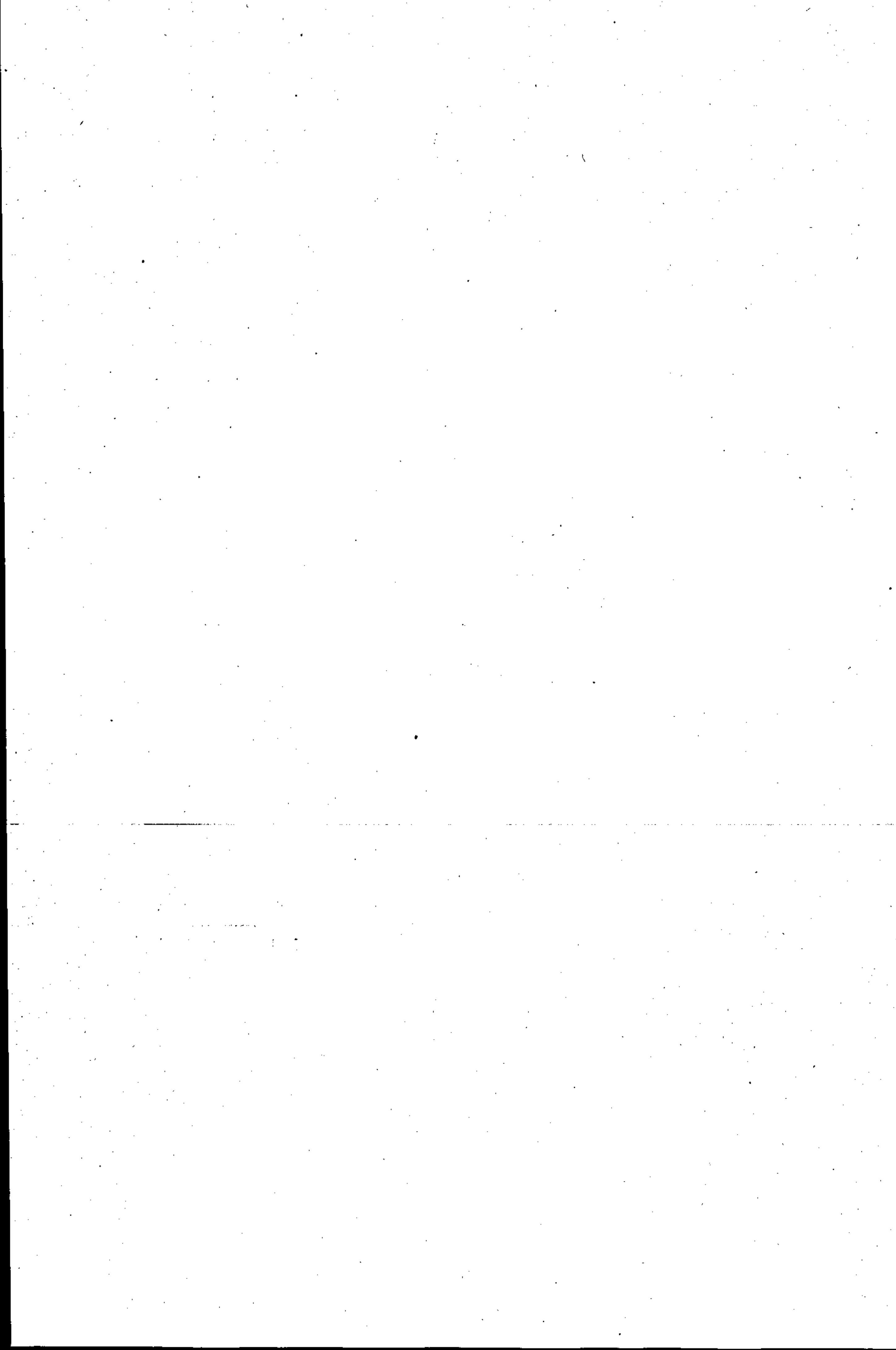
BLLID NO:- D37574/81

LOUGHBOROUGH
UNIVERSITY OF TECHNOLOGY
LIBRARY

AUTHOR/FILING TITLE		
WOODWARD, M		
ACCESSION/COPY NO. 192008/02		
VOL. NO.	CLASS MARK	
-5 JUL 1985 -1 JUL 1986 -1 DEC 1989	LOAN COPY	

019 2008 02



SOME ASPECTS OF THE EFFICIENT USE OF MULTIPROCESSOR CONTROL SYSTEMS

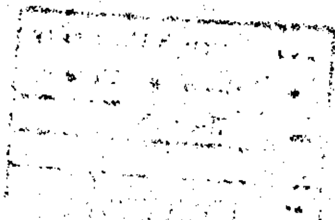
by

Michael Charles Woodward

A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of Doctor of Philosophy of the Loughborough University of Technology January, 1981

Supervisor: I. A. Newman, PhD.
Department of Computer Studies.



© by Michael Charles Woodward, 1981.

Loughborough University
of Technology Library
Date June 81
Class
Acc. No. 192008/02

D E C L A R A T I O N

I declare that the following thesis is a record of research work carried out by me, and that the thesis is of my own composition. I also certify that neither this thesis nor the original work contained therein has been submitted to this or any other institution for a degree.

M.C.WOODWARD

ACKNOWLEDGEMENTS

The author would like to express his thanks to all those who have assisted with the research reported in this thesis: to Dr. I. A. Newman for his direction and encouragement; to Professor D. J. Evans for his interest and to many colleagues for times of discussion. Thanks are also expressed to Mrs. S. Mercer for her dextrous typing. Thanks must also be given to the Science Research Council for the provision of a Research Studentship.

Finally, many thanks must go to the author's parents for their concern and encouragement during the years of research.

C O N T E N T S

Page

CHAPTER 1: INTRODUCTION

1.1. Introduction 2
1.2. Efficiency Considerations 4
1.3. Motivation for Research 6
1.3.1. Hardware Level 6
1.3.2. Systems Software 7
1.3.3. User Software 8
1.4. Proposed Areas of Investigation 9
1.5. Framework of Thesis 10

CHAPTER 2: MULTIPROCESSOR HARDWARE

2.1. Introduction 13
2.2. Multiprocessor Organisation 15
2.3. Commercial Multiprocessors 21
2.4. Multiprocessors in Research 23
2.5. Multiprocessor Systems and Reliability 34
2.6. Memory Contention 36

CHAPTER 3: SOFTWARE CONSIDERATIONS IN MULTIPROCESSORS

3.1. Introduction 41
3.2. Operating System Organisation 43
3.3. Synchronisation 46
3.4. Software Reliability 51
3.5. Parallel Processing 53

CHAPTER 4: THE INVESTIGATION OF A MODEL OF A MULTIPROCESSOR

4.1. Introduction 57
4.2. Model of a Multiprocessor 58
4.3. Derivation of Computing Power 63
4.4. Round-Robin Servicing 67

4.5. Priority Servicing 69
4.6. Constraints for Effective Configurations 74
4.7. Analysis of Performance 76
4.8. Refinement of Servicing Policy 80
4.9. Application of Formulae 87

CHAPTER 5: THE ABSTRACT RESOURCE RING - A SYNCHRONISING TOOL

5.1. Introduction 94
5.2. The Abstract Resource Ring 96
5.3. Multiple Rings 104
5.4. Temporary Resources 108
5.5. A Comparison of Synchronising Tools 112
5.6. Multiple Users of a Resource 125

CHAPTER 6: THE ABSTRACT RESOURCE RING AND RELIABILITY

6.1. Introduction 130
6.2. Classification of Failures 131
6.3. Initial Death Detection 133
6.4. Rigorous Death Detection 139
6.5. Failure Within ARR Routines 147
6.5.1. GETRES Routine 147
6.5.2. PUTRES Routine 147
6.5.3. Recovery Procedure 149
6.6. Addition and Replacement of Processors 151
6.7. Reliable Update 153
6.8. Application of Reliable Update to the ARR 159
6.9. Failures Due to Other Errors 160
6.10. Self Stabilising Techniques 162

CHAPTER 7: PARALLEL PROCESSING AND THE APPLICATION OF THE ABSTRACT RESOURCE RING

7.1. Introduction 167
7.2. System Configuration 168
7.3. Parallel Processing System Design 170

7.4. ARR Implementation 173
7.5. Reliability and Recovery Procedures 178
7.6. Performances 181

CHAPTER 8: GARBAGE COLLECTION - A MULTIPROCESSOR APPLICATION
8.1. Introduction 188
8.2. Definition of Terminology 191
8.3. Lamport's Algorithm 192
8.4. Chaining Algorithm 194
8.5. Comparison of Marking Algorithms 200

CHAPTER 9: CONCLUSIONS
9.1. Summary 206
9.2. Areas for Further Research 209

REFERENCES: 211

APPENDIX 1: AN IMPLEMENTATION OF THE ABSTRACT RESOURCE RING 221

APPENDIX 2: AN IMPLEMENTATION OF THE RELIABLE UPDATE 251

CHAPTER 1

INTRODUCTION

1.1. Introduction

Computer technology, particularly at the circuit level, is fast approaching its physical limitations. As future needs for greater power from computing systems grows, increases in circuit switching speed (and thus instruction speed) will be unable to match these requirements.

Greater power can also be obtained by incorporating several processing units into a single system. This ability to increase the performance of a system by the addition of processing units is one of the major advantages of multiprocessor systems. Four major characteristics of multiprocessor systems have been identified (28) which demonstrate their advantage. These are:-

Throughput

Flexibility

Availability

Reliability

The additional throughput obtained from a multiprocessor has been mentioned above. This increase in the power of the system can be obtained in a modular fashion with extra processors being added as greater processing needs arise. The addition of extra processors also has (in general) the desirable advantage of giving a smoother cost - performance curve (63). Flexibility is obtained from the

increased ability to construct a system matching the user requirements at a given time without placing restrictions upon future expansion. With multiprocessor systems, the potential also exists of making greater use of the resources within the system.

Availability and reliability are inter-related. Increased availability is achieved, in a well designed system, by ensuring that processing capabilities can be provided to the user even if one (or more) of the processing units has failed. The service provided, however, will probably be degraded due to the reduction in processing capacity. Increased reliability is obtained by the ability of the processing units to compensate for the failure of one of their number. This recovery may involve complex software checks and a consequent decrease in available power even when all the units are functioning.

1.2. Efficiency Considerations

The use of multiprocessor systems potentially provides many advantages over single processor systems. However, caution must be expressed as regards the potential of multiprocessor systems. These two aspects are summed up in two well known proverbs:

"Many hands make light work"

"Too many cooks spoil the broth".

A certain overhead has to be faced in the construction of multiprocessor systems. At the hardware level, this overhead is manifest in the cost of interconnection between the processors and memory of the system. This may impose delays within the hardware not experienced by a single processor system. Also, the interaction between processors places an overhead upon realisable processing power. In practical realisations of multiprocessor systems, these overheads must be considered, and it is known that for certain organisation, a limit exists upon the number of processors that may be usefully added to a system (35).

At the software level, similar problems of interaction between the processors arise. If they are actually to co-operate then it is necessary for the processors to synchronise. This may be due to operating system functions or because of interaction between tasks running on different processors. The synchronising overheads can prove to be unnecessarily large if there is a poor choice of synchronising tool.

The interactions between tasks can also impose great inefficiencies. A poorly designed program may impose many more synchronisations upon various tasks than a well designed solution to the same problem. Poor design may, therefore, impose extra costs upon the processing capacity of the system as a whole.

The meaning of the term efficiency is, of course, contentious and a definition of the concept, in the context of multiprocessor systems, is needed to enable an effective discussion of the "efficiency" of such systems to be undertaken. Efficiency may be expressed as the amount of useful work which can be accomplished in relation to the potential capacity of the components. At the hardware level, the potential capacity of a multiprocessor system could be expressed as the sum of the power of the components in terms of work which could be accomplished. The realisable power is reduced by the overheads associated with the interconnection of and interaction between the processors. This available power would be further reduced at the software level by the costs of intercommunication and synchronisation.

1.3. Motivation for Research

The problems associated with multiprocessor systems (indeed with any computer system) may be split into three broad classes:-

i) Hardware

ii) Systems Software

iii) User at Application Software.

If an overall system is to be efficient, that is make good use of the total system resources, all three areas must be considered and given due merit. The power of a system with sophisticated hardware and a well designed operating system may be wasted if badly designed or inappropriate applications are executed on it.

1.3.1. Hardware Level

It is, perhaps, at this level that consideration should first be given to efficiency as, no matter how well designed, software run on poor hardware cannot make it operate faster than is feasible as the maximum power of the system is inevitably limited by the hardware.

For multiprocessors with shared memory, one of the major areas of consideration must be that of memory contention. The degree of memory contention is dependent upon the number of processors accessing the shared memory and the use to which it is put. As will be noted in Chapter Two, some authors have developed complex models to study

the behaviour of multiprocessor systems, yet these are often specialised, being applicable to only a specific class of hardware.

1.3.2. Systems Software

Having designed and built (or purchased) a multiprocessor system, several possibilities lie before the user in the organisation of the software on the machines. Whatever regime is chosen for the multiprocessor, be it master/slave, an anonymous treatment of the processors or a compromise, questions will arise as regards synchronisation between the processors and also as regards recovery on the failure of one (or more) of the processors.

One of the major advantages of multiprocessor systems is their ability to provide processing capabilities even when one or more of the processors have failed. If use is to be made of this ability to recover, then some forms of hardware synchronisation may be unacceptable. As will be seen (Chapter Three), if one processor has lowered a semaphore and all other processors are waiting and the running processor dies then the system may permanently hang waiting for the semaphore to be raised.

Of the software mechanisms that have been developed, most (e.g. critical regions, readers and writers) require a lower level of synchronisation upon which they may be based. Some algorithms have been developed whereby synchronisation may be achieved by software, but rarely are these algorithms considered in terms of reliability or error recovery.

The algorithms also tend to become less efficient as the load placed upon them increases.

1.3.3. User Software

Having obtained an efficient system, the problems at the user level then become apparent. On single processor systems, the bad construction of programs can yield vast inefficiencies in machine usage. Some design methodologies are being popularised nowadays (20,44), and these have been shown to provide improvements in efficiency over many levels, including those of systems analysis and programming.

With multiprocessor systems, the potential for resource wasting increases with the possibility of processes vying for a resource instead of co-operating over its use.

When designing multiprocess (or parallel) programs, care and foresight must be used to develop programs which suitably represent the parallelism of the problem. The techniques that should be used in the detection and exploitation (either human or automatic) of a problem are not yet fully understood, though some progress is being made in this direction (64).

1.4. Proposed Areas of Investigation

There are, therefore, an extremely large number of topics relating to multiprocessor systems which would merit investigation and, indeed, there is much research work currently being undertaken in this area. Since the overall efficiency of a multiprocessor system relies on the efficiency of each of the three areas mentioned above, consideration has been given to a topic from each, though greater emphasis is placed upon the second area.

It was felt, from the above discussion, that, at the hardware level, there was scope for a general model which would be of use in the early stages of a system design exercise and would provide some bounds for the maximum realisable power of a multiprocessor system. The model should take into account the type of interconnection and the type of use to be made of the system.

At the level of systems software, it was decided to investigate the subject of synchronisation between the processors. As was noted above, certain disadvantages exist with the algorithms found in the literature, and it was hoped that a reappraisal of the problem could produce a solution with different operational characteristics.

Finally, a particular user application was chosen for investigation to highlight the difficulties of designing user software for a multiprocessor system.

1.5. Framework of Thesis

Chapter Two discusses the possible organisations of multiprocessor systems and outlines the problems faced at the hardware level with each organisation. Chapter Three deals with the corresponding software organisations and problems. The difficulties of synchronisation between processors are discussed and the existing, published, solutions are described. Some aspects of the current state of research into reliability are also described in the chapter.

In Chapter Four, a model of a multiprocessor system is introduced. This model is then used to develop formulae for bounds which may be placed upon the memory contention experienced by multiprocessor computer systems. Results obtained from these are compared with timings from actual hardware.

Chapter Five deals with the development of a software synchronisation tool (the Abstract Resource Ring or ARR). Two distinct implementations of the basic technique are introduced. The tool is compared with other algorithms found in the literature. In Chapter Six, the ARR is developed with specific reference to reliability and error recovery within multiprocessor systems. In Chapter Seven, the role of the ARR in a parallel processing system is described, including discussion of its use in the realm of reliability.

Chapter Eight, by way of an example, shows the difficulties of writing

"efficient" software for multiprocessor systems.

Finally, the thesis is drawn to a close by bringing together some conclusions and pointing to areas where further research might be pursued.

CHAPTER 2

MULTIPROCESSOR HARDWARE

2.1. Introduction

In 1966 Flynn (31) introduced a classification for digital computers which is in common use today. By observing parallelism in both the instruction stream and the data stream for computers, four classes were identified:-

1) Single Instruction Single Data Stream (SISD)

This is the standard serial uni-processor system

2) Single Instruction Multiple Data Stream (SIMD)

In this classification, a single instruction is executed by several arithmetic units with different data. This yields the array or vector processors

3) Multiple Instruction Single Data Stream (MISD)

This class of hardware would involve a single data item being operated upon by several different instructions. A realistic interpretation of a processor of this class is difficult, although it may include a Dataflow architecture.

4) Multiple Instruction Multiple Data Stream (MIMD)

In this class of hardware lie systems of processors which may operate independently upon different sets of data with different programs yet may also co-operate upon a computation if required.

The latter classification may be subdivided into loosely coupled and tightly coupled multiprocessor systems. Most network systems and distributed computing applications (e.g. (69)) would be examples of loosely coupled MIMD computers. The processors have no shared storage medium, being connected by relatively low speed communication lines only. With closely-coupled multiprocessors, however, the individual processors have access to a shared or common storage medium and may communicate or co-operate through this medium. Usually this storage medium is core (or a similar high speed random access medium), though shared disc or drum systems equally fall into this classification, as would independent machines with separate stores and a high speed memory to memory link.

This thesis is, however, concerned with the shared memory version of the latter group of machines (i.e. closely coupled MIMD systems). In the following section, various hardware organisations for this type of system are described. Some special purpose systems which have been developed by various research teams are then discussed. The chapter closes by describing two further areas of research in multiprocessor hardware.

2.2. Multiprocessor Organisation

The basic model of a multiprocessor system is of a number of processor units connected to memory and input-output devices. It is the manner of this connection which gives rise to the different organisations.

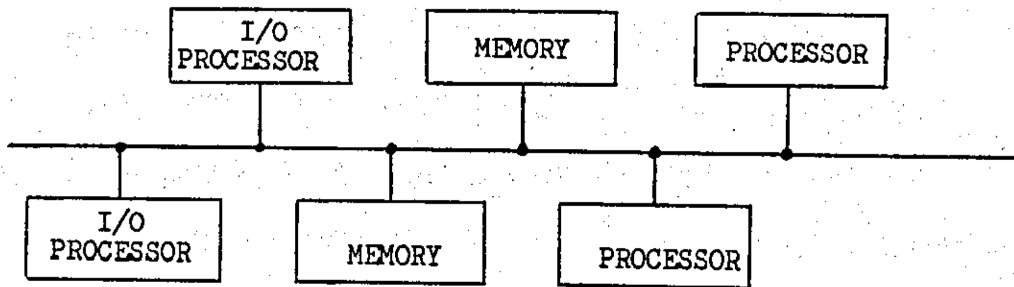
Enslow (28) has "identified three fundamentally different system organisations used in multiprocessors:

- Time shared or common bus
- Crossbar switch matrix
- Multiport memories

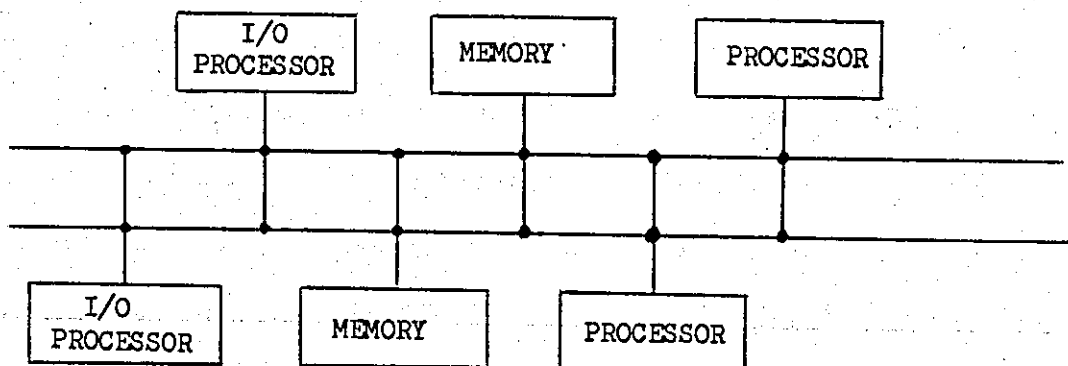
... the entire scope of interconnection schemes is much larger and certainly more complex ..... these categories nonetheless form a useful base for a discussion of the organisation of multiprocessor systems...."

a) Time shared or common bus (Figure 2.2.1)

With this organisation, all the system components (processors, memory modules and I/O devices) are connected by a common communication path (the bus). The operation of this system is in concept simple, though in practice it may be quite complex. A unit wishing to communicate with another must first ascertain that the bus is free. It then places on the bus the address of the requested unit together with any other information required in the communication. Units which may potentially receive communication must inspect the bus for their address being



a) Time-shared/Common Bus Organisation - Single Bus



b) Time-shared/Common Bus Organisation - Dual Bus

Figure 2.2.1.

transmitted. The necessary synchronisation over the use of the bus may be handled by an interface between each component and the bus in co-operation with a single arbitration unit for the bus.

With this organisation, however, as the number of components increases, the load placed upon the bus increases, and the bus may become a bottleneck. Also, if the bus fails, then the system as a whole is unusable. To overcome both these problems, the bus may be duplicated, though this greatly increases complexity.

b) Crossbar switch matrix (Figure 2.2.2.)

With this organisation, the number of connections between processors and memories is increased such that a different access path exists from each processor to each of the memory modules. The important characteristic of these systems is that transfers to or from each memory module can potentially be made simultaneously. Whilst this design is not complex, much circuitry is required to cope with the potential contention at each interconnection in the crossbar. An example given in the literature (29) gives, for a twenty-four 32-bit processor system with 32 memory modules, the number of circuits required in the crossbar switch as two to three times the number required for an IBM System 360 Model 75.

Expansion of this organisation is, however, conceptually straight-

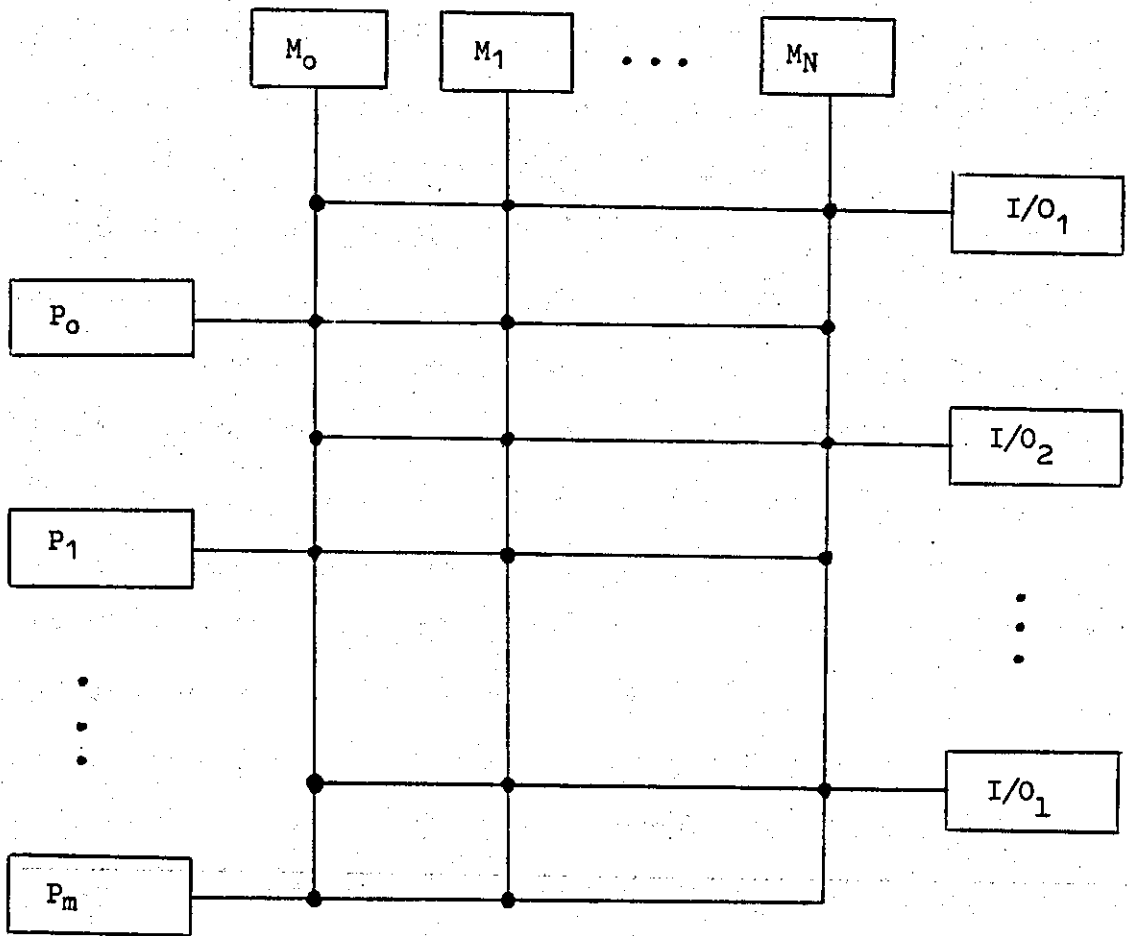


Figure 2.2.2. Cross-bar Switch Organisation

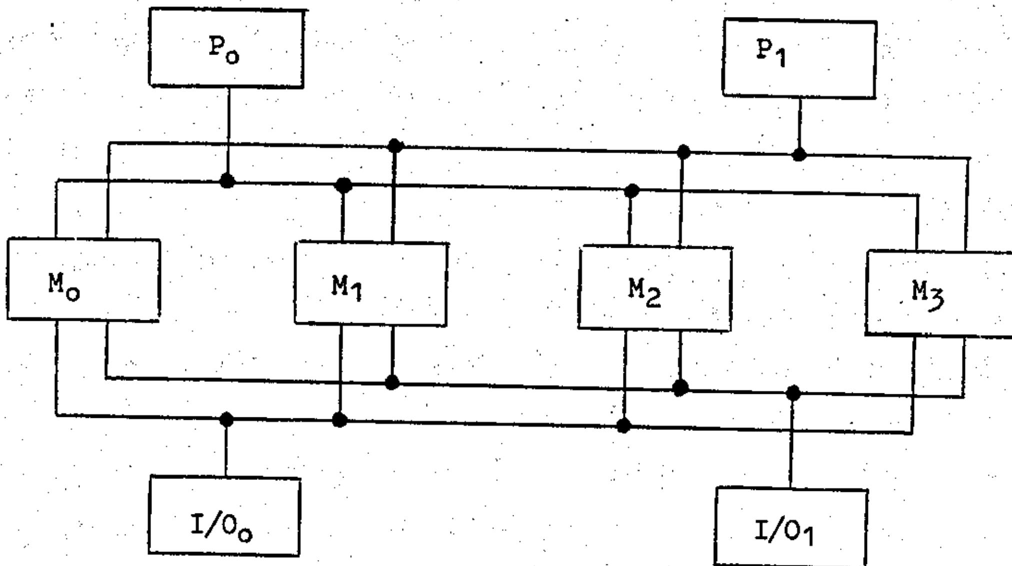
forward requiring only the size of the switch to be increased.

c) Multiport Memories (Fig. 2.2.3)

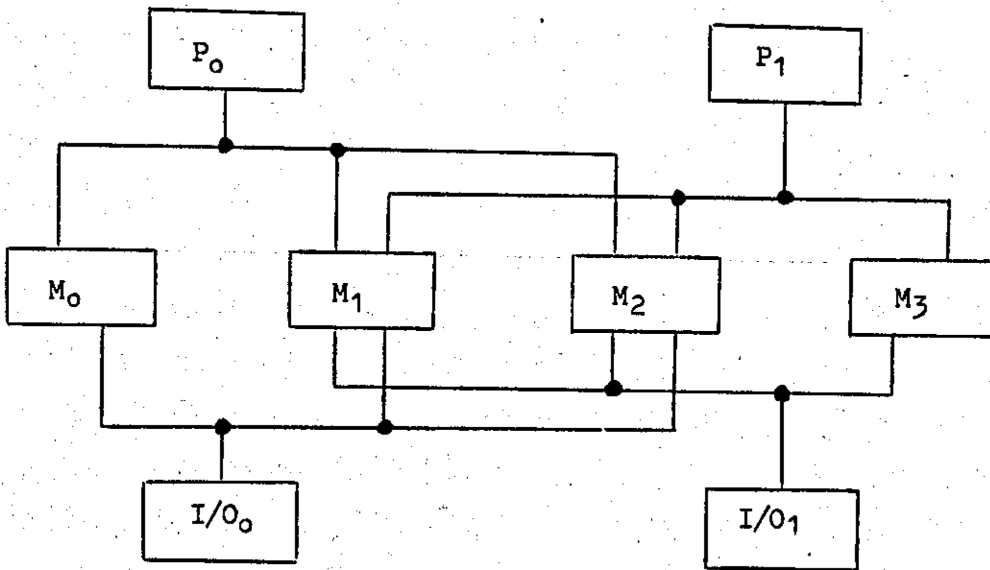
If the logic controlling switching and arbitration, which is distributed among the interconnections in the crossbar, is concentrated at the interfaces to the memory modules then multiport memory systems are obtained. Often, preassigned priorities are given to the parts to reduce the contentions which may arise allowing the system to be configured as required at each installation. One advantage with multiport memory systems is the ease with which private memories (that is memories accessible to only one processor) may be given to each processor. (Figure 2.2.3b) This has advantages with respect to security against unauthorised access of data, but has disadvantages with respect to reliability. Since only the one processor may access data in its private memory, if that processor fails,access cannot be made to the data and it is "lost".

Another disadvantage with multiport memories is due to the fixed number of ports (which is generally small). This restricts the number of processors that can be connected to a single memory module and thus limits the maximum size of the system.

Unfortunately, although this classification is intended to provide a general description of the hardware, many practical systems cannot be neatly assigned to one or other of the categories.



a) Multiport Memory - No Private Memory



b) Multiport Memory - With Private Memory

Figure 2.2.3. Multiport Memory Organisation

2.3. Commercial Multiprocessors

Many computer manufacturers are willing to supply multiprocessor systems. Indeed, many so-called uniprocessor systems are actually multiprocessor systems, with the different processors being given well defined tasks. Examples of such systems are the larger ICL 1900 systems and the CDC 6600, in which specially designed processors are dedicated to the role of peripheral processors, relieving the main processor of this duty.

Some manufacturers, e.g. IBM, CDC and UNIVAC, supply multiprocessor systems with operating systems able to take advantage of the whole configuration. Examples of this are the IBM 370/158 MP and IBM 370/168 MP both of which may be operated under OS/VS2 (1,51). These systems contain no local memory, but contain special hardware to perform some memory mapping as well as handling inter-processor interrupts and the serialisation of processor cycles. The serialisation is required to prevent interruption of instructions requiring several memory cycles (e.g. Test and Set). Hardware is also included to enable one processor to interrogate, or set, the status registers of another. The OS/VS 2 operating system allows the processors to be run in multiprocessor mode or as several uniprocessors. The control program is considered in two parts. One part is concerned with servicing functions local to each processor, the other with global functions of the multiprocessor as a whole. Locks, software flags, are used to prevent several processors performing sections of code

simultaneously. These locks enable software functions to be serialised in a similar manner to the hardware.

Other manufacturers are willing to supply multiprocessor configurations, though without any software to control the system in multiprocessor mode. Examples of these are Ferranti, Texas Instruments and Perkin Elmer. Such systems will contain the hardware necessary to handle bus contention, though in some instances, instructions requiring multiple memory cycles may be interruptable.

2.4. Multiprocessors in Research

Many organisations and research groups are currently investigating the problems peculiar to multiprocessor systems, leading, in some instances, to the building of multiprocessors. Often, however, the hardware designs of these machines cannot be directly related to one of the major classes considered in the previous section.

One of the foremost groups is that at Carnegie-Mellon University. In 1971, a project was started there to develop a multiprocessor computer system based on the PDP-11 minicomputer. This resulted in, the now famous, CMMP system (67). The project arose, not only to perform research in multiprocessor systems but also to provide computational power for existing projects. The organisation of the system is shown in Figure 2.4.1.

Each processing element, up to a design total of 16 in the development system, consists of a processor, some local memory and some local devices. Two crossbar switches have been added. The first connects the processors to shared memory, the second connects them to shared peripherals. Each processor may access all shared devices and all shared memory. The processing elements include interface hardware to these crossbar switches to convert locally generated addresses into addresses suitable for the switch architecture.

The hardware also contains a system clock, providing a clock interrupt to all the processors, and an interprocessor interrupt mechanism.

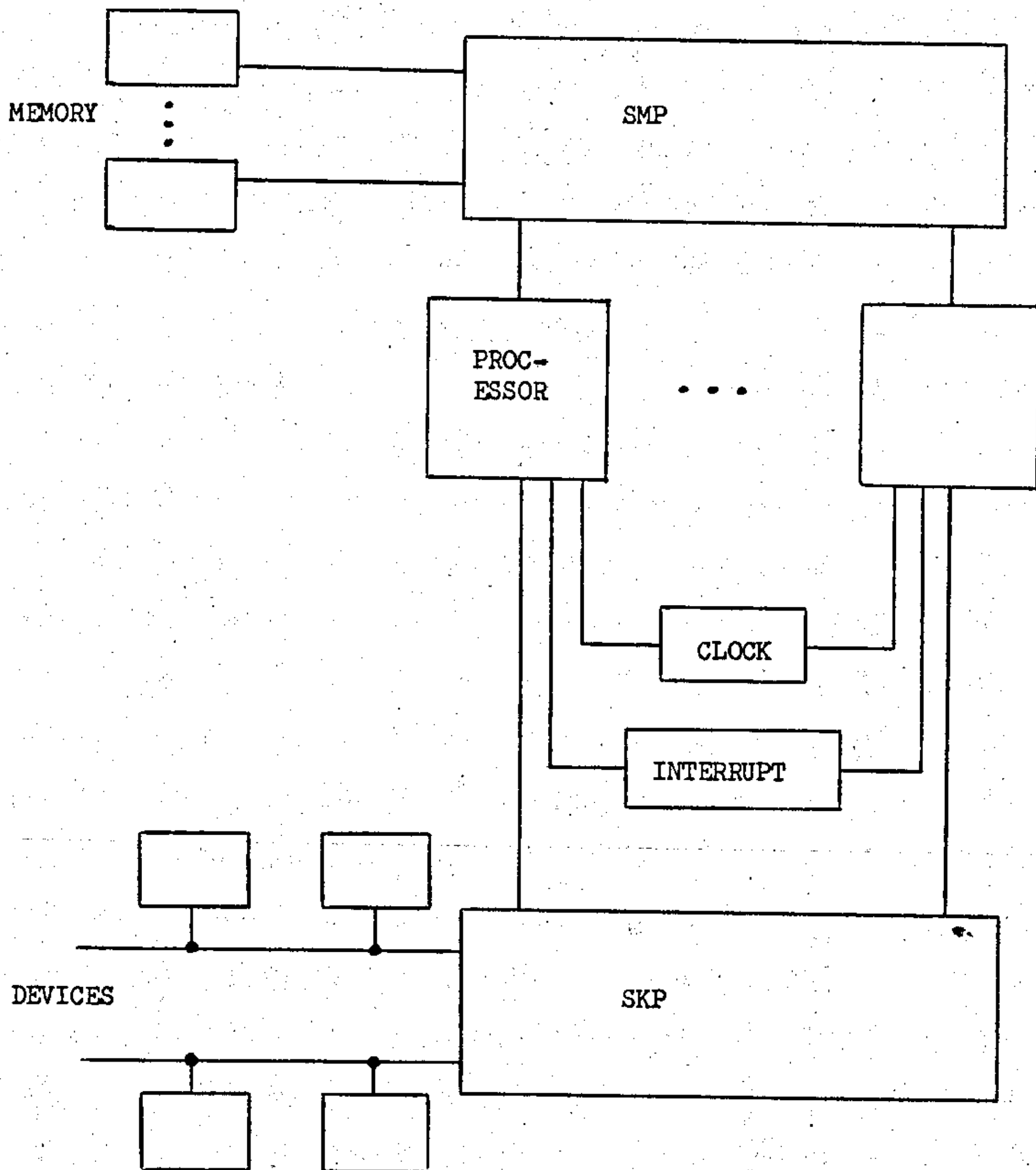


Figure 2.4.1. Basic CMMP Hardware Organisation

With the latter, one processor may interrupt any number of its counterparts at one of several interrupt levels.

One of the areas that provided some design problems was the area of memory contention (see section 2.6.). Calculations based on Strecker's formulae (59) were made during the design stages to attempt to find cost-effective processor and memory configurations. Research was also undertaken in aspects of systems software. This led to the development of the kernel of the operating system, called HYDRA (66). HYDRA is not in itself an operating system, but provides all the mechanisms for building one.

The group are currently developing a multiprocessor system, Cm*, based on microprocessors which "is intended to be a testbed for exploring a number of research questions concerning multiprocessor systems, for example: potential for deadlock, structure for inter-processor control mechanisms, modularity, reliability and techniques for decomposing algorithms into parallel co-operating processes"(60). The hardware design chosen for this system, whilst forming a multi-processor system with all memory sharable, closely links memory modules with processors. A network of buses provides access to non-local memories, as is shown in Figure 2.4.2.

Each processor-memory module contains a local switch (Slocal). This switch provides the first level of memory mapping. References to the local memory are serviced directly. References to non-local memory

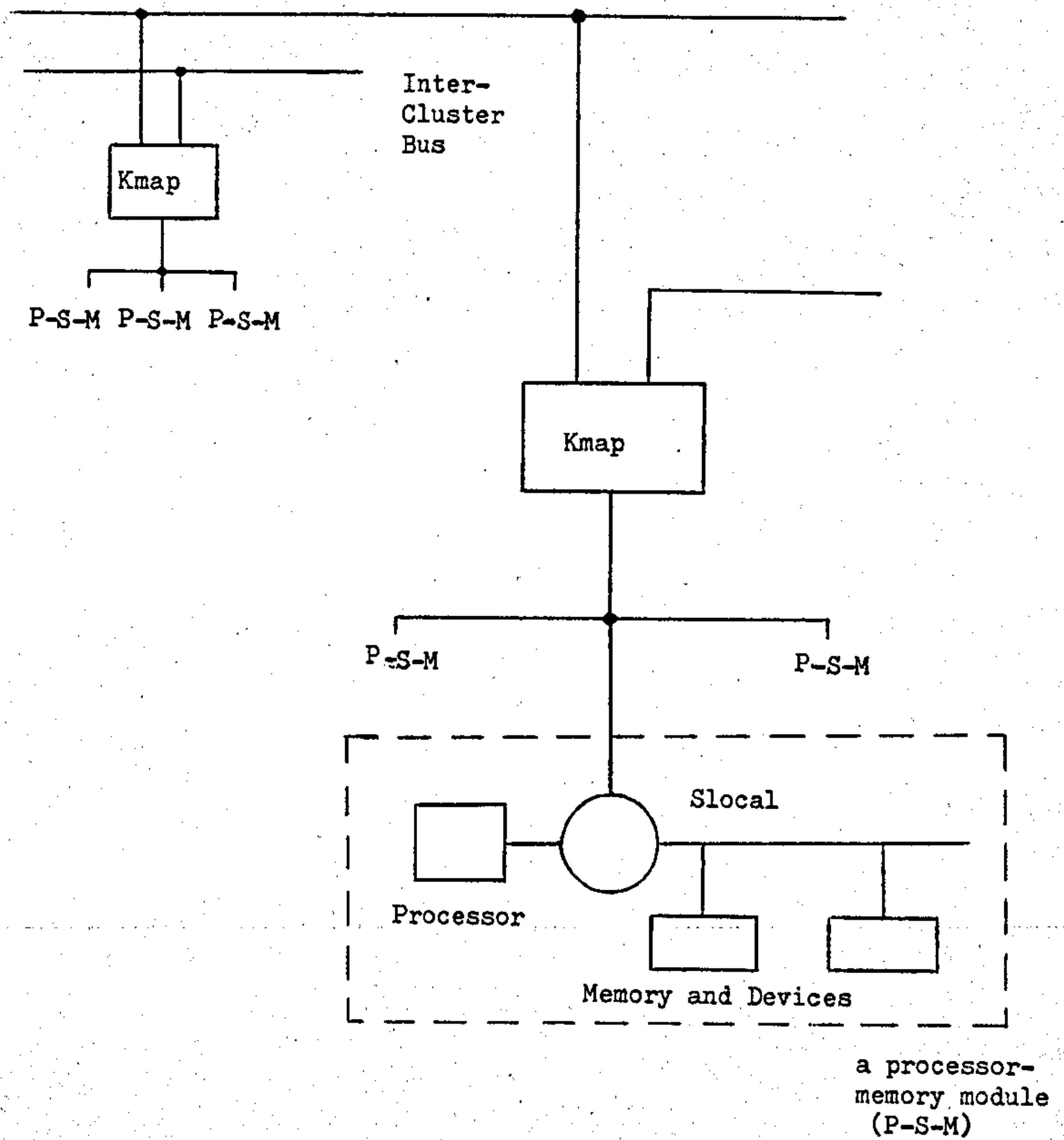


Figure 2.4.2. A Simple Cm* System

modules are placed, by the Slocal, onto a bus connecting the switch to a Kmap processor. The Kmaps are mapping processors which provide the routing mechanism for access to remote memory modules. Each Kmap is connected to several processor-memory modules to give a cluster and the clusters are also connected by buses.

When a Kmap processor receives a request for memory access, the request is sent either to the correct Slocal, if the reference is made to memory within the cluster, or the request is passed to another Kmap for servicing.

This hardware organisation gives highly asymetrical memory access times. Access to local memory suffers minimal degradation, while accesses to remote clusters may experience a large overhead due to the routing of the request. In order to make efficient use of the hardware, a large proportion of memory accesses should be to the local memory. "It has been hypothesized that the local hit ratio would lie in the range 85 to 95 percent, in which case, the effect of non local references would be 'reasonably' small". (61)

A second unusual hardware organisation has been developed by a group in Siemens AG. The SMS 101 (46) is also a multi microprocessor system, but designed with particular reference to problems of the class of large systems of differential equations or on-line process control. In many senses, the system is not strictly a multiprocessor (the processors do not directly share some common store) yet all processors

can access the memories of other processors.

The basic hardware design is shown in Figure 2.4.3. The system comprises a main processor consisting of a processor and memory. This is connected via a single bus to several further processor-memory modules. Each of the modules is interfaced to the bus through a switch. The main processor controls the bus and also the switches in each of the modules. Each of the modules has the capacity for independent program execution.

The operation of the system falls into distinct phases while running a program. Firstly, the main processor distributes the code and data among the modules. Each of the modules then completes its portion of the workload. In the third phase any results or variable changes derived by the modules are distributed to the other processor allowing the cycle to be repeated. The switches are used to govern the distribution of the information derived, allowing it to be directed in a number of ways.

In the United Kingdom, several groups are investigating the problems of multiprocessor systems. One group is concerned with the development of the CYBA-M system (2,26,32).This system consists of up to 16 Intel 8080 microprocessors, each with some private memory. These micro processors are connected, via a switch, to two banks of shared memory. The organisation is shown in Figure 2.4.4. Program segments performing well defined functions are assigned to each processor, indeed the

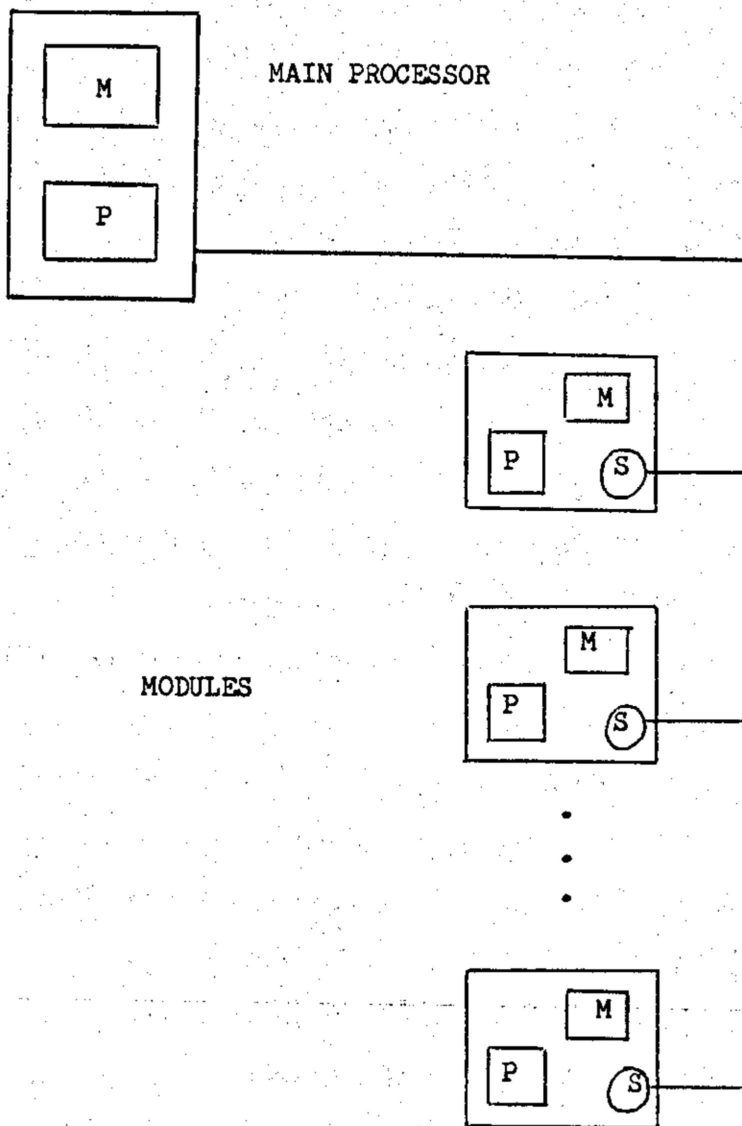


Figure 2.4.3. Basic SMS 101 Design

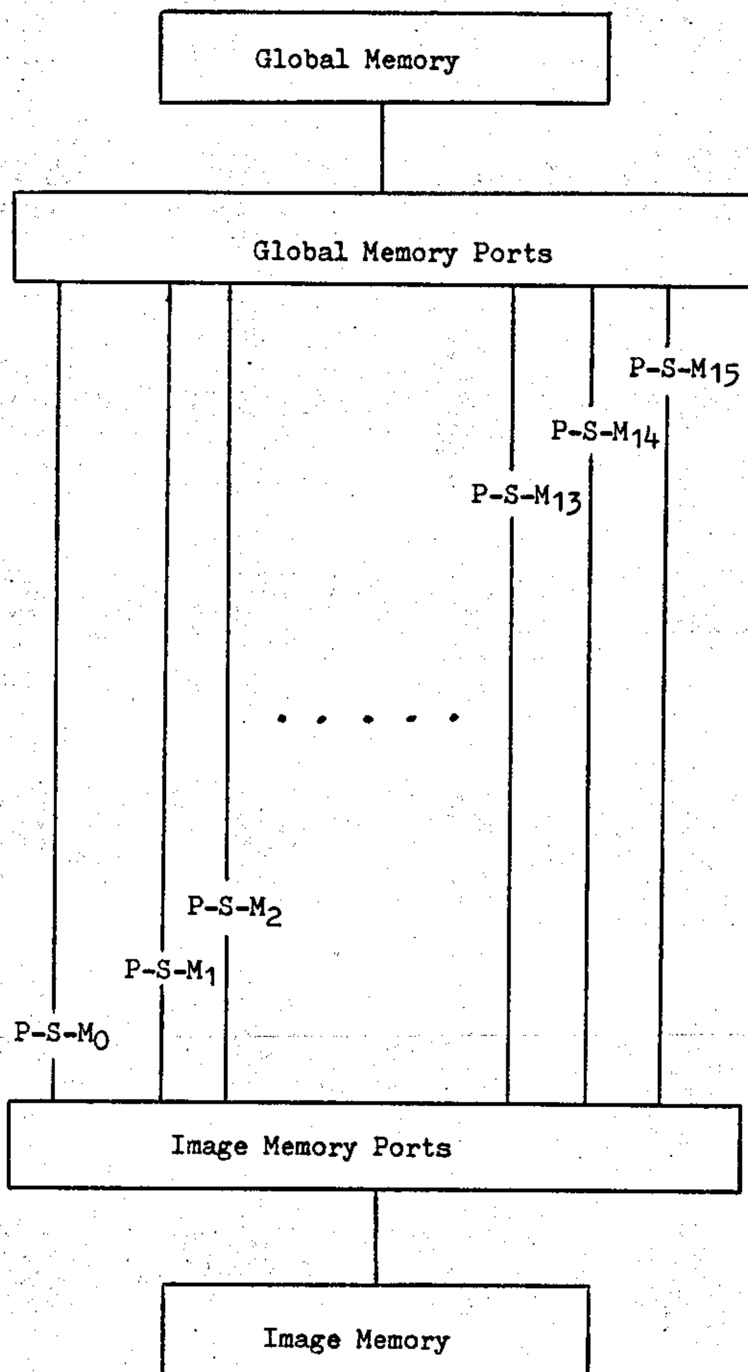


Figure 2.4.4. CYBA-M Hardware Organisation

system is envisaged as a testbed for proving the validity of such assignments. The Global memory is used for inter-process communication. The memory is logically divided into several sections, or lines, each of which is dedicated to a particular communication path. The Image memory is used for accessing peripherals, which are all memory mapped. Again, the memory is partitioned into lines with lines being associated with peripheral registers. Some of the Image memory lines have semaphores associated with them to enable contention over shared peripherals to be resolved. All processors derive their timing from a common clock.

One processor also has connections to the private memories of all the other processors. This processor is used to downline load the program segments to the individual processors and also to provide control and monitoring facilities. To this special processor is attached a keyboard, floppy disc and other peripherals to aid in the setup of the system and the following monitoring.

Another group, at Sussex University, is developing a multiprocessor system which may have application in the office situation (34). The arrangement of this system is of a number of communication highways to each of which several computers (either minis or micros) are attached. The communication highways are themselves interconnected via highway coupler processors (Fig. 2.4.5). The communication highways all use the same protocol, with each processor being interfaced to the highway. This interface includes some buffering of messages to be

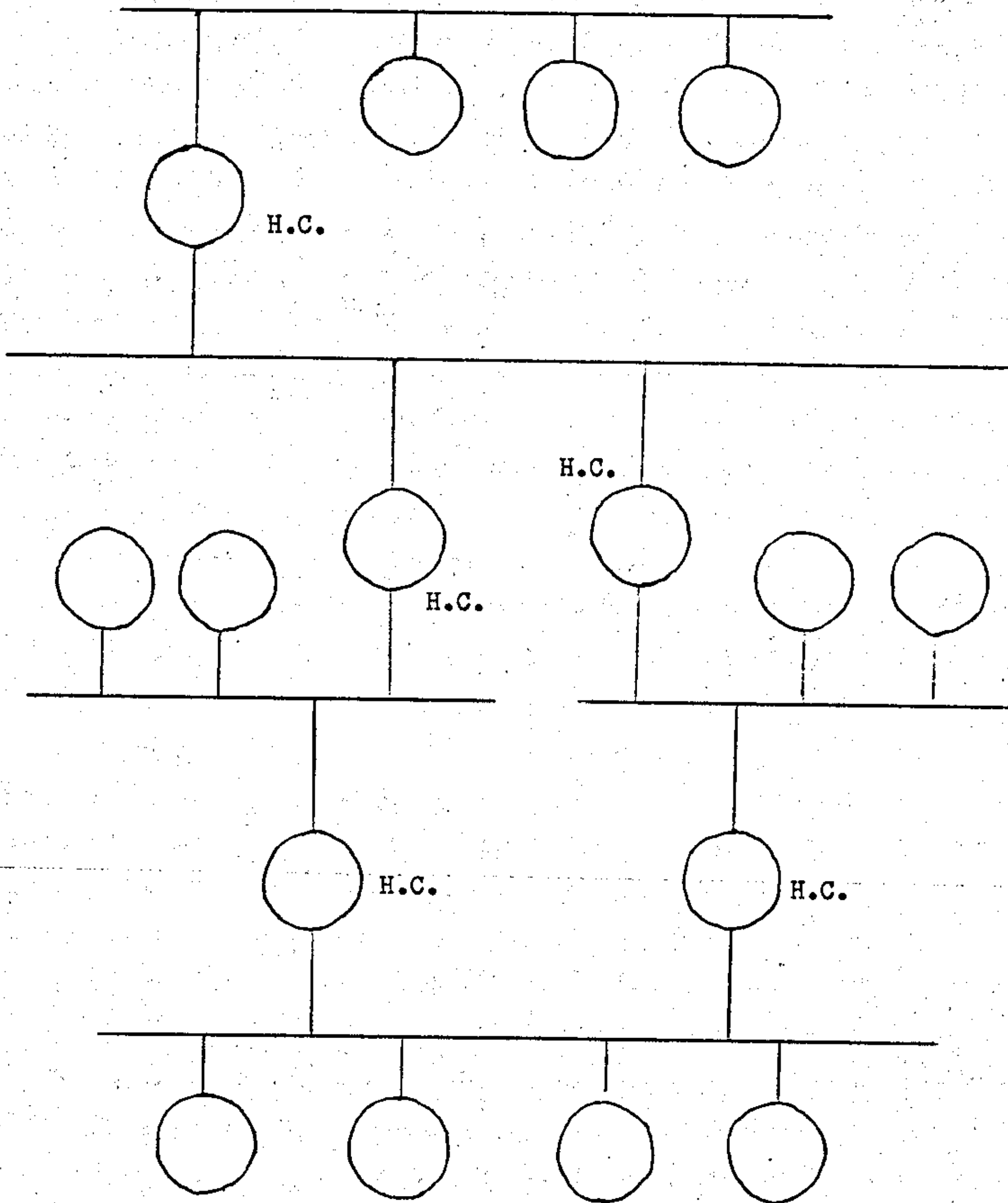


Figure 2.4.5.

transmitted/received on the highway.

It is envisaged that the system would be organised (at the software level) with each processor containing a single application program performing a dedicated function, e.g. a terminal processor or a file handler. Each processor would also contain the necessary software to drive the interface to the highway, this being called the nucleus. As the application programs require service (e.g. access to a file) messages are sent, via the communication network, to the processor running the appropriate service program.

The same group is also investigating the problems at the software to hardware interface in multiprocessor systems (57).

2.5. Multiprocessor Systems and Reliability

One of the major advantages of multiprocessor systems is their ability to continue operation even when one of the processors fails. This ability has been used to advantage in many situations where high availability is one of the system requirements. These applications range from process control to networking. Often, however, special purpose hardware has to be added to enable an adequately high degree of reliability to be obtained.

The TRANSPAC network system (69) in France is typical of many applications where redundancy (that is the duplication of components) is used. In this network, the major routing nodes are dual processors, with many of the other components, including memory modules, being duplicated. One of the two processors at each node operates as the routing processor. The second processor, together with a special hardware module, act as a watchdog over the main processor. If a failure occurs within the processor, then the second processor assumes responsibility for the routing of the network traffic.

Recently, an American Company, Tandem Computers Incorporated, have begun marketing a multiprocessor system, the Tandem Non-Stop System (62). It is claimed, as a consequence of the design and implementation of the hardware and software, that the system can be configured automatically to continue processing despite the failure of any component. A high degree of redundancy is present in the hardware with

most components duplicated and redundancy of a higher order may be incorporated. Some less common features, such as multi-part disc drives, have also been included. However, it appears that the hardware may not be configured to provide memory shared between processors.

A special purpose operating system, the Guardian Operating System, is available and it is claimed that, with the use of the facilities it provides, the failure of hardware components may be made transparent to the users of the system.

2.6. Memory Contention

One subject of particular interest in the field of multiprocessors is that of memory contention (or memory clashing). In a system where several processors are connected to a storage module, it is possible for two or more of the processors to simultaneously request access to the shared storage. In this situation, only one may actually have its request honoured with the others being delayed until they in turn can be serviced.

Many authors have developed statistical models of such situations and have carried out analysis of their performance, and these have appeared in the literature (13,14,etc.). A variety of models have been considered, though each has normally been applicable to a certain type of hardware. A survey of the techniques has been produced by Bhandakar and Fuller (8),but some comments on a few representative papers are given below.

Baskett and Smith (6) consider a model of a multiprocessor consisting of a number of processors and memory modules, each of which may be accessed by all the processors. All the processors and memories are synchronised, that is, all the processors make their requests at the same time with each memory taking the same time to service the requests. If two (or more) processors make a request to the same memory module then only one of the requests is serviced. The access pattern of the processors is random, with all the memory modules

having an equal probability of selection. The authors consider their model particularly applicable to systems where the hardware is bound by the speed of its memory, with emphasis on interleaved memory. They also acknowledge that their model may "describe only a minority of current or proposed multiprocessor systems."

Bhandakar (7) also considers a model in which the processors have no private memory. The model is of a number of processors and memory modules connected by a crossbar switch. The access pattern to the modules is again random, being considered (for each processor) as a sequence of Bornoulli Trials. The phases of a memory access are considered in much more detail with parameters being incorporated into the model to describe the states of the processor and memory during an access. The extra complexity enables Bhandakar to remove the synchronisation constraint present in Baskett and Smith's work. Bhandakar also ignores the effects of input/output operations, as is the general practice in the literature, claiming support from Strecker (59).

Sastry and Kain (56) model a system similar to the above, with a number of processors and memory modules. Each processor can access every memory module, with arbitration logic being incorporated in the memory module to resolve the contention. They direct their investigation towards a situation in which instructions and data are stored in separate memory modules enabling a form of pipelining to be incorporated.

Generally, the analysis adopted to derive formulae from the model is that of Discrete Marcov Chains. This is, indeed, the method adopted by all of the above. Having derived formulae to predict the amount of contention that is experienced by their model, the authors provide simulation results, and occasionally measurements from multiprocessor systems, to support these calculations.

Sastry and Kain, having adopted a model with certain attributes (the separation of code and data) demonstrate the relationship between the memory contention experienced and the parameters of the model defining the attributes. Kurtzburg (47) considers the problem of allocating jobs among a number of memory modules. Having developed his model, the parameters are varied to show how the distribution affects the theoretical memory contention.

Many of the organisations are of a more specialised nature, as is indeed acknowledged by Baskett and Smith. Other models rely on specific organisational decisions to be made by the operating system, the model of Sastry and Kain being such an example. These models, and those making similar assumptions or design decisions, are clearly applicable to a small cross-section of multiprocessor systems.

Other models, for example Bhandakar's require very detailed information on the performance characteristics of the system components. Whilst giving very accurate predictions for the given specification, even slight modification in the hardware may invalidate the accuracy of the

predictions. Also, the more detailed (and, probably, greater quantity of) parameters to the model may make calculations more complex.

In Chapter Four, a model of a multiprocessor is presented which is applicable to a larger number of hardware organisations and, whilst a number of parameters are required, these are not of a highly detailed nature as some of those in the literature.

CHAPTER 3

SOFTWARE CONSIDERATIONS IN MULTIPROCESSORS

3.1. Introduction

The programs written to solve problems often contain discreet sections which do not necessarily have to be executed in a fixed order. On a uni-processor system, the various stages must inevitably be executed sequentially. When a multiprocessor system is used, however, this constraint is removed giving the potential for several parts of a program to be run simultaneously.

In order to exploit the natural parallelism in programs, certain restrictions must be placed upon the software operating on the multiprocessor. The processors must be allocated to the tasks, or parallel sections, within a program and there must be some synchronisation, for example where two or more parallel sections meet (terminate). The synchronisation may be performed purely by software or be based upon some underlying hardware mechanism.

A method must also be provided whereby the user of a multiprocessor system may express the parallelism within his program, either explicitly or implicitly. This may be by the use of language constructs which generate parallel code or by requesting automatic generation of parallel code from a sequential program.

With the availability of several processors in a multiprocessor system, processing may continue despite the failure of one of their number. If this advantage is to be taken, the software on the multiprocessor

must be able to recover from the death of a processor, and possibly retrieve its workload.

In the next section the basic organisation of multiprocessor operating systems is considered. The problems of both synchronisation and reliability are then considered. Finally, the chapter closes with a brief consideration of parallel processing.

3.2. Operating System Organisation

The operating system is that part of the software on a computer that manages the resources (devices, memory, central processor time). The operating system provides the mechanism for the execution of programs and the environment in which they run.

"Three basic organisations have been used in the design of operating systems for multiprocessors: master-slave; separate executive for each processor; symmetric or anonymous treatment of all processors" (28). Each organisation provides different operational characteristics.

With the master-slave organisation, the operating system routines are always executed in the same processor, the 'master'. If one of the slave processors requires a service that must be provided by the operating system, a request must be made to the master processor. This may cause a delay within the slave processor. Since the operating system only runs in one processor, the problems of multiple update of system tables and device access cannot arise. A means whereby communication between the master and the slaves may take place must, however, be provided.

The master-slave organisation has some disadvantages. Foremost amongst these is the reliance of the whole system upon the master processor. If the master fails then the system as a whole will be lost. It may be possible to redesignate one of the slaves as a new master, but this would (probably) require action from either operators or engineers.

Also, if the master cannot keep pace with the service requirements of the slaves, then the idle time of the slaves may increase significantly. Despite being comparatively inflexible, this organisation is relatively simple to implement.

With a separate executive (or operating system) on each machine, the characteristics are very different. Each processor is capable of servicing its own needs and manages its own (local) resources. Each processor, therefore, maintains its own set of tables. Some tables, representing the shared resources, must be shared between the processors and therefore require synchronised access. Thus this organisation gives several co-operating but potentially independent systems. The supervisory code, under this scheme, may be placed in shared memory in which case only one copy need reside in memory, or it may be placed in the local memory of each system. The failure of one of the processors will not cause a catostrophic failure, as in the case of the master-slave organisation, since no one processor provides all the supervisory functions. However, some recovery of the shared tables may be required before the remaining processors may proceed to (correctly) use the shared resources. Some facilities (e.g. some i/o devices) will be lost if they are accessible only through the failed processor.

With the third approach, in which all processors are treated as any other resource, all resources will be shared, that is the tables defining their state will be shared. The mastership "floats" among

the processors, though several may be executing supervisory code at once. Clearly, each shared resource may have only one master, this being decided through the synchronisation required prior to them being accessed. Because no one processor has any special privileges or properties, if one of the processors fails, then only the processing power of the whole system need be affected. Again, system tables may need to be recovered, but the possibility exists for graceful degradation to take place. Also, as a processor acts as one of the system resources, scope exists for better load sharing.

3.3. Synchronisation

In the previous section, it was noted that, for a multiprocessor system, the need for synchronisation between the processors arises in order, to prevent two copies of the executive simultaneously accessing a shared table or device. This need for table lockout occurs not only at the operating system level, but at all levels of software on multiprocessor systems. Brinch Hansen (10) provides a useful survey of synchronising techniques.

The most famous form of synchronisation is the semaphore, originally proposed by Scholten and Dijkstra. A semaphore is basically an integer variable upon which two indivisible operations may be performed. These operations are variously known as P and V, Wait and Signal or Down and Up. The V operation causes the semaphore to be incremented. The P operation causes the semaphore to be decremented unless the value of the semaphore would become negative. In this case, the processor performing the P operation waits until it may be completed. Many examples of the use of semaphores may be found in the literature (11). Brinch Hansen (10) noted, however, that as, originally proposed, semaphores may leave some processors permanently blocked. This may be overcome by assuming some scheduling policy within the P and V operations.

Critical Regions (22) provide a similar technique to semaphores. A critical region is basically an area of code associated with a

shared variable. Each shared variable may be associated with several different code segments. The critical region mechanism ensures that, for each shared variable, only one processor is allowed to execute one of the areas of code associated with that variable. Critical regions provide an excellent medium for describing the use of and protection of shared data structures.

A modification of critical regions leads to the so-called Conditional Critical Regions (38). Not only is a section of code associated with a shared variable, but also a list of conditions to be satisfied before entering the region is given. The region is entered only when all the conditions are satisfied.

The elegance of these tools has led to discussion in the literature (9,18) as to their suitability in certain contexts.

For shared resources, another approach is to create a resource manager process. Processes then wishing to access the resource must make requests to the resource manager. This requires a message queue, to which processes add their requests. The addition of these requests must be an indivisible operation with respect to the processes. That is, if two processes attempt to add a message to the queue simultaneously, one will complete its addition before the second may make its addition and they will not mutually interfere. The resource manager removes messages from this queue, processing the requests as required.

Wirth (65) has noted that the message queueing techniques and semaphores are remarkably similar, a semaphore merely being a queue with no attached messages. An example of this class of tools are Hoare's Monitors (36).

All of these techniques may be used to great advantage upon uni-processor systems where indivisible operations may be guaranteed. However, if several processors are used then these techniques require some lower level of synchronisation upon which they may be based. Brinch Hansen (10) suggested that a hardware lockout device ('arbiter') was required. Indeed, in many multiprocessor systems, such devices have been implemented in hardware, for example the IBM 360/158 MP and 168 MP systems, as described in section 2.3, contain several instructions that may be used for this purpose.

In the absence of special hardware, it becomes necessary to develop synchronising algorithms using standard instruction sets. This problem of performing synchronisation between processors using only read and store instructions, originally proposed by Dijkstra, was first solved by Dekker (22), and generalised by Dijkstra (21). However, as Dijkstra noted, the method is cumbersome and potentially very time consumptive. Furthermore, Knuth (45) noted that one or more processors may be blocked indefinitely since the algorithm relies on a 'first past the post' mechanism, having no memory of the waiting time spent by a processor attempting to gain control.

Several authors (12, 27, 45) have proposed refinements to the algorithm to

reduce the time taken and to introduce some element of scheduling. All these algorithms, however, maintain the basic structure of the original solution. The improvements culminated in an algorithm (48) which guarantees safe access to a resource in a multiprocessor environment on a first-come-first-served basis.

The method adopted in all these cases is to allow one processor access to the shared resource and, when the processor has finished with the resource, it is freed to allow another processor to gain access to it. Thus the resource is alternately in use (or "owned" by a processor) and free. A processor, when it requires access to the shared resource must wait for that resource to become free. Then, if no other processor simultaneously requires the resource, it will become the owner and proceed to use the resource. Complications arise, however, when many processors attempt to gain ownership of a resource simultaneously since there must be a "competition" to decide who becomes the new owner. Indeed, even if a single processor only requires access to the shared resource, it must take part in the "competition" to discover that no other is also attempting to access it.

When this "competition" arises, the processors have to decide which of their number is to become the new owner. As the number of processors requiring access to the resource increases, the decision making becomes more complex and, in a general purpose algorithm, the case where all processors may require access needs to be catered for. As the complexity increases, so does the cost of performing the synchronisation.

This may be observed from the (sometimes complex) looping structure of the algorithms in the literature. This results in the cost (overhead) of synchronisation rising at least proportionally with the number of processors being synchronised. For heavily used system tables, the cost may become unacceptable.

3.4. Software Reliability

Much research is now being carried out in the field of fault-tolerant systems and other areas of increased reliability at the software level. This has led to the design of new languages and methodologies. With multiprocessor systems, the need for reliable software lies not only in obtaining correct programs, but also in withstanding processor (or other component) failure. Since the multiprocessor system contains several processors, there is the potential for performing useful work despite the failure of one of them. However, some recovery of shared data structures may be necessary before resuming the computational workload of the dead processor, if, indeed, the latter is possible.

Of the major manufacturers, IBM provides a process (the Alternate CPU Recovery process (15)) which is invoked on the death of a processor in the tightly coupled multiprocessor system described in Chapter Two. The process is initiated when a special interrupt is received indicating that a processor has died. The use of the ACR process enables various components of the system to be checked and recovery action to be taken as required. The problems facing the ACR, and associated routines, are sometimes complex. The considerable range of states that the processors may be in when the death, and ensuing interrupt, occurs contribute to the complexity of the problem. The recovery relies on the recovery process being able to ascertain much information on the dead processor at its point of death. Once the

recovery is complete, the system is then free to continue running, but providing a degraded service due to the reduced processor power.

Research is also being carried out into techniques for software error recovery (54,55,68). The aim of the group at Newcastle University is to provide a methodology which will not only cope with process failure, but also with errors due to inadequate or faulty design or coding. Due to the complexity of the software required for multiprocessor systems, the ability to withstand some design faults and continue to perform useful work in the presence of errors would be of advantage.

The approach taken is to provide the equivalent, at the software level, of standby components at the hardware level. It is accepted practice to write programs (especially those which are large and complex) in blocks (be they subroutines, procedures or modules, etc.). These blocks may be written in terms of sub-blocks, and so on. Each block may be viewed as providing an operation within the total system. A block is turned into a recovery block by adding an 'acceptance test' at the end of the block and zero or more stand-by blocks (alternates). The acceptance test is a logical expression by which the correct operation of the block may be tested. If the operation has failed, then one of the alternates is used. However, before the alternate is entered, the state of the process is restored to that current just before entry to the block which failed. A software technique for providing this ability to restore a process to an earlier state has been described in the literature (39).

3.5. Parallel Processing

Even when the organisational problems of multiprocessors at the hardware and operating system level have been resolved, there still remains the task of applying the system to the solution of problems in an efficient manner. However, the whole topic of parallel programming has recently gathered momentum due to recent hardware developments. The falling cost of processors and the availability of Array processors, such as the Illiac IV, and Vector processors as well as the multiprocessor systems described above, have contributed to this interest.

The, so called, array and vector processors, which are of the SIMD classification (see section 2.1), consist of a large number (often thousands) of small processing elements attached to a host. Parallelism is obtained, in such systems, by arranging for all the processing elements to perform the same single operation, but on different values. Algorithms to run upon these systems thus need to be formulated in terms of arrays of values upon which operations are performed. This makes such hardware particularly suitable for the solution of large numerical problems.

Research is also being carried out into the automatic detection of parallelism within programs. This research may be partitioned into two main groups:-

a) Statement level

b) Block level

At the statement level, single statements, particularly arithmetic, are considered. It is hoped that techniques to enable these statements to be compiled for optional parallelism may be derived. A survey of such research may be found in the literature (64). However, due to the great frequency of synchronisation required between processors when using this form of parallelism, it is not a viable technique when using a multiprocessor system of the type being considered.

At the block level, several statements can be grouped together and the blocks can be considered for execution in parallel. This technique provides a much more cost effective means of achieving parallelism on a multiprocessor system. As the size of these groups of instructions increases, so the relative cost of the inter-processor synchronisation will diminish, assuming that the groups are mutually independent. Results have been obtained (30) showing that the effective degree of parallelism obtainable is indeed dependent upon the length of these groups.

Proposals have been in existence for many years (19,22) for language extensions to enable parallelism to be expressed in programs. This approach enables programmers to directly insert parallel properties into their programs in a manner which they deem suitable to the application.

The suggestion "that parallel composition of communicating sequential processes is a fundamental program structure method" has recently appeared in the literature (37). A formal notation, based on Dijkstra's guarded commands (24), is presented which allows the communication between processes to be expressed. The communication is of the form of messages and not through shared variables.

CHAPTER 4

THE INVESTIGATION OF A MODEL OF A MULTIPROCESSOR

4.1. Introduction

It was noted in Chapter 2 that many detailed or complex models have been developed in the study of the theoretical computing power which can be realised in a multiprocessor system. Also noted was the fact that these formulae are, in general, specialised to a small class of hardware. It would be valuable if a more general tool were available which would enable an estimate of the maximum power that would be realised from a given multiprocessor system to be evaluated. Conversely, it may be desirable, given a particular workload, to evaluate the number of processors that may efficiently be included in the system.

In this chapter, therefore, a simple model of a multiprocessor system is presented and from the study of this model, an attempt is made to derive a formula for an upper bound to the computer power which may be realised.

4.2. Model of a Multiprocessor

The basic hardware model is of a collection of N, possibly different, processors. The characteristics of each processor are given by two variables, the execution speed of the processor, in instructions per second, and the private memory size, in instructions. These are denoted by r_i and s_i for the ith processor respectively. All the processors are linked to a large block of common memory. Information, either code or data, can be transferred between common memory and the private memory of any of the processors at the rate of l blocks of information per second. Each of these blocks contains b instructions giving an effective common to private (or private to common) memory transfer speed of lb instructions per second. These two parameters represent the line speed and bandwidth of the communication line between common and private memory. A processor may directly access the common memory for an instruction or data word without requiring it to be stored in its own private memory. The time required to perform this operation is expressed as the time to access private memory (inherently included in the processor execution speed) plus a fraction, f, of the transfer time between common memory and private memory. An assumption inherent in the model is that all accesses to common memory suffer some degradation whether memory contention takes place or not. This is due to the need for a contention resolving "black-box" to be placed in the access path to common memory of each processor (see Figure 4.2.1). If required, the degradation caused by this "black-box" may be

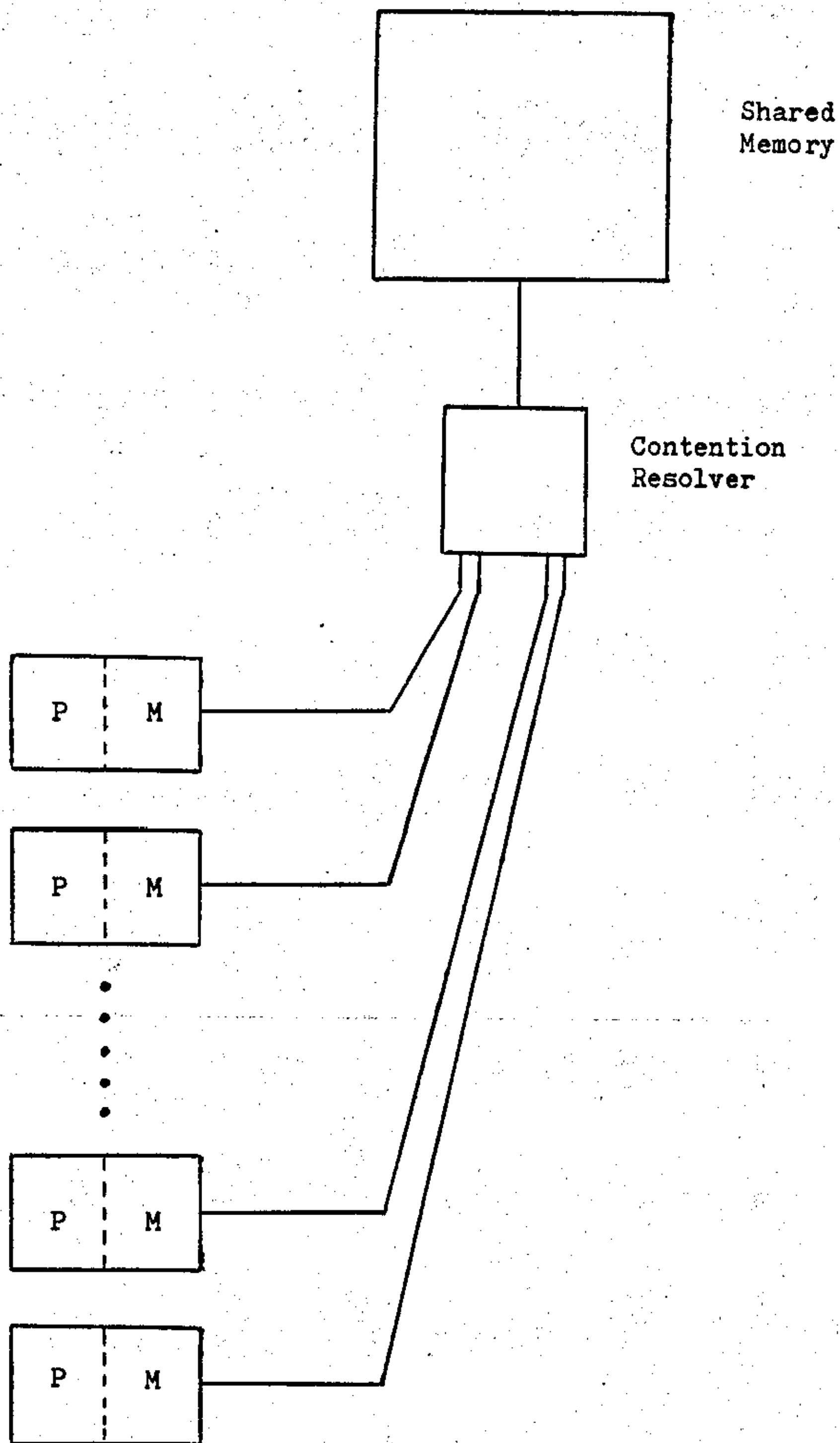


Figure 4.2.1. Modelled Multiprocessor System.

ignored by setting f to zero.

The instruction was chosen as the unit of data since no confusion over existing terminology, largely manufacturer dependant, would arise. With the modularity of current hardware, it may seem that a model catering for multiple memory modules would be necessary, but by the correct choice of the values for the parameters specifying the common memory, the operational characteristics of several blocks of common memory may be obtained.

By suitably altering the values of the parameters, the model can be applied to a variety of hardware configurations, including

a) Many processors each working from private memory using the common memory for communication only

b) Many processors each with no, or very little, private memory linked to a single block of common memory

c) Many processors each with limited private memory, using the common memory as a data base.

The same model may also be used for many processors accessing a common disc system as a variation on a) or c) above. In this case, the data access fraction, f, will have a value of one, since any data accessed must be copied to the private memory before it can be used.

In order to make calculations of computing power, the workload for the multiprocessor system must be incorporated into the model. The unit of work which is most clearly associated with users is that of the program. It would appear that, ideally, a general set of programs, or benchmarks, would be necessary. However, it is not possible to select a set of programs which would be representative of all situations. Furthermore, the theoretical analysis of such a set would be extremely difficult. It was, therefore, decided to examine the operation of the hardware model by postulating that a single program is run repetitively on all the processors.

It is further postulated that the program is initially loaded into the common memory but can only be executed from private memory. The program instructions, therefore, must be copied from common to private memory before execution can take place. Clearly, the private memory may not be sufficiently large to accommodate the whole of the program, in which case several copying operations would be required during the course of the run of the program in a manner analagous to paging (no attempt is made to model this activity but it is implicitly included in the parameter c_i defined below).

The characteristics of the program used in the model are

a) E, the execution length, or number of instructions executed by the processor in completing the program

b) c_i, the transfer or copy size, that is the total number of of instructions that have to be copied from common to

private memory

c) a data access rate of 1 access to common memory per d instructions executed

d) no external input or output operations.

Each of the parameters plays an important role in the model. E, the execution length, is effectively a normalisation constant or scaling function for the evaluation of computing power. The incorporation of c_i into the model allows short regular bursts of high rates of access to common memory. This parameter would be used when investigating systems performing copying operations to or from common and private memories. If no such function is performed, this parameter may be omitted (by setting it to zero). The variation in the parameter d can be used in the investigation of systems using only common memory (d having a value of one or less) through to systems rarely accessing common memory (d being large).

Thus, causing a representative program to be run repetitively on all the processors places no great restriction upon the workload that can be modelled since various classes of program may be considered by suitably varying the parameters of the representative program.

4.3. Derivation of Computing Power

A measure of the computing power of a multiprocessor system is the number of representative programs processed per unit time by the multiprocessor configuration, denoted by Pm.

In order to determine the effective performance of the system, this must be compared with the computing power of the same computers working separately. That is the number of representative programs processed per unit time by the separate processors, P_s.

Taking the model described above, the time for the ith processor to execute the representative program, whilst working separately, would be

$$E/r_i \quad \text{seconds} \qquad 4.3.1.$$

Thus the number of programs executed by the N separate processors in unit time (Ps) is

$$P_s = \sum_{i=1}^{N} r_i/E \qquad 4.3.2.$$

The total time for a program to run in one of the N processors in the multiprocessor configuration has four components, namely

a) the time required to transfer the program from common memory to the private memory of an individual processor

b) the time required to execute the program

c) the time overhead of making data accesses to common memory

d) the time spent waiting to be serviced by the memory. This delay, due to memory contention, may occur in two instances
- i) while copying instructions to private memory
- ii) while performing data accesses to common memory.

The first three components are obtainable from the model directly

a) program copy time

c_i/b transfers are required to copy the program to the private memory of the ith processor. This takes $c_i/(lb)$ seconds

b) execution time

This component is identical to that for the single processor case, that is E/r_i seconds for the ith processor

c) common memory access overhead

The overhead for each data access is f/l seconds. During execution of the program, a total of E/d accesses are made to common memory giving a value of $Ef/(dl)$ seconds for this component. It is implicitly assumed that a data item is of an equivalent size to an instruction, however d could be altered to model other data sizes.

The fourth component, that due to contention over common memory,

is dependent upon the strategy used by the hardware to distribute memory cycles between the processors. In order to obtain bounds for computing power of a multiprocessor, two distinct strategies are considered.

The first strategy treats all processors as strictly equal, and provides a common memory cycle to each processor in strict rotation (Round-Robin). With this strategy, there is the potential for (large) delays while accessing the common memory. Delays will inevitably arise due to memory contention in any practical situation, but it is possible, with this model, for a processor to wait for a memory cycle even if no other processor is accessing the memory. Under these circumstances this theoretical strategy gives a greater common memory access overhead than would practically be experienced due to memory contention alone, and when included in performance calculations it will therefore provide lower performance figures than could be experienced in practice.

In contrast to the first strategy, the second imposes an inherent order upon the processors. A memory cycle will always be allocated to the highest processor in this ranking list currently making a request, thus giving a Priority servicing policy. To obtain an upper bound for performance an assumption is made about the ordering of the memory requests from the processors. It is assumed that the memory requests made by the processors are synchronised so that no processor ever waits for service from the common memory unless all

the memory cycles are being used by the processors of higher rank. Thus no overheads or delays are experienced due to common memory contention provided that the total number of requests made by the processors does not exceed the capacity of the memory. There is still, however, a delay due to accessing the shared memory via the interface hardware.

Since, with this strategy all common memory cycles are being used, Priority represents the maximum processing power. When all the memory cycles have been used, further processors may not access the memory. This limit to processor power will be discussed in Section Six of this chapter.

In the next two sections, the formulae for the computing power of a multiprocessor system are derived for the two memory servicing policies.

4.4. Round-Robin Servicing

As noted above, the waiting time (the component dependent upon the memory servicing policy) arises in two situations. Firstly, the waiting time while copying is the time required for the N-1 memory cycles between each copy. These N-1 cycles take (N-1)/l seconds, and hence the total time spent waiting by the ith processor while copying is

$$C_i (N-1) / (lb) \text{ seconds} \qquad 4.4.1.$$

The second factor in the waiting time is due to waiting for a memory cycle while making a data access to common memory. The elapsed time between accesses is d/r_i seconds for the ith processor. After this time, the processor has to wait for its next memory cycle. The time spent waiting, y_i, is therefore

$$y_i = xN/l - d/r_i \text{ seconds} \qquad 4.4.2.$$

where x is the minimum integer such that

$$(xN) / l \geqslant d/r_i \qquad 4.4.3.$$

That is, it is on the xth memory cycle due to the processor since its last access that its next request is honoured.

This overhead is for each of the E/d accesses, giving a total waiting time, while performing data accesses, for the ith processor of

$$(y_i E) / d \text{ seconds} \qquad 4.4.4.$$

where y is given in equation 4.4.2.

The total time to run a representative program on the ith processor with Round-Robin common memory servicing, T_R, may now be evaluated as the sum of the four components

$$T_{Ri} = C_i/(lb) + E/r_i + Ef/(dl) + (C_i(N-1)/(lb) + (y_iE)/d) \qquad 4.4.5.$$

simplifying,

$$T_{Ri} = (NC_i)/(lb) + E(1/r_i + f/(dl) + y_i/d) \qquad 4.4.6.$$

Hence, the number of programs completed per unit time on processor i is

$$1/T_{Ri} \qquad 4.4.7.$$

and the total number of programs run on the system as a whole (J_{Ri}) is given by

$$J_{Ri} = \sum_{i=1}^{N} (1/T_{Ri}) \qquad 4.4.8.$$

and expanding

$$J_{Ri} = \sum_{i=1}^{N} (1/(C_iN/(lb) + E(1/r_i + f/(dl) + y_i/d))) \qquad 4.4.9.$$

where y_i is given in 4.4.2.

4.5. Priority Servicing

As mentioned in Section Three, the processors are assumed to be exactly synchronised and that no processor waits for servicing unless all common memory cycles are taken by processors of higher priority. When all memory cycles are being utilised by a number of processors, any other processors added to the system (at a lower priority) will be unable to access the common memory.

The processing power of the configuration under this form of common memory servicing can be evaluated by considering the operation of the processors in priority order.

Since the highest ordered processor experiences no delay, the time taken to complete a representative program on this processor T_{P_1} is the sum of the first three components

$$T_{P_1} = C_1 /(lb) + E/r_1 + Ef/(dl) \text{ seconds} \qquad 4.5.1.$$

Since only the first and third components involve usage of the common memory, there is a period of time during which the common memory is free, given by

$$E/r_1 \text{ seconds} \qquad 4.5.2.$$

The processor with second highest priority will take

$$T_{P_2} = C_2 /(lb) + E/r_2 + Ef/(dl) \text{ seconds} \qquad 4.5.3.$$

to run the representative program and can, therefore, potentially complete

$$T_{P1}/T_{P2} \qquad 4.5.4.$$

programs in time T_{P1}. Since each run of the representative program requires access to the common memory for a time of

$$C_2/(lb) + Ef/(dl) \text{ seconds} \qquad 4.5.5.$$

The time spent accessing common memory in time T_{P1} is given by the product of equations 4.5.4. and 4.5.5., that is

$$(T_{P1}/T_{P2})\,(C_2/(lb) + Ef/(dl)) \text{ seconds} \qquad 4.5.6.$$

If the time given by 4.5.6. is less than, or equal to, the execution time of the first processor, given by 4.5.2, then the assumption made regarding memory clashing may be applied and, therefore, all common memory accesses made by the second processor overlap the execution time of the first processor.

A smaller amount of time will remain when the common memory is not being accessed. This time is given by the difference between equations 4.5.2. and 4.5.6., namely

$$E/r_1 - (T_{P1}/T_{P2})\,(C_2/(lb) + Ef/(dl)) \text{ seconds} \qquad 4.5.7.$$

The argument may be continued for subsequent processors until the free time of the common memory is inadequate to allow the common

memory accesses of the next processor, denoted by N_L, to be satisfied.

Systems with fewer than N_L processors will, therefore, from 4.5.4. complete

$$\sum_{i=1}^{N} Tp_1 / Tp_i \qquad 4.5.8.$$

representative programs in time Tp_1, where Tp_i is the time for the ith processor to complete the representative program (cf.4.5.1.). Hence the number of representative programs executed in unit time on a multiprocessor system with fewer than N_L processors and a Priority servicing policy for common memory, Jp, is

$$Jp = (1/Tp_1) \sum_{i=1}^{N} (Tp_1/Tp_i) \qquad 4.5.9.$$

and simplifying

$$Jp = \sum_{i=1}^{N} (1/Tp_i) \qquad 4.5.10.$$

or $$Jp = \sum_{i=1}^{N} (1/(G_i/(lb) + E/r_i + Ef/(dl))) \qquad 4.5.11.$$

This throughput represents each of the N processors working at maximum speed. When the number of processors reaches or exceeds the capacity of the common memory, the $N_L{}^{th}$ processor cannot achieve

its maximum throughput and all subsequent processors will be unable to access common memory and therefore perform no useful work. Thus the throughput obtained from a configuration with N processors where $N \geqslant N_L$ lies between that obtained for a configuration with $N_L - 1$ processors and that obtained from a system with N_L processors as given by formula 4.5.10.

Graph 4.5.12. shows a typical Priority curve with the characteristic cut-off.

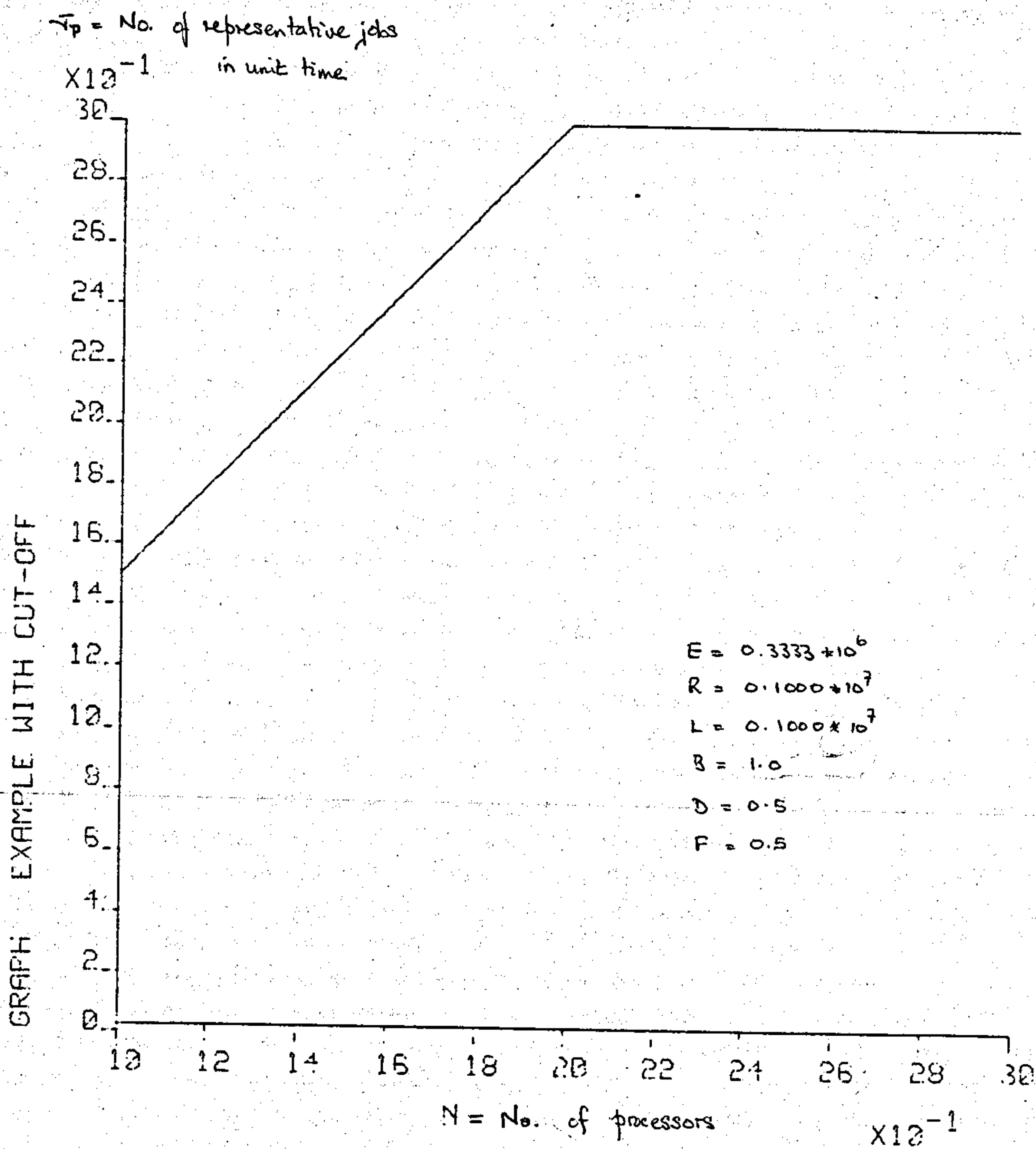


Figure 4.5.12. Priority Curve Showing Cut-off

4.6. Constraints for Effective Configurations

With Priority servicing used to allocate shared memory cycles, it was demonstrated that there was a limit to the number of processors which could access the memory. A system containing a greater number of processors would inevitably lead to a waste of resources since some of the processor could not perform useful work, being unable to access the shared memory. This limit is, from 4.5.7.,

$N_{L-1} = 1 + \text{max integer } k \text{ such that}$

$$((E/r_1) - \sum_{i=2}^{k} (T_{P1}/T_{Pi}) \; (C_i/(lb) + Ef/(dl))) > 0 \qquad 4.6.1.$$

From the original specification of the model, the Priority common memory servicing strategy gives the highest possible throughput since the slowdown factor is due only to the hardware interconnection and no memory clashing factor is included.

The Priority servicing strategy makes optimum use of memory cycles, with no time being wasted due to contention between the processors. Any cut off which exists with the Priority servicing must, therefore, apply to all other servicing strategies. Given parameters which characterise both the constituent processors in a multiprocessor configuration and the workload to be placed upon the system, a limit to the useful number of processors may be evaluated. In practice, it might be anticipated that this ideal situation would be unattainable,

in which case the effective maximum number of processors which could usefully be connected would be less than that given by N_L.

4.7. Analysis of Performance

In Section Three, it was postulated that the effective performance of the multiprocessor computer system could be found by comparing the computing power of the multiprocessor (P_m) with that of the computers running separately (P_s). This may be accomplished by expressing P_m as a percentage of P_s. The effective performance (EP) may therefore be evaluated for the two servicing strategies using P_s(4.3.2), J_R (4.4.9) and J_P (4.5.11).

Round-Robin: $EP_R = 100\ J_R/P_s\ \%$ 4.7.1.

$$= \frac{100\left(\sum_{i=1}^{N} \left(1/(C_i N/(lb) + E(1/r_i + f/(dl) + y/d))\right)\right)}{\left(\sum_{i=1}^{N} r_i/E\right)} \% \qquad 4.7.2.$$

Priority: $EP_P = 100\ J_P/P_s\ \%$ 4.7.3.

$$= \frac{100\left(\sum_{i=1}^{N} \left(1/(C_i/(lb) + E/r_i + Ef/(dl))\right)\right)}{\left(\sum_{i=1}^{N} r_i/E\right)} \% \qquad 4.7.4.$$

These formulae describe a situation where the processors and local memories have different characteristics. In practice, most multiprocessor systems might be expected to consist of combinations of identical (or near identical) processors. The formulae 4.7.2. and

4.7.4. can be simplified in this case. In the remainder of this chapter it will be assumed that all the processors are identical. This, however, places no restrictions upon conclusions drawn in later sections.

If all processors are assumed to be identical, all subscripts disappear and the summations may be replaced by a multiplication factor. The equations 4.3.2., 4.4.9. and 4.5.11. for P_S, J_R and J_P respectively may be simplified to give

$$P_S = Nr/E \qquad 4.7.5.$$

$$J_R = N/(CN/(lb) + E(1/r + f/(dl) + y/d)) \qquad 4.7.6.$$

with $y = xN/l - d/r$

where x is the minimum integer such that

$xN/l \geq d/r$

$$J_P = N/(C/(lb) + E/r + Ef/(dl)) \qquad 4.7.7.$$

Rewriting equations 4.7.2. and 4.7.4. with these simplified equations, values for the effective performance will be given by

Round-Robin:

$$EP_R = 100(N/(CN/(lb) + E(1/r + f/(dl) + y/d)))/(Nr/E) \; \%$$

4.7.8.

Priority:

$$EP_P = 100\,(N/(C/(lb) + E/r + Ef/(dl)))/(Nr/E) \; \% \qquad 4.7.9.$$

Simplifying, these become

$$EP_R = 100\ E/(r(CN/(lb) + E(1/r + f/(dl) + y/d)))\ \% \quad 4.7.10$$

$$EP_P = 100\ E/(r(C/(lb) + E/r + Ef/(dl)))\ \% \quad 4.7.11$$

These two formulae for effective performance apparently provide the bounds on the performance of the multiprocessor system which were sought. However, by observing the predictions of the formulae for a particular choice of the parameters (shown in Fig.4.7.12), it is seen that under some circumstances the efficiency achieved with the Round-Robin servicing is equal to that with the Priority servicing. This clearly violates the upper-lower bound hypothesis.

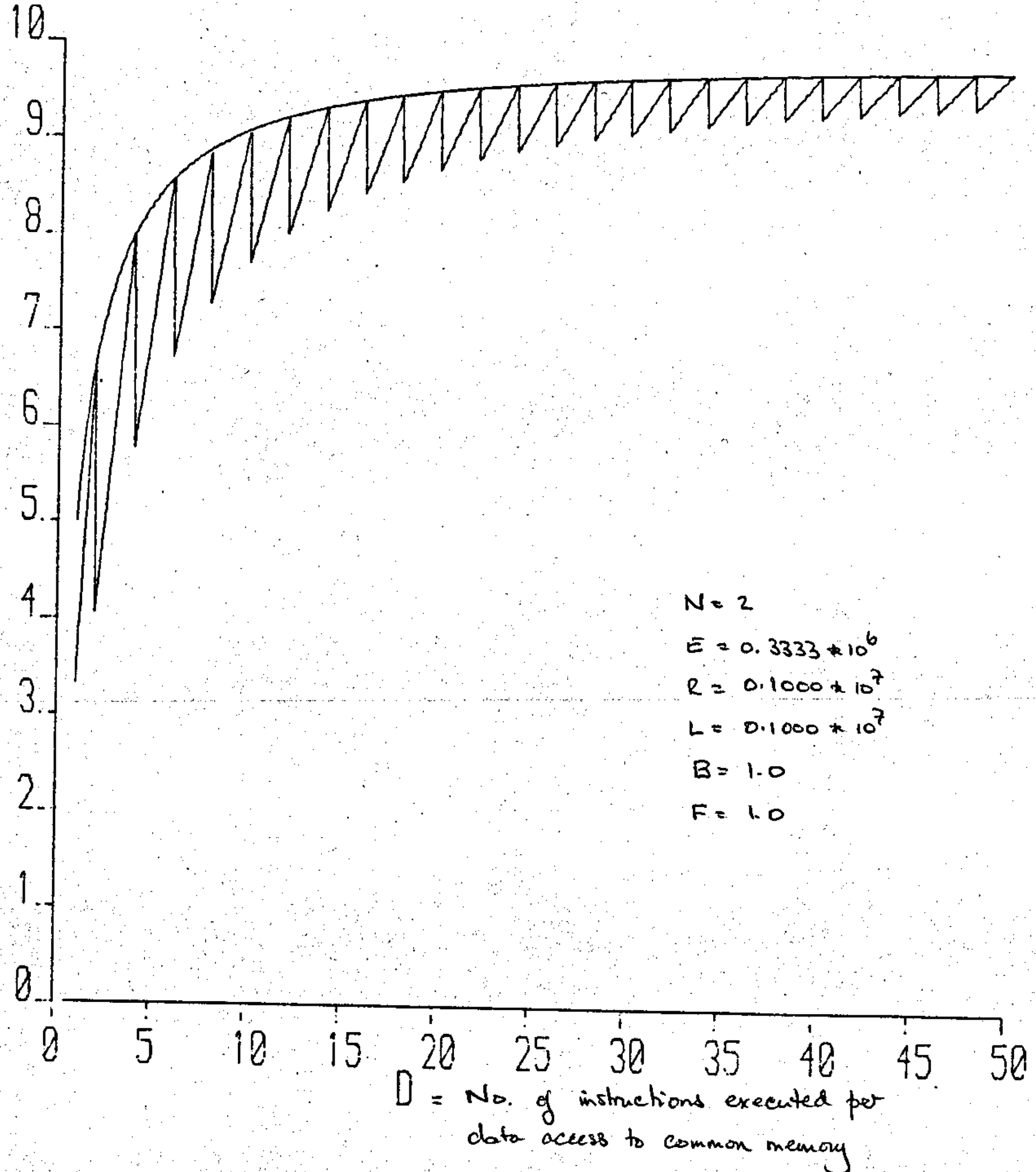


Figure 4.7.12. Peaking characteristic of Round-Robin servicing policy.

4.8. Refinement of Servicing Policy

If the denominator of the right hand side of equation 4.7.11 is denoted by v then equations 4.7.10 and 4.7.11 may be rewritten in terms of v as

$$EP_R = 100\ E/(v + (N-1)rC\ /(lb) + E/d)\ \% \qquad 4.8.1.$$

$$EP_P = 100\ E/v\ \% \qquad 4.8.2.$$

$$\text{where } v = rC/(lb) + E + rEf/(dl) \qquad 4.8.3.$$

The two extra terms in the denominator of equation 4.8.1 are due to the waiting times while copying from common memory and while making data accesses to the common memory. The deficiency in the Round-Robin strategy now becomes apparent. If these two extra terms,

$$(N-1)rC\ /(lb) + E/d \qquad 4.8.4.$$

can become zero, or very small, the Round-Robin strategy instead of reflecting the case where there is memory interference, becomes equivalent to the Priority servicing strategy. This will occur, in general, if both y and C themselves become very small. In practice both of these conditions may hold. C would be small if the private memory of the individual processors is large and very little copying were required in relation to execution length. The waiting time for a data access to common memory (y) can be zero if the access is requested when a cycle is offered, that is when (from equation 4.4.2)

$$d/r = x\ (N/l) \qquad 4.8.5.$$

where x is some integer.

The possibility for the waiting time to become zero will clearly give rise to a "peaking" characteristic to the function defined by 4.7.10, as has been seen in Fig. 4.7.12.

In practice, common memory requests do not occur at strictly regular time points, but are distributed about these time points. While the mean arrival time may be coincident with the offering of a memory cycle, the mean waiting time will not be zero.

This can be illustrated by considering the case in which the probability of arrival of a common memory request can be represented by an arbitrary distribution function with a mean at the point at which a memory cycle is offered. This is illustrated in Figure 4.8.6.

Any request which arrives before the memory cycle is offered must wait until it is offered, while any request that arrives afterwards must wait for the next cycle. Thus the mean waiting time is

$$\int_{t = t_1}^{T_A} h\ (T_A - t)\ \delta t + \int_{t = T_A}^{T_2} h\ (T_B - t)\ \delta t \qquad 4.8.7.$$

and this must have a non-zero value. In the general case where the mean is not coincident with the offer of a cycle, the mean waiting time can be expressed as

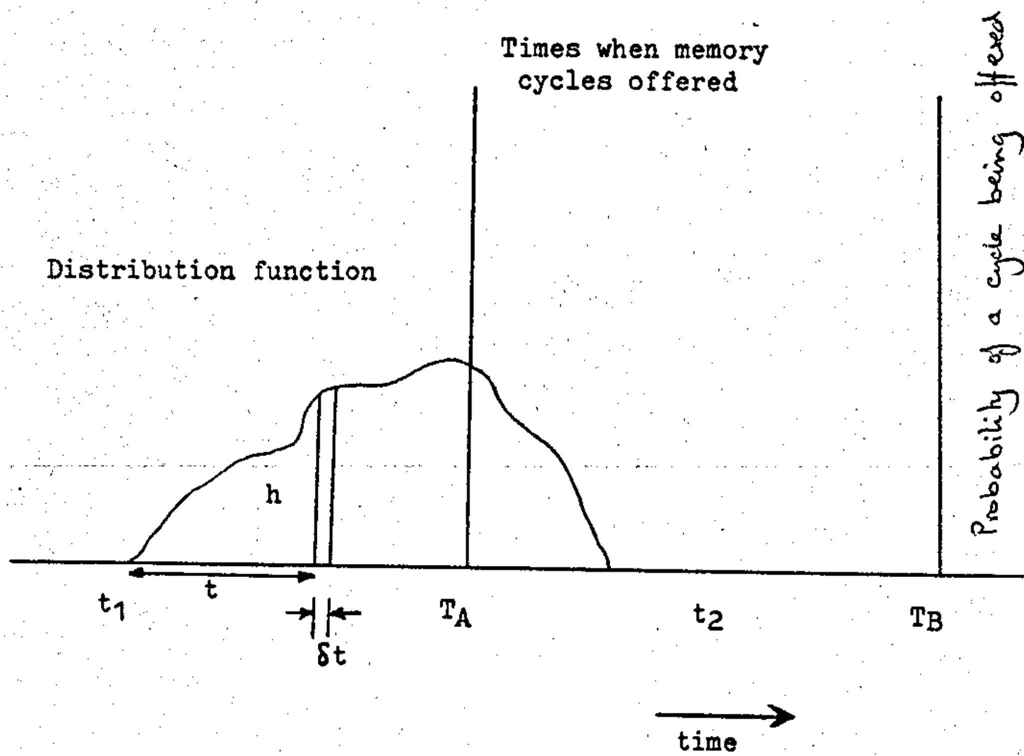


Figure 4.8.6. Distribution of Memory Requests

$$y = g\,(xN/l - d/r) \qquad 4.8.8.$$

where g is a function which reflects the actual distribution of data access requests.

Various distributions were investigated, and Table 4.8.10 shows the mean waiting time for the four distributions shown in Figure 4.8.9. To produce the table, the following values were chosen for the parameters of the distribution as shown in Figure 4.8.6

$$t_2 - t_1 = T_B - T_A = 1$$

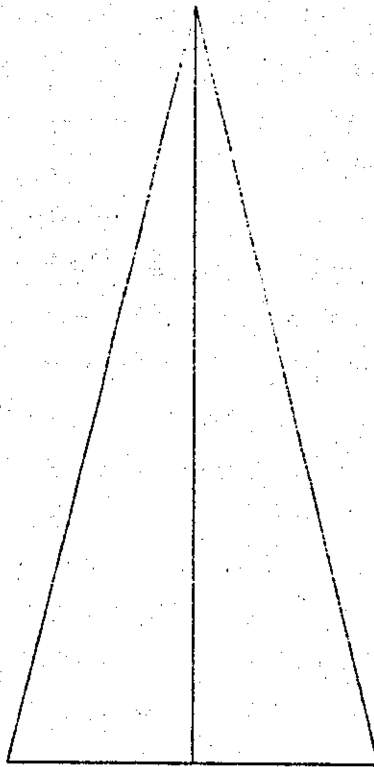
$$\text{the mean of } g,\ \mu(g) = (t_2 + t_1)/2 \qquad 4.8.11.$$

This choice of parameters describes a situation where a request can arrive at any time between two successive offerings of a memory cycle. The value of M in the table is the distance

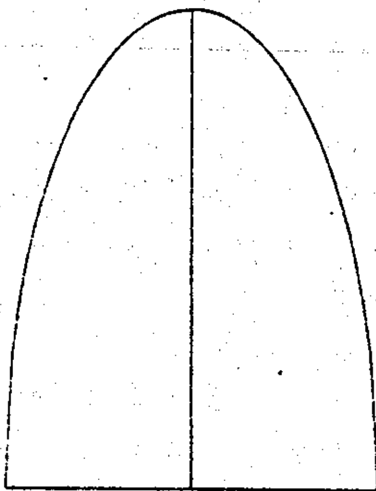
$$M = \mu(g) - T_A \qquad 4.8.12.$$

that is, the offset of the mean from the offer of a memory cycle. From the table it can be seen that the three distributions give similar waiting times so, for ease of calculation, the triangular distribution is adopted throughout the remainder of this chapter.

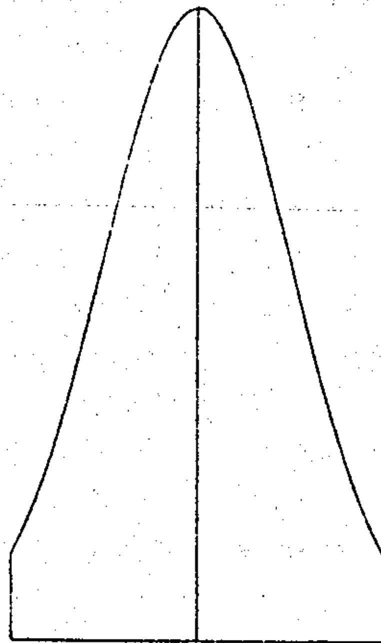
Figure 4.8.13 shows the same graph as Figure 4.7.12, but with the triangular arrival distribution applied to smooth the peaking.



a) Triangular



b) Eliptical



c) Truncated Normal

Figure 4.8.9. Arrival Distributions

M	Triangular	Elliptical	Truncated Normal*
0	0.501	0.501	0.503
0.1	0.581	0.527	0.566
0.2	0.621	0.548	0.605
0.3	0.620	0.558	0.607
0.4	0.580	0.548	0.569
0.5	0.500	0.500	0.502
0.6	0.420	0.452	0.436
0.7	0.380	0.443	0.399
0.8	0.381	0.453	0.401
0.9	0.421	0.474	0.440
1.0	0.501	0.501	0.503

* Truncated at 95% confidence limits

Table 4.8.10. Comparison of Arrival Distributions

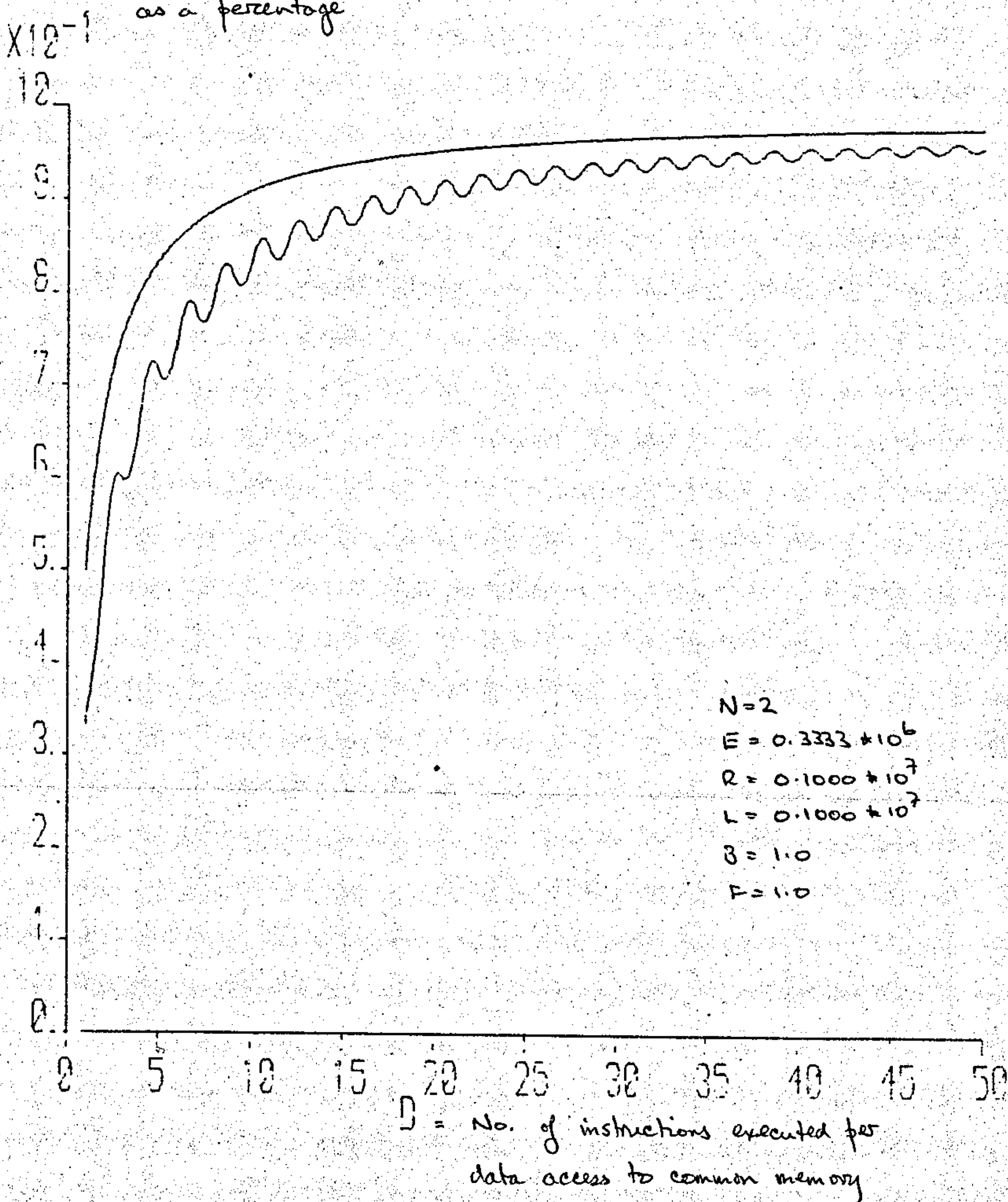


Figure 4.8.13. Smoothed Peaking of Round-Robin Servicing Policy

4.9. Application of Formulae

In this section examples of the use of the formulae are given showing comparisons with both simulation studies and practical results obtained from multiprocessor systems.

The primary test-bed for the formulae was a simulation program written in BASIC. The simulation program contained variables corresponding to the major parameters of the model presented in this chapter. The variables cover the number of processors, the frequency of access to shared memory (for both data access and program copying) and the memory speed.

Each of the processors in the simulation model would repeatedly execute the program, specified by the memory access parameters, until a pre-specified number of time steps had been completed. When the simulation finished the number of representative programs executed by each processor was reported. Memory accesses in the simulation model were not made at strictly regular intervals, an element of randomness being incorporated into their arrival. This randomness represented the situation where memory requests were evenly distributed over the memory cycle.

Two algorithms were encoded for the resolution of memory contention. The first of these corresponds to the Priority servicing policy. At each memory cycle, the processors are searched in order as in

the Priority policy. The second corresponds to the Round-Robin policy, with memory cycles being offered in strict rotation.

Due to the ideal representation of the hardware inherent in the Priority model (that is no memory contention), it would be expected that results obtained from the formula would overestimate the throughput as determined by the simulation. Also, this overestimate would increase as the potential for contention increases. The results for the Round-Robin servicing would, however, be expected to correspond more closely to those from the simulations.

Table 4.9.1. shows some results obtained from the simulation studies. It is seen that the results correspond to those expected, with greater discrepancy being shown in the Priority servicing. Also, as the frequency of accesses to the memory increases (either by increasing the data access rate or by increasing the number of processors), there is a drop in actual performance obtained from the simulation.

Experience for predicting the performance of real hardware was obtained using the dual Interdata Model 70 system within the Department of Computer Studies at Loughborough University. The memory of the system is 1 micro second. The memory contention resolving mechanism is complicated due to the fact that the shared memory is physically attached to one of the processors. When the other processor wishes to access the shared memory, it bids for

	PRIORITY				ROUND-ROBIN			
	THEORY	SIMULATION			THEORY	SIMULATION		
		Max	Av	Min		Max	Av	Min
1)N=2;D=5	90.9	82.1	81.7	81.7	76.9	78.0	77.8	77.7
2)N=5;D=5	90.9	73.9	73.9	73.7	62.3	62.6	62.4	62.3
3)N=2;D=50	99.0	98.1	98.1	98.0	97.0	97.1	97.0	97.0
4)N=5;D=50	99.0	98.0	98.0	98.0	94.3	94.5	94.4	94.3

Notes: 1) No program copying

2) Memory Speed = 1 micro second

3) All values show effective performance in %

Table 4.9.1. Comparison of Theory with Simulation Results

access to the memory and suffers a delay of 1 micro second. Also, while it is accessing the shared memory, the first processor may not access its own private memory.

Two programs were used in a test of the formulae. These programs involved access to the shared memory, but at differing rates. Values for the parameters to the formulae were obtained from the programs and these were used to obtain comparative results. Table 4.9.2. shows the results obtained. The potential expansion of the system can be found by evaluating the formulae for a greater number of processors. Figure 4.9.3. shows the curves for the first of these two tests, and it can be seen that the processor limit is 20.

In (29), a formula, developed by UNIVAC, is cited for evaluating the extra performance achievable from the addition of extra processors. Results are quoted for the 1108 system. Table 4.9.4 shows the corresponding predictions based upon the formulae derived in this thesis. The values adopted for the parameters are an instruction time and memory access time of 1 μsec, with memory being accessed in one word units. It is assumed that common memory is accessed every instruction with accesses to common memory increasing access time by one eighth.

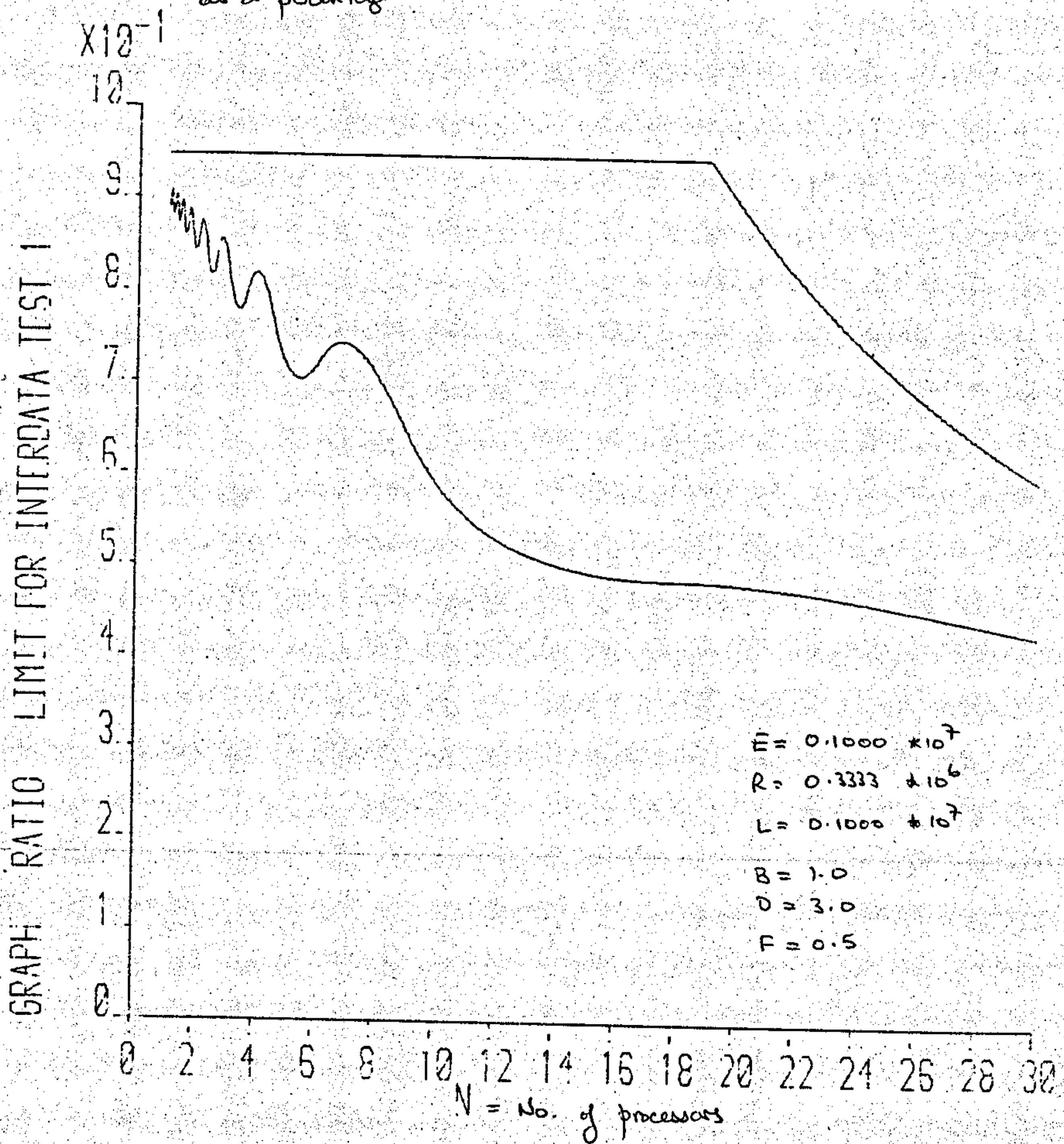


Figure 4.9.3. Graph for Interdata Test 1

	Observed	Priority	Round-Robin
TEST 1	90.8%	94.7%	85.7%
TEST 2	70.4%	75.0%	60.0%

Table 4.9.2. Comparison with Timings from Dual Interdata Model 70

UNIVAC FORMULA WITH 2 PROCESSORS	85.9%
PRIORITY	91.4%
ROUND-ROBIN	49.7%

Table 4.9.4. Comparison with UNIVAC Formula

CHAPTER 5

THE ABSTRACT RESOURCE RING

- A SYNCHRONISING TOOL

5.1. Introduction

This chapter, and that following, are concerned with the description of the development of a reliable synchronising tool to enable resource sharing and mutual exclusion within multiprocessor systems. Again, the model of a multiprocessor is of several processes connected to same shared memory but without any hardware synchronisation available, except that required to prevent multiple accesses to shared memory.

As discussed in Chapter Three, existing software solutions to the synchronising problem in these circumstances have some inherent deficiencies. These include the potentially large amount of computational time required to synchronise, when demand becomes high, and the possibility, with some of the algorithms, that one or more of the processes can be blocked, indefinitely. In this chapter, we approach a solution by re-appraising the problem and in the following chapter the synchronising tool, so developed, is investigated with respect to reliability.

As has been noted, the time is spent in discovering a new owner for the resource and the ensuing "bartering". If the method of discovering the new owner could be modified, or removed, then the cost of synchronisation may be reduced. One method whereby this may be achieved is to make the "resource free" state illegal and give the current resource owner the responsibility of locating a new owner and passing ownership, instead of merely relinquishing the resource.

As will be seen later, this technique, which may be termed a resource master technique, has performance advantages when the shared resources are reasonably heavily used but means extra overheads when the resource is used infrequently.

The resource sharing takes place between processes on the different processors. The problem of resource sharing may, therefore, be split into two phases

i) the sharing of the resources between processors (or more correctly between the schedulers on the processors)

ii) the distribution of the resource between the processes on a particular processor.

The latter problem can be readily handled by existing techniques, it being exactly the problem faced on a standard uni-processor with the scheduling system acting as a master or controller. Consideration is therefore given to the former phase, that is the sharing between processors where no mastership exists.

5.2. The Abstract Resource Ring

The problem of managing access to a single resource will first be considered and this will later be generalised to cover the management of several resources.

A data structure will be required to represent the current ownership of the resource and those processors wishing to use it. Clearly, this must be placed in the shared memory of the multi-processor system if all processors are going to access it. It will also be necessary to have algorithms to access and alter the fields of the data structure to enable the required resource sharing to take place.

A node is required in this data structure for each processor which may wish to access the shared resource. A suitable ordering of the nodes is in the form of a closed ring. Each node is required to maintain information on whether the processor requires use of the resource and also whether the processor is the current owner or not. This may be held in two boolean fields known as WANT (which if set indicates that the processor requires the resource) and CAN (which if set indicates that the processor is the current owner). Also, a separate field (NEXT) containing a pointer to the next node on the ring is required. The whole data structure must be accessible to all the processors, and each field must be individually addressable. This structure is known as an Abstract Resource Ring (ARR).

Two algorithms are required, firstly to enable a processor to gain access to the resource and secondly to relinquish it. The algorithm for gaining access to the resource (GETRES) consists of setting the WANT flag and then, conceptually, looping inspecting the CAN flag until it is set. Once the CAN flag is set, then the processor has become the owner of the resource and may freely use it.

The second algorithm, to relinquish the resource (PUTRES), consists of clearing the WANT flag then inspecting the WANT flags of the other processors. When one is found set then ownership (indicated by the CAN flag) may be passed. This is accomplished by the processor clearing its own CAN flag and then setting that of the requesting processor. The second processor will then discover that its CAN flag is set and will then start to use the resource. These two algorithms are shown in Figure 5.2.1.

In order to demonstrate that these basic algorithms can provide a satisfactory resource sharing tool, it is necessary to show that only one processor may become the owner of the resource.

<u>Theorem</u>

If all accesses to the Abstract Resource Ring are made only through the GETRES and PUTRES algorithms, then the number of set CAN flags can never increase.

```
getres =
        begin
            i: = our processor number;
            WANT of node [i] :=  set;
            while CAN of node [i] =  clear do
                    nothing
            od;
        end;

putres =
        begin
            i: = our processor number;
            WANT of node [i] := clear;
            j: = i;
            while WANT of node [j] =  clear do
                    advance j to next processor number
            od;
            CAN of node [i] : = clear;
            CAN of node [j] : = set
        end;
```

Figure 5.2.1. Basic GETRES and PUTRES algorithms

Proof

i) Consider firstly the GETRES algorithm. In this algorithm, the CAN flags are not assigned to, only the CAN flag of the node corresponding to the processor is inspected. Therefore the number of set CAN flags cannot increase by using GETRES.

ii) The PUTRES algorithm has two steps involving the alteration of CAN flags. Firstly, that in which the CAN flag of the current owner is cleared and secondly that of setting the one of the new owner. If the number of set CAN flags is not to increase then two conditions must be fulfilled -

a) The CAN flag must be set prior to clearing, otherwise the number set increases by 1 i.e. PUTRES must not be executed unless the CAN flag is set

b) The resource should not be passed to more than one new processor i.e. PUTRES should not be executed twice in the same machine.

The first condition can be met by ensuring that the CAN flag is set prior to passing the ownership. The second by ensuring, within the operating system, that a PUTRES of the resource is not started twice.

If both of these conditions are met then firstly the number of set CAN flags is decremented and then incremented, leaving the total unchanged. If the Abstract Resource Ring is initialised with a single CAN flag set (a single owner) then there can never be more

than a single owner following a sequence of GETRES and PUTRES operations and the necessary resource protection is obtained.

When the PUTRES algorithm is invoked, a search is made of the ARR for another processor to pass ownership of the resource to. If none is found the algorithm does not terminate, but continually loops. Clearly, this is highly undesirable since it may be some time before the resource is required again. To overcome this excessive use of processor time a separate PUTRES activity is created to dispose of the resource. This may be a separate process or a function of the operating system. This activity periodically checks the resource ring, attempting to relinquish ownership until the resource can be disposed of.

Basically, the problem is to decide when next to check whether it is possible to pass ownership. Two strategies may be employed in determining this time:-

a) Periodic restart

b) Interrupt restart

With solution a), the PUTRES activity is restarted periodically, that is, after each search of the ARR, the activity suspends itself for a period of time. It may also be incorporated into a section of the operating system which is executed periodically, for example the scheduler. To reduce system overheads with the latter implementation, a flag should be set when the resource is owned but

not wanted so that the scheduler only performs the check when the flag is set.

With the second solution, the interrupt restart, the PUTRES activity is only restarted when another processor requests ownership, that is as a function of the GETRES algorithm. A mechanism is, therefore, required whereby a processor performing a GETRES may restart the PUTRES activity in another processor (if present).

A mechanism whereby this may be accomplished is by using interrupts. If a hardware path corresponding to the Abstract Resource Ring is formed such that each processor may raise an interrupt in its successor processor, then when a GETRES is initiated, an interrupt can be sent to the successor. Clearly, the successor need not be the owner so whenever an interrupt is received by any processor it must be passed to its successor. Thus the interrupt will circulate round the ring. When the processor with the PUTRES activity is interrupted, it should restart the activity. As a consequence, the resource ownership will be passed to the requesting processor.

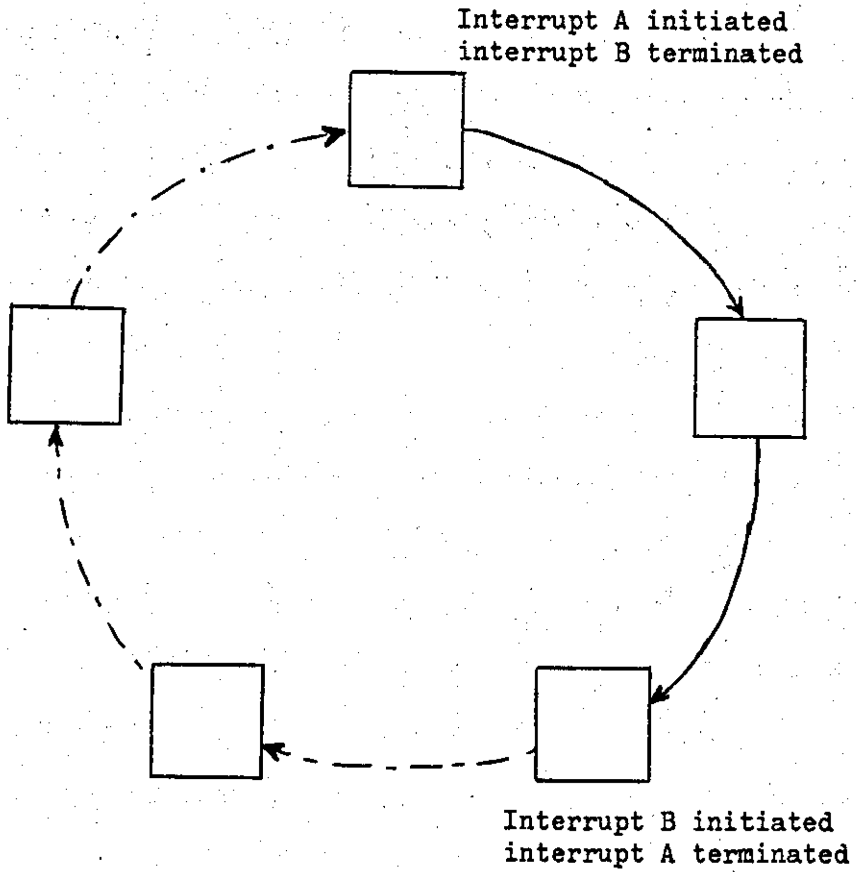
As the interrupt is passed round the ring, it will eventually reach the processor which initiated the cycle. Clearly, there is no need for the interrupt to pass any further. If each processor maintains a count which is increment each time an interrupt is sent and decremented when one is received then the interrupt should be passed only if the count is negative.

Since interrupt cycles are started when a GETRES is initiated then two (or more) interrupt cycles may be in progress simultaneously if several processors request the resource (see Fig. 5.2.2.). However, when one processor receives an interrupt, it is not passed on if there is one outstanding, so the many interrupt cycles are coalesced into one.

The performance characteristics of the two solutions (the Periodic Restart and Interrupt Restart) are different with each performing better under certain conditions. With the periodic solution, the PUTRES activity may be needlessly restarted if the periodic time is too short. However, if the time is too large, there may be excessive delay in passing the resource. There is, however, no requirement for an interrupt path to exist between the processors.

With the interrupt restart, if a GETRES unilaterally causes an interrupt to be sent then one could be issued while the resource is still in use. Also, the interrupt path must be created.

With both solutions, the PUTRES activity and GETRES must be non-interruptable with respect to each other (except for the waits). This is to prevent the resource, in a "partially-passed-on" state being claimed by the GETRES causing the basic assumptions to be violated.



Path of Interrupt A ⟶

Path of Interrupt B —·—·→

Figure 5.2.2. Multiple Interrupt Cycles

5.3. Multiple Rings

So far, the discussion has been based upon a single resource. However, in a multiprocessor system many resources will be shared and each will therefore need protecting with a synchronising mechanism. Therefore the mechanism described above needs extending with several resources.

The function of the Abstract Resource Ring will be split into two parts and each will be considered separately, these being

a) the handling of the ring nodes

b) the operation of the PUTRES activity.

Firstly, the basic ring structure and operation. Clearly, a ring structure similar to the structure already devised will be required for each resource. Since every resource may not be used by all the processors, the resource rings may not be identical. The rings need only contain nodes for those processors which may access the resource. The functions of GETRES and PUTRES also need to be modified to include a parameter giving the identification of the resource required. Each processor will require a routing table to convert this identification into a pointer to the appropriate node. One simple technique whereby this may be accomplished is by numbering each resource and using that number as an index to a row of pointers. If this scheme is followed, a structure of the type shown in Figure 5.3.1. is obtained.

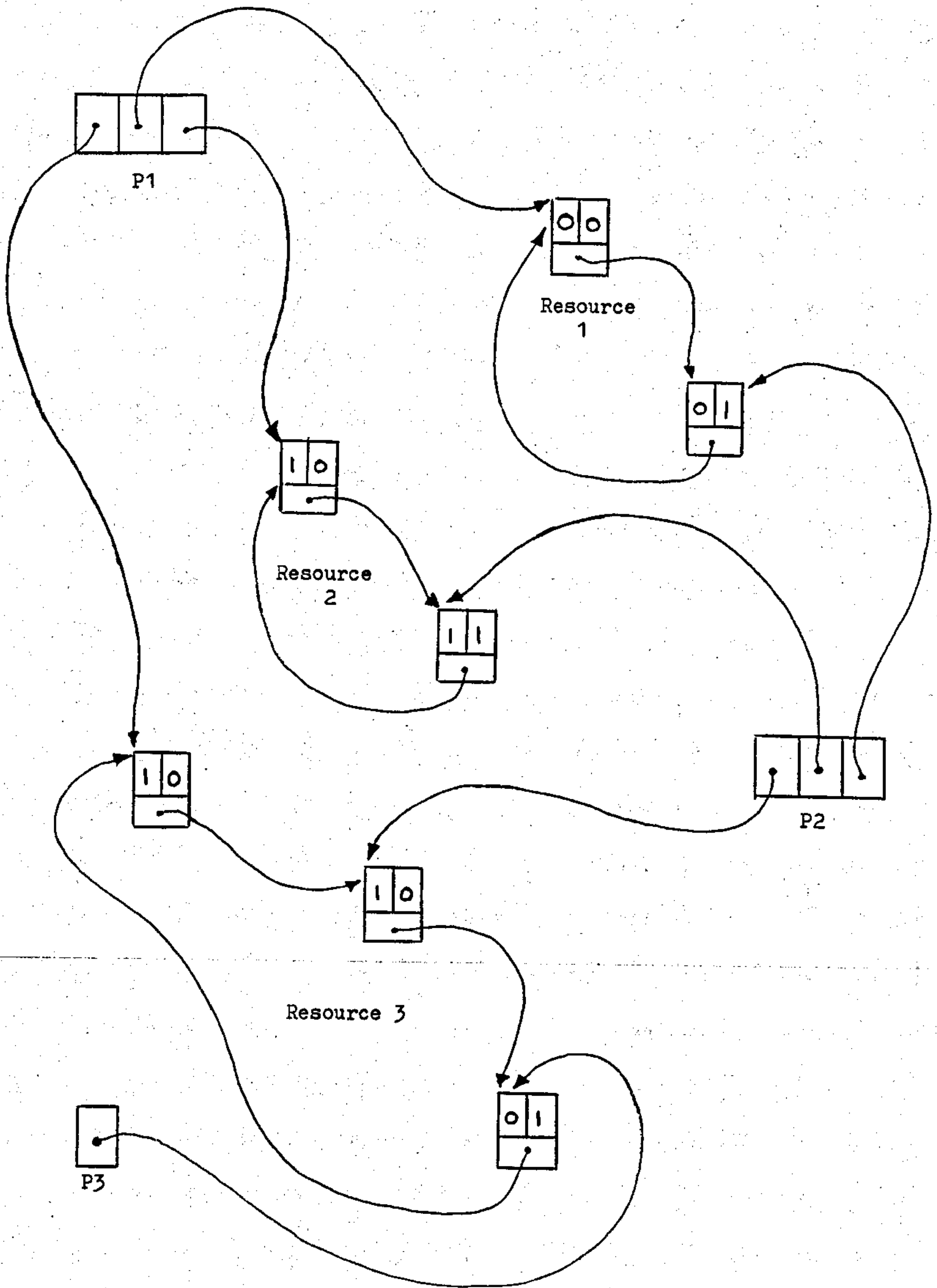


Figure 5.3.1. Multiple Resource Ring Structure

With the PUTRES Activity, using the second solution (the interrupt wakeup mechanism), complications arise if multiple PUTRES activities are in existence on a particular processor, as may, in general, be the case. When an interrupt is received from the predecessor, the question arises as to which of the PUTRES activities should be restarted. If multiple PUTRES activities are created then either some message needs to arrive with the interrupt to indicate for which PUTRES activity it is intended or all the PUTRES activities should be resumed. Another disadvantage with this solution is the potential number of interrupts circulating, and the associated counting complexity. A more rational approach would be to unify the mechanism. The PUTRES activities could be merged into a single routine, which could check for resources owned but not wanted, with an interrupt manager being created. When the GETRES routine decides an interrupt should be issued, a request is made to the interrupt manager. When an interrupt is received, the interrupt manager will restart the resource checker and then perform the necessary counting and pass the interrupt if required.

Clearly, the sending of two interrupts in quick succession will frequently make little difference in response. Some of the interrupt requests from the GETRES routine may be ignored by the interrupt manager, for example, if it has just passed an interrupt round the ring or if two processes perform a GETRES for different resources in quick succession.

With the first solution, that of periodic restart, the existence of

multiple PUTRES activities causes no difficulties with restart. The only disadvantage is the potential number of activities which may be in existence and the corresponding overhead within the scheduling system and possible reduction in the number of user processes which can be supported. If several PUTRES activities would consume too many scheduler resources (e.g. items in the scheduler list), a single resource checking procedure could be adopted as for the interrupt restart. If the PUTRES activity is incorporated into the scheduler, then a count of owned but not wanted should be maintained. The scheduler then need only check if the count is non-zero.

5.4. Temporary Resources

It has been assumed in the previous sections that the ring structure was a permanent part of the system. It is reasonable that, for certain permanent shared system resources, the ring structure should be created at system initialisation, in the same way as other system tables, with a node for each processor in the system. However, many of the resources used in the system will be of a transitory nature, being required only during the running of certain sets of complementary programs. It would be possible to create a number of rings at system initialisation time which may be used for these transient resources. However, this may cause unwanted interaction between two (otherwise independent) programs which happen to be using one particular resource ring for two completely different transient resources. Some mechanism must therefore be provided to enable dynamic creation of resource rings.

We require a procedure for uniquely creating rings, adding new nodes to existing rings and distinguishing between the different resources. One possible solution would be to maintain a table giving identifying information about the temporary resources and a pointer to a node on the ring. A system resource ring will also be required to protect this shared table as it is a sensitive resource. This ring may suitably be called CREATE and the table RESOURCES.

A processor running a process requiring access to a temporary resource

must first call an allocation routine to obtain the resource number of the temporary resource. After all the processes referencing this temporary resource have completed, the processor should remove itself from the resource ring by calling a deallocation routine.

The allocation procedure claims ownership of the CREATE resource to obtain access to the RESOURCES table. The table is inspected to see if a resource ring for that resource already exists. If a ring exists, then a new node is added to the ring for the processor. Adding a node to one of the rings consists merely of altering the pointer and not the value fields. Since the pointers are only modified when a new node is added to (or removed from) the ring and the corresponding processor must own the CREATE resource, only one processor may be modifying the pointers. The addition should be made in a way such that the ring is never broken, that is, the pointer (NEXT) field of the new node should be set to point to its successor before the NEXT field of its future predecessor is altered. If a ring does not exist a free resource number is chosen and the description of the resource is entered in the RESOURCES table. A ring consisting of a single node is created and a pointer to this node is placed in the entry for the new resource. In both cases, the number of the temporary resource is returned.

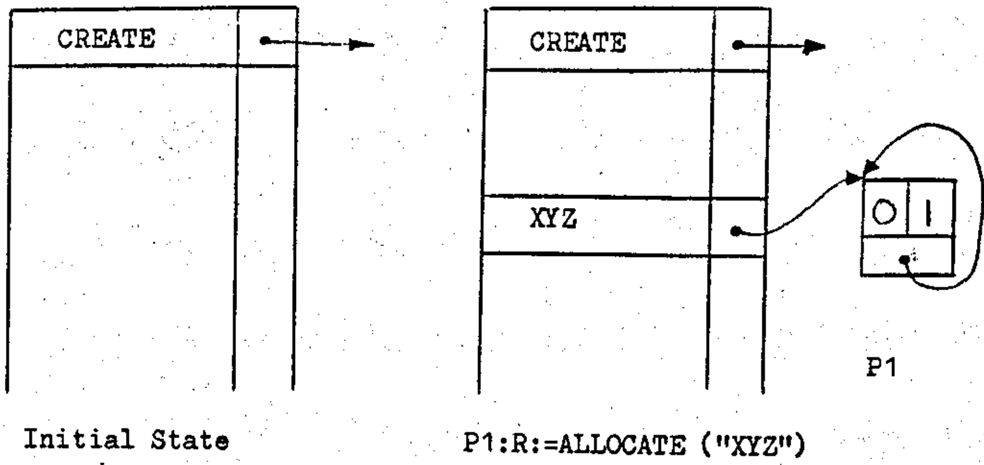
The deallocate procedure operates in the opposite manner. Firstly, both the resource to be deallocated and the CREATE resource are

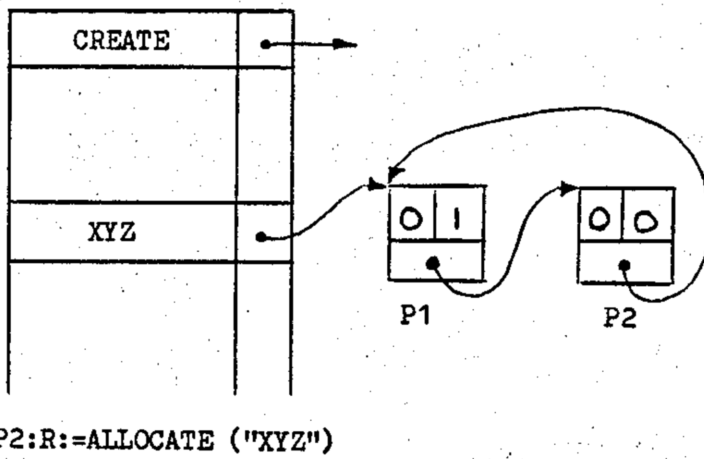
claimed. This is necessary to prevent several nodes being removed simultaneously and also to prevent another processor searching the ring while the node is being removed. Note that CREATE should be claimed last to prevent possible deadlock.

If the processor performing the deallocate is the only processor on the resource ring, then the entry for that temporary resource is removed from the resources table, enabling that entry to be used for another temporary resource in the future, and CREATE is released.

If, however, other processors are still on the ring, then the processor performing the deallocate must wait for one of the other processors to request the resource. While waiting, however, the CREATE resource should be released to allow other processors access to the RESOURCES structure. As with PUTRES, this waiting can be achieved more readily by creating a separate activity to allow the scheduler to continue. When a request is made, the processor should remove itself from the ring and pass ownership to the requesting processor.

The operation of these two procedures is shown pictorially by the state of the data structures at various stages in Figure 5.4.1. A possible implementation of these procedures will be found as part of Appendix 1.





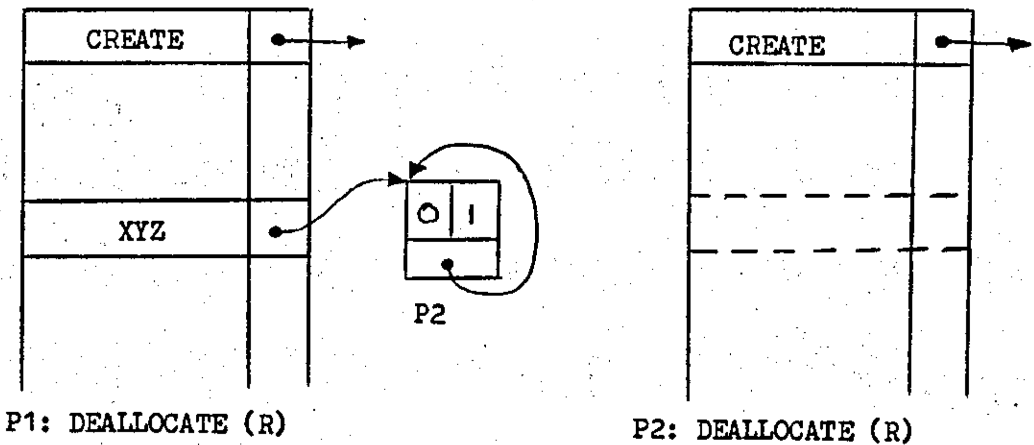


Figure 5.4.1. Example of operation of Allocate and Deallocate routines.

5.5. A Comparison of Synchronising Tools

For a synchronising algorithm to be a viable tool in a multiprocessor system, it must not consume too many of the system resources during operation. Two factors, at least, are a useful indication of the performance of such an algorithm. These two factors are the amount of time during which the resource is requested but is unowned and the amount of time between becoming owner of the resource and being able to use it. These may be thought of as the times between requesting a resource and being allocated it and from being allocated it to using it. The Abstract Resource Ring will be compared with two other synchronising tools found in the literature. These are firstly Dekker's original solution to the problem as described by Dijkstra (22) and secondly a more recent solution devised by Lamport (48). These two algorithms are reproduced in Figure 5.5.1. The algorithms will be compared on the two characteristics noted above.

Firstly, algorithm response time. Ideally, a processor should be able to use a resource immediately after it has been passed (or gained) ownership. By inspection of the algorithms, it is seen that both the Dekker and Lamport algorithms contain multiple loops. In particular, both algorithms require a processor to inspect the state of all other processors with possible secondary loops in certain circumstances. In contrast, however, the algorithm for the Abstract Resource Ring contains only a single tight loop upon a single variable.

In order to investigate this static cost of accessing a resource

```
claim  =
            begin
               i: = our processor number;
label:         while  turn <> i   do
                    c [ i] : = 1;
                    if b[turn]  =  1 then
                         turn: =  i
                    fi
               od;
               c [ i] : =  0;
               for j : = each processor number except ourselves do
                    if c  [j]  =  0  then
                         goto label
                    fi
               od
            end;

release  =
            begin
                    i : = our processor number;
                    turn: = 0;
                    c  [i]: = 1;
                    b  [i]: = 1
            end;
```

where b and c are arrays dimensioned 0 to N, both initialised to 1, and turn is initialised to 0. Processor numbers range from 1 to N.

Figure 5.5.1 a) The Dekkar algorithm

```
claim =
        begin
            i : = our processor number;
            choosing [ i] : =  1;
            number  [ i] : = 1+ maximum of number [ 1] to number [ N] ;
            choosing  [ i] : = 0;
            for  j : = each processor number do
                while choosing [ j] < > 0   do
                            nothing
                od;
                while number [ j]<> 0    and
                      (number [ j] , j ) <(number  [ i] , i ) do
                            nothing
                od
            od
        end;
```

```
release =
        begin
            i : = our processor number;
            number [i] : =  0
        end;
```

where choosing and number are dimensioned 1 to N, both initialised to 0 and $(i,j) < (k,l) \equiv (i < k)$ or $((i = k)$ and $(j < l))$

Figure 5.5.1. b) The Lamport algorithm

empirically, the algorithms of Lamport and the ARR were encoded on a single processor system, but with the data structure that would be required for several. Calls to the GETRES and PUTRES routines were placed in a loop. Table 5.5.2 gives the times obtained with various numbers of processors. The cost of the nested loops can be observed in the times given in the table.

Secondly, what may be called wasted resource time. This is the time during which at least one processor requires the resource, but due to the transitional state of passing ownership (or gaining ownership) the resource remains unowned, or owned by a processor which does not require the resource. With the Dekker and Lamport algorithms, this cost factor is due to the 'bartering' nature of the algorithms and the fact that the resource is freed after it has been used by a processor. With the Abstract Resource Ring, this overhead may be incurred when a processor performs a PUTRES but no processor requires the resource. If a processor later requires the resource, it will be unable to obtain ownership immediately, but will have to wait for the owning processor to check for unwanted resources. The ARR therefore contains some tuning facility in that the frequency of checking for resources may be altered either by changing the frequency with which interrupts are sent or the time step between reactivations of the PUTRES activity. If the frequency is increased, the overhead of wasted resource time will decrease, but the cost of performing the check will increase.

Loop Size	ARR	Lamport's Algorithm*		
		N = 4	N = 8	N = 16
1000	0.81	0.94	1.06	1.31
10000	8.03	9.44	10.64	13.04
20000	16.05	18.88	21.29	26.08
30000	24.07	28.35	31.95	39.15

* N gives the number of processors in the ring

Note: all times are in seconds

Table 5.5.2. Response time comparison

Shared resources which are heavily used, that is when one processor releases the resource another requires it will suffer negligible overhead with the Abstract Resource Ring since a PUTRES will always be completed with no requirement for a delay following a retry. However, the overheads for the Dekker and Lamport algorithms will increase with the number of processors taking part in the resource sharing. Both these algorithms have a section of code which, ideally, would be executed by a single processor at one time. Checks have to be made for multiple execution of that section of code, with possible retries in the case of Dekker's algorithm. As the number of processors increases, so does the possibility of simultaneous execution of the critical section of code by several processors and correspondingly the potential overhead of the algorithms.

It is worth noting that if a resource is heavily used (as mentioned above) then the only overhead associated with the Abstract Resource Ring is the cost of locating the new owner within the PUTRES routine, that is, the cost of searching the ring structure.

To confirm these predictions, the performance of the algorithms was tested under simulation conditions. The simulations were of a coarse-grained nature, with an algorithm-step as opposed to a machine instruction being executed by each processor in turn. This is a sufficient formulation of the algorithms, since no action between algorithm steps may affect the synchronisation being performed, and each step is a single action.

The simulation program was written in the BASIC language and enabled a number of resources to be shared amongst a number of processors. Both the number of resources and the number of processors were supplied as input data. Each processor would randomly choose one of the available resources, claim that resource, hold it for a number of algorithm steps and then relinquish the resource. A further number of algorithm-steps would elapse before that processor would again choose a resource and repeat the cycle. The size of the time periods holding and not holding the resource were specified by input data. The three algorithms were incorporated into the simulation program.

The three algorithms were compared under various configurations and workloads. Figure 5.5.3 shows graphs drawn from some of the results obtained from the simulations. All the graphs show the operation with six resources being shared. Two of the graphs a) and b) show results for a varying number of processors while graphs c) and d) show results for varying workload, that is frequency of resource access. For each variant, a graph is given showing the two critical measures of the performance of the algorithms. The wasted time, expressed as a percentage of total elapsed time, is shown in graphs a) and c) and graphs b) and d) show the total resource usage expressed as a percentage of total possible resource usage.

From these graphs, it can be seen that a performance similar to that predicted is obtained. As the load upon the resource sharing mechanism is increased, either by increasing the number of processors

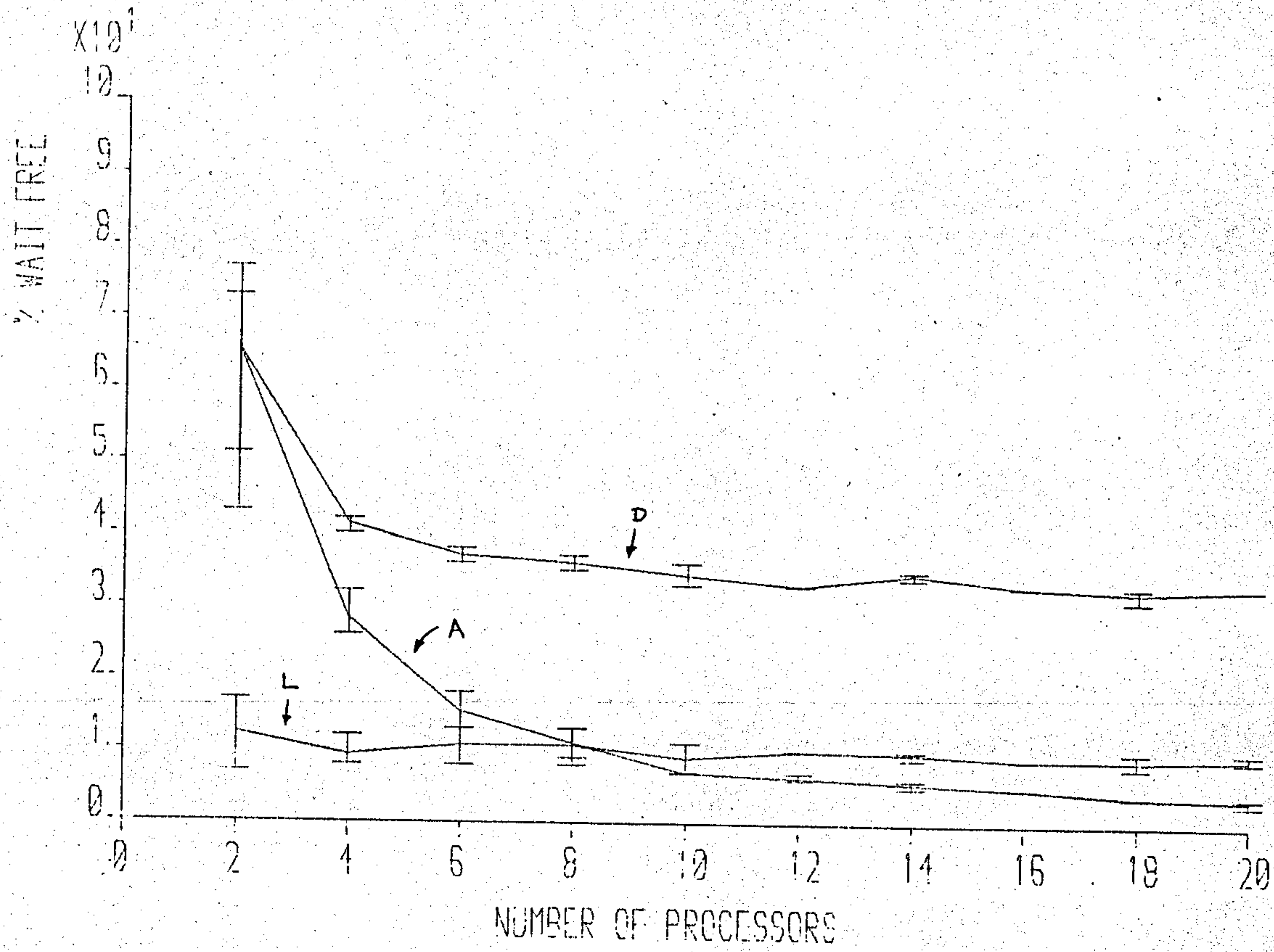


Figure 5.5.3. a)

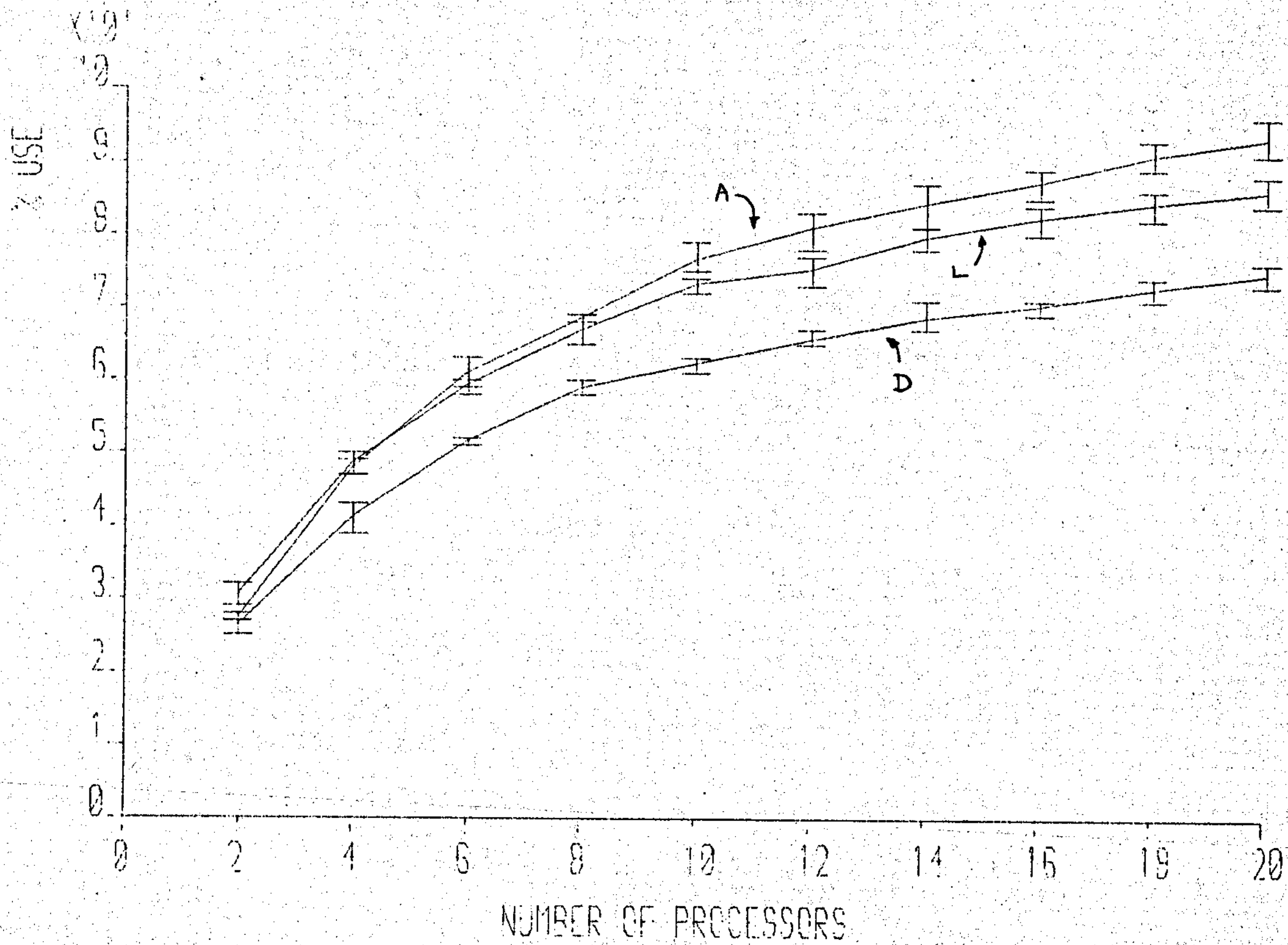


Figure 5.5.3. b)

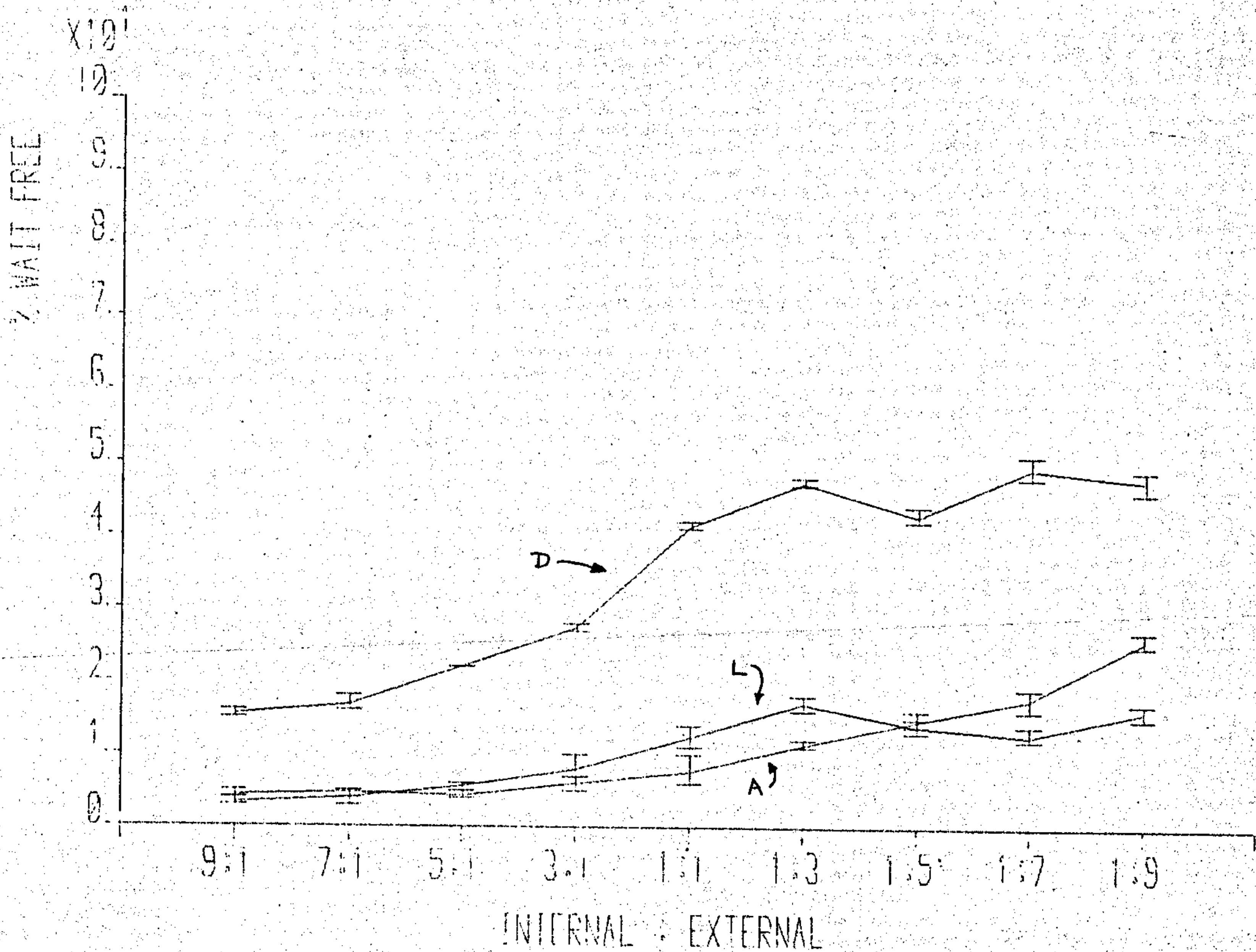


Figure 5.5.3. c)

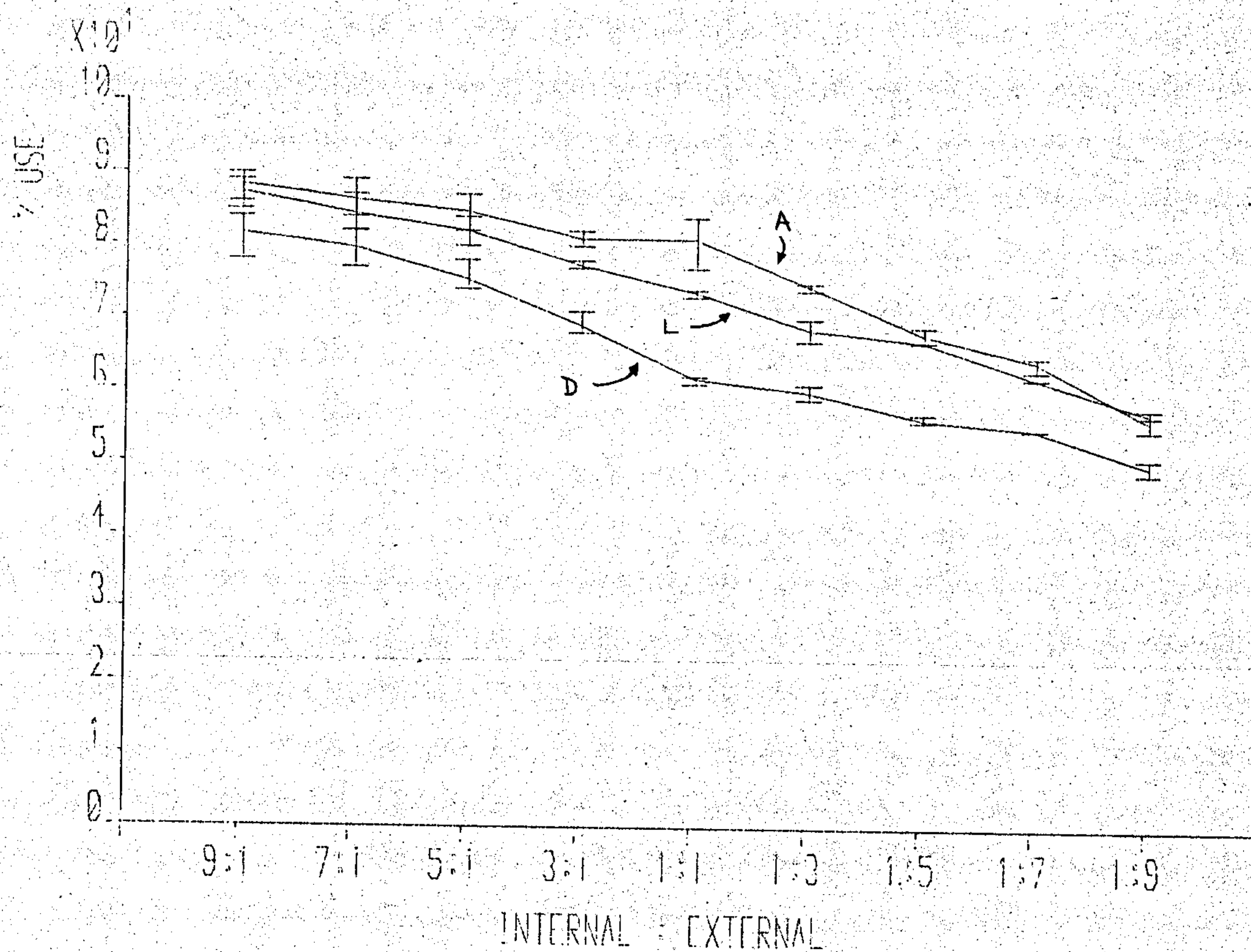


Figure 5.5.3. d)

or by increasing the frequency of access to the mechanism, so the performance of the Abstract Resource Ring improves against that of the other two algorithms. Under light usage, where frequent use of the PUTRES activity will be required, the ARR performs poorly compared to Lamport's algorithm. As the number of processors increases, the Abstract Resource Ring rapidly improves in performance and with a heavy workload gives considerably improved performance (only half of the overheads) against Lamport's algorithm.

Therefore the Abstract Resource Ring is most suited to the protection of heavily used resources, in particular potential system bottle necks. In Chapter Seven it will be shown that even with less frequently used resources the ARR gives acceptable performance.

Another aspect that should be considered when comparing the various algorithms is their ability to distribute the resource usage among the processors. As was noted in Chapter Three, some synchronisation algorithms may allow processors to remain blocked indefinitely if resource usage is heavy. The algorithm developed by Dekker falls into this category. Lamport, however, has developed an algorithm which guarantees service on a first-come - first-served basis.

With the Abstract Resource Ring, however, some scheduling may be incorporated. If the standard searching algorithm is used then no processor will be blocked, the use of the resource being on a form of Round-Robin. However,this search algorithm may be replaced by

another which locates the next user of the resource on another basis, for example on priority. This adds an extra dimension of flexibility to the ARR.

5.6. Multiple Users of a Resource

In the preceding sections of this chapter, it has been assumed that for each protected resource, only a single processor may access that resource at a given time. However, a class of problems have been described, the readers and writers problem (17), in which several types of resource use exist. With some of these types it is possible for several users to simultaneously access the shared resource.

With the Abstract Resource Ring as described, this is not directly attainable. It may also be necessary, within the scope of multiple users, to periodically reduce the number of processors allowed to access the resource. For example, a file may be read by any number of processors, but when one requires to write to that file, it may be necessary to stop any other reading and writing.

Two very similar solutions to this problem are presented in this section, the first using the Abstract Resource Ring in its current format, the second using a modified form of the ARR.

Firstly, the role of the Abstract Resource Ring may be modified. Instead of providing access to the resource directly, the ARR provides access to information on the current usage of the resource. With this technique a processor obtains access to this information block by calling GETRES. Then, after inspecting the data block, the processor decides whether it is able to use the resource or not,

making note in the data block as necessary. Access to the data block is then released by calling PUTRES. This approach requires that the code handling the usage information block be placed in the user program. This may place unwanted management responsibilities upon the user, although great flexibility may be achieved by careful structuring of the data block.

The second approach involves modification of the Abstract Resource Ring. As will be seen in the next chapter, the modification improves error recovery capabilities of the ARR. If the ARR data structure is altered (in some sense inverted) to consist of a node per processor as before, but consisting of only a WANT flag and pointer to the next node in the ring. The CAN flags can be replaced by a single location for each resource ring. This location will contain the name (number or other identification) of the processor currently owning the resource. The GETRES procedure now loops inspecting this new OWNER location until the processor's identification is placed in it. PUTRES places the name of the new owner in the location rather than clearing and setting the CAN flags.

If multiple users of a resource are required, they may be incorporated into the ARR by providing several OWNER locations for each ring. The number of OWNER locations would specify the maximum number of simultaneous users of the resource. A call to the GETRES routine would specify the resource required and also the number of ownership locations required. The processor would loop within the GETRES routine until the required number of ownership locations contained

its name and would then be able to use the resource.

Unfortunately, this algorithm is not sufficiently strong to counteract a possible deadlock. This may be shown by considering a case where four ownership locations exist and two processors require three of these locations each. It would be possible for the processors to obtain two of the ownership locations each thus blocking the other, and the resource.

This problem may be overcome by only allowing a single processor to obtain "multiple ownership" at any one time. This may be accomplished by adding another location, say MULTIPLE, similar to the OWNER locations. Before a processor may attempt to obtain multiple ownership, its identification must be placed in MULTIPLE. The problem arises when two processors partially claim multiple ownership but insufficient ownerships remain to complete either, the problem attacked by the Banker's Algorithm (22). Since any processor between one passing an ownership and the multiple requester which will have a request honoured requires only a single ownership of the resource, the ownership will be used and then be passed on, eventually to the multiple requester. After sufficient ownerships have circulated and been claimed by the multiple requester, it will use the resource. A consequence of this strategy of having a single multiple owner is that a processor requiring multiple ownership of a "higher order" than that which it already has must not retain any of its ownerships until its identification is placed in MULTIPLE since the ownerships

may be required by another processor. This implies that, in general, a processor may not increase its ownership while keeping those it has.

Once a processor has achieved its required number of ownerships, it may pass the MULTIPLE location to another processor since it may only relinquish the ownerships it has, without any possible deadlock.

The PUTRES and PUTRES ACTIVITY must be modified to pass all the ownerships held but only passing to a processor which has need of an ownership, there being no advantage in passing on ownership to a processor which already has its requirements met.

CHAPTER 6

THE ABSTRACT RESOURCE RING

AND RELIABILITY

6.1. Introduction

> "The use of computers in on-line control situations and for other applications giving rise to ever-more stringent reliability and availability specifications, resulted in the construction of systems including two or more central processing units ...... As a result of the multiplicity of units in such multiprocessing systems, failure of any one would degrade, but not immobilize, the system, since a supervisor program could re-assign activities and configure the failed unit out of the system." (50)

If the potential for increased reliability in a multiprocessing system is to be realised, then care must be taken to ensure that the shared resources, including system tables, cannot be corrupted or lost due to the failure of a system component (e.g. the central processing unit). In this chapter, a brief classification of failures is made then the design of the Abstract Resource Ring is re-analysed and an alternative implementation is discussed which enables graceful degradation of the multiprocessor system to take place for one of the classes of failure.

6.2. Classification of Failures

Failures may be categorised into two main groups, namely

1) Hardware failures

2) Software failures

Each of these groups may be subdivided into the following two partitions

a) Cessation of operation

b) Fault in operation

Examples of the type of failure in the four subgroups are

1a) Cessation of operation of a processor may arise if the operator switches off a processor or if a power failure occurs

1b) Faults in hardware can arise in many ways evidencing themselves in such phenomena as 'dropped bits' in memory accesses, a failure in addressing, etc.

2a) Cessation of process execution may arise because of a system deadlock, or a scheduler malfunction

2b) Faulty operation of a process may be evidenced in "random" corruption of code or data due to incorrect coding.

Whilst perfect security and reliability is clearly desirable it can never be achieved in the hardware. At best, the probability of failure can be reduced to a suitably low level. Many of today's reliable systems provide their reliability at a heavy cost in terms

of duplicated components and special logic. Yet with the current state of the art , many areas of potential error are being overcome. For example, store protection and addressing mechanisms have largely overcome the problem of user programs corrupting system code and data. Thus, reliability against a certain type of failure can frequently be achieved in a cost-effective manner. In the bulk of this chapter, consideration will be given to providing reliability to cover class 1a) of failures above, with respect to the CPU only. In sections 9 and 10 of the chapter, brief consideration is given to other errors.

The standpoint from which the solution presented in the next section was taken was to provide a version of the Abstract Resource Ring which would allow the remaining processors to continue to share the resources after the failure of one or more processors.

6.3. Initial Death Detection

The starting point for the investigation was the Abstract Resource Ring with a single resource using the interrupt mechanism to ensure that the resource ownership would be transferred. However, this arrangement will not work as it stands if one or more of the processors on the interrupt ring ceased operation ("died"). Two possibilities could arise:

a) (see Figure 6.3.1.) The interrupts would not complete a cycle of the ring, proceeding no further than a dead processor. In the Figure, processor A can never receive an interrupt to cause it to pass the resource, so it will be lost to all processors except A itself

b) A second, and possibly more catastrophic, situation is that the dead processor owned the resource when it died. The resource would then remain unusable.

Clearly some action is required when a processor dies. This action is required in two phases, firstly the death of the processor must be detected and secondly recovery action must be taken for the dead processor. It is worth noting that for the system to continue to give a response, though degraded, the only recovery that must be taken is that of the shared resources. This is because the processors are assumed to be otherwise independent. Thus it appears that recovery action is only necessary if the dead processor was actually using, that is had ownership of, the shared resource. This

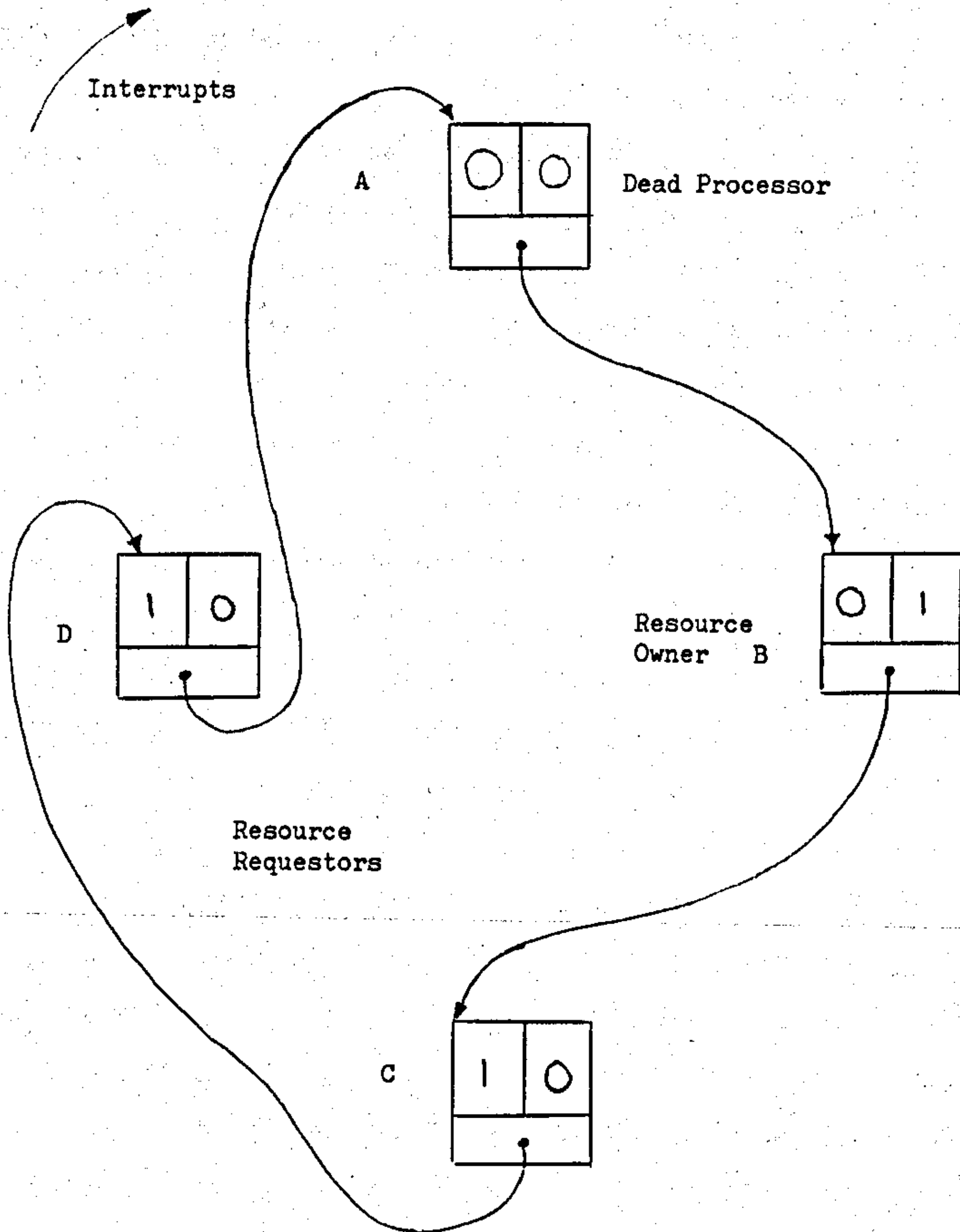


Figure 6.3.1.

information is readily accessable from the ring data structure by inspection of the CAN/WANT flags for the resource to which the dead processor has access. This makes the Abstract Resource Ring system a very good medium for death detection and error recovery initiation.

The first solution followed naturally from the constraint which must be placed upon the recovery:-

> only one processor may perform recovery action on the death of another.

A suitable candidate for the processor performing the death detection is the predecessor of the dead processor, this always being unique in a ring structure. If, when an interrupt from the ring is received by a processor, it acknowledges receipt of that interrupt by sending a reply to its predecessor, then the predecessor may ascertain whether its successor is dead or alive. If a reply is received, within a suitable time span, then the successor is assumed to be alive, otherwise it is deemed to be dead. Once a processor has discovered that its successor is dead, recovery action may be taken.

The form of the error recovery, for the single resource, is shown in Figure 6.3.2. Firstly, the state of the WANT flag of the dead processor is remembered and it is then cleared. This step is to prevent the resource ownership need lessly passing to the dead processor. The processor performing the recovery should now wait for a sufficient length of time for a processor which may be in the

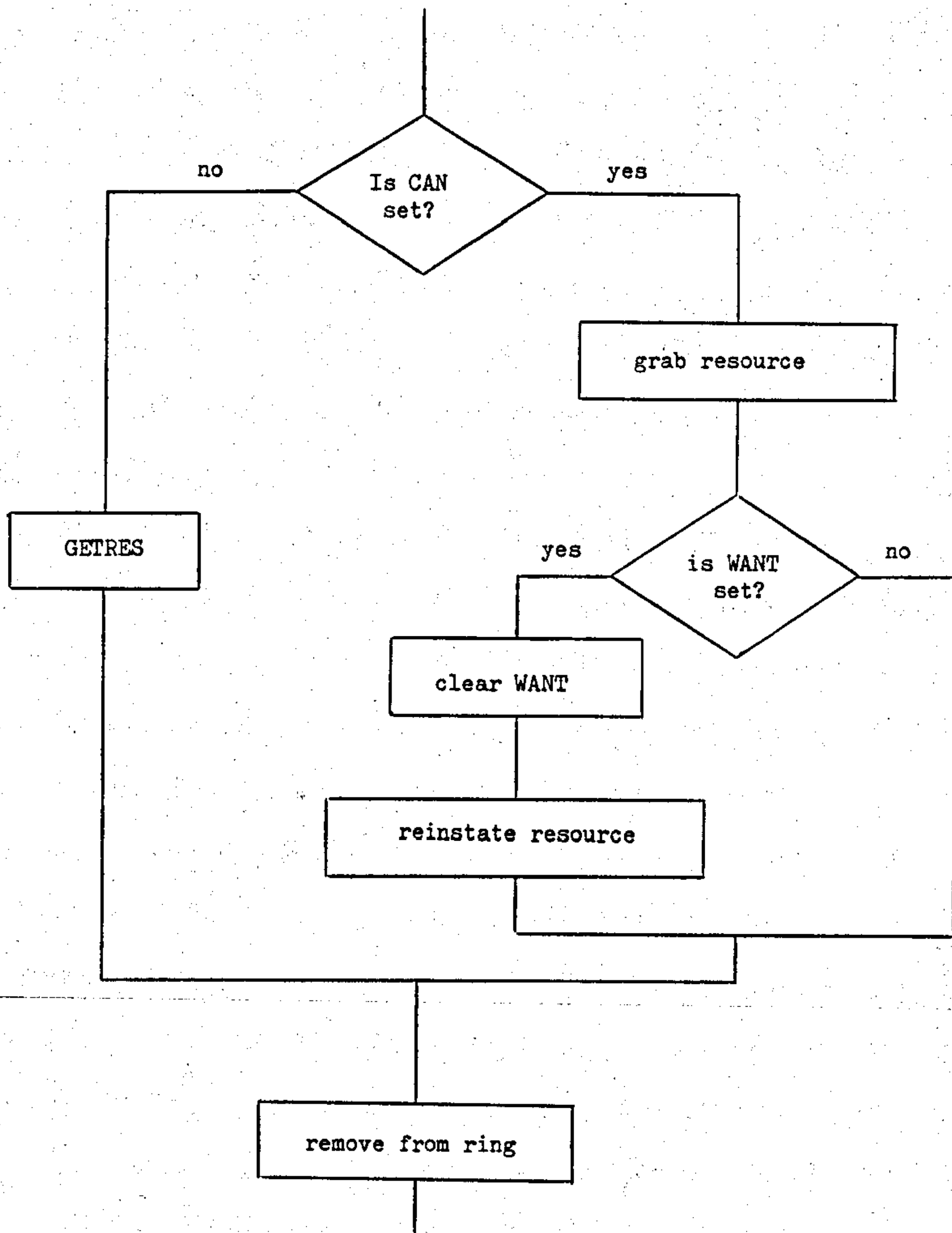


Figure 6.3.2. Basic Flowchart for Recovery Procedure

progress of passing ownership to the dead processor.

Secondly, the CAN flag is inspected. As stated above, no recovery action on the resource is required unless the CAN flag is set. If this situation arises, the ownership of the resource may be forcibly acquired by the (unique) predecessor of the dead processor (termed "grabbing") by clearing the CAN flag of the dead processor and setting its own. The need for the constraint above now becomes apparent. If several processors attempted to recover from the death of another processor, then more than one of them could become the owner of the resource during the recovery period. This would violate the basic premise of the mechanism.

If both the CAN flag and WANT flag of the dead processor were set then not only did it own the resource, but it was potentially using it. In this case,some recovery action must be taken to check the internal consistency of the resource. This may involve a number of steps, for example comparing forward and backward pointers within a data structure etc. In section seven of this chapter, a description of a technique is given whereby a shared data structure may be updated in a manner such that it may be restored to a self consistent state even if the update was only partially made.

Having returned the resource to a useable state, if necessary, recovery action needs to be made to the ARR structure. This recovery is required even if the dead processor was not using the resource. The ring structure mechanism will not function since interrupts

cannot pass the dead processor. The dead processor must be removed from the ring, and the corresponding interrupt path needs to be reformed. Prior to removing the node corresponding to the dead processor, the recovery processor should obtain the resource, if it does not own it, by performing a GETRES. The condition is laid down that a processor may only remove a node from the ring if it owns the corresponding resource. Since only a processor performing a PUTRES, and hence owning the resource, may inspect the nodes of other processors, no other processor may be inspecting the ring while the node of the dead processor is being removed(by the resource owner). The removal is easily accomplished by replacing the NEXT field of the recovery processor's node with that of the dead processor.

With the above mechanism, a system sharing a single resource may gracefully degrade in the presence of a single failing processor. However, many deficiencies remain in the system. In the next section, these deficiencies will be presented and solutions to them will be given.

6.4. Rigorous Death Detection

The following deficiencies can be observed in the recovery aspect of the Abstract Resource Ring as described in the previous section:

i) recovery takes place within a single resource environment only

ii) in general, recovery cannot be made from multiple deaths (see below)

iii) a processor in a repeated stop/start state may be deemed dead, but "come back to life" and potentially cause havoc by using a resource ownership which has been removed from it

iv) the potential need for operator intervention to reconnect lines to ensure that the interrupt path corresponds to the Ring Structure.

To show the validity of point ii) above, consider Figure 6.4.1. Processors A and B have both died, with B owning and using the resource - none of processors A, C or D want the resource. On discovering the death of processor A, as shown in the flowchart of Figure 6.3.2, processor D performs a GETRES on the shared resource. However, that GETRES can never be satisfied since the owning processor is dead and cannot be recovered. That is, a form of deadlock arises.

The development of the Abstract Resource Ring will be described, and it will be shown how the developments overcome the deficiencies above.

The first matter to be considered is the recovery with multiple

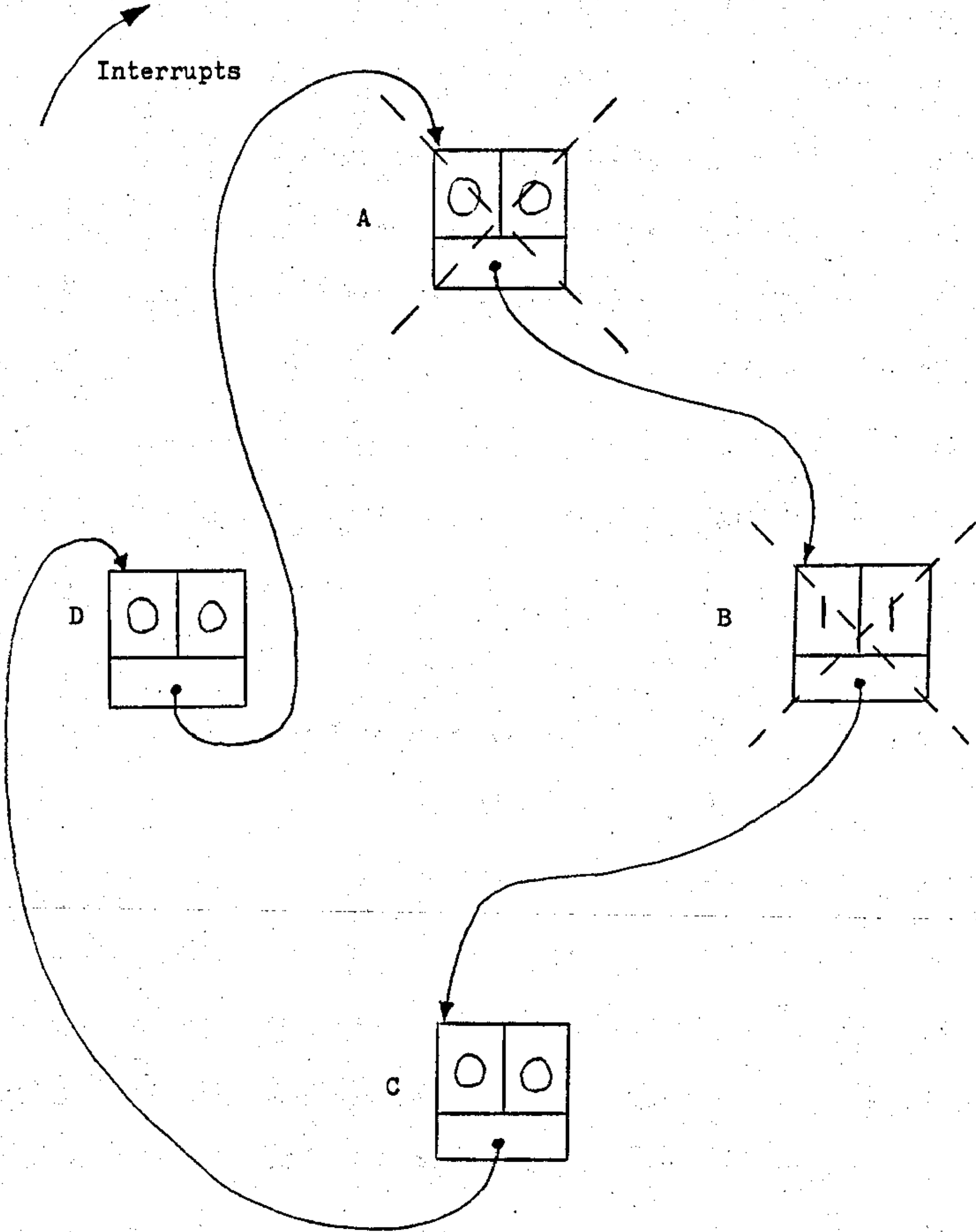


Figure 6.4.1.

rings. In section 5.3. it was postulated that a single process on each processor (the interrupt manager) should administer the interrupt prompting mechanism. Any prompting interrupts would then circulate the Ring Structure passing to every processor, not just those capable of sharing a particular resource. This led to the conceptual splitting of the Abstract Resource Ring into two classes of rings:

a) "Software" Rings - the ring structures used within the sharing of particular resources

b) "Hardware" Ring - a ring structure showing the physical ordering of the processors, and used by the interrupt manager on each processor.

This breakdown of functions naturally allows processor death detection to take place within the context of the Hardware Ring, on a similar principle to that employed with a single resource. The death detection, therefore, becomes independent of the actual resource sharing.

The interrupt manager's function is modified to include the reply/ timeout mechanism, as proposed in the previous section, to enable the death detection to take place. When a death is detected, the recovery process must firstly rebuild the Hardware ring by removing the node for the dead processor and arranging (with possible operator intervention) for the interrupts to be sent to the new successor. This may be accomplished because of the independence of the Software and Hardware Rings. It has been seen that, with a single resource,

detection of multiple deaths was severely handicapped. But for the interrupt manager operating under the Hardware Ring, the death detection may be continued before recovery of any of the Software rings is started.

Having rebuilt the Hardware Ring, the recovery process may then perform any recovery necessary for each Software Ring to which the dead processor is attached. The operations performed will be directly comparable to those for the single resource case. The WANT flags of all resources not owned by the dead processor should first be cleared to prevent it becoming an owner, and thus increasing the cost of recovery. For a particular ring to which the dead processor is attached, it may be that the recovery processor has no node as it is a temporary resource. In this case, before any necessary grabbing of the resource or other recovery action which may be necessary can take place, the recovery processor must add itself to the ring by the technique described in section 5.4. Since this technique relies on the CREATE resource, it may be necessary to overlap recovery procedures if a second (or later) death caused the (temporary) loss of CREATE. Once the recovery of the resource is completed, the processor should remove itself from the ring. Also, removal of the dead processor's node is not as straightforward as in the single resource case, since the recovery processor need not be the predecessor of the dead processor on a Software Ring. The Software Ring may need to be searched to locate the predecessor. The need to own the resource before removing a node is again required since several processors may otherwise be searching the ring.

With the Software Rings, however, (except in the case of transient resources (see section 5.4.)) there is no strict need to remove the nodes of the dead processors. The reason for the removal of nodes of dead processors was to enable further death detection to be performed. Since death detection operates independently of the Software Rings, the removal of nodes need not take place. If the nodes are removed then it reduces the size of the Software Ring, reducing searching costs, however when the processor is brought back into the system after being repaired the cost of adding it back to the rings from which it had been removed must be paid.

Thus, by separating the resource sharing and error recovery aspects of the Abstract Resource Ring, deficiencies i) and ii) above have been overcome.

The need for operator intervention (point iv)) is due to the need for an interrupt path existing between adjacent processors. Some multiprocessors systems have an inter-processor interrupt mechanism (51), yet others do not. In the latter case, external I/O ports may need to be used back-to-back (as in the system described in Chapter Seven). When a processor dies, rearrangement of cabling may therefore be necessary in order to keep the physical interrupt path corresponding to the internal Hardware Ring structure. This operator intervention may be undesirable, and is potentially error prone. An alternative approach to the interrupt mechanism was therefore sought.

The solution to the problem proved straightforward once the principles

concerned had been isolated. The basic requirement, from the error recovery aspect, is that a processor indicates (or fails to indicate) that it is alive. As has been suggested previously, two basic methods may be used to achieve this indication. It may be either

i) on demand

or ii) periodic.

The interrupt mechanism is an example of the first type. A processor indicates that it is alive by replying (on demand) to an interrupt. The alternative might be expected to be of the second type.

If each processor maintains a local clock variable readable by all processors, and this clock is guaranteed to be correct (to within a fixed accuracy) to the "real time" maintained by the system as a whole, then a processor may safely be deemed dead if its local time is outside the required accuracy. A simple realisation of this would be to have a location containing system time and have each processor copy this time into their local time locations, say every second. If the difference between system time and the local time of any processor exceeded one second, then that processor would be assumed dead. A degree of safety may be added by using a cruder accuracy for checking the copy.

These local times would replace part of the nodes in the Hardware Ring, and the interrupt manager would be replaced by a process which periodically checked the local times to detect dead processors. This death checking process needs to compare the local time for its successor

against the current system time. If the processor is dead then recovery action should be taken. However, if multiple deaths are to be handled then the following processor should be inspected. This should continue until all the dead processors immediately following the checking processor are found.

With this technique, no operation intervention is required during the recovery action overcoming point iv) above. However, the problem of maintaining the system clock arises. Initialisation of any clock requires operator interaction, and so when this action is performed, the system clock can be initialised. Each successive processor may then initialise its local time from the system clock. If all processors are given the responsibility of maintaining the system clock according to the rule:-

> "Each periodic interrupt, the local time is advanced. If this time is later than system time then the system time is advanced"

then some advantages follow. If all the processors are operating correctly, then the local time on each will advance in step. If, however, one of the processors dies (or stops), its local time will lag behind the system time maintained by the others, and if it restarts, it will not reset the system time. This difference between system time and local time may be used to improve trapping of the stop/start effect of point iii) above. If the local time of the processor bears a greater discrepancy to the system time than the guaranteed accuracy then the system on that processor could deem itself dead, and terminate any further access to shared resources on the assumption that recovery action had been taken. Note, however, that

if the processor was about to update a shared resource or the ring structure when it stopped it may, on restart, continue with that update, causing possible corruption of data if recovery had taken place. Local time would need to be reset as a specific act if a processor is legitimately restarted after a failure.

If the local time accuracy is not a fixed quantity but made flexible for each processor, then when a processor is about to enter a known stop/start state (for example single-shot operation) the accuracy could be made very crude. This operation would clearly be a function of the Hardware Ring, with the accuracy for each processor being stored in its node on that ring.

From the above discussion, the periodic scheme for handling the function of the Hardware Ring has certain advantages. The first of these is the ability of the total system to reconfigure itself. without the need for operator intervention. In some applications, this may be of some importance. Also, the stop/start state may be more easily handled and, with the periodic technique, a processor may incorporate some self checking against stop/start. However, it does require a clock to be present on each machine in the ring. Against the periodic scheme the arguments of the previous chapter may be raised, that is needless restart of the checking processes and the ensuing processor overhead.

6.5. Failure within ARR Routines

So far, no consideration has been given to the consequences of a processor failing during any of the Abstract Resource Ring routines. Those procedures which need to be considered are GETRES, PUTRES, the allocate and deallocate routines and the recovery procedure.

Of these, the allocate and deallocate come under the class of general resources, since they involve a data structure protected by a resource ring (CREATE). Recovery techniques may be applied to them as to any other data structure.

The remaining three, however, need separate consideration.

6.5.1. GETRES routine

The only operation this routine performs upon the ARR data structures is to alter the value of the WANT flag. If a processor fails during execution of this routine the ARR will appear either with or without the appropriate WANT flag set. Neither of the two conditions is illegal, so failure within GETRES is safe.

6.5.2. PUTRES routine

The consideration given to PUTRES also applies to the PUTRES Activity. The act of clearing the WANT flag cannot affect the legality of the ARR data structure if a failure occurs, nor can the searching of the ring. However, the passing of ownership between processors poses difficulties. The processor passing the resource needs to clear its

own CAN flag and set that of the second processor. This clearly takes more than one operation on most computers, so the processor may fail between the two steps.

If the processor clears its own flag prior to failing, then the ring appears to have no owner. However, if the setting and clearing operations are interchanged and the CAN flag of the new owner is set prior to failing, then after recovery two owners of the resource exist. Both situations break the basic condition for correct operation of the Abstract Resource Ring.

The technique of reliable update,described in section seven of this chapter, may be used to guarantee a legal state of the ring structure data.

Another solution may be obtained by adopting the OWNER location technique described in section 5.6.(that is of maintaining a location holding the identification of the current owner of the resource rather than many CAN flags). The problem arises because a single piece of information, that is the current owner of the resource, has been distributed amongst several nodes. In general, this distribution of information raises many reliability problems. In this context, if the distributed ownership information is replaced by the single OWNER cell, the passing of ownership becomes a single operation and difficulties with failure no longer remain, since the location will either contain the old owner or the new owner of the resource.

The technique of section 7 of this chapter is the generalisation of this technique.

6.5.3. Recovery Procedure

Two possible illegal conditions may arise if a failing processor was executing a recovery procedure.

a) The death of the recovering process may occur after the original processor has been removed from the ring but before it has had complete recovery action taken over its resources.

b) An invalid structure within one of the resources may be generated due to partial recovery being performed upon it.

Consider the latter possibility first. If the processor was actually performing recovery upon the resource, then it must have ownership of the resource. The recovery procedure itself should be constructed in a manner which, if it is being performed as a processor which dies, the recovery may be restarted.

The ring structure is altered on two occasions (see Figure 6.3.2.). Once when nodes are removed and once when the resource is grabbed. The removal of a node involves the changing of a single location in the node (the NEXT field) and is a single operation and is therefore safe with respect to a failure. However, the grabbing of a resource is not a single operation, but it is an operation corresponding to that of the PUTRES routine. Using the single ownership location as described earlier, this operation may be made safe.

The second major condition to be considered is to be able to continue the recovery of the original dead processor. If the interrupt mechanism is employed, the node in the Hardware Ring for the dead processor is removed before any recovery action is taken. So the death of the first processor cannot be rediscovered if the recovery processor dies. Yet, as has been noted, if the node remains, it may not be possible to recover from multiple deaths. The node must therefore be removed. It then remains that a processor must maintain a list of processors from whose deaths it is recovering in order that they may be resumed on its own death, if the need arises. The addition to this table must take place before the node is removed from the Hardware Ring.

With the periodic death detection, no extra action need be taken. Since processors may discover more than one death at a time, the need to remove a node from the Hardware Ring does not arise, so rediscovery of a death is possible and a partial recovery cannot be lost. A consequence of this is that when the periodic restart of the PUTRES activity is used, a dead processor should not be removed from the Hardware Ring until it has had all its resources recovered.

6.6. Addition and Replacement of Processors

Once a processor has failed and been repaired, it would be desirable to be able to add it back to the system, restoring it to full power. Also, the addition of new processors may be possible. Some method must therefore be found whereby nodes can be added to both the Hardware and Software Rings. The rings may be split into two groups

a) those to which the new processor has to be added by another e.g. the CREATE resource

b) those to which the new processor may add itself e.g. one of the temporary resources.

The criterion upon which a processor is added is dependent upon the criterion for removing a node. For some nodes, the removal criterion is that the predecessor on the Hardware Ring must remove the node (corresponding to group a) above), whereas for others, the criterion is the ownership of the CREATE resource (corresponding to group b) above).

For class a) of rings above, another processor is requested to add nodes for the new processor to each of the rings necessary. The nominated processor should be the new processor's predecessor if the new processor is present on the Hardware Ring. If the new processor is being added to the Hardware Ring, it should be placed as the successor of the adding processor in order to reflect the detection criterion. The request may either be by operator command or a request from the processor via a message system, or any other convenient method.

For those rings governed by the CREATE resource, the processor may add itself by calls to the Allocate routine (see 5.4.).

6.7. Reliable Update

In this section, and those following, brief consideration is given to other reliability aspects. In this section, an algorithm is described which permits alteration of multilocation values to be performed safely despite the class of failures under consideration. The procedure assumes that updates by different processors are mutually exclusive, i.e. that synchronisation already exists.

The difficulty with updates of multilocation values is that they are not usually point (indivisible) operations with respect to the failing of a processor. That is, the update takes several steps (instructions) and the processor may fail between any two steps. Thus, after the processor has failed, part of the new and part of the old values are found in the locations. A procedure, known as "Reliable Update" was developed whereby a point operation is introduced into the update. The extra reliability obtainable by the application of this procedure is achieved at the expense of both storage and processing time.

The straightforward update fails because we have a data structure changing from one state to another over several steps. The point operation is introduced to show when the change from the old values to new takes place. In order for this to be possible, we require two sets of locations. One contains the old values and the other the new values. Before the complete update is made, the old values are used. Once the new values are stored, the locations containing the new

values are used. The point operation is introduced to indicate which set of values is to be used.

The operation of the procedure may be demonstrated by considering the update of the table shown in Figure 6.7.1. The table contains a count of elements, followed by that number of elements and then the sum of the elements.

If we now wish to add the number 4 to the table, three locations must be changed. The count of elements must become 5, 4 must be added to the table (say in the sixth location) and a new total must be placed in the seventh location.

In order to be able to perform this update using the Reliable Update procedure, each entry of the table is duplicated. A single bit (or bistable) is associated with each entry. The two values for each entry are known as "value" and "new value" and the bistable is known as "indicator". There must also be another bistable for the table as a whole, called "flag". Initially, all bistables are assumed to be zero and the correct entries for the table are in the value fields (see Figure 6.7.2.).

The procedure falls into four steps, namely

i) For each entry to be changed, store the new value in the corresponding new value field and set indicator (the order of the two operations is irrelevant)

4	12	5	3	6	26	......	

Figure 6.7.1. Origional Table

Flag [0]

Value	4	12	5	3	6	26		
New value	-	-	-	-	-	-		
Indicator	0	0	0	0	0	0		0

Figure 6.7.2. Table for use with Reliable Update

ii) Set flag

iii) Copy all the altered entries from the new value field to the value field and then clear indicator (the copy must be performed first)

iv) Clear flag.

The state of the table after each phase of the procedure is shown in Figure 6.7.3., and an example of the coding is given in Appendix 2.

Clearly, if the update is completed by the process performing it, then the data structure conforms exactly to the assumptions made about it on entry, that is all bistables are zero and the correct values are all in the value fields. Two conditions need to be satisfied for the procedure to be able to withstand a failure of the type proposed in section two of this chapter:

a) correct and consistent values may be obtained from the data structure after the failure

b) the data structure must be able to be brought in line with the assumptions made about its state before entering the procedure.

The crucial phase in the procedure is step ii) and it is this step which provides the indivisible operation for the update. If this step is completed then the data structure is considered to be updated. If it is not, then no update has been made to the data structure.

To obtain a correct value from the table, the following rule should

4	12	5	3	6	26	-			
5	-	-	-	-	4	30			
1	0	0	0	0	1	1	0		0

[0]

a) After Step i)

4	12	5	3	6	26	-			
5	-	-	-	-	4	30			
1	0	0	0	0	1	1	0		0

[1]

b) After Step ii)

5	12	5	3	6	4	30			
5	-	-	-	-	4	30			
0	0	0	0	0	0	0	0		0

[1]

c) After Step iii)

5	12	5	3	6	4	30			
5	-	-	-	-	4	30			
0	0	0	0	0	0	0	0		0

[0]

d) After Step iv)

Table 6.7.3. Table During Reliable Update

be used.

Rule:- The correct value is contained in the value field unless both flag and the corresponding indicator are set in which case it is in the new value field.

Before flag is set, according to the rule, the value field is used for the correct value, giving the appearance of the table not being updated. However, when the flag is set, the values for the entries which have been changed are found in the new value field since their indicator is set. The copying phase returns the data structure to its initial state with modified values.

To recover the data structure one of two operations is performed depending upon the state of flag. If the flag has not been set then the update has not taken place and all that is required is to clear all the indicators which are set (the contents of the new value fields being irrelevant). If, however, the flag is set then the update may be completed by the recovery process performing the remaining copy steps required to bring the data structure to a correct state. The flag should then be cleared.

This procedure is a candidate for safe update of shared resources such as the RESOURCES table.

6.8. Application of Reliable Update to the ARR

If the implementation of the Abstract Resource Ring based on the OWNER flag is used the overhead of the Reliable Update need not be imposed on the basic structures and routines. If each field of the ring structures can be implemented using a single location then no special security measures need be taken (see section five of this chapter).

However, if the proposal of section 5.4. for dynamic creation of rings is incorporated then the Reliable Update must be used. As was noted, the RESOURCES table is a shared resource. Access to the resource is only made while the CREATE resource is owned, and access is therefore made by only one processor. As such the RESOURCES table is a candidate for use with the Reliable Update. This will impose an extra overhead upon the Allocate and Deallocate routines.

With that addition to the Allocate and Deallocate routines, the complete Abstract Resource Ring mechanism can be maintained correct and consistent even in the presence of multiple failures of the class considered.

6.9. Failures Due to Other Errors

In this section, brief mention is made of other types of failure and their effect upon the operation of the Abstract Resource Ring mechanism.

Clearly, as with all systems enabling resource sharing, the possibility of deadlocks is present. The problems of deadlocks have been known for some time (16). Two basic methods can be used to overcome deadlock problems. Firstly, by use of pre-emption to force a process to release (temporarily) a resource (40) and secondly to prevent deadlock from arising in the first place (33,40).

Brinch Hansen describes the Hierarchal Resource Allocation technique (11) for deadlock prevention. This is the technique used within the current implementations of the Allocate, Deallocate and Recovery routines. Each of these routines requires the use of two resources (the CREATE resource and one other) and so a potential for deadlock exists. If the resources are claimed in one fixed order (the same for all processors) and are released in the reverse order, the deadlock cannot take place. So the three routines always claim the CREATE resource last and release it first.

The possibility of deadlock within the ARR routines has, therefore, been removed. However, by bad design of a total system based upon the Abstract Resource Ring, deadlocks could still arise.

A second area to which consideration must be given is that of corruption of the data structures. With all systems, simple or complex, some data is crucial to the safe running of the whole system. The data structures for the ARR fall into this category. At best, corruption may merely cause a delay in the system by unsolicited setting of a WANT flag. Various levels of degradation may be experienced up to complete system failure, for example a single Software Ring may be corrupted causing the loss of one resource only or major corruption may take place requiring the system to shutdown. Extra checks may be incorporated to validate the various ring structures on access, but this will naturally lead to an increase in overheads and still cannot guarantee consistency.

6.10. Self Stabilising Techniques

This section is concerned with the adaption of some theoretical work performed by Dijkstra. One specific case of data corruption, due either to hardware or software failure, is the setting of multiple CAN flags within a Software Ring. This implies that several processors may (wrongly) use the resource. The question posed by this situation is whether it is possible to return from this erroneous or illegal state to the correct state of having just a single owner of the resource. It should be noted that with the version of the ARR having the single ownership location this problem cannot arise.

Dijkstra has published a paper (23) on self-stabilising systems in which he presents examples of systems where, by applying only valid state-transitions within a system, the system will return to a valid state from an invalid state within a finite time. Each system has a (finite) number of privileges and with each privilege there is a corresponding state transition. At each step a daemon, either centralised or distributed, chooses one of the privileges existing and the corresponding state transition is made. The system is said to be self stabilising if it will return to a legitimate state irrespective of the privilege chosen at each step by the daemon.

If the ARR could be made self stabilising, then it would be able to recover from the illegitimate state with multiple CAN flags set. Whether, in practice, this is desirable is questionnable since for

a period of time the critical resource may be accessed by several processors potentially damaging the resource beyond repair.

Dijkstra provides three examples of systems which have the self stabilising property. The first of these causes a single privilege to circulate amongst the finite-stale machines in the system. This system may, therefore, possibly be allowed to provide the facilities provided by the Abstract Resource Ring.

We follow the notation of Dijkstra, that is

L refers to the state of the left hand neighbour of a machine

S refers to the state of the machine itself

R refers to the state of the right hand neighbour

to which is added

W refers to the secondary state of the machine, and corresponds to the WANT flag of the Abstract Resource Ring.

For the system to be described, L, S and R are all represented by integer value in the range 0 to N, where there are N machines in the system. W is a boolean value giving true or false.

A system which describes the operation of the ARR is given by the following privileges and state changes

for the bottom machine:

$\underline{\text{if}}$ L = S $\underline{\text{and}}$ $\underline{\text{not}}$ W $\underline{\text{then}}$ S: = S + 1($\underline{\text{mod}}$ N + 1) $\underline{\text{fi}}$ 6.10.1

for the other machines:

if $L \neq S$ *and not* W *then* S: = L *fi* 6.10.2

for all machines:

if GETRES called *then* W := *true* *fi* 6.10.3

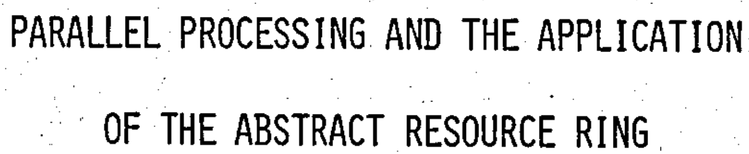
if PUTRES called *then* W := *false* *fi* 6.10.4

The following physical interpretation may be placed upon these rules. Rules 6.10.3 and 6.10.4 govern the setting and clearing of the WANT flag when GETRES and PUTRES are called. Rules 6.10.1 and 6.10.2 cause the ownership to permanently circulate unless a WANT flag is set, in which case ownership will rest with that machine until the WANT flag is cleared. It should be noted that the ownership (indicated by the presence of the privilege) is passed to all processors, not just those wishing to use the resource, so a much greater frequency of checking for unwanted resources must be performed.

Dijkstra provides no proof for his assertion that the system he describes is self-stabilising, but assuming it is, it can be argued that the system described above is also self-stabilising. As mentioned above, we have kept within the constraints of the original system. That is, each of the finite state machines has K states, where K is greater than the number of machines. In the above system of N machines, each machine has N + 1 states. Also, at each step at least one machine will have one of the privileges given in 6.10.1 or 6.10.2 or will be using the resource and will eventually cause PUTRES to be called causing the state change given in 6.10.4. Thus the system

above reduces to that given by Dijkstra but with a delay placed upon the privileges 6.10.1 and 6.10.2 while a machine uses the resource.

CHAPTER 7



PARALLEL PROCESSING AND THE APPLICATION OF THE ABSTRACT RESOURCE RING

7.1. Introduction

In this chapter, the application of the Abstract Resource Ring to an SRC project (under grant BRG 7010) awarded to the Department of Computer Studies at Loughborough University is described. The project comprised three distinct sections including the development of a parallel processing system and the investigation of algorithms run on the system.

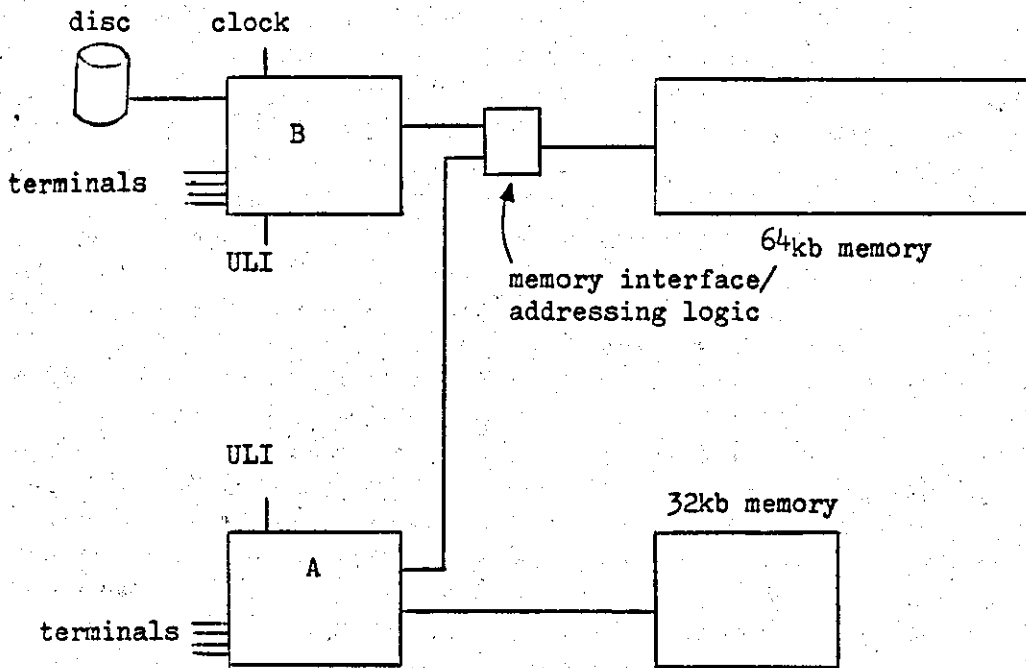
In the next section, the system as delivered by the manufacturer is described. Then the overall design of the parallel processing system and the role of the Abstract Resource Ring is outlined. Details are then given of the implementation of the ARR describing the basic operation and the error recovery capabilities. Finally, performance figures are given for various aspects of the ARR.

7.2. System Configuration

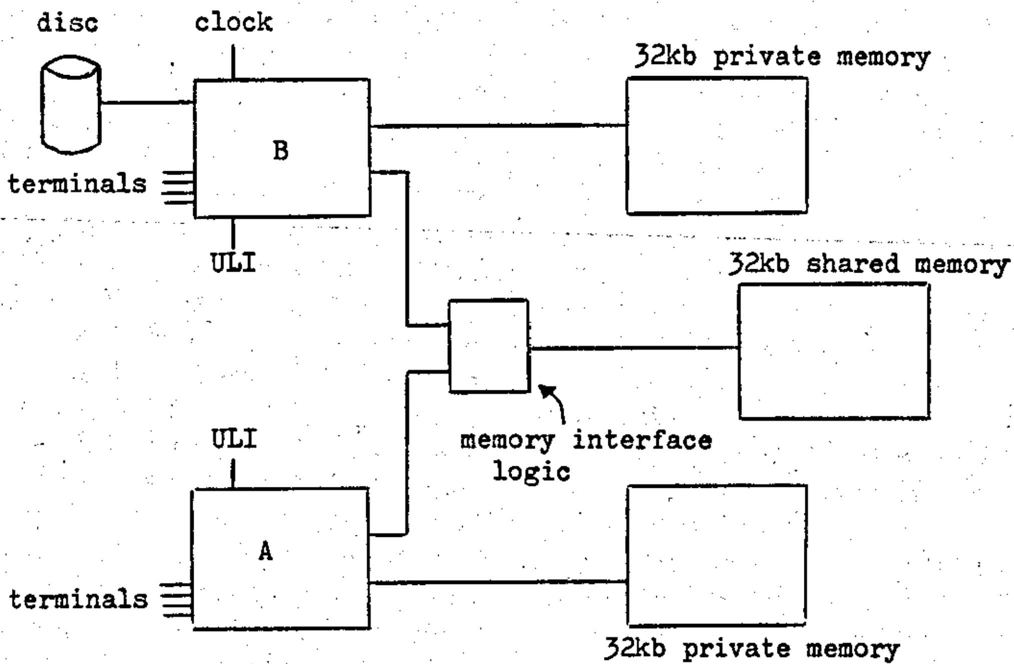
The hardware supplied to the department consisted of an Interdata Model 55 dual processor system (42) with various peripherals. The two processors are known as systems A and B. System A (a Model 70 processor) has 32kb core memory while system B (originally a Model 50 processor but since upgraded to a Model 70) has 64kb of core memory. Both processors have several I/O ports capable of supporting terminals and each has a general I/O interface board, known as a Universal Logic Interface (ULI) (43). System B also has a 9.6 Mb disc system and a clock.

The Model 55 system also includes hardware to enable sharing of core store. Switches are provided to allow various address ranges of store to be shared, but of those which may be obtained only one is of interest. In running parallel programs, the system is configured so that System A has access to the top (high address) 32kb of System B's memory. This gives the symmetric configuration shown in Figure 7.2.1, with the two processors having 32kb private memory and sharing 32kb of common memory. The address space is the same for both machines, that is the common memory is addressed from 32k to 64k-1 by both processors.

Various items of software were delivered with the system including a Disc Operating System (DOS) (41), compilers for FORTRAN and Assembler and various utilities. DOS, however, was not designed to run the dual configuration, being a crude interactive, single user, non-multi-programming system to run only one processor.



a) Physical Configuration



b) Logical Configuration

Figure 7.2.1. Dual Interdata Configuration

7.3. Parallel Processing System Design

Much of the design of the parallel processing system arose from the nature of the operating system as supplied by the manufacturer. At an early stage it was decided that the parallel processing system would run as a subsystem under DOS and that both processors would run an independent version of DOS. That is, the operating system would remain largely unaltered and all necessary synchronisation and resource management would be handled by the parallel processing sub-system. Also, since DOS is a uni-processing system, the "program" run by each processor would be the parallel processing system scheduler.

A parallel program is considered to be one which initially consists of a single stream of instructions. This stream may divide into several parallel branches (which may or may not consist of similar sequences of code). These branches later merge together at a single point to reform the original single stream. Any one of the parallel streams may itself branch and then rejoin. At any one time, a number of streams of code may exist and each is considered as one of a set of parallel processes any of which may be executed.

The parallel processing scheduler providesall the facilities necessary for the creation and deletion of parallel processes, and the maintaining of the correct hierarchal ordering of the processes. The scheduler maintains a list (in shared memory) of the processes to be run together with process ordering information.

Each processor, then, runs the scheduler which searches the scheduler list for a process (synonymous with a parallel path) which it may run. When one has been located, the scheduler jumps to that process, causing it to be executed. On completion of the process, return is made to the scheduler. Two routines, based on standard parallel statements, were developed to enable a process to enter the scheduler. The first, FORK, causes new paths (processes) to be created and the second, JOIN, causes several paths to be merged together.

More detailed information may be found elsewhere (4,5).

No synchronising hardware or software was available with the system, yet an obvious need for such a tool existed. It was decided, therefore, that the Abstract Resource Ring should be used to provide the synchronising facilities required for the parallel processing system. In fact, the ARR was originally designed to meet this problem. The use of the ARR would arise in two situations. Primarily, the ARR would be used by the parallel processing system itself to protect its own access to the scheduler list. Secondly, an interface to the ARR would be provided to enable high level constructs, such as critical regions, to be implemented within user programs. Just as no synchronising mechanism existed on the machines, so no interprocessor interrupt was available. An external interrupt path had, therefore, to be created. It was decided to use the ULIs available on the machines since they were easier devices to control and operated at much greater

speeds (1.9 MBytes/Sec) than the terminal parts. Also, small quantities of data could be transmitted using only the control lines.

7.4. ARR Implementation

In this section, the implementation of the Abstract Resource Ring for the parallel processing system is described. Two different implementations have been made, and both will be discussed. The first (and original) implementation provides death detection facilities, but no error recovery is included whereas the second implementation provides full error recovery capabilities.

The original implementation was based upon the "on request" philosophy, that is it employs interrupt sending for passing the resource and for death detection.

The interrupt manager functions of the ARR mechanism were incorporated into the driver for the ULI. This interrupt manager can be entered in two contexts, either by an interrupt being raised on the ULI or by the GETRES routine requesting the manager to send an interrupt. The interrupt manager has power to ignore requests from GETRES if it deems that interrupts may arrive too rapidly at the other processor.

Due to the philosophy and design of DOS, many of the functions of the ARR which were described in terms of individual processes may not be encoded as such. The process to check for unwanted resources, which should be initiated by the interrupt manager when an interrupt is received from the predecessor, was incorporated into the interrupt manager while the death detection and reporting was distributed between

the interrupt manager and the GETRES routine.

The GETRES routine raises the resource request flag, then loops (for a fixed maximum number of times) inspecting the CAN flag. If this flag is set then return is made from the routine. If, however, the flag is not set within the number of loops then a request is made to the resource manager for an interrupt to be sent. The GETRES routine then waits in a secondary loop (also of a fixed size) inspecting not only the CAN flag but also a reply word. This reply word, cleared by the GETRES routine, is set every time a reply is received by the interrupt manager. The GETRES routine may leave this second loop prematurely on two counts. If the CAN flag is set then return is made from the GETRES routine, it now being irrelevant to current needs whether the processor is dead or not. This is only true of a two processor system since in this case there can never be the need for one processor to check its successor for death on behalf of a third which actually requires a resource. The GETRES routine also leaves the second loop if the reply word is set, returning to the start of the first loop to wait before sending another interrupt.

If, however, the second loop is completed before a reply is received, then the other processor is deemed dead, a message is reported by the GETRES routine for the operator and the parallel program is abandoned with no error recovery taking place. Note that an infinite loop is an acceptable solution, as the parallel processing system is running in a uniprogramming environment. The algorithm is shown in Figure 7.4.1.

```
getres  =
        begin
            i : = our processor number;
            WANT of node [i] : = set;
            while  true  do
                count : = time before next interrupt sent;
                while count > 0  do
                    if CAN of node [ i] =  set then
                            return
                    fi;
                    count : =  count - 1
                od;
                send an interrupt;
                count : = time allowed for reply;
                while count>0  and no reply received do
                    if CAN of node [ i]  =  set then
                            return
                    fi;
                    count : = count - 1
                od;
                if count = 0 then
                    the other processor is dead;
                    report error;
                    stop
                fi
            od
        end;
```

Figure 7.4.1. Implementation Algorithm of GETRES

The software was written with a fixed number of resources (eight) of which one represents the synchronisation within the parallel processing scheduler, and the remainder are for use by application programs.

The second implementation of the ARR provides the same user interface as the previous one. However, full death detection and error recovery procedures are incorporated, giving a powerful system for the user.

The improved system also uses the interrupt mechanism for notification purposes but it is built into a modified version of the DOS system allowing the resource checking and error recovery to appear as separate processes.

With this implementation, the GETRES routine requires no communication with other parts of the ARR, so having set the WANT flag, a single loop upon the CAN flag is adequate. Again, an infinite loop is allowable since, as will be seen, the other processes are interrupt driven and are run to completion. As with the earlier implementation, each time the loop is completed, an interrupt is sent before the loop is restarted.

The ULI driver has incorporated into it not only the code to drive the ULI but also the code to enable the initiation of the process which checks for unwanted resources and code to start a recovery process if a dead processor is detected. Whilst all the steps of the recovery process are not required for the two processor situation,

it being possible to treat this as a special case, the software has, nevertheless, been designed with more processors in mind. Indeed, more processors may be added to the system with only minimal modification being required to the data structure. The recovery process is invoked if the successor on the hardware ring fails to reply to an interrupt within a fixed time. This process enters the name of the dead processor in the table of dead processors, and proceeds to remove it from the hardware ring. Having removed the dead processor from the hardware ring, the PUTRES Activity is initiated.

The PUTRES Activity not only checks for unwanted resources and attempts to pass them to another processor but also checks each resource ring to see if the successor of this processor is still alive. If this processor is dead (i.e. its name is present in the table of dead processors) the node for the ring is removed and if the resource was owned and being used by the dead processor, the integrity checking/recovery process for that resource is initiated. The identification of this process is contained within the nodes of the ring. If no process identification is contained in that field of the node, then it is reported that the resource has been reinstated, but without an integrity check. Currently, only the parallel processing resource has any recovery incorporated and this recovery is described in the next section.

The PUTRES Activity is also initiated at periodic intervals.

7.5. Reliability and Recovery Procedures

In this section, the reliability aspects incorporated into the parallel processing system, that is for the associated resource, will be described.

With several processors corporately working on a parallel program, it would be desirable to have the program completed even if one of the processors failed. This may include a processor failing while executing one of the parallel processes (paths). The need may therefore arise for a path to be restarted by a different processor in order to complete the program as a whole. To be able to restart a path, the variables for that process must be restored to their value before the path was originally started. Also independence must exist between that path and any other.

Information upon the current state of each path (that is whether it is being executed or not and if so by which processor) is maintained within the scheduler list. It is therefore possible to discover if a processor which has died was executing one of the parallel processes. Part of the function of the recovery routines is to search the scheduler list for any paths being undertaken by the dead processor and to make them restartable by another processor.

Currently, the function of restoring the path to its original state has to be performed by the applications programmer. Some routines have

been written whereby, prior to starting a path, the initial values of variables which could be altered may be saved. Within the path, the process may interrogate the scheduler to discover if the path has been restarted. If it has then a further routine enables the saved variables to be returned to their original value. The scheduler maintains information on which variables have been saved by which paths. When paths are successfully completed, any space occupied by variables held for that path is freed for future use. Figure 7.5.1. gives an example of the use of these routines.

The reliable version of the Abstract Resource Ring has been used on an experimental basis. A number os parallel algorithms have been run on the dual processor system, and failure has been induced by switching off the power to one processor. The error recovery routines have functioned, although, with some algorithms, the saving of variables has proved expensive in time. Some of this overhead can, however, be attributed to the need to make these routines callable explicitly for the FORTRAN source, which incurs checking by the run time system. Ideally, the calls to the variable saving routines would be inserted automatically by a "parallel FORTRAN" compiler with much of the run time checking removed.

With some algorithms, notably those of an iterative nature, the inclusion of variable saving has proved unnecessary, as the formulation of the algorithm will withstand data that is not completely updated.

```
      $SHARED  X (20, 3)
C     INITIALISE SHARED ARRAY X
           :
C
C     OBTAIN A NEW AREA TO SAVE MODIFIED VARIABLES
C
      $SAVEI
C
C     NOW SAVE THE ARRAY X - TYPE IS REAL
C
      DO 10 I = 1, 20
      DO 11 J = 1, 3
      $SAVE  REAL, X (I, J)
   11 CONTINUE
   10 CONTINUE
C
C     NOW GENERATE PARALLEL PROCESSES - ONE PER COLUMN
C
      $DOPAR  100 I = 1, 20
C
C     CHECK TO SEE IF THIS PATH HAS BEEN RESTARTED
C
      IF (RESTRT (DUMMY).EQ. 0) GOTO 20
C
C     YES - IT HAS BEEN RESTARTED
C         - RESTORE OUR COLUMN OF X
C
      DO 19 J = 1, 3
      $REST  REAL, X (I, J)
   19 CONTINUE
   20 CONTINUE
C
C     REMAINDER OF PATH MODIFIES THE COLUMN OF X
           :
C
C     END OF THE PATH - WHEN ALL PATHS TERMINATE SO
C     WILL THIS SAVE AREA
C
  100 $PAREND
```

Figure 7.5.1. Example Use of the Restart Routines

7.6. Performances

In this section, results are presented for various algorithms run on the parallel processing system at Loughborough. The times shown in Table 7.6.1 are given for the original implementation (labelled ARR in the table), the implementation with added reliability (RARR) and, for comparison, an implementation of Lamport's algorithm (see section 5.5.) (L).

Times are given for:-

i) the total elapsed time of the programs, that is the time taken from starting the program until the last processor finished. (T)

ii) the processor idle times, that is the time when processors were either waiting for a path to execute or for a resource to be passed to them. (I)

iii) the nett processing time, that is the time when the processor was performing the algorithm (which is given by i) - ii)) (N)

and iv) the total nett processing time, that is the total of iii) for the two processors. (Total)

The table also shows the time taken when the same algorithm is run on a single processor both with and without the ARR parallel control software. The timings for four different algorithms have been given, these being:-

Program	Processor	ARR A	ARR B	RARR*[2] A	RARR*[2] B	L A	L B
i) Matrix	T	4.88	4.88	6.28	6.28	4.86	4.86
Multiplication	I	0.15	0.01	1.56	1.46	0.12	0.00
(RBMTX1)	N	4.73	4.87	4.72	4.82	4.74	4.86
	(total)	(9.60)		(9.54)		(9.60)	
(uniprocessor - with/without ARR 9.56/9.49)							
ii) Eigenvalue	T	17.78	17.78	27.15	27.15	17.66	17.66
Solver	I	0.96	0.62	9.56	10.63	0.95	0.56
(RBEIGR)	N	16.82	17.16	17.59	16.52	16.71	17.10
	(total)	(33.98)		(34.11)		(33.81)	
(uniprocessor - with/without ARR 34.55/34.20)							
iii) PDE	T	26.18	26.18	40.11	40.11	25.78	25.78
Solver	I	1.49	0.39	14.94	14.65	1.34	0.06
(RBDIF4)	N	24.69	25.79	25.17	25.46	24.44	25.72
	(total)	(50.48)		(50.63)		(50.16)	
(uniprocessor - with/without ARR 50.85/49.84)							
iv) Adaptive	T	24.02	24.02	24.78	24.78	24.02	24.02
Quadrative	I	6.30	0.01	6.99	0.47	6.30	0.02
(RBINT2)	N	17.72	24.01	17.79	24.31	17.72	24.00
	(total)	(41.73)		(42.10)		(41.72)	
(uniprocessor - with/without ARR 42.54/42.53)							

Glossary on page 181

Table 7.6.1. Performance Figures From Dual Interdata 70 System

Notes: 1) All times are shown in seconds. 2) The times for the RARR do not include overheads for variable saving.

i) Matrix Multiplication

This program performs the multiplication of two square matrices.

ii) Eigenvalue Solver

This program evaluates the eigenvalues of a system using a bisection algorithm based upon sturm sequences.

iii) PDE Solver

This program solves a set of partial differential equations using a successive line over-relaxation method.

iv) Adaptive Quadrature

A function is integrated over a given interval with the integration being performed over a sequence of interval bisections until a required accuracy is obtained.

From the table of times, various comments may be made. Firstly, comparing the two implementations of the Abstract Resource Ring, it is seen that the use of the reliable implementation gives a greater total elapsed time for the completion of the program. Most of this increase in time is attributed to the idle processor time, which in turn is due to a lower frequency of interrupt sending with the RARR system. However, when placed within the context of a general multiprocessing system, this spare processor time may be rescheduled to other processes capable of being run giving a higher processor utilisation than is presented in the Table. Timings with a much lower processor idle time may be obtained by tuning the RARR system to each particular algorithm. In practice, this would probably be

unrealistic, so no modifications were made to the RARR software between the running of the various programs.

Considering the nett times of the two implementations, it is seen that the reliable version of the software does impose an overhead in processor time, in general of the order of one per cent. This increase in processor time is due to the cost of the improved death checking.

Comparing the first implementation of the Abstract Resource Ring and that of Lamport's algorithm, as would be expected, Lamport's algorithm gives better timing figures. However, the gain is not dramatic. From section 5.5., it may have been expected that, with only two processors, a large increase in speed would be attained, yet when viewed within the context of a complete algorithm, the reduced overheads of synchronisation become less apparent. It should be noted, however, that as the number of processors increases, the ARR will perform more favourably than Lamport's algorithm. As yet, however, a system with more than two processors is unavailable to test this hypothesis.

The original implementation of the Abstract Resource Ring has been used for a period of over three years, and many parallel algorithms have been run (3). Some of the algorithms have been completed in just over half the time when run on the two processor system as compared to a single processor system, that is close to the theoretical

limit.

Another aspect of performance which must be considered is the amount of memory occupied by the Abstract Resource Ring and its associated routines. Table 7.6.2 shows the amount of core required by the various aspects of the ARR and the reliability and recovery routines. In comparison, the parallel processing subsystem occupies a total of some 4.5 kb.

	ARR	RARR
GETRES/PUTRES/DRIVERS etc.	0.5 kb	0.8 kb
Reliability/Error Recovery	-	0.7 kb
Data and Messages	0.4 kb	0.8 kb
Variable Saving Code	-	0.8 kb
Variable Saving Data	-	1.0 kb
TOTAL	0.9 kb	4.1 kb

Notes 1) On average 1 instruction occupies 3 bytes

Table 7.6.2. Implementation Sizes for Dual Interdata 70.

CHAPTER 8

GARBAGE COLLECTION - A MULTIPROCESSOR APPLICATION

8.1. Introduction

In this chapter, consideration is given to a particular problem that has been applied to multiprocessors of the type being investigated in order to show that a parallel solution should be developed in its own merits and not necessarily be many coordinated copies of a uniprocessor solution.

Within list processing systems, nodes are repeatedly added to and removed from the various lists. The storage locations in the memory space available to the list processing system tend to be allocated for use in a particular list and then freed. It is clearly desirable to reclaim these freed cells for subsequent use, and there are a number of techniques whereby this may be accomplished. The one that is of particular interest for the ensuing discussion is Garbage Collection which was first proposed by McCarthy (52) and used in the LISP 1.5 system (53).

Using this technique, the problem of storage reclaimation is (often) ignored until the list of available cells (free list) becomes empty. When this arises, the list processing is temporarily suspended and a garbage collection process locates cells which have become free and adds them to the free list.

The basic garbage collection algorithm falls into four phases:-

1) Marking phase in which all accessible nodes are marked.

2) Relocate phase in which all accessible nodes are compacted into a single contiguous area.

3) Update phase in which all pointers to relocated nodes are changed.

4) Reclaim phase in which the inaccessible cells are collected to form the new free list.

Of these phases, numbers 2) and 3) may be omitted if desired.

Interest has recently arisen in using multiprocessor systems for list processing (58). With this scheme, one processor would perform all the list processing operations, while a second would perform the garbage collection function. By splitting the operation of the total system between two processors, the garbage collection may be run in parallel with the list processing, not just when the free list becomes exhausted. In this way, an improved response to the users should be achieved.

Lamport (49) has taken the solution for the dual processor list processing / garbage collection problem developed by Dijkstra et al (25) and expanded it to incorporate multiple list processors (mutators) and multiple garbage collectors.

Consideration will be given to the marking phase of the garbage collection and it will be shown that the marking algorithm used by Lamport may be improved by aligning it more with the inherent structure

of the system.

Firstly, the terminology will be introduced, then the algorithm adopted by Lamport will be described. A different solution will be developed and finally results will be presented to show the performance of the two algorithms.

8.2. Definition of Terminology

The list structure to which consideration will be given consists of a collection of list cells (nodes). Each node consists of some (and possibly no) data fields and an ordered sequence of pointers to other nodes (edges). The node from which an edge emanates will be called its source and that to which it points the destination. Some of the edges are distinguishable as null edges, that is the edge does not connect two nodes but acts as a terminator.

If an edge connecting two nodes (A and B)exists and B is the destination of that edge then B is (one of) the successors of A and A is a predecessor of B. Nodes having no successors are known as terminal nodes (or terminals).

Some of the nodes, called root nodes, are fixed. A node is said to be reachable (or accessible) if there is a path to it from a root via reachable nodes. A non-reachable node is called a garbage node.

8.3. Lamport's Algorithm

Lamport introduces an extra field into the nodes for use during the marking phase. This field is intended to hold a colour which may be one of black, grey or white, and indicates at which of the stages of the marking phase the node is.

Operations are introduced to change the colour of a node to a specific value. Also introduced is a shading operation which changes a white node to grey but leaves other colours unchanged. These operations on a node are required to be indivisible with respect to the list processing system (i.e. they must be point operations).

The node space is divided into several (not necessarily disjoint) subsets. A marking process (marker) is assigned to each of the subsets. No details are given as to the method of division, so a physical division seems simplest. Initially all nodes are white.

The operation of the marking algorithm commences with the roots being shaded. Then each marker searches its subset of nodes. When a grey node is located by any one of the processors then it shades all the successors of that node and colours the original node black. All the markers are then requested to restart the search of their portion of the node space. The marking terminates when no grey nodes exist, i.e. all reachable nodes have been coloured black. The garbage (unreachable) nodes are then those that remain white.

Several points may be made about this algorithm. Firstly, no attempt is made to use the structure of the list within the algorithm itself. All reachable nodes may be located by chaining down the list structure from the roots. This leads to a second point, that all the garbage nodes will have to be inspected, possibly several (and in some cases many) times. This time is, of necessity, "wasted" since a garbage node, by definition, cannot become grey. This is an inevitable consequence of dividing the node space in physical subsets.

Further, the synchronisation between the markers is non-trivial, despite the fact that Lamport glosses over the problems. The ability for one marker, on discovering a grey node, to cause all others to restart the search of their subspace requires a "communication path" between every pair of markers. Also, when a marker completes the search of its subspace,no guarantee can be given that it has completed its work as another marker may later discover a grey node. Only when all the markers have completed searching their own subspaces can the marking process terminate. This requires each marker to monitor the state of all the others(potentially requiring much communication traffic or frequent access to shared variables). Again, on a particular implementation it may be possible for all the other markers to "appear" to have completed their marking, yet for a restart request to be received or, worse still, in transit.

8.4. Chaining Algorithm

Since it was noted that objections may be raised against the above algorithm, due to its lack of correspondence to the data structure an algorithm more closely aligned with the data structure was developed. The algorithm, described below, marks the reachable nodes by searching down the list structure and hence has been given the name Chaining Algorithm.

In order to partition the list space, and thus enable several markers to operate, the concept of a subject is introduced with the Chaining Algorithm. Each marker is allocated a section of the total list structure and marks the nodes contained in this sublist. Once a marker has a sublist it may proceed independently of the other markers (thus reducing the synchronisation overheads). However, to enable marking to be equitably distributed between the markers, an additional list, the subtree list, is introduced.

This list contains the roots of unmarked sublists. Initially, the list contains the roots of the whole list structure. The list can be kept short, with possibly one entry for each marker since this list represents work yet to be allocated to a marker. The colour yellow is introduced for a node contained within the subtree list, so the roots of the list structure are initially coloured yellow. Also, the term "uncoloured" is introduced for a node which is either white or grey.

When a marker is initially started, or whenever it has completed the marking of a subtree, it removes a node from the subtree list to discover the section of the list which it is to process. This node is shaded. The marker then refills the subtree list, by adding the uncoloured successors of the subroot to the list until either the list is filled or only one uncoloured successor remains. Those nodes added to the subtree list are coloured yellow. At all stages in the remainder of the algorithm, yellow nodes are treated as black when encountered by a marker since the nodes following them are guaranteed to be marked at a later stage.

The remainder of the algorithm, shown in outline in Figure 8.4.1, is as follows. The marker maintains two pointers to the subtree it is processing, the root of the subtree and the node which it is currently inspecting. Both of these initially point to the root of the subtree. If only one uncoloured successor of the current node exists then that node is shaded, the current node is coloured black and both the subroot and current pointers are advanced to the successor. This process is repeated until a node with several or no uncoloured successors is met. If the current node has some uncoloured successors then one is chosen. It is shaded and the current pointer is advanced to it. This shading and advancing is repeated until the current node has no uncoloured successors. When this situation arises, the current node is coloured black and the current pointer is set to the subroot. The whole of this procedure is then repeated until the subroot is coloured black. When that occurs, the subtree for which the marker was responsible

```
marker =
    begin

        while   subtree list is not empty do

                remove node from subtree list;

                shade node;

                refill subtree list;

                while   subroot is not black do

                        while  number of uncoloured successors = 1 do

                               shade successor;

                               colour node black;

                               advance to successor setting as subroot

                        od;

                        while number of uncoloured successors > 0  do

                              choose one successor;

                              shade successor;

                              advance to successor

                        od;

                        colour current black;

                        current: = subroot

                od

        od

    end;
```

Figure 8.4.1. Algorithm for a marker

has been marked and a new root is chosen from the subtree list. The marker terminates when it cannot obtain a node from the subtree list.

With a simply connected list structure (that is one containing no closed loops and no interconnection between sublists), the algorithm is guaranteed to be correct and to terminate, the list structure appearing as many independent lists each with its own marker. Furthermore, the only synchronisation required between the markers is when accessing the subtree list. If the addition to and the removal of a node from this list are independent, then the overheads of the synchronisation when accessing the subtree list may be reduced. If one marker is attempting to refill the subtree list then the overheads may again be reduced by allowing further markers to by-pass the refilling stage of the algorithm. The initial phase of the marking algorithm then becomes as in Figure 8.4.2.

If the list structure is not simply connected but the subtrees have common nodes (but still without loops) then consideration must be given to the possible events at the intersection points. The simplest possibility to consider is that one marker colours the common node yellow or black before any other marker accesses that node. When another marker reaches the node, it will proceed no further. If the intersection node is white or grey then the structure beyond the node needs to be inspected and several markers may attempt to colour the subtree. This will have the same effect as several passes down the branch by a single marker, that is, the several markers will jointly

```
marker =

    begin

        while subtree list is not empty do

            remove node from subtree list;

            shade node;

            if no other marker is refilling the subtree list then

                refill subtree list

            fi;

            .
            .
            .
            .
```

Figure 8.4.2. Modified Initial Stage for a Marker

colour the nodes below the intersection point.

If two markers attempt to update the colour of the intersection node simultaneously, then one must complete its update after the other. The node then becomes that colour. Whichever colour is finally given to the node, it is valid for at least one of the markers,and this marker will complete the colouring.

However, with the algorithm as described, a list structure containing cycles (closed loops within the edges) may cause a marker to permanently loop. To overcome this, some intelligence may be given to the markers. If, while chaining down through the successors, the marker visits an excessive number (e.g. more than the maximum height of the structure or more than the total quantity of nodes) of nodes without reaching a terminal (or a yellow or black node), then it may assume that a loop exists and arbitrarily colour the current node yellow and add it to the subtree list. In this way, a terminating condition is placed within the loop. Loops will therefore reduce the efficiency of the algorithm due to wastage in searching the loops.

8.5. Comparison of Marking Algorithms

Empirical testing of the algorithms was carried out using a simulated multiprocessing system. The algorithms were tested and compared with a number of types of list structure. Four types of structure were chosen to exercise the algorithms under a variety of conditions. These types were:-

a) Linear List

b) Curtain

This structure consists of many linear lists emanating from a single root

c) Highly Interconnected

In this structure, each node has many branches with a large number of nodes shared between subtrees. Two versions of each structure were generated, the second being the mirror image of the first, that is the subtrees that were placed left to right from a node in one version were placed right to left in the other.

d) Random

The interconnection was generated randomly.

Each of the first three structures were used with both a high and a low proportion of the node space consisting of reachable nodes. All structures were loop free. Lamport's algorithm was performed twice, once with the markers searching from low addresses to high addresses

and secondly from high addresses to low. Table 8.5.1. shows some of the results obtained from the simulation studies when the node space consisted of 100 nodes.

From the Table it can be seen that, with one exception, the Chaining Algorithm out performs Lamport's algorithm on each of the values tabulated. In most cases, the number of nodes visited is vastly reduced (often by a factor of 50 or more). Also the costs of synchronising the markers is reduced. The overall improvement obtained from the Chaining Algorithm can be observed from the elapsed times given in the Table.

The structure with which the Chaining Algorithm performs least well is one with high interconnectivity. Yet even with this structure, the synchronisation overheads are minimal. This is of great advantage since a synchronisation will (in general) be much more expensive than a node visit.

The first highly-interconnected structure provides a pathalogic case for the Chaining Algorithm. In order for the blackening of the nodes from the terminal nodes towards the subroots to take place, the sublists need to be traversed many times. This is partly due to the high interconnection which will yield a high degree of overlapping subtrees and partly due to the greater number of successors which each node has.

As is known for programs designed for uni-processor systems,

MARKING ALGORITHM

COMPARISON TABLE

Type	G. N.	M	Chaining Algorithm: Node Vstd	Chaining Algorithm: W. P.	Chaining Algorithm: Time	Lamport Up: Node Vstd	Lamport Up: Syn	Lamport Up: Time	Lamport Down: Node Vstd	Lamport Down: Syn	Lamport Down: Time
Linear List Dense	0	1	100	0	0:26	5150	100	8:56	5150	100	9:00
	0	5	100	24	1:19	5710	500	9:26	5730	500	9:28
Linear List Sparse	95	1	5	0	0:02	305	5	0:32	400	5	0:42
	95	5	5	24	0:06	125	25	0:13	600	25	1:01
Curtain Dense	0	1	103	0	0:28	5150	100	9:11	5150	100	9:14
	0	5	103	27	0:33	2251	200	3:58	3370	350	5:46
Curtain Sparse	84	1	19	0	0:06	908	16	1:37	908	16	1:38
	84	5	19	27	0:09	959	80	1:40	987	80	1:42

Table 8.5.1 Simulation Results for Marking Algorithms.

High	1	1	2310	0	6:03	5148	99	9:02	5051	99	8:57
Inter-			1086	0	3:05						
Connect											
Dense	1	5	4335	42	12:46	4110	400	6:52	3875	400	6:31
			1297	45	4:17						
High	51	1	55	0	0:16	2717	49	4:50	2432	49	4:44
Inter-			250	0	0:54						
Connect											
Sparse	51	5	90	38	0:34	2346	215	4:00	2704	245	4:38
			105	27	0:36						
Random	40	1	186	0	0:44	4060	60	7:07	2200	60	3:56
One											
	40	5	189	42	0:55	1465	170	2:32	2612	255	4:28
Random	79	1	84	0	0:18	1893	21	3:19	428	21	0:47
Two											
	79	5	84	24	0:52	1227	100	2:04	1201	100	2:02
Random	75	1	82	0	0:17	1943	25	3:24	782	25	1:24
Three											
	75	5	82	24	0:52	1219	105	2:04	1380	125	2:20

Table 8.5.1 continued.

NOTES

1). Two sets of results are given for the High Inter-Connectivity with the Chaining Algorithm, these being for a structure and its symetrical pattern.

2). Lamport's algorithm is performed twice with the markers searching from low addresses to high addresses (Up) and searching from high addresses to low addresses (Down).

KEY

Type	The formation of the list structure.
G.N.	The number of garbage nodes in the structure.
M	The number of markers employed.
Node Vstd	The number of nodes visited during the marking phase.
W.P.	The number of time steps during which a marker was waiting on the 'listfront' semaphore.
Time	The elapsed time (in minutes and seconds) for the simulation of the marking phase.
Syn	The number of times, in total, that the markers were restarted at the beginning of their subspace.

Table 8.5.1 continued.

pathalogical data can greatly increase the processing time. Similar problems may also arise in programs designed for multi-processor systems. This is evidenced by the three-fold improvement in the performance of the Chaining Algorithm for the High-inter-connectivity Structure when the mirror image of the structure was used.

It has been noted, and indeed Lamport himself states, that synchronisations are costly operations. By considering the problem above in the light of the potential synchronisation, it has been reduced to a small level in the Chaining Algorithm. Lamport, however, by adapting an often used uniprocessor solution has maintained a potentially high level of synchronisation, and its inherent cost.

CHAPTER 9

CONCLUSIONS

9.1. Summary

Multiprocessor computer systems may provide many benefits over similar uniprocessor systems. However, it is possible to use a multiprocessor in an unsuitable application or to use one inappropriately in an application which may take advantage of a multiprocessor organisation. Indeed, such pitfalls exist for conventional uniprocessor systems. For a multiprocessor system to be utilised to advantage, consideration should be given to all aspects of the system, that is the hardware, the operating system software and the application software.

At the hardware level, many organisations of the processors and memory exist, ranging from array processors to multipart memory systems. Each of the many possible organisations has certain operational characteristics which make it most suitable for a particular class of problem. If an application from another class is implemented on that organisation, poor performance may be obtained from the system.

A simple model of a multiprocessor system was introduced (Chapter 4). The parameters of the model allow the processor and memory characteristics and the memory access pattern to be specified. The model was then analysed, with reference to the memory access pattern, and formulae were derived and an upper bound was placed upon the performance which could be expected from the modelled system. It was also shown that, for any particular access pattern, there is a

practical limit to the number of processors that should be attached to the shared memory if each is to accomplish useful work. A formula giving that limit was also derived (4.6.1.).

Whilst an application is executing on a multiprocessor, coordination will be required between the parallel paths as they are being executed. In Chapter 5, a tool, the Abstract Resource Ring (ARR), whereby the paths may synchronise, was described. The ARR is based on a 'Resource Master'technique. Comparisons were made between the ARR and two algorithms found in the literature. It was shown that, as the load placed upon the synchronisation method increased so the performance of the ARR increased whereas that of the other solutions deteriorated.

The ability for a multiprocessor system to withstand the 'death' of one of the processors within the system was discussed, with particular reference to the Abstract Resource Ring. It was shown that the ARR could be adapted to detect the failure of one of the processors and cause appropriate recovery action to be taken. This recovery action may include reconfiguration of the system as viewed by the supervisory software.

The Abstract Resource Ring has been used as the synchronising tool within a parallel processing system at Loughborough University. Figures may be found (Table 7.6.1) giving the performance of the system for a number of test programs. Comparison was made between two implementations of the ARR and one of the synchronisation tools described in the literature. The parallel processing system has also

provided a testbed for the reliability aspects of the ARR as discussed in Chapter 6.

Finally, to highlight the difficulty in designing multiprocessor applications, an example found in the literature was considered. A new solution to the problem of multiprocessor garbage collection was developed. This solution takes advantage of the inherent structure of the problem, and, in most circumstances, shows improvement in performance over the published algorithm, as is shown in Table 8.5.1.

9.2. Areas for Further Research

Within this thesis, a number of topics within the subject of multiprocessor systems have been considered. However, as stated earlier, the subject is vast with many areas where worthwhile research may be carried out. In the following subsections, areas are suggested where the research reported in this thesis may be extended.

a) Hardware Model Evaluation

It was claimed that the model presented in Chapter 4 applies to a large range of multiprocessor organisations. However, due to the lack of available hardware, this hypothesis has not been extensively tested. As more multiprocessor systems become available, further tests could be performed. Indeed, with the cheapness of microprocessor technology, it may be feasible to build small systems to test the hypothesis.

Also, two classes of memory were considered, private and shared. The relationship between the sizes of private and shared memory and their function (whether to store code or variables etc.) could be investigated, possibly with reference to a particular algorithm. This may yield new understanding on the relationship between hardware and application program.

b) Abstract Resource Ring

It was shown that the Abstract Resource Ring had the desirable effect that under high load conditions the overheads associated with its use

were reduced. However, under low load conditions its performance deteriorated such that one of the published solutions became a more viable tool to be used for synchronisation. It would be of advantage if the ARR could be modified so that its performance under low load improved. This would provide a synchronisation tool suitable for all contexts.

c) Algorithm Structure

The example of multiprocessor garbage collection, considered in Chapter 8, shows that the relationship between an application and its implementation on a multiprocessor system is not fully understood. This is one area which may be fundamental to all multiprocessor operation. If any automatic parallelisation is to be achieved with any success, more understanding of the underlying structure of a problem and the consequential interactions and synchronisations between the parts is required.

REFERENCES

1. Arnold, S.J. et al. "Design of Tightly-Coupled Multiprocessing Programming", IBM System Journal, Vol. 13 No. 1 (1974) pp.60-87.

2. Aspinall, D. and Dagless, E.L. "Overview of a Development Environment", Microprocessors and Microsystems, Vol. 3 No. 7 (1979) pp. 301-305.

3. Barlow, R.H. "Parallel Algorithms for Sorting, Quadrature and Eigenvalue Determination", Report No. 44, Dept. of Computer Studies, Loughborough University (1977).

4. Barlow, R.H. et al. "Historical Survey of the Implementation of Parallel Programming on the Interdata Dual Processor Computer", Report No. 40, Dept. of Computer Studies, Loughborough University (1977).

5. Barlow, R.H. et al. "Implementing Parallel Processing on a Production Minicomputer System", Report No. 58, Dept. of Computer Studies, Loughborough University (1977).

6. Baskett, F. and Smith, A.J. "Interference in Multiprocessor Computer Systems with Interleaved Memory", CACM, Vol. 19 No. 6 (1976) pp. 327 - 334.

7. Bhandakar, D.P. "Analysis of Memory Interference in Multiprocessors", IEEE Trans. on Computing, Vol. C-24 No. 9 (1975) pp. 897 - 908.

8. Bhandakar, D.P. and Fuller, S.H. "A Survey of Techniques for Analyzing Memory Interference in Multi-Processor Systems", Technical Report, Carnegie-Mellon University, Pittsburgh (1973).

9. Brinch Hansen, P. "A Comparison of Two Synchronizing Concepts", Acta Informatica, Vol. 1 (1972) pp. 190-199.

10. Brinch Hansen, P. "Concurrent Programming Concepts", ACM Computing Surveys, Vol. 5 No. 4 (1973) pp. 223 - 245.

11. Brinch Hansen, P. "Operating System Principles", Prentice-Hall Inc., Englewood Cliffs, New Jersey (1973).

12. de Bruijn, N.G. "Additional Comments on a Problem in Concurrent Programming Control", CACM, Vol. 10 No. 3 (1967) pp. 137 - 138.

13. Burnett, G.J. "Performance Analysis of Interleaved Memory Systems", PhD Thesis, Princeton University, Princeton, New Jersey (1970).

14. Burnett, G.J. and Coffman, E.G. "Analysis of Interleaved Memory Systems Using Blockage Buffers", CACM Vol. 18 No. 2 (1975) pp. 91 - 95.

15. Casey, D.P. and Wasserman, R.S. "Alternate CPU Recovery", IBM Technical Disclosure Bulletin, Vol. 16 No. 6 (1973) pp. 2005 - 2010.

16. Coffman, E.G. et al. "System Deadlocks", ACM Computing Surveys, Vol. 3 No. 2 (1971) pp. 67 - 78.

17. Courtois, P.J. et al. "Concurrent Control with 'Readers' and 'Writers' ", CACM, Vol. 14 No. 10 (1971) pp. 667 - 668.

18. Courtois, P.J. et al. "Comments on 'A Comparison of Two Synchronising Concepts' ", Acta Informatica, Vol. 1 (1972) pp. 375 - 376.

19. Conway, M.E. "A Multiprocessor System Design", AFIPS Conference Proceedings, Vol. 24 FJCC (1963) pp. 139 - 146.

20. Dahl, O-J et al. "Structured Programming", pub. Academic Press, New York (1972).

21. Dijkstra, E.W. "Solution of a Problem in Concurrent Programming Control", CACM, Vol. 8 No. 9 (1965) p. 569.

22. Dijkstra, E.W. "Cooperating Sequential Processes", Technological University, Eindhoven, The Netherlands (1965), reprinted in "Programming Languages", Genuys, F.(Ed), Academic Press, New York (1968).

23. Dijkstra, E.W. "Self-Stabalising Systems in Spite of Distributed Control", CACM, Vol. 17 No. 11 (1974) pp. 643 - 644.

24. Dijkstra, E.W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs", CACM, Vol. 18 No. 8 (1975) pp. 453 - 457.

25. Dijkstra, E.W. et al. "On-the-fly Garbage Collection: an Exercise in Cooperation", to be published.

26. Dowsing, R. "Software for CYBA-M", Microprocessors and Microsystems, Vol. 3 No. 7 (1979) pp. 306 - 310.

27. Eisenberg, M.A. and McGuire, M.R. "Further Comments on Dijkstra's Concurrent Programming Control Problem", CACM, Vol. 15 No. 11 (1972) p. 999.

28. Enslow, P.H. "Multiprocessors and Other Parallel Systems: An Overview and Introduction", Multiprocessor Systems, Infotech State of the Art Report No. 29, Infotech International Ltd., Maidenhead (1976) pp. 219 - 262.

29. Enslow, P.H. "Multiprocessor Organisation - A Survey", ACM Computing Surveys, Vol. 9 No. 1 (1977) pp. 103 - 129.

30. Evans, D.J. and Barlow, R.H. "An Analysis of the Performance of a Dual Minicomputer Parallel Computer System", Report No. 59, Dept. of Computer Studies, Loughborough University (1978).

31. Flynn, M.J. "Very High Speed Computing Systems", Proceedings of the IEEE, Vol. 54 No. 12 (1966) pp. 1901 - 1909.

32. Gilchrist, B. (Ed). "A Multi-microprocessor - CYBA-M", Information Processing 77, IFIP, North Holland Pub. Co. (1977).

33. Habermann, A.N. "Prevention of Systems Deadlocks", CACM, Vol. 12 No. 7 (1969) pp. 373 - 377; 385.

34. Halsall, F. and Fenesan, A.E. "Software Aspects of a Closely Coupled Multicomputer System", Computer and Digital Techniques, Vol. 1 No. 1 (1978) pp. 21 - 26.

35. Heart, F.E. et al. "The PLURIBUS Multiprocessor System", Multiprocessor Systems, Infotech State of the Art Report No. 29, Infotech International Ltd., Maidenhead (1976) pp. 307 - 330.

36. Hoare, C.A.R. "Minotors: An Operating System Structuring Concept", CACM, Vol. 17 No. 10 (1974) pp. 549 - 557.

37. Hoare, C.A.R. "Communicating Sequential Processes", CACM, Vol. 21 No. 8 (1978) pp. 666 - 677.

38. Hoare, C.A.R. and Perroth, R.H. (Eds.) "Towards a Theory of Parallel Programming" in "Operating Systems Techniques", Academic Press, New York (1973).

39. Horning, J.J. et al. "A Program Structure for Error Detection and Recovery", Proceedings of the Conference on "Operating Systems: Theoretical and Practical Aspects", IRIA (1974) pp. 177 - 193.

40. Howard, J.H. "Mixed Solutions for the Deadlock Problem", CACM Vol. 16 No. 7 (1973) pp. 427 - 430.

41. Interdata Inc., "Disc Operating System (DOS) Reference Manual", publication number 29 - 293, Interdata Inc., Oceanport, New Jersey (1974).

42. Interdata Inc., "Model 50/55 Communications Processor Reference Manual", publication number 29 - 249, Interdata Inc., Oceanport, New Jersey (1972).

43. Interdata Inc., "Universal Logic Interface Instruction Manual", product code M48 - 013, Interdata Inc., Oceanport, New Jersey (1975).

44. Jackson, M.A. "Principles of Program Design", Academic Press, New York (1975).

45. Knuth, D.E. "Additional Comments on a Problem in Concurrent Programming Control", CACM, Vol. 9 No. 5 (1966) pp. 321 - 322.

46. Kober, R. et al. "SMS 101 - A Structured Multimicroprocessor System with Deadlock-Free Operation Scheme", Euromicro Newsletter, Vol. 2 No. 2 (1976) pp. 56 - 64.

47. Kurtzburg, J.M. "On the Memory Conflict Problem in Multiprocessor Systems", IEEE Trans. on Computing, Vol. C-23 No. 3 (1974) pp. 286 - 293.

48. Lamport, L. "A New Solution of Dijkstra's Concurrent Programming Problem", CACM, Vol. 17 No. 8 (1974) pp. 453 - 455.

49. Lamport, L. "Garbage Collection with Multiple Processes: An Exercise in Parallelism", Proceedings of the International Conference on "Parallel Processing", Walden Woods (1976) pp. 50 - 54.

50. Lehman, M. "A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors", Proceedings of the IEEE, Vol. 54 No. 12 (1966) pp. 1889 - 1901.

51. MacKinnon, R.A. "Advanced Function Extended with Tightly-Coupled Multiprocessing", IBM System Journal, Vol. 13 No. 1 (1974) pp. 32 - 59.

52. McCarthy, J. "Recursive Functions of Symbolic Expressions and their Computation by Machine", CACM, Vol. 3 No. 4 (1960) pp. 184 - 195.

53. McCarthy, J. et al. "LISP 1.5 Programmer's Manual", MIT Press, Cambridge, Mass (1962).

54. Randell, B. "Research on Computing System Reliability at the University of Newcastle Upon Tyne 1972/73", Technical Report No. 57, Computing Laboratory, University of Newcastle (1974).

55. Randell, B. "System Structure for Software Fault Tolerance", Proceedings of the International Conference on "Reliable Software", Los Angeles (1975) pp. 437 - 449.

56. Sastry, K.V. and Kain, R.Y. "On the Performance of Certain Multiprocessor Computer Organisations", IEEE Trans on Computing, Vol. C-24 No. 11 (1975) pp. 1066 - 1074.

57. Science Research Council. "Distributed Computing Systems Annual Report Sept. 78 - Sept. 79" (1979).

58. Steele, G.L. "Multiprocessor Compactifying Garbage Collection", CACM, Vol. 18 No. 9 (1975) pp. 495 - 508.

59. Strecker, W.D. "An Analysis of the Instruction Execution Rate in Certain Computing Structures", PhD Thesis, Carnegie - Mellon University ARPA Report (1971).

60. Swan, R.J. et al. "Cm* - A Modular, multi-microprocessor", AFIPS Conference Proceedings, Vol. 46 NCC (1977) pp. 637 - 644.

61. Swan, R.J. et al. "The Implementation of the CM* Multi-microprocessor", AFIPS Conference Proceedings, Vol. 46 NCC (1977) pp. 645 - 655.

62. Tandem Computers Inc. "TANDEM Non-Stop Systems", Sales Literature, Tandem Computers Inc., California (1978).

63. Ward, quoted in "Introduction", Multiprocessor Systems, Infotech State of the Art Report No. 29, Infotech International Ltd., Maidenhead (1976) p. 18.

64. Williams, S.A. "Approaches to the Determination of Parallelism in Computer Programs", PhD Thesis, Loughborough University (1978).

65. Wirth, N. "On Multiprogramming, Machine Coding and Computer Organisation", CACM, Vol. 12 No. 9 (1969) pp. 489 - 498.

66. Wulf, W.A. "Hydra: The Kernel of a Multiprocessor Operating System", CACM, Vol. 17 No. 6 (1974) pp. 337 - 345.

67. Wulf, W.A. and Bell, C.G. "C.mmp - A multi-mini-processor", AFIPS Conference Proceedings, Vol. 41 Part 2 FJCC (1972) pp. 765 - 777.

68. Yau, S.S. and Cheung, R.C. "Design of Self Checking Software", Proceedings of the International Conference on "Reliable Software", Los Angeles (1975) pp. 450 - 457.

69. Proceedings of "Computer Networks", an Advanced Course, Dublin (1977) to be published.

APPENDIX 1

AN IMPLEMENTATION OF
THE ABSTRACT RESOURCE RING

This Appendix consists of a listing of an implementation of the Abstract Resource Ring. The implementation, which is based upon the periodic restart, is written in Algol 68R.

```
 1
 2    'C'
 3         VARIABLES AND CONSTANTS
 4    'C'
 5
 6        'INT'N,                          'C'  NO OF PROCESSORS                        'C'
 7             P,                          'C'  NO OF RESOURCES                         'C'
 8             PROCNO,                     'C'  LOCAL PROCESSOR NUMBER                  'C'
 9             NO OF SYS RES;              'C'  NO OF PERMANANT (SYSTEM) RESOURCES      'C'
10        'INT'CREATE=1;                   'C'  "CREATE" RESOURCE NUMBER                'C'
11
12        [1:P,0:N,1:2]'INT'RING;      'C'  THE RING STRUCTURE                          'C'
13        'MODE''RESOURCEDEFN'='STRUCT'('INT'NODE,'STRING'NAME);
14        [1:P]'RESOURCEDEFN'RESOURCES;  'C'  "RESOURCES" TABLE                         'C'
15        [1:NO OF SYS RES]'REF''STRING'SYSNAME;  'C'  NAMES OF SYSTEM RESOURCES        'C'
16        [1:N,1:2]'INT'LOCAL TIME;    'C'  TIMING ARRAY FOR DEATH CHECKING             'C'
17
18    'C'
19         PROCEDURES
20    'C'
21
22        'PROC'WAIT='VOID':
23        'BEGIN'
24    'C'
25         A PROCEDURE WHICH CAUSES THE CALLING ACTIVITY TO BE SUSPENDED
26         FOR A MINIMUM TIME
27    'C'
28             'NIL'
29        'END';
30
```

```
31          'PROC'FAULT=([]'CHAR'MESS):
32          'BEGIN'
33    'C'
34           A PROCEDURE WHICH CAUSES THE CALLING ACTIVITY TO FAIL FOR THE
35           GIVEN REASON
36    'C'
37               'NIL'
38          'END';
39
40          'PROC'TELL OPERATORS=([]'CHAR'X,'INT'Y,[]'CHAR'Z):
41          'BEGIN'
42    'C'
43           A PROCEDURE WHICH CAUSES A MESSAGE TO BE PRINTED ON THE OPERATOR
44           CONSOLE
45    'C'
46               'NIL'
47          'END';
48
49          'PROC'INITIATE=('REF''PROC'('INT')P,'INT'X):
50          'BEGIN'
51    'C'
52           A PROCEDURE TO CREATE A NEW ACTIVITY TO BE ADDED TO THE
53           SYSTEM SCHEDULE
54    'C'
55               'NIL'
56          'END';
57
58          'PROC'ALREADYDOING=('REF''PROC'('INT')P,'INT'X)'BOOL':
59          'BEGIN'
60               'BOOL'ANS;
```

```
61    'C'
62         A PROCEDURE WHICH RETURNS TRUE IF THE ACTIVITY SPECIFIED BY
63         THE PARAMETERS IS ON THE SYSTEM SCHEDULE, ELSE FALSE IS RETURNED
64    'C'
65              ANS
66        'END';
67
68        'PROC'KILL=('REF''PROC'('INT')P,'INT'TX);
69        'BEGIN'
70    'C'
71         A PROCEDURE WHICH REMOVES THE ACTIVITY SPECIFIED BY THE
72         PARAMETERS TO BE REMOVED FROM THE SYSTEM SCHEDULE
73    'C'
74              'NIL'
75        'END';
76
77        'PROC'REINSTATE RESOURCE=('INT'RESOURCE NO);
78        'BEGIN'
79    'C'
80         A PROCEDURE WHICH RECOVERS THE SPECIFIED RESOURCE TO A
81         CORRECT AND SELF CONSISTANT STATE
82    'C'
83              'NIL'
84        'END';
```

```
 1
 2    'C'
 3         NEXT PROC NO  ROUTINE
 4    'C'
 5
 6        'PROC'NEXT PROC NO=('INT'I)'INT':
 7        'BEGIN'
 8            'IF'I>=N'THEN'1'ELSE'I+1'FI'
 9
10    'C'
11         DELIVER NEXT PROCESSOR NUMBER IN CYCLIC ORDER
12    'C'
13
14        'END';
```

```
 1
 2   'C'
 3        PUTRES ACTIVITY
 4   'C'
 5
 6        'PROC'('INT')PUTRES ACTIVITY:=('INT'RESOURCE NO);
 7        'BEGIN'
 8             'BOOL'EXIT:='FALSE';
 9             'WHILE''NOT'EXIT'AND'RING[RESOURCE NO,PROCNO,1]=0'DO'
10             'BEGIN'
11                 'INT'J:=PROCNO;
12                 'WHILE'J:=RING[RESOURCE NO,J,2];'NOT'EXIT'AND'J#PROCNO'DO'
13                      'IF'RING[RESOURCE NO,J,1]=1'THEN'
14
15   'C'
16        SOMEONE WANTS IT
17        ***   GIVE IT IN A SAFE UPDATE   ***
18   'C'
19
20                           RING[RESOURCE NO,0,1]:=J;
21
22   'C'
23        ***   END OF SAFE UPDATE   ***
24   'C'
25
26                           EXIT:='TRUE'
27                      'FI';
28                 'IF''NOT'EXIT'THEN'WAIT'FI'
29
30   'C'
```

```
31          WAIT FOR A SPELL THEN TRY AGAIN; UNLESS GOT RID OF IT
32    'C'
33
34                'END'
35          'END';
```

```
 1
 2   'C'
 3        GETRES PROCEDURE
 4   'C'
 5
 6       'PROC'GETRES=('INT'RESOURCE NO);
 7       'BEGIN'
 8
 9   'C'
10        CHECK VALID RESOURCE NO
11   'C'
12
13             'IF'RESOURCE NO>P'OR'RESOURCE NO<1'THEN'
14                  FAULT("INVALID RESOURCE NUMBER")
15             'FI';
16             'IF'RING[RESOURCE NO,PROCNO,2]#=1'THEN'
17
18   'C'
19        IF ON THE RING
20   'C'
21
22                  RING[RESOURCE NO,PROCNO,1]:=1;
23
24   'C'
25        SET WANT FLAG
26   'C'
27
28                  'IF'ALREADY DOING(PUTRES ACTIVITY,RESOURCE NO)'THEN'
29                       KILL(PUTRES ACTIVITY,RESOURCE NO)
30                  'FI';
```

```
31
32    'C'
33         KILL OFF PUTRES ACTIVITY IF GOING - WEVE TAKEN IT BACK
34    'C'
35
36                  'WHILE'RING[RESOURCE NO,0,1]#PROCNO'DO'
37                        WAIT
38               'ELSE'
39                  FAULT("CANNOT GET RESOURCE AS NOT ON RING")
40               'FI'
41         'END';
```

```
 1
 2    'C'
            PUTRES ROUTINE
 4    'C'
 5
 6          'PROC'PUTRES=('INT'RESOURCE NO);
 7          'BEGIN'
 8
 9    'C'
10          CHECK VALID RESOURCE NO
11    'C'
12
13                'IF'RESOURCE NO>P'OR'RESOURCE NO<1'THEN'
14                      FAULT("INVALID RESOURCE NUMBER")
15                'FI';
16                'IF'RING[RESOURCE NO,PROCNO,2]#=1'THEN'
17
18    'C'
19          IF ON THE RING
20    'C'
21
22                      RING[RESOURCE NO,PROCNO,1]:=0;
23
24    'C'
25          CLEAR OUR WANT FLAG
26    'C'
27
28                      'IF'RING[RESOURCE NO,0,1]=PROCNO'THEN'
29
30    'C'
```

```
31           IF WE OWN AND OTHERS ON THE RING
32     'C'
33
34                       'BEGIN'
35                            'INT'J:=PROCNO;
36                            'BOOL'DONE:='FALSE';
37                            'WHILE'J:=RING[RESOURCE NO,J,2];J#PROCNO'AND'
38                                       'NOT'DONE'DO'
39                                 'IF'RING[RESOURCE NO,J,1]=1'THEN'
40
41     'C'
42           SOME ONE WANTS IT
43           ***   GIVE IT TO THEM WITH A SAFE UPDATE   ***
44     'C'
45
46                                       RING[RESOURCE NO,0,1]:=3;
47                                       DONE:='TRUE'
48
49     'C'
50           ***   END OF SAFE UPDATE   ***
51     'C'
52
53                                 'FI';
54                            'IF''NOT'DONE'THEN'
55                                 INITIATE(PUTRES ACTIVITY,RESOURCE NO)
56                            'FI'
57                       'END'
58                 'FI'
59              'FI'
60           'END';
```

```
 1
 2   'C'
 3         DEALLOCATE ACTIVITY
 4   'C'
 5
 6        'PROC'('INT')DEALL ACTIVITY:=('INT'RESOURCE NO);
 7        'BEGIN'
 8
 9   'C'
10         WAIT FOR SOME ONE TO REQUEST THE RESOURCE,
11         THEN GIVE IT UP AND DELETE US
12   'C'
13
14              'BOOL'EXIT:='FALSE';
15              'WHILE''NOT'EXIT'DO'
16              'BEGIN'
17                   GETRES(CREATE);
18                   'IF'RING[RESOURCE NO,PROCNO,2]=PROCNO'THEN'
19
20   'C'
21         JUST US LEFT ON THE RING - REMOVE FROM RESOURCES
22         ***   START OF SAFE UPDATE   ***
23   'C'
24
25                        NODE'OF'RESOURCES[RESOURCE NO]:=-1;
26                        NAME'OF'RESOURCES[RESOURCE NO]:="";
27                        RING[RESOURCE NO,PROCNO,2]:=-1;
28
29   'C'
30         ***   END OF SAFE UPDATE   ***
```

```
31    'C'
32
33                      EXIT:='TRUE';
34                      PUTRES(CREATE)
35                 'ELSE'
36                 'BEGIN'
37                      'INT'J;
38                      J:=PROCNO;
39                      'WHILE'J:=RING[RESOURCE NO,J,2];J#PROCNO'AND''NOT'EXIT'DO'
40
41    'C'
42         SEARCH ROUND THE RING FOR SOMEONE WHO WANTS THE RESOURCE
43    'C'
44
45                           'IF'RING[RESOURCE NO,J,1]#0'THEN'
46
47    'C'
48         PROCESSOR J WANTS THE RESOURCE
49    'C'
50
51                           'BEGIN'
52                                'INT'K;
53                                K:=PROCNO;
54                                'WHILE'RING[RESOURCE NO,K,2]#PROCNO'DO'
55                                     K:=RING[RESOURCE NO,K,2];
56
57    'C'
58         K IS OUR PREDECESSOR
59         REMOVE US AND GIVE THE RESOURCE AWAY
60         ***   SAFE UPDATE COMING   ***
```

```
61    'C'
62
63                                       RING[RESOURCE NO,PROCNO,1]:=0;
64                                       RING[RESOURCE NO,0,1]:=J;
65                                       NODE'OF'RESOURCES[RESOURCE NO]:=K;
66                                       RING[RESOURCE NO,K,2]:=RING[RESOURCE NO,PROCNO,2];
67                                       RING[RESOURCE NO,PROCNO,2]:=-1;
68
69    'C'
70          ***   END OF SAFE UPDATE   ***
71    'C'
72
73                                       EXIT:='TRUE';
74                                       PUTRES(CREATE)
75                                   'END'
76                                   'FI'
77                           'END'
78                           'FI';
79                           'IF''NOT'EXIT'THEN'
80                               PUTRES(CREATE);
81                               WAIT
82                           'FI'
83                       'END'
84                   'END';
```

```
 1
 2   'C'
 3        ALLOCATE TRANSIANT RESOURCE ROUTINE
 4   'C'
 5
 6       'PROC'ALLOCATE=('STRING'RESOURCE NAME)'INT':
 7       'BEGIN'
 8
 9   'C'
10        CREATE A NEW TEMP RESOURCE WITH THE GIVEN NAME & RETURN RESOURCE NUMBER
11   'C'
12
13             'INT'HOLE:=-1,J:=1;
14             'BOOL'EXIT:='FALSE';
15             GETRES(CREATE);
16             'WHILE'J<=P'AND''NOT'EXIT'DO'
17             'BEGIN'
18                  'IF'HOLE=-1'AND'NODE'OF'RESOURCES[J]=-1'THEN'
19                       HOLE:=J
20                  'FI';
21                  'IF'NAME'OF'RESOURCES[J]=RESOURCE NAME'THEN'
22                       EXIT:='TRUE'
23                  'ELSE'
24                       J'PLUS'1
25                  'FI'
26             'END';
27
28   'C'
29        IF J<=P THE RESOURCE ALREADY EXISTS ELSE HOLE IS FIRST FREE SLOT
30   'C'
```

```
31
32              'IF'J<=P'THEN'
33                  'IF'ALREADY DOING(DEALL ACTIVITY,J)'THEN'
34                      KILL(DEALL ACTIVITY,J)
35                  'ELSF'RING[J,PROCNO,2]=-1'THEN'
36
37      'C'
38          ADD US IF WE ARE NOT ON THE RING
39          ***   SAFE UPDATE   ***
40      'C'
41
42                      RING[J,PROCNO,1]:=0;
43                      RING[J,PROCNO,2]:=RING[J,NODE'OF'RESOURCES[J],2];
44                      RING[J,NODE'OF'RESOURCES[J],2]:=PROCNO
45
46      'C'
47          ***   END OF SAFE UPDATE   ***
48      'C'
49
50                  'FI'
51              'ELSE'
52
53      'C'
54          NO RESOURCE - PUT IT AT HOLE
55      'C'
56
57                  J:=HOLE;
58
59      'C'
60          ***   SAFE UPDATE COMING   ***
```

```
61    'C'
62
63                      NAME'OF'RESOURCES[J]:=RESOURCE NAME;
64                      NODE'OF'RESOURCES[J]:=PROCNO;
65                      RING[J,PROCNO,2]:=PROCNO;
66                      RING[J,0,1]:=PROCNO;
67                      RING[J,PROCNO,1]:=0
68
69    'C'
70           ***   END OF SAFE UPDATE   ***
71    'C'
72
73               'FI';
74               PUTRES(CREATE);
75               PUTRES(J);
76
77    'C'
78           GET RID OF RESOURCE
79           RETURN RESOURCE NUMBER
80    'C'
81
82               J
83          'END';
```

```
 1
 2   'C'
 3        DEALLOCATE TEMPORARY RESOURCE ROUTINE
 4   'C'
 5
 6       'PROC'DEALLOCATE=('INT'RESOURCE NO);
 7       'BEGIN'
 8
 9   'C'
10        CHECK VALID RESOURCE NO
11   'C'
12
13            'IF'RESOURCE NO>P'OR'RESOURCE NO<1'THEN'
14                 FAULT("INVALID RESOURCE NUMBER")
15            'FI';
16
17   'C'
18        TRAP DEALLOCATION OF PERMANANT RESOURCES
19   'C'
20
21            'IF'RESOURCE NO<=NO OF SYS RES'THEN'
22                 FAULT("CANNOT DEALLOCATE PERMANANT RESOURCE")
23            'FI';
24            'IF'RING[RESOURCE NO,PROCNO;2]#-1'AND'
25                      'NOT'ALREADYDOING(DEALL ACTIVITY;RESOURCE NO)'THEN'
26
27   'C'
28        IF POINTER IS SET - IE ON THE RING
29   'C'
30
```

```
31              'IF'ALREADY DOING(PUTRES ACTIVITY,RESOURCE NO)'THEN'
32                  KILL(PUTRES ACTIVITY,RESOURCE NO)
33              'FI';
34              GETRES(RESOURCE NO);
35              GETRES(CREATE);
36              'IF'RING[RESOURCE NO,PROCNO,2]=PROCNO'THEN'
37
38   'C'
39       IF WE ARE ONLY PERSON ON THE RING
40       REMOVE THE RESOURCE FROM THE LIST
41       ***   START OF SAFE UPDATE   ***
42   'C'
43
44                  NODE'OF'RESOURCES[RESOURCE NO]:=-1;
45                  NAME'OF'RESOURCES[RESOURCE NO]:="";
46                  RING[RESOURCE NO,PROCNO,2]:=-1;
47
48   'C'
49       ***   END OF SAFE UPDATE   ***
50   'C'
51
52                  PUTRES(CREATE)
53              'ELSE'
54                  PUTRES(CREATE);
55                  INITIATE(DEALL ACTIVITY,RESOURCE NO)
56
57   'C'
58       START THE DEALLOCATE ACTIVITY IF OTHERS ON THE RING
59   'C'
60
```

```
61                    'FI'
62               'FI'
63       'END';
```

```
 1
 2   'C'
 3         RECOVERY PROCEDURE
 4   'C'
 5
 6        'PROC'('INT')RECOVERY:=('INT'DEADPROC);
 7        'BEGIN'
 8
 9   'C'
10         INITIATED WHEN A DEAD PROCESSOR IS FOUND
11         THE SINGLE PARAMETER IS THE NAME OF THE DEAD PROCESSOR
12   'C'
13
14              'FOR'I'TO'P'DO'
15                   'IF'RING[I,DFADPROC,2]#-1'THEN'
16                        'IF'RING[I,0,1]#DEADPROC'THEN'
17                              RING[I,DEADPROC,1]:=0
18                        'FI'
19                   'FI';
20
21   'C'
22         CLEAR ALL WANT FLAGS OF RESOURCES NOT OWNED
23   'C'
24
25              UAIT;
26
27   'C'
28         UAIT FOR ALL PUTRES TO THE DEADPROC TO FINISH
29   'C'
30
```

```
31              'FOR'I'TO'P'DO'
32              'BEGIN'
33                  'BOOL'FLAG:='FALSE',FLAG1:='FALSE';
34                  'IF'RING[I,DEADPROC,2]#-1'THEN'
35
36    'C'
37          FOR EACH RING HE IS ON
38    'C'
39
40                      'IF'RING[I,PROCNO,2]=-1'THEN'
41
42    'C'
43          ADD US TO THE RING IF NOT THER ALREADY - NOTE WE ALREADY OWN CREATE
44    'C'
45
46                          FLAG:='TRUE';
47
48    'C'
49          ***   START OF SAFE UPDATE   ***
50    'C'
51
52                          RING[I,PROCNO,1]:=0;
53                          RING[I,PROCNO,2]:=RING[I,NODE'OF'RESOURCES[I],2];
54                          RING[I,NODE'OF'RESOURCES[I],2]:=PROCNO;
55                          NODE'OF'RESOURCES[I]:=PROCNO
56
57    'C'
58          INCASE DEAD PROCESSOR WAS THERE
59          ***   END OF SAFE UPDATE   ***
60    'C'
```

```
61
62                    'ELSE'
63                         FLAG1;=RING[I;PROCNO,1]=1
64
65    'C'
66         REMEMBER IF WE WANT THE RESOURCE
67    'C'
68
69                    'FI';
70                    'IF'RING[I,0,1]=DEADPROC'THEN'
71
72    'C'
73         GRAB RESOURCE IF HE HAS IT
74         ***   USING A SAFE UPDATE   ***
75    'C'
76
77                         RING[I,PROCNO,#];=1;
78                         RING[I,0,1];=PROCNO;
79
80    'C'
81         ***   END OF SAFE UPDATE   ***
82    'C'
83
84                         'IF'RING[I,DEADPROC,1]=1'THEN'
85                              RING[I,DEADPROC,1];=0;
86                              REINSTATE RESOURCE(I)
87
88    'C'
89         RECOVER THE RESOURCE IF HE WAS USING IT
90    'C'
```

```
 91
 92                      'FI'
 93                  'ELSE'
 94                      PUTRES(CREATE);
 95                      GETRES(I);
 96                      GETRES(CREATE)
 97
 98    'C'
 99         GET THE RESOURCE PROPERLY
100    'C'
101
102                  'FI';
103
104    'C'
105         REMOVE HIM FROM THE RING
106    'C'
107
108                  'FOR'J'TO'N'DO'
109                      'IF'RING[I,J,2]=DEADPROC'THEN'
110
111    'C'
112         WE HAVE FOUND HIS PREDECESSOR
113         ***   START OF SAFE UPDATE   ***
114    'C'
115
116                          RING[I,J,2]:=RING[I,DEADPROC,2];
117                          RING[I,DEADPROC,2]:=-1
118
119    'C'
120         ***   END OF SAFE UPDATE   ***
```

```
121    'C'
122
123                               'FI';
124
125    'C'
126          REMOVE US IF WE WERE NOT ORIGIONALLY ON
127    'C'
128
129                         'IF'FLAG'THEN'
130                              DEALLOCATE(I)
131                         'ELSF''NOT'FLAG1'AND'I#CREATE'THEN'
132
133    'C'
134          GET RID OF THE RESOURCE IF WE DID NOT WANT IT
135    'C'
136
137                              PUTRES(I)
138                         'FI'
139                    'FI'
140               'END';
141               TELL OPERATORS("PROC ";DEADPROC;" IS DEAD");
142
143    'C'
144          SET AS RECOVERED IN LOCAL TIME
145    'C'
146
147               LOCAL TIME[DEADPROC,1]:=-1
148          'END';
```

```
 1
 2    'C'
 3         DEATH CHECKING ROUTINE
 4    'C'
 5
 6         'PROC''VOID'DEATH CHECK:='VOID':
 7         'BEGIN'
 8              'INT'I:=PROCNO;
 9
10    'C'
11         FOR EACH PROCESSOR FOLLOWING US TILL EITHER OURSELVES
12         OR A LIVE PROCESSOR IS MET
13    'C'
14
15              'WHILE'I:=NEXT PROC NO(I);I#PROCNO'AND'LOCAL TIME[PROCNO,1]>
16                        LOCAL TIME[I,1]+LOCAL TIME[I,2]'DO'
17
18    'C'
19         RECOVER HIM IF NOT ALREADY DOING SO AND NOT BEEN RECOVERED BEFORE
20    'C'
21
22                   'IF''NOT'ALREADY DOING(RECOVERY,I)'AND'LOCAL TIME[I,1]>=0'THEN'
23                        INITIATE(RECOVERY,I)
24                   'FI'
25         'END';
```

```
 1
 2    'C'
 3         SCHEDULAR STARTUP PROCEDURE
 4    'C'
 5
 6        'PROC'SCHED STARTUP='VOID';
 7        'BEGIN'
 8
 9    'C'
10         ALLOCATE ALL SYSTEM (FIXED) RESOURCES AND KILL OFF THE REST
11    'C'
12
13              'FOR'J'TO'NO OF SYS RES'DO'
14              'BEGIN'
15                   NODE'OF'RESOURCES[J]:=1;
16
17    'C'
18         GIVE SYSTEM RESOURCES TO PROCESSOR 1
19    'C'
20
21                   NAME'OF'RESOURCES[J]:=SYSNAME[J];
22                   RING[J,1,1]:=0;
23                   RING[J,0,1]:=1;
24
25    'C'
26         SET 1 AS OWNER
27    'C'
28
29                   INITIATE(PUTRES ACTIVITY;J);
30
```

```
31    'C'
32          INITIATE PUTRES ACTIVITY FOR PROCESSOR 1
33    'C'
34
35                      RING[J,1,2]:=2;
36                      'FOR'K'FROM'2'TO'N'DO'
37                      'BEGIN'
38
39    'C'
40          CHAIN ALL THE PROCESSORS TOGETHER
41    'C'
42
43                           RING[J,K,1]:=0;
44                           RING[J,K,2]:=NEXT PROC NO(K)
45                      'END'
46                 'END';
47                 'FOR'J'FROM'NO OF SYSRES+1'TO'P'DO'
48                 'BEGIN'
49
50    'C'
51          KILL OFF THE REST
52    'C'
53
54                      NODE'OF'RESOURCES[J]:=-1;
55                      NAME'OF'RESOURCES[J]:="";
56                      'FOR'K'TO'N'DO'
57                      'BEGIN'
58                           RING[J,K,1]:=0;
59                           RING[J,K,2]:=-1
60                      'END'
```

```
61              'END'
62          'END';
```

APPENDIX 2

AN IMPLEMENTATION OF THE RELIABLE UPDATE

This Appendix consists of a listing of an implementation of the Reliable Update algorithm discussed in Section 6.7. The implementation is written in Algol 68R.

```
 1
 2  'C'
 3
 4      SYSTEM PROCEDURE
 5
 6  'C'
 7
 8      'PROC'FAULT=('STRING'S):
 9      'BEGIN'
10
11  'C'
12
13      A PROCEDURE WHICH CAUSES THE CALLING PROCESS TO FAIL FOR THE GIVEN REASON
14
15  'C'
16
17          'SKIP'
18      'END';
19
20  'C'
21
22      BASIC MODE DEFINITIONS
23
24  'C'
25
26      'MODE''VALUE'='UNION'('INT','REAL','CHAR');
27      'MODE''BISTABLE'='INT';
28      'MODE''SAFEENTRY'='STRUCT'('VALUE'VALUE,NEWVALUE,'BISTABLE'BS);
29
30  'C'
```

```
31
32        SAFE UPDATE PROCEDURE
33
34    'C'
35
36        'PROC'SAFEUPDATE=([]'REF''SAFEENTRY'TABLE,[]'VALUE'NEWVALUES;
37                'REF''BISTABLE'FLAG);
38        'BEGIN'
39
40    'C'
41
42        THE ARRAY "TABLE" CONTAINS POINTERS TO THE ENTRIES TO BE CHANGED;
43        THE NEW VALUES TO BE INSERTED ARE GIVEN IN ARRAY "NEWVALUES";
44        AND THE FLAG IS GIVEN BY "FLAG"
45
46    'C'
47
48            'INT'I:='UPB'TABLE;
49            'IF'I#'UPB'NEWVALUES'THEN'
50                FAULT("BAD PARAMETERS")
51            'FI';
52
53    'C'
54
55        FAIL IF DIFFERENT NUMBER OF ENTRIES AND NEW VALUES
56
57        STEP A
58
59    'C'
60
```

```
61            'FOR'J'TO'I'DO'
62            'BEGIN'
63                NEWVALUE'OF'TABLE[J]:=NEWVALUES[J];
64                BS'OF'TABLE[J]:=1
65            'END';
66
67    'C'
68
69        STEP B
70
71    'C'
72
73            FLAG:=1;
74
75    'C'
76
77        STEP C
78
79    'C'
80
81            'FOR'J'TO'I'DO'
82            'BEGIN'
83                VALUE'OF'TABLE[J]:=NEWVALUE'OF'TABLE[J];
84                BS'OF'TABLE[J]:=0
85            'END';
86
87    'C'
88
89        STEP D
90
```

```
 91    'C'
 92
 93              FLAG:=0
 94
 95    'C'
 96
 97         UPDATE OVER
 98
 99    'C'
100
101         'END';
102
```