation would be a start I/O on most systems.)

Peripheral Request Completion :- This signals the end of the peripheral operation and re-initiates the process. (Equivalent operation is the interrupt on I/O completion.)

Create Process :- Used to identify a new process to the system and obtain a control segment.

Initiate Process :- Used to place the process on a queue ready to execute on a processor.

Terminate Process :- This causes the process to be deleted from the system.

Allocate Segment :- Creates the capability for a storage segment and assigns available storage pages.

Release Segment :- Deletes the capability for a storage segment and releases allocated storage pages.

Call Processor :- Used to transfer a process executing on one processor type to another processor type. The processor information is saved so that return can be made at some later time.

Processor Return :- Used to transfer a process back to the processor type whose information was saved by the use of the call processor function.

Chain Processor :- Transfers a process executing on one processor type to another processor type without saving the current processor information.

Create File :- Creates the control segment for a file.

Delete File :- Destroys the capability for the file.

APPENDIX 3

NUCLEUS CONTROL AND STATUS STRUCTURES

Since capabilities can only be created by the nucleus, some form of status structure is required to indicate which process and file segments have been allocated.

The simplest approach, and that which meets the requirements of the nucleus as specified, would be the use of two bit matrices, with the indices being process/file number and the segment number. If a bit in the matrix is one, then the segment represented by that bit has been allocated. A process or file number has been allocated if segment zero has been allocated. Using this scheme it is not possible to determine the number of copies of any allocated capability, and therefore when a segment is released the nucleus does not know whether all capabilities have been destroyed.

An alternative scheme is to have two matrices of counters, indexed as before. If the counter is zero then the segment has not been allocated. The counter is set to one on allocating the segment. It is incremented when a copy of the capability is stored in a storage segment and decremented when a copy of the capability is destroyed by a storing operation. The counter indicates the number of capabilities in storage plus one and is used to ensure that on a segment release no additional capabilities exist for the segment.

This scheme ensures greater integrity of the capability scheme but places greater structural requirements on the system. These requirements are that on each store operation both the item being stored and that being overwritten be examined to determine whether they are a capability.

The nucleus may find that some segments and process or file numbers remain allocated in its storage structure without capabilities existing to reference them. The second of these structures can be used to perform a clean up operation either on system termination or initiation, since at these times the only capabilities which will be in existence are those in the storage structure. Consequently all segments whose counter is one can be deleted, freeing storage pages, process/file numbers and segments.

Other structures could be used to ensure the integrity of the capabilities but these two provide for the status requirements of the nucleus.

ACKNOWLEDGEMENTS.

The author wishes to acknowledge the assistance of his wife, Marilyn, who typed this thesis and had to survive on a limited budget while this research was being carried out. The support of the Department of Computer Science was appreciated and enabled this work to be completed.

The author is greatly indebted to his faith in the Lord, Jesus Christ, which has greatly encouraged him and grown stronger during the period of this research. The author believes that without this faith this work would not have been completed and the author's family would not have survived this period. As the author's relationship with the Lord has grown, the author's ability to withstand pressures has developed and given him confidence to complete this work and other tasks which have been placed before the author. The objective for completing this research has also changed as a result of the author's faith. Initially the completion of this thesis was regarded as a status symbol but now it represents another step in God's plan for the author's life and growth.

Rejoice in the Lord always;
again I will say, Rejoice.

Philippians 4:2

BIBLIOGRAPHY

- ANDREWS, G.R. and others. Language features for Process Interaction. Operating Systems Review, Vol. 11: No. 2. 1977.
- ATKINSON, T.D. Architecture of Series 60/Level 64. Honeywell Computer Journal, Vol. 8: No. 2. 1974.
- ATKINSON, T.D. and others. Modern Processor Architecture. Proceedings IEE, Vol. 63: No. 6. 1975.
- ATWOOD, J.W. I/O Supervision in the Project SUE Operating System. Computer, Vol. 6: No. 11. 1973.
- ATWOOD, J.W. Concurrency in Operating Systems. Computer, Vol. 9: No. 10. 1976.
- BACHMAN, C.W. The Evolution of Storage Structures. Communications of the ACM, Vol. 15: No. 7. 1972
- BARBAGCI, M.R. and others. Evaluation of the CFA Test Programs via Formal Computer Descriptions. Computer, Vol. 10: No. 10. 1977.
- BARNES, G.H. and others. The Illiac IV Computer. IEEE Transactions on Computers, Vol. C-17: No. 8. 1968.
- BAUERLE, D.A. The Organisation of Fast Store in the B6700. Infotech State of the Art Report, No. 17.
- BELL, C.G. and others. The Evolution of the DEC System 10. Communications of the ACM, Vol. 21: No. 1. 1978.
- BELL, C.G. and NEWELL, A. Computer Structures : Readings and Examples. McGraw-Hill. 1971.
- BERKLING, K. A Computing Machine based on Tree Structures. IEEE Transactions on Computers, Vol. C-20: No. 4. 1971.
- BONDY, L.B. and others. Putting Supervisory Routines into Hardware. Proceedings of IFIP Congress on Information Processing, Vol. 7. 1977.
- BORGERSON, B.R. and others. The Evolution of the Sperry Univac 1100 Series : A History, Analysis, and Projection. Communications of the ACM, Vol. 21: No. 1. 1978.
- BRINCH HANSEN, P. Operating System Principles. Prentice-Hall. 1973.
- BROWN, G.E. and others. Operating System Enhancement through Firmware. SIGMICRO Newsletter, Vol. 8: No. 3. 1977.
- BROWN, R.R. and others. The GM Multiple Console Time Sharing System. Operating Systems Review, Vol. 9: No. 4 and Vol. 10: No. 1.
- BUZEN, J.P. I/O Subsystem Architecture. Proceedings IEEE, Vol. 63: No.6. 1975.

- BUZEN, J.P. and GAGLIARDI, U.O. The Evolution of Virtual Machine Architecture. AFIPS, Proceedings of NCC, Vol. 42. 1973.
- CAPON, P.C. Order Codes that suit Programming Languages. Infotech State of the Art Report, No. 17.
- CARPENTER, B.E. and DORAN, R.W. The Other Turing Machine. Massey University Computer Unit Report, No. 23.
- CASE, R.P. and PADEGS, A. Architecture of the IBM System/370. Communications of the ACM, Vol. 21: No. 1. 1978.
- CHU, Y. An Algol-like Computer Design Language. Communications of the ACM, Vol. 8: No. 10. 1965.
- CHU, Y. Computer Organisation and Microprogramming. Prentice-Hall. 1972.
- CHU, Y. High-Level Language Computer Architecture. Academic Press. 1975.
- CHU, Y. Evolution of Computer Memory Structure. AFIPS, Proceedings of NCC, Vol. 45. 1976.
- COHEN, E. and others. Protection in the Hydra Operating System. Operating Systems Review, Vol. 9: No. 5. 1975.
- COOK, D. The Cost of using the CAP computer's protection facilities. Operating Systems Review, Vol. 12: No. 2. 1978
- CORBATO, F.J. and VYSSOTSKY, V.A. Introduction and Overview of the MULTICS System. AFIPS, Proceedings of FJCC, Vol. 27. 1965.
- CORBATO, F.J. and others. Structure of the MULTICS Supervisor. AFIPS, Proceedings of FJCC, Vol. 27. 1965.
- COX, P.R. Machine Independent Operating Systems : A Functional Approach to Design. Infotech State of the Art Report, No. 1.
- DALEY, R.C. and others. A General-Purpose File System for Secondary Storage. AFIPS, Proceedings of FJCC, Vol. 27. 1965.
- DALEY, R.C. and others. Virtual Memory, Processes, Sharing in MULTICS. Communications of the ACM, Vol. 11: No. 5. 1968.
- DAVIS, A.L. The Architecture and Systems Method of DDM1 : A Recursively Structured Data Driven Machine. SIGARCH Newsletter, Vol. 6: No. 7. 1978.
- DENNIS, J.B. and MISUNAS, D.P. A Preliminary Architecture for a Basic Data Flow Architecture. SIGARCH Newsletter, Vol. 3. 1975.
- DENNIS, J.B. and VAN HORN, E.C. Programming Semantics for Multiprogrammed Computations. Communications of the ACM, Vol. 9: No. 3. 1966.
- DIJKSTRA, E.W. The Structure of the 'THE' Multiprogramming System. Communications of the ACM, Vol. 11: No. 5. 1968.
- DIJKSTRA, E.W. Hierarchical Ordering of Sequential Processes. Acta Informatica, Vol. 1: No. 2. 1971.

- DORAN, R.W. A Computer Organisation with an Explicitly Tree-Structured Machine Language. Australian Computer Journal, Vol. 4: No. 1. 1972.
- DORAN, R.W. Virtual Memory and Display Registers. Massey University Computer Unit Report, No. 14.
- DORAN, R.W. The ICL 2900 Computer Architecture (Compared with the Burroughs B6700). Massey University Computer Unit Report, No. 20.
- DORAN, R.W. Virtual Memory. Computer, Vol. 9: No. 10. 1976.
- FABRY, R.S. Capability - Based Addressing. Communications of the ACM, Vol. 17: No. 7. 1974.
- FEUSTAL, E.A. Advantages of Tagged Architecture. Transactions on Computers, Vol. C-22: No. 7. 1973.
- FORD, W.S. and others. Low - Level Architecture Features for Supporting Process Communication. Computer Journal, Vol. 20: No. 2. 1977.
- FOSTER, C.C. View of Computer Architecture. Communications of the ACM, Vol. 15: No. 7. 1972.
- FRIEDER, G. Microprogramming and Operating Systems. Infotech State of the Art Report, No. 23.
- GAGLIARDI, U.O. Trends in Computing - System Architecture. Proceedings of IEEE, Vol. 63: No. 6. 1975.
- GAMMAGE, N.D. Addressing Structures : The Focal Point of Fourth Generation Architecture. Infotech State of the Art Report, No. 1.
- GILLOI, W.K. and BERG, H.K. Data Structure Architectures - A Major Operational Principle. SIGARCH Newsletter, Vol. 6: No. 7. 1978.
- GLASER, E.L. and OLIVER, G.A. System Design of a Computer for Time - Sharing Applications. AFIPS, Proceedings of FJCC, Vol. 27. 1965.
- GOLDBERG, R.P. Architecture of Virtual Machines. AFIPS, Proceedings of NCC, Vol. 42. 1973.
- GORDON, R.L. Impact of Automated Memory Management on Software Architecture. Computer, Vol. 6: No. 11. 1973.
- HABERMANN, A.N. and others. Modularisation and Hierarchy in a Family of Operating Systems. Communications of the ACM, Vol. 19: No. 5. 1976.
- HERBERT, A.J. A New Protection Architecture for the Cambridge Capability Computer. Operating Systems Review, Vol. 12: No. 1. 1978.
- HOARE, C.A.R. Monitors : An Operating System Structuring Concept. Communications of the ACM, Vol. 17: No. 10. 1974.
- HOPKINS, W.C. and others. A Microprogrammed Implementation of an Architecture Simulation Language. SIGMICRO Newsletter, Vol. 8: No. 3. 1977.

- HORNING, J.J. and RANDALL, B. Process Structuring. Computer Surveys, Vol. 5. 1973.
- HUXTABLE, D.H.R. and others. The Hardware/software Interface of the ICL 2900 range of Computers. Computer Journal, Vol. 20: No. 4. 1977.
- IBBETT, R.N. and CAPON, P.C. The Development of the MU5 Computer System. Communications of the ACM, Vol. 21: No. 1. 1978.
- ILLIFFE, J.K. Basic Machine Principles. MacDonalD/American Elsevier, 1972.
- KEEDY, J.L. On Structuring Operating Systems with Monitors. Australian Computer Journal, Vol. 10: No. 1. 1978.
- LAMPSON, R.W. and others. Reflections on an Operating System Design. Communications of the ACM, Vol. 19: No. 5. 1976.
- LASKI, J. Theoreticians' Work as Informing Principles of Machine Architecture. Infotech State of the Art Report, No. 17.
- LAVINGTON, S.H. The Manchester Mark I and Atlas : A Historical Perspective. Communications of the ACM, Vol. 21: No. 1. 1978.
- LAWSON, H.W.Jnr. and MALM, B.U. The Datasab Flexible Central Processing (FCPU) Background, Concepts, Basic Design and Applications. Infotech State of the Art Report, No. 17.
- LEVIN, R. and others. Policy/Mechanism Separation in Hydra. Operating Systems Review, Vol.9: No. 5. 1975.
- LIPOVSKI, G.J. and DOFY, K.L. Developments and Directions in Computer Architecture. IEEE Computer, Vol. 11: No. 8. 1978.
- LISKOV, B. The Design of the VENUS Operating System. Communications of the ACM, Vol. 15: No. 3. 1972.
- MARILL, T. and STERN, D. The Datacomputer - A Network Data Utility. AFIPS, Proceedings of NCC, Vol. 44. 1975.
- MILLS, P.M. Control Functions for a Multiprocessor Architecture. Operating Systems Review, Vol. 11: No. 1. 1977.
- MORRIS, D. and others. A Virtual Processor for Real Time Operation. Software Engineering, Vol. 1. 1969.
- McKEAG, R. M. and WILSON, R. Studies in Operating Systems. Academic Press. 1976.
- NEEDHAM, R.M. and others. The Cambridge CAP Computer and its Protection System. Operating Systems Review, Vol. 11: No. 5. 1977.
- NEEDHAM, R.M. The CAP Filing System. Operating Systems Review, Vol. 11: No. 5. 1977.
- von NEUMANN, J. and others. Planning and Coding of Problems for an Electronic Computing Instrument. 1946 as reprinted in The

- Origins of Digital Computers edited by B. Randell.
- ORGANICK, E. Computer System Organization, The B5700/B6700 Series. ACM Monograph Series, Academic Press. 1973.
- OSSANNA, J.F. and others. Communications and I/O Switching in a Multiplex Computing System. AFIPS, Proceedings of FJCC, Vol. 27. 1965.
- OUTRED, C.F.J. Variable Microprogramming and Its Implications. Infotech State of the Art Report, No. 23.
- OZKARAHAN, E.A. and others. RAP - An Associative Processor for Data Base Management. AFIPS, Proceedings of NCC, Vol. 44. 1975.
- POPEK, G.J. and FARBER, D.A. A Model for Verification of Data Security in Operating Systems. Communications of the ACM, Vol. 21: No. 9. 1978.
- RANDELL, B. The Origins of Digital Computers (Selected Papers). Springer-Verlag. 1975.
- REDDI, S.S. and FEUSTEL, E.A. A Conceptual Framework for Computer Architecture. ACM Computing Surveys, Vol. 8: No. 2. 1976.
- ROSEN, S. Electronic Computers: A Historical Survey. ACM Computing Surveys, Vol. 1: No. 1. 1969.
- ROSIN, R.F. Supervisory and Monitor Systems. ACM Computing Surveys, Vol. 1: No. 1. 1969.
- RUSSELL, R.M. The Cray-1 Computer System. Communications of the ACM, Vol. 21: No. 1. 1978.
- SCARROTT, G.G. Fourth Generation Computer Design. Infotech State of the Art Report, No. 1.
- SCARROTT, G.G. System Design Objectives for the 1970's. Infotech State of the Art Report, No. 17.
- SCHROEDER, M.D. and SALTZER J.H. A Hardware Architecture for Implementing Protection Rings. Communications of the ACM, Vol. 15: No. 3. 1972.
- SMITH, D.R. Computer Structure Language (GSL). Proceedings of International Symposium on CHDL and Their Applications, 1975.
- STEEL, R. Another General Purpose Computer Architecture. Computer Architecture News, Vol. 5: No. 8. 1977.
- STEVENS, W.P. and others. Structured Design. IBM System Journal, Vol. 13: No. 2. 1974.
- STRACHEY, C. The Interaction of Software Engineering and Machine Structure. Infotech State of the Art Report, No. 1.

- van WIJNGAARDEN, A. and others. Revised Report on the Algorithmic Language ALGOL 68. Acta Informatica, Vol. 5. 1975.
- WILKES, M.V. Time-sharing Computer Systems. MacDonalD/American Elsevier, 1968.
- WILNER, W.T. Design of the Burroughs B1700. AFIPS, Proceedings of NCC, Vol. 41. 1972.
- WILNER, W.T. Burroughs B1700 Memory Utilization. AFIPS, Proceedings of NCC, Vol. 41. 1972.
- WILNER, W.T. Structured Programs, Arcadian Machines and the Burroughs B1700. in Lecture Notes in Computer Science, Springer-Verlag.
- WIRTH, N. The Programming Language PASCAL. Acta Informatica, Vol. 1. 1971.
- WIRTH, N. MODULA: A Language for Modular Multiprogramming. Software-Practice and Experience, Vol. 7: No. 1. 1977.
- WULF, W.A. C.MMP A Multi-mini-processor. Infotech State of the Art Report, No. 17.
- WULF, W.A. and others. Overview of the HYDRA Operating System Development. Operating Systems Review, Vol. 9: No. 5. 1975.
- WULF, W.A. and others. HYDRA: The Kernel of a Multiprocessor Operating System. Communications of the ACM, Vol. 17: No. 6. 1974.

ADDENDA

- DIJKSTRA, E.W. and others. Structured Programming. Academic Press. 1972.
- HANLAN, A.E. Content Addressable and Associative Memory Systems. IEEE Transactions, EC - 15, 4. 1966.
- PARNAS, D.L. On the Criteria to be used in decomposing systems into Modules. Communications of the ACM, Vol. 15: No. 12. 1972.