An investigation into Multiprocessor Systems based on UNIX

Welten, P.J.M.

Award date:
1989

Eindhoven University of Technology
Department of Electrical Engineering
Group Digital Systems

An Investigation into Multiprocessor Systems based on UNIX

Master Thesis
By P.J.M Welten

This is the final report on the graduation work done at the Eindhoven University of Technology, department of Electrical Engineering, Digital Systems Division

By:	P.J.M Welten
Supervisor:	ir. A.G.M Geurts
Supervising professor:	Prof. ir. M.P.J Stevens

Time period:	01-02-1988 / 17-02-1989
Report Date:	17-02-1989
Report Code:	

ABSTRACT

Multiprocessors become more and more important, both for technical as for other environments. The Unix^R operating system has also gained importance in recent years and gains still (Now entering the Consumer market!). This report describes an investigation into tightly coupled multiprocessors based on Unix and based on VME. An overview of Multiprocessor aspects is given. Restrictions to the field of investigation, such as only considering Unix and VME, are stated. The system architectures Master-Slaves, Uni-Kernel, Multi-Kernel with shared but without private memory and Multi-Kernel with shared plus private memory are discussed. Aspects that are discussed, although not always for every architecture mentioned, include: System Startup, Interrupts, Inter Processor Communication, Mutual Exclusion and Scheduling.

SAMENVATTING

Multiprocessoren worden steeds belangrijker, zowel voor technische als voor andere omgevingen. Het Unix^R besturingssysteem heeft de laatste jaren ook aan belangrijkheid gewonnen en doet dat nog steeds. Dit rapport beschrijft een onderzoek naar sterk gekoppelde multiprocessoren die Unix als besturingssysteem hebben en die op de VMEbus geïmplementeerd zijn. Er wordt een overzicht gegeven van de verschillende aspecten die bij multiprocessoren van belang zijn. Tevens wordt het onderzoeksterrein ingeperkt. Bijvoorbeeld omvat het onderzoeksterrein geen andere besturingssystemen dan Unix en is er ook niet naar andere processor-geheugen netwerken dan de VMEbus gekeken.
De volgende systeem architecturen zijn nader onderzocht: Master-Slaves, Uni-Kernel, Multi-Kernel met gemeenschappelijk maar zonder privaat geheugen en Multi-Kernel met gemeenschappelijk en privaat geheugen. De volgende aspecten zijn bekeken, hoewel niet altijd voor alle genoemde architecturen: Opstart van het systeem, Interrupts, Inter Processor Communicatie, Wederzijdse Uitsluiting and Proces Scheduling.

Key words: Multiprocessor, Unix, VME, parallel processing, scheduling, simulation, load balancing, process migration, common bus.

ACKNOWLEDGEMENTS

I would like to thank the following people for their support:

Fist of all I thank Ellen, my wife, for encouraging me during my study and for buying me time by doing more than her share (in spite of her own job) in taking care of the children and household,

ir. A.G.M Geurts for his day to day guidance,

Prof. ir. M.P.J Stevens for enabling me to work on the topic at all and for his overall guidance,

the staff of the Digital Systems Division for their effort to diminish my workload in other group activities,

A. Chamboné and W. Brant for making the drawings to this report,

ir. F. Butzelaar for letting me use his PLOTUTIL program, which considerably speeded up presentation and interpretation of simulation results,

and all others that contributed in any way, they were too many to be mentioned on a single A4 sheet.

Peter

CONTENTS

ABSTRACT i

ACKNOWLEDGEMENTS ii

CONTENTS iii

1. Introduction 1

2. Multiprocessoring Systems 3

2.1 Memory Map 6
2.2 Interconnections 7
2.3 Symmetry 9
2.4 Input/Output Structure 10
2.5 Interrupts 10
2.6 Inter Processor Communication 12
2.7 Reserved Storage Areas 14
2.8 The Operating System 14
2.9 User Facilities for Parallel Processing 16
2.10 Error Recovery 17
2.11 Performance 18

3 Target Multiprocessor 19

3.1 Aspects 19
3.1.1 Target Interconnections (VME) 20
3.1.1.1 Data Transfer Bus 22
3.1.1.2 Priority Interrupt Bus 25
3.1.1.3 DTB Arbitration Bus 25
3.1.1.4 Utility Bus 26
3.1.1.5 VMSbus 26
3.1.2 Memory Map 29
3.1.3 Interrupts 32
3.1.4 Inter Processor Communication 32
3.1.5 Operating System (Unix^R System V) 32
3.1.5.1 System Startup 33
3.1.5.2 System Call Interface 34
3.1.5.3 Mutual Exclusion 35
3.2 Configurations 35

4 Master-Slaves Architecture 37

4.1 Introduction 37
4.2 Mutual Exclusion. 37
4.3 Interrupts 39
4.3.1 IO Interrupts 41
4.3.2 Clock Interrupt 41
4.3.3 Summary 42
4.4 Inter Processor Communication 42
4.4.1 Explicit IPrC 44
4.4.1.1 Unix System V IPC-like Implementation 44
4.4.1.2 Private Mailboxes 51
4.4.1.3 VMSbus 52
4.4.1.4 P1296 specification-like Implementation 53
4.4.2 Implicit IPrC 54
4.4.3 Conclusions 56
4.5 System Startup 57

5 Uni-Kernel Architecture 61

5.1 Mutual Exclusion 61
5.1.1 Mode Broadcasting 62
5.1.2 Kernel Border 63
5.2 Interrupts 63
5.2.1 General 63
5.2.2 Clock Interrupt 65

6 Multi-Kernel SMO Architecture 69

6.1 Unix Critical Regions 69
6.1.1 Synchronization Mechanisms 70
6.1.1.1 Hardware Synchronization Primitives 70
6.1.1.2 Software Synchronization Primitives 76
6.1.1.3 Summary 77
6.1.2 Detectability of Critical Regions 77
6.2 Process Scheduling 80
6.2.1 What is Scheduling? 80
6.2.2 How to Schedule 81
6.2.3 Types of Policies 83
6.2.4 Optimal Response Time 84
6.2.5 Optimal Processor Utilization 85
6.2.5.1 Threshold Policies 88
6.2.5.2 Process Behavior based Policies 92
6.2.6 Congestion Problems with Scheduling 94
6.2.7 General Requirements for Scheduling 98
6.3 Scheduling in a Unix based Multiprocessor 99
6.3.1 Single Processor Unix Scheduling 100

6.3.2 Practical Requirements for Scheduling 103
6.3.3 Multiprocessor Scheduling 103
6.3.3.1 Simulation Model 105
6.3.3.2 Conclusions Simulation 114
6.3.3.3 Queue Manipulation Times 115
6.3.4 Summary 120

7 Multi-Kernel PSM Architecture 121

8 Proposed Multiprocessor 123

9 Conclusions and Recommendations 126

Appendix A Abbreviations and Definitions 128

Appendix B VMEbus - AM codes 130

Appendix C Simulation Results 132

Appendix D Chapter 2 in Tables 136

Appendix E Implementation Conditions 139

Literature 140

Index 144

1. Introduction

Progress in technology, computer architecture, design tools (e.g. CAE) etcetera have led to an enormous increase in speed and capabilities of Computers. However despite this steep growth, the speed and capabilities of computers continuously run short in fulfilling the user's demands. These demands always seem to be one step ahead of temporary capabilities.
One way to meet the user's demands has been to combine processors and/or computers into a working-force, better capable of distributing the workload and thus better servicing each and every user. A multiprocessor is one such working-force, special in the sense that its processors share a common memory.
To investigate into the multiprocessor phenomenon a project was started by the Digital Systems Division of the Department of Electrical Engineering from the Eindhoven University of Technology. It is in this project that I was granted to fulfil the final assignment, a requirement to earn the dutch 'ir' (M.Sc) degree. The purpose of my assignment ("investigate Multiprocessors based on Unix^{R}") was two-fold. First the wish existed to arrive quickly at an actually working multiprocessor, which could then serve as the basis for subsequent design and experiments, second one also wanted to dispute performance issues. As a result of this dual nature the reader will notice that parts of this thesis are near the implementation level while others are at the system analysis level.

The contents of this report is as follows. In chapter 2 our definition of a Multiprocessor is given and several of its aspects are discussed in a general way. The reader will appreciate that due to the strong relation between these aspects some overlap occurs in their treatment. Chapter 3 is meant to restrict the vast area of multiprocessor issues to an area comprising only what is of immediate relevance to our group. This chapter also presents four Multiprocessor architectures which are subject of the subsequent four chapters.
The Master-Slaves architecture is highlighted in chapter 4. The Master processor in this configuration is allowed to run both supervisor as user code whereas the slaves run only user code. A derivative of the Master-Slaves architecture, the Uni-Kernel architecture, poses us for a new range of problems in chapter 5, but, once these are solved, breaks the way for the Multi-Kernel architectures featuring in chapters 6 and 7. Chapter 6 was meant to discuss the simplest of the two Multi-Kernel configurations, the one in which all main memory is shared, in its entirety. However we only managed to look at scheduling in detail. A simulation model has been built to assess the performance and the effects of changes in the proposed scheduling subsystem.
The Multi-Kernel architecture also employing private memory is the subject of chapter 7. Unfortunately time was too short to uncover the details of this configuration. The main benefit to be expected is the diminished bus load. As the bus is the first encountered bottleneck in the performance-raising process the performance of this last architecture may be expected to be highest.

As a final result of the investigation we present some routes towards a multiprocessor of the 'Multi-Kernel with private memory' architecture in chapter 8. One of these routes will implement the architectures mentioned one at a time, letting the more complex architectures evolve from the simpler ones. Which route will be followed depends on group policy.

Finally, a number of conclusions are given in chapter 9. Some of these conclusion have a wider significance then just for the case discussed in this thesis.

2. Multiprocessoring Systems

Since the advent of the computer, engineers and scientists continuously improved its performance. Now we are meeting the limits of nature's laws in trying to achieve further improvements especially with respect to speed. Speed improvement, as the main way to boost performance, can be achieved in a number of ways.
The first way is to use better technologies for realizing the components that make up the computer. Here two aspects can be distinguished namely **miniaturization** and the **technology** itself. There may still be room for improvement in the latter aspect (e.g. optical technologies), the former however is now approaching the limits set by molecule size and speed of light.
Another way of improving performance is to use better architectures. Improving the architecture of the processor by applying pipelining and other such techniques is one way of doing it, putting more processors at work to satisfy the user community is another. The latter solution not only improves bare speed but is also more intelligent in distributing the resources (processors) over the users thus achieving a higher per processor utilization rate.
In the last decade we have seen methods to put more processors to work in a combined fashion. These methods receive considerable attention and are the subject of many industrial and scientific research projects. This field of work can be divided in two area's, the area of Distributed Computers and the area of Shared Memory computers called Multiprocessors in this report. Note that other authors use different classifications. Tanenbaum [TANE81] for example classifies these systems from a communications point of view. Distributed Computers are, in his classification, systems that manifest themselves at the Application Layer of the ISO-OSI model while Shared Memory systems do so at the Network Layer. Flynn [FLYN72] presents the well-known classification where the number of instruction streams and the number of data streams are the distinguishing characteristics. In his classification Distributed Computers as well as Shared Memory computers are of the Multiple Instruction Multiple Data (MIMD) type. The other three classes are SISD, SIMD and MISD. The latter is an empty class, the SISD is the well-known uniprocessor and the SIMD class contains the so called array (or vector) processors. Array processors perform the same operation on pieces of the total data and they do so simultaneously. This is achieved by having more than one identical processing unit under control of one control unit.

We are concerned with multiprocessors in this report and rely on the definitions given in the next two paragraphs.

Distributed Computer Systems form the class of computers that we wish to exclude from our field of attention. The term Distributed Computer System perhaps is very confusing since most people working in this area define its meaning in their own way. Beside the vagueness of its meaning the term Distributed Computer System also has a number of full or near synonyms like **distributed computing system** or **network computing system** etcetera.

Definition 2.1

We will understand a **distributed computer system** to be a set of computers or processors that communicate with each other in order to jointly service user demands. However these computers will not possess a shared memory through which that communication takes place. Instead they use LANs or long haul networks to that purpose.

Though this class of multicomputers is a very interesting one, it is not the subject of this report and therefore not further pursued here.

The other area of research is concerned with the so called **shared memory** multicomputers. Here also various terms are used to describe this field. The majority of these have no rigid definition. Again we will therefore define our term and its interpretation here. In doing so we rely heavily on the terminology and definition thereof as given by Satyanarayanan [SATY80].

Definition 2.2

A **multiprocessor** is a system that contains 2 or more processors which share all or part of their main memory, and that employs its processors dynamically with user (sub)tasks (Hence, processors will interact). The processors are capable of performing a set of general tasks and thus are not dedicated to one (for example IO) task.

Again one is easily confused since the term multiprocessing exists also. Multiprocessing however is used to describe the situation where processes or tasks are logically (and in some cases physically) executed in parallel. This includes the parallel execution of processes occurring in a multiprocessor but also the 'parallel' execution achieved through timesharing on a uniprocessor (multiprogramming). If we wish to specifically speak of parallel execution by more than one processor we will use the term **multiprocessoring** to indicate so.

The remainder of this report will mainly deal with an aspect of multiprocessors namely its operating system. For this reason multiprocessors are discussed in this chapter in much greater detail.

Let us first mention the advantages we hope to obtain from multiprocessoring. A few have already been mentioned but more exist:

Per job run time is shorter

In general this will be true as soon as one is able to exploit the job's inherent parallelism. When run times are shorter system response time is shorter which is to the immediate advantage of the user. There are three different ways for a shared memory multiprocessor to exploit the parallelism in a job. The first way is to write the programs using a library of functions which are geared toward parallel execution. The second way is to present a sequential program to the compiler and have it 'parallelize' as much as possible. The third way is to have the user recognize the parallelism which he or she should then code into the program.

Enlarges throughput

When compared to a uniprocessor system with a processor having equal capabilities as those applied in the multiprocessor, this is evident. More computing power renders a larger throughput for a typical job mix.

Another reason that throughput increases is caused by what is generally called **load balancing**. Load balancing is the technique used to have each processor do an equal share of the total workload. This enlarges the already mentioned per processor utilization rate thus preventing some processors to be overloaded while others leisure.

Other resources than processors and processor time are also put to use in a more efficient manner since more processors can utilize them.

Fault tolerant

Due to the multiple occurrence of some components the multiprocessor may, when it is arranged that way, overcome or alleviate negative effects caused by hardware faults or software errors. A multiprocessor that has this capability will show high availability and high reliability ratings.

More Economic

This does not refer to higher throughput or faster response. For these a comparison should be made between an N-processor multiprocessor and N uniprocessors. This comparison would greatly depend on the type of jobs to be run, the organizational environment in which the machines would function etcetera.

Instead we are referring to the multiprocessors potential capability to grow in small relatively cheap increments thus providing user communities with a gradually 'upgradeable' machine both in quantity as in quality. Economic risks may therefore be much lower as opposed to buying one machine that one may outgrow earlier than expected. The latter situation forces one to buy a larger machine, will disrupt continuous development or affairs and, even worse, may force one to adapt to a new computing environment.

Human like

Human systems usually are parallel processing systems. Adapt tools such as computers to this notion.

Although this is a remark typical of a very high level view of multiprocessors and as such merely serves as a general guideline for most systems it is true in essence. Therefore it must be advantageous, at least in the long run, to build systems that resemble human systems. An attempt in this direction is the **connection machine** [HILL86].

Though most present multiprocessors do not as closely resemble a human system as the connection machine does, they do resemble to such an extent that it may be profitable.

What types of multiprocessors exist to date? When one investigates into the existing types one is surprised with the variety encountered. Therefore we do not discuss different types as a whole. We will first enumerate possibilities per a specific aspect of a multiprocessor. From these different possibilities one may assemble a (imaginative) multiprocessor restricted of course by the mutual compatibility of these possibilities. Then we will review some possible (global) configurations of multiprocessors.

In the next paragraph the different aspects that are discussed are:

Memory map
Interconnections
Symmetry
Input/Output Structure
Interrupts
Inter Process(or) Communication or IP(r)C
Reserved Storage Areas
Operating System and its Structure
User facilities for Parallel Processing
Error Recovery
Performance

In chapter 3 we will choose from and/or restrict the possibilities.

2.1 Memory Map

There are different levels on which one can define a memory map. We have a virtual memory level, a functional level and a physical level. The virtual level provides the memory as it is 'seen' by the processes in a system. Virtual memory is projected onto main and secondary memory by a Memory Management System. The Memory Management System may take several forms, from an empty module (where the virtual level does not really exist) to one containing an address translator and a paging subsystem. The structure of the Memory Map at the virtual level has an influence on the efficiency of uniprocessors and multiprocessors alike (See for example [OUST88]). There are little or no differences between a uniprocessor and a multiprocessor at this level, they have been hidden by the memory management system.

At the functional level we do have a clear difference between uni- and multiprocessors. We consider only main memory and find that we may distinguish private and shared main memory. The adjectives *private* and *shared* can refer to physical access of memory but also to the way it is used. When memory is accessible only by one processor, this memory is physically private to that processor. Otherwise the memory is physically shareable. Parts of physically shareable memory can be used privately by one processor or can be shared by several. We then speak of logically private and logically shared memory. Note that logically private memory can be physically private or physically shared. Normally physically private memory should be used for logically private memory because of efficiency reasons. Nowadays products are marketed that are able to switch parts of memory between the private and shareable address space. Often these products employ dual ported memory. Because of the flexibility with which a design engineer employing such products can manipulate the address spaces it is not worth-while to discuss the physical access. Instead we will assume reference to the (logical) *use* of memory when we speak of private or shared memory below.
In a uniprocessor only private memory exists since there are no other processors to share it with. In a multiprocessor we have shared memory as implied by definition 2.2. However this does not preclude the use of private memory with each processor. Private memory can be used for several purposes. First of all we could load the operating system code into it. This would eliminate accesses to shared memory for this code and hence would reduce shared memory access contention in highly populated multiprocessors. Also we could explicitly load process contexts into private memory for those processes that are or will be in execution on the associated

processor. This too would reduce contention for shared memory access. Private memory can also be used as a cache for data residing in shared memory. There are two reasons to cache shared data. Once again contention for shared memory is reduced. Furthermore it may speed up the memory access.
On the basis of their use of main memory, multiprocessors can be divided into two classes. The fist class uses shared memory only, the second class also uses private memory. Let us name multiprocessors that do not have private memory per processor Shared Memory Only(SMO) multiprocessors and those that do have such private memory Private plus Shared Memory(PSM) multiprocessors.
The different uses of private memory discussed above give the PSM an edge on the SMO machine. Other properties however imply the contrary. There is for example the possible need to relocate a process context from one private memory to another in the PSM when trying to balance the load. Also context sharing is more difficult in the PSM.
Besides the use of main memory to provide the processes with their virtual memory other uses of this memory exist. It is possible to reserve a separate address space for each of these purposes. Thus we could have an address space for the operating system, one for the processes and one for inter processor communications.

At the physical level memory may be inhomogeneous. This may be caused by different access times to different modules but also by the geographical location of modules. The latter aspect has two sides to it, the access time and the decoupling of processor-memory communication networks. Placing a subset of the memory modules and a subset of the processors in the system together and connecting them by a local communication network (a cluster) allows for the simultaneous use of all these local networks in the system. The only time that this is not the case is when the local networks are involved in an inter cluster communication. Generally intra cluster communications will be faster than inter cluster communications. Several policies can be applied to take advantage of this speed difference. Such policies usually concern themselves with the optimal placement of either processes with processors (in 'possession' of process contexts) or with process contexts with processors (in 'possession' of processes). We will therefore refer to these policies as Context Placement Policies.

2.2 Interconnections

Three sets of interconnections are of importance in a multiprocessor. These sets are the set of interconnections between processors and memory modules, the set of connections between processors and, when full caches are present, cache-cache connections.

The **processors-memory modules** communications are very frequent both in SMO as in PSM multiprocessors. This urges to implement the interconnections in a very efficient way. Traditionally the time-shared bus has been used for this purpose but lately new techniques have been applied. These new techniques are usually based on communications theory and can, as is custom in that theory, be split up in circuit-switching, message-switching and packet-switching techniques. The time-shared bus technique can be seen as one of the circuit-switching techniques as will be explained shortly.

The circuit switching techniques can be subdivided into the crossbar connection scheme and multiport connection scheme. In the first scheme a matrix of switches is used that allows every processor to communicate with every memory module. The crossbar switch is often implemented as a set of parallel time-shared busses in a one-bus-per-memory-module fashion. If only one memory module exists in such an implementation this configuration degenerates into the single time-shared bus situation. A time-shared bus can be looked upon in two different ways depending on the way one defines a processor memory logical connection. When one defines every memory access from each processor as a complete conversation (compare to a human phone call), the one available circuit (= the bus) is essentially switched between users. When one, on the other hand, defines processor-memory communication as one long conversation from system power up till system power down, then the operation must be characterized as Time Division Multiplex (TDM). This last view gave rise to the time-shared attribute.

In the multiport connection scheme every processor has its own input port on every memory module. In essence the switching function has been integrated in the memory modules.

In both schemes there may exist multiple requesters for a bus, so some kind of arbitration mechanism is needed.

Packet Switching is the other technique often used for communication between computers. It can also be used for processor-memory communications. A packet is a piece of a message encapsulated in headers and trailers in such a way that it can be routed through a network. During its journey through the network it is regenerated and packed again at every exchange it meets. This packet switching technique has several advantages over circuit switching, the main one being the higher utilization of circuits due to the (logical, not physical) multiplexing of circuits between many connections.

This logical switching is achieved by routing each packet independently through the network. The packet thereby uses only 1 network segment (f.e. an inter-exchange connection) at a time, leaving all segments it already passed and all segments still to pass free for other packets to be transferred. Furthermore a connection only 'consumes' circuits when there indeed are messages to be sent. When both ends are 'quiet' the available circuits are used in other connections. For further detail refer to [TANE81].

While packet switching obviously is a nice technique for communications networks of various topologies and composed of a large number of segments, it is not immediately evident that it is suitable as the communications mechanism between processors and their main memory in a multiprocessor. Especially the delays caused by the packing and unpacking would seem to render the method unfit for this purpose. The first multiprocessor to prove usefulness of packet switching in this respect was Carnegie-Mellon's CM*. For an introductory description of this computer see [SATY80].

The dominating factor in processor-memory communications is the achievable bandwidth. This bandwidth is limited and will therefore limit the number of processors in a multiprocessor. An interesting possibility to implement processor-memory communication is to make use of optical data paths such as fibers. Fibers have a very high bandwidth and can be used over large distances. The larger distance opens the possibility to design multiprocessors that are not confined to one cabinet. As soon as high speed transducers and associated IC's are commonly available one could replace conventional methods, like the time-shared bus, with fibers.

The **processor-processor** communications may use point-to-point processor links with the drawback of a quadratic increase in the number of links when the number of processors increases. At the cost of arbitration delay and congestion one could eliminate the quadratic increase by using a time-shared bus again. Other topologies such as the hypercube strike a good balance between number of physical links and bandwidth.
Last but not least the processors could make use of the existent processor-memory connections to communicate. In that case the time-shared bus and crossbar switch situations still allow for direct inter processor communications, in the multiport situation it necessarily is a two-step processor-memory-processor operation that would do the job. (This last method would need an interrupt to be effected or polling to be done).

2.3 Symmetry

A symmetrical system will show more universal software and hardware than an asymmetrical system, it will also have a higher degree of fault-tolerance, a higher up-time etcetera. Further process switching and inter processor communication are likely to be much smaller in a fully symmetric situation. This can be explained by the fact that when a need for some service occurs in a main processor that same processor is capable of handling it, avoiding the need for inter processor communication to request the service or the need to relocate the requesting process to a processor that is capable of handling the request.

Of course the benefits mentioned are gained to a degree that is related to the machine level where symmetry starts. Benefits will be few when only the user interface makes the machine appear to be symmetrical while all lower levels are asymmetric.

The symmetry can be present across the boundary between main processors and IO processors but may be restricted to main processors only. In the former case the operating systems running on both types of processor can be the same to a high degree.

For a number of reasons, for example hardware restrictions, one may be forced to asymmetry in one area while achieving full symmetry in others.

Although symmetry is a praiseworthy characteristic of a system in many respects, an asymmetric system has its special advantages too. One could for example have one or more vector- and/or floating point processors next to the normal main processors to speed up such operations [TEST86].

2.4 Input/Output Structure

In every computer the way input and output is performed has large effects on system performance and response time. In the earliest computers IO was performed by the one and only processor present in the system. Gradually more and more specialized hardware was developed to perform IO tasks leading to disk controllers, network controllers etcetera. At first these controllers were running under the regime of the main processor. With the advent of multiprocessors IO controllers often serve more masters.
This trend of shifting intelligence from main processor(s) to IO processors has even led to the migration of parts of the operating system to the IO processors. Moreover, systems have been developed where IO is performed on separate dedicated boards running (almost) the same operating system as the main processors do [ANNO85]. Pushing this one step further, that is employing the IO processor with main processor tasks, returns us to the starting point where main processor and IO processor are one and the same though this time in cooperation with other such processors.
An important factor here is that communication network IO can, and in some cases does, replace traditional In- and output. Think for example of terminals connected via Ethernet or using a network to access a shared printer. In this sense multiprocessors may exhibit 'distributed computer'-like properties.

The structure of the IO subsystem may have a high degree of symmetry or just the opposite. An example of the latter case is the situation where every IO processor communicates with one main processor, the master. Every IO request must then be directed to, and results are coming from, the master processor. Usually this architecture, with the master representing a (potential) IO bottleneck, is chosen to easily solve the problems induced by the synchronization needs. The complete synchronization including the handling of interrupts is taken care of by the master processor. With respect to synchronization the master-slaves configuration is equal to the single processor system .The communication between master and non-master main processors stays undiscussed here. (Communication may even be made obsolete through process switching).
The other extreme is to allow every main processor to either perform IO itself or to communicate with any IO processor. Naturally this is a much more desirable situation. First of all this will be a two-party instead of a three-party communication (Switching a process from one processor to another is seen as a form of communication) thus reducing the overhead for context switching and inter processor communication. Second, if a main processor breaks down, no devices become unreachable which adds to the fault tolerance of the system. Third, when interrupts are used for IO, the latency times can be less. (can be, because this strongly depends on the implementation.)

2.5 Interrupts

An interrupt is a logical control signal. Making the signal active is called an interrupt request. A request can be accepted, delayed or ignored. When an interrupt request is ignored nothing happens to the current flow of instructions in execution on the addressed processor. When an interrupt request is delayed it is said to be pending. At some point in future it will be accepted. When a processor accepts an interrupt request it stops execution of the current flow of

instructions, usually saves some of its registers, and will start executing another flow. The new flow of instructions is said to 'handle' the cause of the interrupt. The sequence of instructions that make up this flow form the Interrupt Handler (IH) for that interrupt. The last instruction in the IH flow ('return') will cause the processor to resume the previous flow of instructions. On most systems interrupts can be nested so that the resumed flow may be from an interrupt handler too. Interrupt requests can be made by activating hardware lines but also by the execution of special software instructions (extra code, supervisor call). Some conditions in the processor may also give rise to an interrupt request.
Several events occurring in a computer system are traditionally signalled by means of interrupts. Some of these events are presented below:

a. Supervisor Calls
b. Divide by Zero, Overflow, Illegal Instruction
c. Page Faults, Protection Violation
d. Inter Processor Communication
e. IO completion/errors
f. Clock
g. Hardware Faults

In a multiprocessor it is not always trivial which processor a request should be addressed to. For the cases a,b and c it is obvious that the processor where the event occurs must accept the resulting request. Inter processor communication interrupt requests are purposely fielded to the receiving processor(s) and must therefore be accepted by them. For the cases e and f the situation is more complex in that more ways exist to handle these events.
In these cases we may often choose a processor to accept the request. The question therefore arises which processor such an interrupt request should be directed to and/or which interrupt request a processor should accept. The policies used can be static or dynamic. In the first case it is decided on beforehand which processor receives and accepts what type of interrupt requests. In the latter this decision is taken the moment the request is generated. As an example of a dynamic policy a request caused by an IO completion could be directed to the processor that currently runs the requesting process.
For IO events the situation is somewhat different in the presence of IOP's than without these. When no IOP's are present it is common that the peripheral devices directly interrupt the main processor that controls them. This processor consequently has to accept and service the request. When IOP's are present they deal with these low level interrupts and may themselves interact with the main processors in a more sophisticated way.
For the case of the clock event a possible implementation is to have a single timer interrupt one main processor which communicates the clock event through IPrC messages to all other processors as part of its interrupt handling.
The occurrence of Hardware Faults should generally be made known to the entire processor community. Unfortunately some faults place one or more processors out of working order so that normal IPrC cannot be used to alert the sane processors. Because of this hardware faults are generally detected and signalled at the hardware level. The signalling usually involves a system wide interrupt line that interrupts all processors.

2.6 Inter Processor Communication

When one distributes processes that mutually communicate across processor borders, an underlying Inter Processor Communication mechanism (IPrC) must exist that will be used to support this Inter Process Communication (IPC). Inter process communication is a necessity when more processes combine to accomplish one task. Processes will have to synchronize their execution and they will have to exchange data. For this reason most uniprocessor multitasking operating systems have some forms of IPC like pipes, mailboxes, sockets, message passing and signals. All of these forms have to be supported by the Inter Processor Communication subsystem.
IPC is not the only area where IPrC is needed. It is needed in all of the following areas:

- System Startup
- IO Communication (when IO processors are present)
- IPC
- Performance Tuning (e.g. Load Balancing)
- System Monitoring (e.g. Tracing)
- Error Recovery

Many mechanisms to perform IPrC exist. Some general characteristics can be distinguished that are useful in discussing the merits of certain implementations:

Data	versus	Events
By Reference	versus	By Value
Polling	versus	Interrupts
Hardware	versus	Software

The first characteristic gives the impression that events and data are entirely different things that can be transported. In essence they are not that different, an event is a communication through which no data is transferred apart from a possible identification of the event. The main part of an event transfer is the synchronization of the two (or more) processors that take part in the communication. Data communication also needs this synchronization but above that, a mechanism to either move the data or to pass a pointer. In view of the above we could conclude that one universal IPrC mechanism can be developed that supports both types of transfer. However due to the differences that exist between the two types, different methods of transfer may be profitable and in cases are necessary. An example of a situation in which a dedicated method is needed to convey events lies in the field of error recovery. A processor that has broken down will probably not react to the normal IPrC. To either reset it or disconnect it from the bus special hardware is needed. This too is a form of IPrC.

Data transport can be done *by reference* or *by value*. Transporting a message by value means that the message is copied in its entirety from a location at the sender site to a location at the receiver site. When a message is transferred by reference, the sender prepares the message in shared memory and then copies a pointer to the receiver. Transport by reference is not always supported by the architecture at hand, the latter usually is. The multiprocessor as we defined it in definition 2.2 always has shared memory and will therefore allow transport by reference. In general, transport *by reference* is more efficient. This is because the message is not copied and therefore no time is lost performing the copying action. Also the communication event is short. The communication event is defined as the actual transfer of the message and its duration is defined as the time the communication channel between sender and receiver is occupied. When passing the message by value this duration will equal the amount of time necessary to copy the message. When passing it by reference the duration equals the amount of time to transfer the pointer. A short duration of the communication event will cause less contention for the channel when more messages need to be transferred simultaneously. However there are a number of implementation characteristics that may cause transport by value to perform better than transport by reference. A very important factor is where messages are placed in a memory space when that space is inhomogeneous. Also the way a message is used may be of importance. Suppose for example that in a system employing the time-shared bus a message is placed in shared global memory reachable only via this bus. If the receiver then heavily references the message, the bus load, a crucial factor in bus'ed multiprocessors, could become greater than when a straight copy of the message had occurred.

Apart from error recovery most inter processor communication can be entirely done by software. Through the application of the polling technique one could do entirely without special hardware support. Normally at least interrupts are used to draw the attention of the communication partner(s). Since the interrupt subsystem is usually available and of universal nature, here too one can hardly speak of special hardware support.
The last few years more and more special hardware is designed to perform IPrC [FORC87], [RAMA87], [RAP86], [VERS87]. Besides the growth of the number of multiprocessors and distributed computers another phenomenon causes this extra interest. This phenomenon is the tendency to more and more extract the communication from a system and treat and design it explicitly as a separate function. Having the major part of the communication done in specialized hardware relieves the processor from the communication overhead and increases the speed. Reduced costs for communication in turn leads to an increased use of communication intensive systems.

A different view on IPrC is its location within the operating system. Starting from the layered model used by A.M. Lister [LIST75] and shown in figure 2.1 IPrC appears to represent an extra segment in the bottom layer. It needs services from the First Level Interrupt Handler (FLIH) and it may need mutual exclusion primitives.

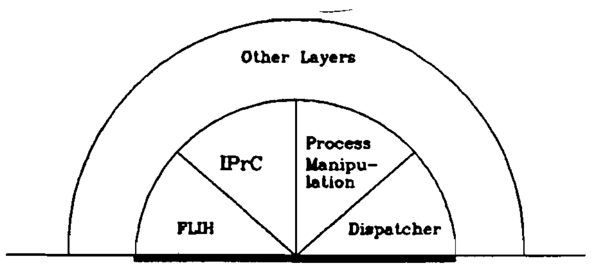


Figure 2.1 Location of IPrC in Kernel

2.7 Reserved Storage Areas

In a multiprocessor different address spaces may exist. In fact each processor may have an entirely different address space of which only a part is shared with others. Even addresses in the shared region need not have the same absolute value seen from the processor side. Address translation mechanisms may be used to project the processor address region onto the same physical addresses for all processors. However certain parts of memory are reserved for special use by hardware. Think for example of interrupt vector tables. When using more than one processor in a system more than one such area exists usually occupying the same physical processor addresses. When only one global memory is available or when it is desired to keep address translation from processor space to system space simple these area's may be relocated or hidden from the system map to avoid a possible clash. There are a number of methods to do so. The first and easiest possibility is to make the area's physically equal. The purpose of such an area must naturally allow for this. When such an area contains code, it has to be re-entrant. If the area contains dynamic data, mutual exclusion must be taken care of. The operating system code and data structures could for example be placed in such an area.
A second method is to hide the areas from the shared memory map. In this case they are only accessible to the owning processor. Thus there could for example be N private area's at the lower end of the total address range and 1 common area above that.
Thirdly when the reserved areas are not tied to specific processor addresses it is possible to simply place them at convenient and distinct address ranges. The fourth and already a little more elaborate possibility is to map each processor owned area onto different sets of addresses in shared memory.

In the early days of multiprocessors the reserved areas could pose quite some practical problems. At present memory management support is widely and cheaply available either as separate IC's or even integrated into one chip with the processor. For this reason present systems combine the private area's mentioned in method 2 with the mapping of method 4 to provide a very sophisticated and user configurable address map. Some systems even offer dynamic configurability where the map can be changed under program control. Parts of memory may then be switched from private to shared space and back.

2.8 The Operating System

Imagine a group of people that work together to accomplish a large task. The task is composed of subtasks that can be partially done in parallel. Sometimes a subtask cannot be started until another one (or more) has been finished. In some way or another these people have to find an organization in which they can do the job. We all know several of these organizations, starting with the familiar boss-employees one. In this situation the boss has all relevant information about when which subtasks should be done in order to perform the overall task. Sometimes he even knows he has to put (say) 3 people on a subtask, no more and no less, in order to have it done efficiently. This boss will tell his employees what to do and at what time, he will arrange the order in which his employees are allowed the use of shared equipment and when the equipment is very expensive, he will sometimes operate it himself.

At the other extreme we find an organization form where all people are alike and take counsel together to finally finish the assignment.
Between these two forms of organizations many others are possible.

Like the people above the processors in a multiprocessor are in need of an organization too. It is the operating system that has to provide for this. The multiprocessor operating system organization often resembles the human organizations described above. With operating systems too there are a lot of possible forms. The three main forms are:

- Master-Slaves
- Central Operating System used by All Processors
- An Operating System per Processor

A Master-Slaves operating system can be compared to the Boss-Employees organization. In this operating system one processor, the Master, plays approximately the role of Boss. The Slaves in such a system can only execute application processes and must request the Master to perform the operating system actions necessary for their progress. Scheduling of jobs is also done by the Master. Since the Master is the only one that runs operating system code, this code need not be re-entrant. Also operating system data structures need not be locked for there is only one processor that is allowed access. Interrupts that need handling which involves manipulation of operating system data structures can conveniently be sent to the Master.
Converting a uniprocessor operating system to a Master-Slaves type often concentrates on the interaction between a slave and the master. Although a slave can be placed into the supervisor mode by a system call it does not have any supervisor rights. It therefore requests the required service from the master and continues processing another process or idles. The master on the other hand services the requests from the slaves until the process returns to user mode. If the master is not dedicated purely to operating system tasks it could execute a process in user mode when no operating system requests are pending. The efficiency of an N processor system rapidly decreases with N when compared to N uniprocessors due to the operating system bottleneck. This type of operating system has been mainly applied in the early days of multiprocessors as a fast route from existing uniprocessors to multiprocessors.

A Central Operating System in use by all processors makes larger multiprocessors possible. However this is at the cost of mutual exclusion measures that have to protect the operating system tables. Also it necessitates the OS code to be re-entrant. Each processor in the system executes operating system code as soon as this ~~is~~ necessary for the progress of the current process. (synchronous system calls). Other operating systems functions, though related to a process can be performed by any processor in the system. An example is the handling of interrupts for IO operations. (asynchronous system calls). A system under regime of such an operating system may provide graceful degradation.

Having an executive per processor with private data structures brings us in the twilight zone between true multiprocessors and distributed computers. The access conflicts that could arise in the previous operating system do not occur here. However other, performance degrading, phenomena occur as we have seen before (load unbalance, process context relocation). This type of system will show a profit mainly there where communications with shared memory and other processors are slow or where a high average processing load exists.

Besides the organization with respect to the processor community other aspects of the operating system play a role. Many of these aspects are also found in uniprocessor systems. For example an OS could be made to hide possible hardware asymmetries which makes the system easier to use. On the other hand in some applications the user needs to exploit the existing asymmetry to fully exploit the systems capabilities (e.g. Real Time environment).
Operating systems for multiprocessors or any system that supports parallel processing must provide operating system primitives that enable the user to manage the parallelism available in the system. Among these are primitives such as synchronization primitives, shared memory primitives, requests for dedicated processor(s), user invoked process suspension and resumption, user controlled scheduling etcetera.

2.9 User Facilities for Parallel Processing

Users of multiprocessor systems have different attitudes. We distinguish between users that rely on the system's built in 'parallelizing' capabilities and those that wish to explicitly indicate parallelism in their programs. For both types of user the multiprocessor system should provide the means they desire.
For the first type of user a number of tools have been developed that recognize and exploit parallelism in a user's sequential code. Among these tools we first of all find compilers. For example, the Alliant FX/Fortran compiler detects loops that can be executed in parallel and automatically generates code containing concurrency control instructions [TEST86]. Of course this compiler is targeted towards the specific Alliant hardware. In general multiprocessors the basic building block for parallel computation is the process. With most present multiprocessors these processes are too large and too inflexible, due to expensive context switching and inter process communication, to be used for fine-grain parallelism as achieved by compilers. Much effort is invested today in achieving an operating system that supports inexpensive (or fine-grain) processes. One route taken is to provide multiple so called threads within one single traditional process. These threads (light-weight processes) use the resources possessed by the process (such as address space, file descriptors etc.) and have a minimal context which makes switching fast. When light weight processes are available parallelizing compilers can be of use in general multiprocessors too. A less complex approach could be to provide libraries that contain as much parallelism as possible.

The second type of user will need a language in which they can easily express parallelism. This means that explicit language constructs must exist to delimit code that can be executed in parallel, to synchronize such parallel code, and to provide communication between these pieces of code. Such a user will also need very powerful tools to design and debug parallel programs. As a matter of course these language constructs must be supported by operating system primitives as discussed in the previous section.

2.10 Error Recovery

We already mentioned at the start of this chapter that recovering from an error can be done much more sophisticated on a multiprocessor than on a uniprocessor. It is possible to hide many errors from the user apart from a possibly noticeable performance degradation. Having the potential to recover from errors does not mean it is easy to design recoverability into a system. When an error occurs some part of the system does not function as specified meaning that one has to deal with in part unpredictable behavior. An illustration of the complexity one has to deal with when attempting to fully protect users from suffering, is the following example.
A processor breaks down while running a process that holds several locks. The recovery system will first have to signal every other processor in the system that a colleague is malfunctioning. This will prevent processors from interacting with the malfunctioning one. Then the process needs to be recovered. The most elegant way is to restart it from where it was forced to stop. This will also avoid any problems with the locks. In general this will not be possible for there is no way to question the faulty processor as to the progress of the process. It is also difficult to obtain the current point of execution from the process context when it is at all reachable (= not in private memory). Usually some backtracking procedure must be used to find a point from which a restart is possible or alternatively the process could be restarted from scratch. Since the process held locks it may have left user data structures and kernel data structures in an inconsistent state. These structures will have to be salvaged before **any** process is allowed to further manipulate them. (An error must therefore be detected as soon as possible!).

The above example clearly shows the complexity involved in recovering from some errors. It also shows that a need for a hardware signalling system exists in order to warn all sane processors as soon as possible (See 2.6 IPrC).
Functions and mechanisms of error recovery therefore include:

- Hardware generated messages (on error) to every processor
- Watchdog timers to detect software crashes
- Capability to inhibit requests from faulty requestors.
- Isolation of faulty processors
- Resetting/restarting of faulty processors
- Salvaging of corrupt data structures
- Removal of deadlocks caused by the error
- The error recovery system must **not** propagate fault effects through the system.
- Restart the system from a fresh instance of the operating system
- etcetera.

It will take a lot of effort to reach the sophistication of recovery as in the example above. However much less effort is needed to have a recovery that outperforms that of a uniprocessor.

2.11 Performance

There are several ways to judge the performance of computer systems. Often the performance of multiprocessors with N processors is compared to N independent uniprocessors. There are two dimensions to this comparison. First, some multiprocessors are designed to enlarge the throughput. Second, some are meant to decrease response times. Multiprocessors optimized for one of these dimensions commonly achieve profits in the other dimension too. It is evident that running a program on a set of N independent uniprocessors will not show decreased response times when compared to a single uniprocessor. A multiprocessor however may show this effect. When comparing the pure CPU-capacity that can be used for user processing between an N processor multiprocessor and N uniprocessors, a processor always looses. This is because the marginal benefit of each added processor decreases as a consequence of, amongst others, the following phenomena:

- Memory Contention
- Processors-Memory Network contention
- Cache validity delays
- More complex Scheduling
- More complex locking
- More OS overhead in IO
- More multiprogramming
    - longer queues
    - fuller hash tables
    - etc.

A lot of research is going on to diminish the effects of these phenomena.
So it seems that a multiprocessor has less throughput than a corresponding set of uniprocessors. This would be true when the workload would be always evenly spread over the processors in both cases. In practice this is not the case. The multiprocessor is capable of balancing the load automatically which is not the case for N independent uniprocessors. In practice (dependent also on organizational aspects) a multiprocessor will perform better.

3 Target Multiprocessor

In this chapter we will do two things. Whenever possible (or unavoidable) we will determine the properties we would like our multiprocessor to have. Otherwise we will restrict the range of possibilities for a system property. Note that this chapter gives trivial or a priori arguments most of the time and that detailed discussions of certain topics can be found in chapter 4.
With every choice the arguments that led to it are mentioned. The reader is stressed to note that some of the arguments are not pure technical but are also based on organizational aspects. Most of the choices made are captured in "Conditions". This gives clarity and makes referencing easy. Although the word condition is usually used to mark strict limits we will at times withdraw or alter one on our route to our ultimate multiprocessor. Next to conditions we also define "Design Strives" as guidelines to arrive at certain favorable characteristics of the system.

We will stick to the outline also used in chapter 2 and first deal with the aspects (though not all) of the multiprocessor and then look into the question of what configuration to choose.

3.1 Aspects

Let us first recall from chapter 2 the aspects of a multiprocessor that we should look into:

Memory Map
Interconnections
Symmetry
Input/Output Structure
Interrupts
Inter Process(or) Communication
Reserved Storage Area's
Operating System
User facilities for Parallel Processing
Error Recovery
Performance

Since the work presented in this report is at the very beginning of the project we will not discuss the items

User facilities for Parallel Processing
Performance
and Error Recovery

but leave these for discussion until they are due. Error Recovery however will occasionally be touched upon in treating the other aspects.
Symmetry of the system is a property we highly appreciate. We will not devote a special paragraph to this aspect but use it as an argument in making choices about the system configuration.

Design Strive 3.1
The system must be as symmetric as possible.

Since the group Digital Systems had already put considerable effort into the development of a uniprocessor minicomputer based on the UNIX version 7 operating system and the VMEbus from 1982 until 1986 [BOKH82], it was natural to try to put that experience to use. For this reason the target multiprocessor should be built around the VMEbus and should use the Unix operating system:

Condition 3.1
The implementation must be based on the VMEbus.

Condition 3.2
The Operating System to be used is Unix^R System V.

This restricts the number of choices in some of the aspects mentioned.
Let us now discuss the aspects one by one:

3.1.1 Target Interconnections (VME)

Because of the forementioned starting point this paragraph really comes down to a discussion of the VMEbus though placed in the light of multiprocessor systems where appropriate.

The VMEbus dates back to the late 1970's when Motorola developed the 68000 microprocessor. To support this processor a development system was built that had the predecessor of the VMEbus as its backplane. This predecessor was called the VERSAbus. During the evolution of the VERSAbus a european division of Motorola's in Munich, Germany developed some eurocard--sized boards using a backplane for those boards derived from the VERSAbus. They called it VERSAbus-E. Shortly after this, Motorola licensed a number of companies to second source the 68000. These companies were also invited to support a standard bus for 68000-based developments. One agreed upon supporting the VERSAbus-E which was renamed VMEbus. Since then several specifications were made for this VMEbus. The latest revision is from the IEEE P1014 Standard Committee and is known as Draft 2.0 .
Apart from the specifications themselves numerous descriptions have been written ([KAPL81], [TIMM87]) so that we will restrict ourselves here to what is necessary to understand the rest of this report.

The VMEbus specification describes mechanical, electrical and functional aspects of the VMEbus. The specification also includes descriptions of 2 other busses: the VSB (VME Subsystem Bus) and VMS (VME Serial Bus) bus.

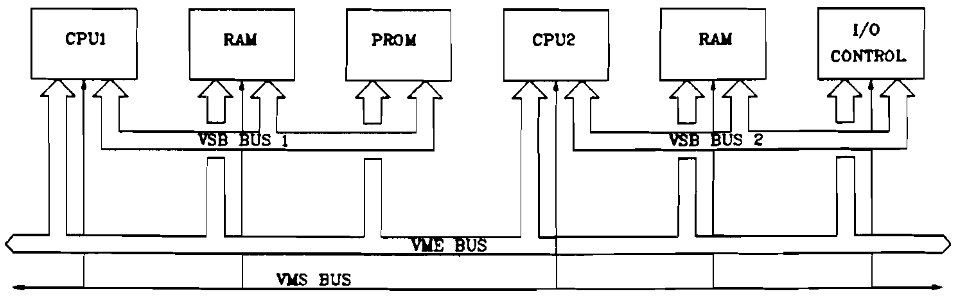


Figure 3.1 VMEbus based System

The VSBbus is a bus private to a VME board processor. Its purpose is to give that processor direct access to additional memory, IO , and other functions. This bus will not be further described here.

The VMSbus is a self-arbitrating high-speed serial bus that occupies two signal lines on the backplane. It also shares the reset line and ground with the VMEbus. The VMSbus is as such an integral part of the VMEbus as opposed to the VSBbus which is better characterized as an add-on bus. The VMSbus supports such functions as intelligent semaphores, broadcasting, simultaneous polling and fault tolerant schemes. We will describe the VMSbus in paragraph 3.1.1.5.

The VMEbus bears an important feature namely that it is not only specified as a group of signal lines and their electrical interfacing details but that functional modules are specified that define and therefore standardize a number of operations on the bus. The bus thus consists of the signal lines, the interface logic and the functional modules.

The signal lines are grouped into 4 buses, each of which having its own associated functional modules. The 4 buses are the Data Transfer Bus (DTB), the Priority Interrupt Bus, the DTB Arbitration Bus and the Utility Bus.

3.1.1.1 Data Transfer Bus

The Data Transfer Bus holds three groups of signal lines that, in traditional microprocessor terms, would be called data bus, address bus and control bus. To this Data Transfer Bus 4 functional modules are defined known as:

master
slave
location monitor
bus timer

The **master** initiates data transfer bus cycles in order to transfer data between itself and a slave. There are 7 basic types of data transfer bus cycles: read, write, unaligned data transfer, read-modify-write, block transfer, address only and interrupt acknowledge cycles. The read-modify-write cycle will prove to be of particular importance to us.

The **slave** detects data transfer bus cycles initiated by a master and, when those cycles specify its participation, transfers data between itself and the master.

A **location monitor** holds an address or a number of addresses set by the user. It continuously monitors the data transfer bus to see whether such an address is used in a data transfer cycle and if so it interrupts its processor.
Some companies now provide value passing location monitors. Associated with the location monitor is a register. When writing the register from the VMEbus a local interrupt is generated and the data is stored in the register, available for the interrupt handler to investigate. This mechanism can be used for inter processor communication. For example, a message is placed in shared memory by processor A which subsequently notifies processor B by writing a pointer, identifying the message, to a location monitored by B. B's interrupt handler will now be able to locate the message.
Other forms of the location monitor occur, bearing features such as message broadcast, message buffers etc. [FORC87]. With this broadcasting method an 8 bit message can be transferred to a maximum of 20 boards in one standard VMEbus write cycle (about 330 ns). Unfortunately this type of broadcasting is not standardized in the VME specifications.
(Note: The type of broadcasting mentioned here should not be confused with the Broadcast Bus signals as defined by the VME specification. The latter signals are each sent over a dedicated signal line on the backplane (See Utility Bus below).)

The **bus timer** terminates the data transfer cycle if it takes too long. It is useful in fault tolerant environments to hide hard- and software failures. Accessing a malfunctioning or non-existent slave will not hang the system when such a bus timer is used.

An important issue not treated yet is the phenomenon of Address Modifier lines. (For short: AM lines). These 6 lines form part of the group of DTB address lines and essentially define 2^6 (= 64) logical address spaces. The AM code of each data transfer cycle thus determines to or from which address space the transfer takes place.

There are numerous ways of using the AM lines.
Let us name a few:

System Partitioning

Complete microcomputer systems may coexist in one VME rack. Mutual independence with respect to address space is achieved through the use of disjunct sets of address modifier codes.

Memory Map Selection

Slaves may be designed to respond at different addresses, depending upon the address modifier required. This allows the master using the bus to place the system resources in selected map locations (or eliminate them from the map) by providing different address modifier codes.

Privileged Access

The AM code could be interpreted as a privilege level. Masters not having the correct privilege level should be denied access.

Distributed Memory Management

Memory Management logic is often used in systems to allocate and translate segments dynamically. A collection of these segments is assigned to each active task. Each time the operating system switches from one task to the next, it must either change the contents of the segment registers or select another set of segment registers (the latter approach being much faster).
Address Modifier codes may be used as segment register selectors. In this case, the Master places the AM codes on the bus which indicate to memory management logic on the slave boards which set of segment registers to use.

Addressing Range

The VMEbus provides 31 address lines to allow direct addressing to over 4 billion bytes. For most slaves however the extra logic required to decode all 31 address lines is a needless expense. For this reason, the VMEbus defines 3 addressing ranges:

Short addressing	64 kbytes
Standard addressing	16 Mbytes
Extended addressing	4 Gbytes

A group of Address Modifier codes is set aside for each type of addressing. Slaves receiving a short address AM code ignore the upper 16 address lines(A16-A31). Slaves receiving a

> standard address AM code ignore the upper eight lines (A24- A31). When receiving an extended address AM code, the slave decodes all 31 address lines.
>
> Slave boards which do not decode address lines A24-A31 should not respond to extended address AM codes. Slave boards which do not decode address lines A16-A31 should not respond to either extended or standard AM codes.

From the above one may observe that the VME specification uses up most of the 2^6 (64) AM codes available. For arbitrary use by the user 2^4 (16) codes are set aside.
In appendix B the AM codes are listed along with their intended purposes.

The VMEbus does not have a standard mechanism for message passing or broadcasting. Since the need for such services is high implementers were required to develop their own method of inter processor communication. But for lack of a standard, each implementer's IPrC differs.
For broadcasting the situation is even more complex. The VMEbus specification (IEEE 1014 Rev C.) does not specify a 'broadcast cycle') but it also does not forbid one. To provide broadcasting on the VMEbus an implementer would first have to define a 'broadcast cycle' and then the broadcast protocol. This is what FORCE computers have done. Their *patented* implementation is called FORCE Message Broadcast (FMB) [FORC87]. The reason of discussing it here is to show the possibility of broadcasts on the VMEbus.
FORCE Computers have defined a broadcast cycle as a normal write cycle with two additional criteria. The first is that all slaves, responding to the address written to, must synchronize on the falling edges of the Data and Address Strobe signals in order to correctly time their respective Data Acknowledge and Bus Error signals. The second criterium is that when one (or more) of the participating slaves detects an error, it asserts its Bus Error signal within 150 ns after the Data Strobe signal has gone active. The other slaves must not drive their Data Acknowledge signal valid before 200 ns after the Data Strobe has been asserted.
The Broadcast Protocol is as follows (if we interpret the scarce information correctly). The broadcast message is a byte. The CPU that wishes to broadcast this message byte to a number of other boards sends a 32 bit word to the Broadcast Reception address. The 32 bit word is composed of 3 fields. The first field is the 8 bit message. The second field is a 21 bit field, for each possible slot in a VME system a bit. For those boards that must receive the message the 'slot-bit' is set. The third field is 5 bit wide and seems to be unused. The receiver inspects the data word upon arrival and discards it when its 'slot-bit' was not set. Otherwise it places the message byte into an 8 byte deep FIFO and optionally interrupts the local processor.
The above message broadcasting can easily be extended to broadcast longer messages by incrementing the VMEbus address in between ~~word~~-broadcasts. The first word could still hold the 21 bit field to address the receivers, while the successor words could hold pure data. One aspect that needs to be considered is the signalling of the end of the message and the number of bytes it contains. This could be done through a dedicated broadcast location separate from the one the first broadcast word was sent to. The message byte to this location would have to contain the byte count of the message and the local CPU would have to be interrupted. To complete the extension the unused 5 bit field may be used for message identification.

3.1.1.2 Priority Interrupt Bus

The Priority Interrupt Bus consists of 7 interrupt request lines, one interrupt acknowledge line, and an interrupt acknowledge daisy-chain. To this bus the following 3 functional modules are specified:

Interrupter
Interrupt Handler
Iack Daisy-Chain driver

The **Interrupter** is the module that initiates interrupt cycles on the VMEbus. It does so by driving one of the 7 request lines. When it receives an interrupt acknowledge signal it will send extra information about itself or the cause of the interrupt. It uses the Data Transfer Bus to that purpose.

The **Interrupt Handler** receives the interrupt requests from one or more interrupt request lines. Upon reception it will ask the highest priority interrupter for additional information using an interrupt acknowledge type of DTB cycle. During the interrupt acknowledge cycle the interrupter is addressed by placing its level on the three least significant bits of the address bus. As said, the interrupter will send an interrupt vector to the handler which will subsequently invoke the appropriate interrupt server.

The **Iack Daisy-Chain Driver** activates the daisy-chain whenever an interrupt handler acknowledges an interrupt request. The acknowledgement of the interrupt handler is done by driving the interrupt acknowledge line *IACK** (A * denotes a low level active signal) low. This *IACK** line runs the full length of the backplane and is therefore accessible to every board. Hence it is not important where the interrupt handler is located. The *IACK** line is the input to the daisy chain. It is therefore connected to the *IACKIN** pin of slot A1. When the board at slot one did not generate an interrupt it will pass on the low level on its *IACKIN** pin to its *IACKOUT** pin otherwise it will make *IACKOUT** high. The backplane connects all *IACKOUT** pins of a slot to the *IACKIN** pin of the successor slot (except for the last). Thus boards down stream will only receive an acknowledge when no boards upstream the daisy chain intercept it. The daisy-chain thus assures that only one interrupter sends the additional information asked for by the handler when more than one has generated an interrupt request.

3.1.1.3 DTB Arbitration Bus

When more than one master or interrupt handler need the Data Transfer Bus some mechanism must allocate the bus. The functional module that performs this task is the **DTB Arbiter** or just **Arbiter**. The Arbiter uses the arbitration bus to do its job. This arbitration bus consists of 4 bus request lines, 4 daisy-chained bus grant lines and 2 other lines called *bus clear* and *bus busy*.
The complementary module to the Arbiter is called a **Requester**. The Requester resides on a board where there exists at least one master or one interrupt handler module. On demand of such a master or interrupt handler the Requester requests the Data Transfer Bus using one of the 4

request lines. It then waits for a bus grant acknowledge to arrive via the daisy-chain associated to the request line used.
When the acknowledge comes in from the Arbiter the Requester seizes the bus by making the *bus busy* line active. It then signals the local module in need of the DTB that it is available.

The Arbiter grants control of the Data Transfer Bus to only one requester. It considers requests on Bus Request Lines to have a priority. This priority depends upon the request line itself and on the priority scheme used. When for example a fixed priority scheme is used the bus request line *BR3** has highest priority. A round-robin scheme is also specified.
Some arbiters drive the *bus clear* line when they detect a request for the bus from a requester whose priority is higher than the one currently using the bus.

3.1.1.4 Utility Bus

This bus is not of particular concern to multiprocessor issues with exception perhaps of the SYSFAIL* line in this bus. This broadcast line plays a role in the startup sequence of the system after power up. This sequence is somewhat different for multiprocessor systems then for uniprocessor systems as we will see later on. However the more primitive start up is the same for both and it is good to be familiar with that sequence so that subsequent discussions of the higher level start up sequence have a sound basis.
The VMEbus specification recommends all boards in a VME system to keep the SYSFAIL* line active (= low) until they are fully initialized. This insures that when SYSFAIL* becomes inactive all boards are ready to run and may perform start up sequences involving VME communications. The latter typically lie in the operating system domain and will be discussed in chapter 4.

3.1.1.5 VMSbus

The VMSbus is a serial bus that includes three lines: Serial Clock (*SERCLK*), Serial Data (*SERDAT**), and System Reset (*SYSRESET**). *SYSRESET** is used to initialize all modules on the serial bus. *SERCLK* is driven by a high_current totem-pole driver from the SERIAL CLOCK module, of which there is one per system. The *SERCLK* has a cycle time of 312.5 nanoseconds. One data bit is transferred per *SERCLK* cycle. *SERDAT** is a wired-OR line which can be driven with an open-collector driver by any module. A "one" on *SERDAT** results when one or more modules drive it low. A "zero" results when no module is driving *SERDAT**, so that the backplane terminating resistors pull the signal high.
The only object that can be sent over the serial bus is called a **frame**. The following frame types exist:

Data Transfer Frame
Semaphore Set Frame
Flip-Flop Set Frame
Flip-Flop Reset Frame
Token Passing Frame
Canceled Frame

All frames but the first and last convey one bit of information. Usually this bit is used to set or reset a flip-flop or register bit. The flip-flop (register bit) can be interrogated by the local processor or may interrupt the processor. Semaphore Set and Token Passing frames are special in that they are turned into cancel frame when illegally used (e.g. set operation on already set semaphore). A frame is composed of subframes and optionally a data part as depicted in figure 3.2.

	Header	Frame Type	[Data]	[Frame Status]	Jam Bit
nr of bits:	26	3	(max 32x8)	3	1

Figure 3.2 VMS Frame Structure

Every frame contains all subframes depicted except for the **canceled frame**. This frame lacks the **Frame Status** subframe. The cancelled frame does not convey any information and is therefore not further discussed.

The exact contents of the several subframes is not important in this context. Their accumulated length however is. The total length of all subframes, excluding the Data subframe, is 33 bits.

The attractions of the VMSbus are multiple, we mention:

Intelligent Semaphores

The VMSbus makes it possible to have "Intelligent Semaphores". Intelligent Semaphores as defined in the VMSbus specification are distributed semaphores of the binary type. Every board that is concerned with the semaphore has its own local copy of it (in fact there is no 'original'). As usual two operations are possible on the semaphore, the set and reset operation. The VMSbus guarantees that a (re)set operation (re)sets all copies in an indivisible action. Simultaneous attempts to set a semaphore are serialized by the arbitration of the bus. A requester of insufficient priority retires from the bus upon detection of a higher priority requester. An attempt to set an already set semaphore will be canceled by the receiver(s). (The receiver(s) turn the setting frame into a canceled frame by forcing the *SERDAT** line low during the Frame Type field transmission: a Frame Type with value '111' identifies a canceled frame).
When a processor on a board needs to set the semaphore it first inspects the local copy. If the semaphore is set it refrains from trying to set it via the VMSbus. This eliminates useless loading of this bus. If the semaphore is not set the processor performs the (VMSbus)set operation to set it.

In case the local copy is set the processor has two options. It can loop until the local semaphore gets free. This is a busy form of waiting typically used on "short-term" semaphores normally used to protect data structures in main memory. Alternatively the local semaphore can be made to interrupt the processor as soon as it gets reset. This is more efficient for "long-term" semaphores typically used for controlling access to a physical device like a printer. For this last case the hardware could even contend for the semaphore independently and interrupt the processor the moment it has acquired the semaphore.

Quoting from the specification [VMS83]:

> "For "long-term" semaphores the VMSbus offers a great improvement. A processor can turn the entire operation of setting the semaphore over to on-board serial bus hardware, and be interrupted only when the semaphore has been set and the associated resource is actually available."

Still we need to be critical about the speed with which a semaphore (and any flipflop by the way) can be set through the VMSbus. The frame to set a semaphore consists of 33 bits (that is: a frame without data). These take 33 times the clock time (312.5 nano seconds), equalling about 10 microseconds. In 10 microseconds a modern RISC processor at 25 MHz executes 250 of its instructions! This means that we indeed should apply this type of semaphore with slow devices only. Multitasking processors working with these semaphores will very likely be made to switch processes when setting such a semaphore, even if it is free!

Token Passing

Whenever in a system a group of interchangeable resources exist it may make more sense to have a **token** in the system for each of these resources than to use semaphores that have to be polled until one is found to be available. Users of the semaphore pass tokens that are sent to them on to their successor user when they do not need the resource at that particular time. Otherwise they retain the semaphore, preventing other users to use the semaphore. The VMSbus provides up to 1024 tokens to be created and passed from board to board with very little software overhead. The VMSbus also guards against duplication or loss of a token.

Low Priority Data Transfer

Though not particularly well suited for fast point to point communication of data, the VMSbus is capable of transferring data between two processors. It can thus be used for inter processor communications of a low priority. This priority should then be a parameter to IPrC calls or should be deduced from the type of messages sent. Sending low priority data over the serial bus takes some of the burden from the parallel bus. The bus could even be used for transport of (non-graphical) terminal data. The data rates involved there are rather low as are the amounts of data.
To mitigate the low bit rates, the bus can be tailored to broadcast messages in a very efficient way. Multiple receivers can simultaneously pick up a message from the bus. In fact this feature also underlies the intelligent semaphores already mentioned. There a special event (set or reset semaphore) was broadcasted, here other events or data may be broadcasted.

Fault Tolerance

Events communicated over the VMSbus may set or reset plain flipflops at every receiver end. In this way local interrupts can be generated or processors may be halted or reset. This adds significantly to the system ability to recover from errors.
The fact that the serial bus provides a secondary path for IPrC also adds to that property. Suppose for example that a processor breaks down in such a way that it blocks any transfers on the main bus. To reset this processor might help but cannot be performed without the use of the serial bus. (One might even make the hardware such that the parallel bus drivers are disconnected on remote command.).

The VMSbus is specified as an integral part of the VMEbus precisely for support of multiprocessor systems. The reason that it stays undiscussed in many other works dealing with both VME and multiprocessors probably is its speed. This speed is rather low, 3.2 Mbit per second is the maximum bit rate. (Remember: It is a serial bus). The effective bit rate however isn't by far that high. A lot of overhead is incurred by the mandatory signalling frames that must be used on the bus. For example to transmit **N** bytes over the bus an overhead of 33 bits is present. **N** may range from 1 to 32. At best the overhead will be as low as about 13 percent for **N** equalling 32. At worst the overhead is 413 percent, that is, over four times as many signalling than data bits are sent. In this picture we did not consider collision of frames, cancelled frames or loss of synchronization by which even more overhead is incurred. In the case of a loss of synchronization (probably very rare!) the bus is "jammed" for at least 512 cycles which amounts to 160 microseconds.
Hence the VMSbus is not the most suitable bus for pure point to point data transfer at high rates. Its attractions must lie elsewhere.
For the Event Frames the situation must be judged differently. Indeed, the overhead is 3300 percent here but we have the possibility of broadcasting and we have the security aspects mentioned that ease the pain. Providing the security aspects, like prevention of token-duplication etc., would bring extra costs with methods like semaphores. So, Event Frames are not as inefficient as they seem to be.

Recalling paragraph 2.2 we see that, when using the VMEbus, the *processors-memory* modules communications either use the VMEbus or are local to a board. A third alternative is to use a VSBbus to bypass the VMEbus. For our purposes this last alternative is equal to local memory. For memory connected via the VMEbus we thus have the time-shared bus situation with its inherent bandwidth limit. For the VMEbus the bandwidth is 40 Mbytes/sec which is not large enough for a multiprocessor to be efficient. Since *processor-processor* communications will also use the VMEbus, the urge to reduce VMEbus traffic is even greater.
Because the use of the VMEbus was an a-priori choice we will have to find solutions within its possibilities.
The process of finding solutions for the traffic problem is strongly influenced by the way the memory map is structured. Solutions for the *processors-memory* communications are therefore discussed in the next paragraph.

3.1.2 Memory Map

The virtual level of memory as discussed in paragraph 2.1 is entirely determined by the operating system. Since we have chosen Unix System V we know how the memory map for each process will look like:

A map as used by a process consists of a region of code in the lowest portion of virtual memory. The code region can be shared with other processes. The code region is fixed in size. The data region, containing static and dynamic data, is therefore placed directly on top of the code region. At the upper end of virtual memory the stack segment is placed. When the stack grows it does so towards lower addresses, when the amount of dynamic data rises the data region grows towards higher addresses. Note that when data must be shared between two processes shared memory must be explicitly requested via the System V IPC package.

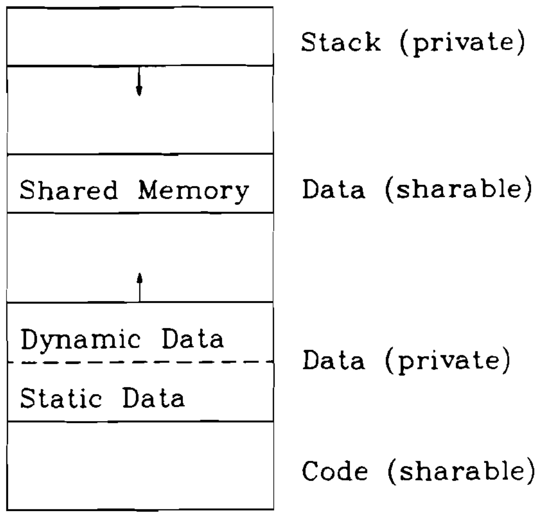


Figure 3.3 Memory Map Unix Process

At the physical level we may study a number of memory architectures in order to compare them and subsequently choose a suitable one from them. In a multiprocessor that has a global bus, usually other, local, buses exist too. Memory modules can be connected to the local bus, to the global bus or to both. We will refer to memory that is connected to a local bus as Local Memory since it will normally reside on the same board as the other components connected to that bus. Consequently memory that is connected to the global bus only is referred to as Global Memory. Using these terms we can set up a number of architectures differing in the location and accessibility of the memory modules. In the figure below 4 of these architectures are presented in one single figure. The figure uses the MSBI (Master, Slave, Bus, bus Interface) notation defined in [CONT85].

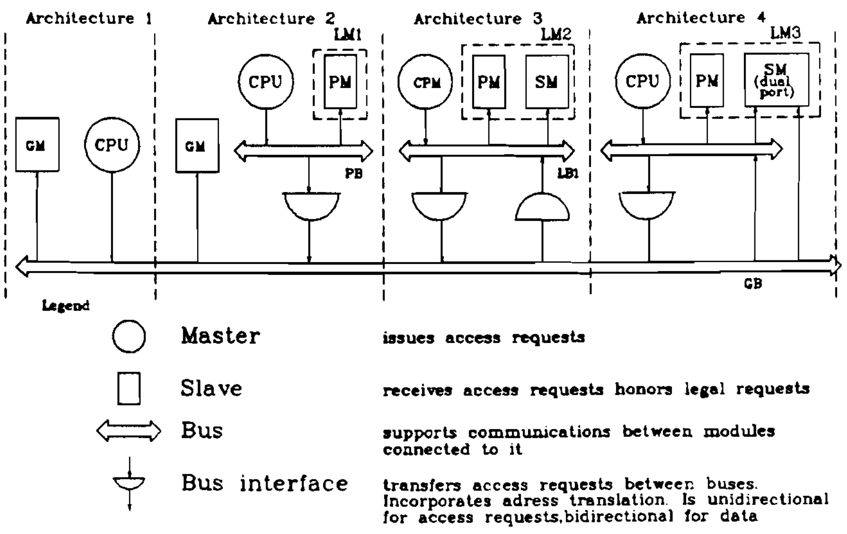

Architecture 1
Architecture 2
Architecture 3
Architecture 4
LM1
LM2
LM3
CPU
PM
SM
SM (dual port)
GM
PB
LB1
GB
Legend
Master
issues access requests
Slave
receives access requests honors legal requests
Bus
supports communications between modules connected to it
Bus interface
transfers access requests between buses. Incorporates adress translation. Is unidirectional for access requests,bidirectional for data

Figure 3.4 Location of Memory Modules

The LM1 module represents Local Memory that can only be used as private memory since no access path from the global bus exists (it is physically private). The LM2 and LM3 modules represent Local Memory too, they can serve as private memory and/or as shared memory. To access the Shared Memory in LM2 from the global bus ownership of the local bus is needed, preventing the local processor to access any memory. Access to shared memory in LM3 causes much less contention, the local processor is no longer denied access to its private memory. Global Memory is represented by the GM module. This memory too can be used as (logically) private and/or shared memory.

In [CONT85] an extensive comparison has been made between architectures 2,3 and 4 and a variation on architecture 4 not presented here. The conclusion from [CONT85] is that architecture 4 gives best performance in all cases, where system load and number of processors were varied. We will therefore adopt architecture 4, replicated in full below. This choice has the fortunate side-effect that all other architectures presented above can be emulated by the chosen one so that diverse experiments can be done.

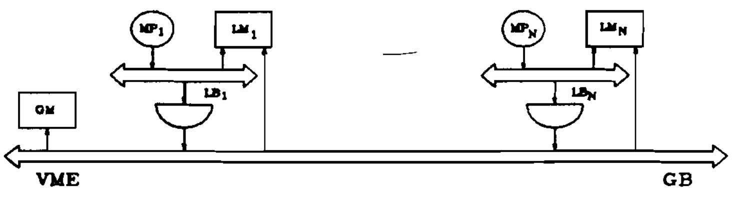


Figure 3.5 Chosen Location of Memory Modules

We added a global memory module to allow for a processor independent shared memory to store the operating code and data structures in. First of all this is where they will be in the early days of the project and second it is where they will be during several discussions in this report.

Eventually this Global Memory module may be eliminated. The Local Memory modules can be used as private memory, as a cache, as shared memory or as a combination of these.

3.1.3 Interrupts

The target system will, since based on the VMEbus, provide us with VME interrupts and Location Monitors as discussed in 3.1.1. Both mechanisms can be used to implement a subset of the traditional single processor interrupts. The VME interrupt mechanism is not symmetric while the Location Monitor mechanism is (or at least can be made to be).
To comply with Design Strive 3.1 (the symmetry strive) we will not use the VME interrupts to handle events that may occur while the system operates normally. Instead, we will use the VME interrupts to handle abnormal events such as hardware faults:

Condition 3.3.1
VME interrupts shall only be used for the signalling of hardware faults.

How the remaining events (e.g. IO events) are dealt with is discussed in paragraph 4.3.

3.1.4 Inter Processor Communication

As explained earlier IPrC must support various type of communication. One of these types is inter process communication (IPC). Unix System V contains an IPC package that provides for message passing, shared memory and user level semaphores. The IPC package uses data structures in main memory. By placing these structures in shared memory and protecting them against corruption in a multiprocessor environment the existing IPC mechanism can be used in the multiprocessor too. For this reason we will not discuss IPrC to support IPC. The main remaining function of IPrC then is the MP-IOP communication. We will discuss some methods to support this communication. The mechanisms will be made as universal as possible in order to keep the possibility to perform IPC with them. In this way we may eventually discard the System V IPC package and replace it with our own thus unifying the message system.

Apart from the restrictions already present due to earlier choices (VME for example) no further limits are imposed on the IPrC mechanisms on beforehand.

3.1.5 Operating System (Unix^R System V)

Due to the existing knowledge about Unix in our group and its availability to the academic world, Unix has been selected as the operating system for our multiprocessor system. Unix is not the most suitable operating system to be implemented on multiprocessor hardware. This is due to the fact that Unix has been specifically written for a single processor system thereby using the simplest techniques possible to protect shared resources from corruption. Exactly these techniques pose some problems since they do not protect these data structures on a multiprocessor system as will be explained in this paragraph and the section 4.2. Nevertheless the choice for Unix is justified since it also has a number of advantages. Among these are the availability of lots of utility and application programs, the relative ease of migration over processor types and the trend for standardization of Unix-based operating systems as apparent from the existence of standardization groups like POSIX, X/OPEN and the OSF (Open Software Foundation).

Since the construction of a multiprocessor system under Unix starts from single processor Unix, a compact description of single processor Unix is given in the next paragraph. The adaptations to be made in order to arrive at the target multiprocessor system are discussed in chapters 4,5,6 and 7.

As for the VMEbus we do not intend to fully describe the Unix operating system for uniprocessors here. Numerous descriptions of Unix have been made ([BACH86], [LION77] for version 6, etc) and some are readily available to consult (e.g. [BACH86]). We will only describe those parts of single processor Unix that are discussed for a multiprocessor in subsequent chapters of this report and that are not sufficiently treated there.

We will present the system startup, system Call interface and mutual exclusion, however all in a very brief manner.

3.1.5.1 System Startup

We assume an intact file system called "root" to exist on the "system disk". In "root" a file called "unix" is present that contains the object code for the operating system. If we then power up the system, a bootstrap program will start running from ROM[1]. The bootstrap loader loads the contents of block 0 of the "system disk" in main memory. The block 0 contents is the object code of a second loader program that is given control by the bootstrap program and that will load the operating system code from file "unix" in main memory from address 0 upwards. Control is then transferred to address 0 and initialization of the system starts.
After initializing the peripheral devices, MMU and OS data areas, the MMU is 'switched on'. A number of variables and data structures are then initialized, for example: 'maxmem', 'swapmap', character buffers for serial IO devices, block buffers for block devices (e.g. disk), inode table, etcetera.

---

[1] We present the most common procedure.

Following this the first process is manufactured artificially. (That is, it is the only process that is not created using *fork*, the system call meant to create processes). This process 0 spawns a second process with Process IDentifier (PID) 1 and puts itself to sleep to allow this new process to run. (It has by then become the *swapper* process, the process that is responsible for moving processes to the swap space on disk when main memory is full (and vice versa)). Process 1 begins with copying an array of code from the kernel space into its own code region and transfers control to it. The code now running overlays the address space of process 1 with the program stored in "/etc/init" and control is transferred to this program. (The reason for this indirect way of doing is explained in [BACH86]). The program now running as process 1 is responsible for the initialization of new processes. Therefore process 1 is known as *init*. *Init* reads the file "/etc/inittab" to find out which processes it should spawn. Typically it spawns *getty* processes for each terminal in the system. The *getty* process allows an (authorized) user to log in and starts a login shell. When the user stops working (properly) the *getty* process exits and *init* will re-spawn one. The moment the *getty* process exits it may still have child processes. These are assigned *init* as their parent. If the user did not indicate otherwise *init* will terminate the child processes. As an aside, both process 0 (*swapper*) and process 1 (*init*) may spawn daemon processes. The daemons created by process 0 always run in kernel mode while those created by process 1 are user-level processes. Daemon processes provide services that are necessary throughout the lifetime of the system. For example the page-reclaiming process *vhand* is spawn by process 0 while printer-spoolers and such are spawn by process 1.
This concludes the system startup sequence. We will return to this subject in section 4.5, then for the multiprocessor case.

3.1.5.2 System Call Interface

A process that requires a specific service from the operating system does a socalled **system call** (or supervisor call).To the process and the programmer it appears as if a system call is just another function call. However this is not so, as in the execution of a system call the processor mode has to change from user to supervisor (or kernel) mode. This change of mode happens only on certain events as indicated in the figure below:

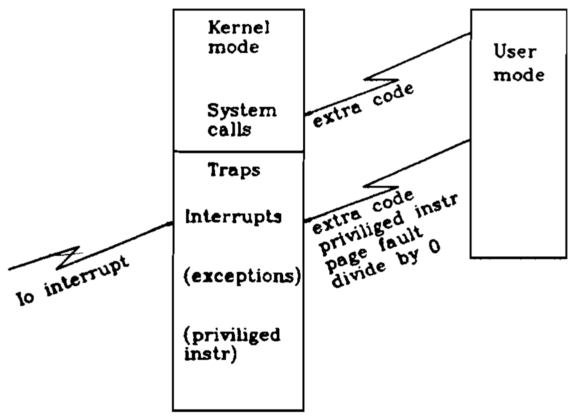


Figure 3.6 Enter Kernel-Mode

One of the processor's instructions is a socalled **trap** or **extra code** which causes an internal interrupt to occur and in this way switches the processor into the supervisor mode. The Unix system calls are implemented using a library of functions, the **C-library**. The functions in this library pass the Trap-Handler (TH) a number that identifies the system primitive that must be invoked. The TH then copies the appropriate amount of parameters from the users stack to the *u_area* and calls the system routine requested. When the system routine returns the TH inspects by looking at an error field in the *u_area* (u.u_error) whether errors have occurred. It then manipulates registers in the user context, that was saved when the trap occurred, in such a way that they either present the return value(s) of the system call or an error indication. The TH then returns to user mode where the C-library routine continues. The library function in its turn inspects the return values and returns a value to the user process.

3.1.5.3 Mutual Exclusion

Mutual exclusion is achieved through two techniques. The first technique is a very simple one and covers the entire spectrum of possible situations, but one, where critical regions of code must be protected. The only case where the first technique is inadequate is when interrupts occur. Then the second technique must be applied.

The first technique is to split up the operating system code in two parts, the kernel part and the non-kernel part. The kernel contains, among other things, all code that deals with resources shared by processes. The non-kernel part is formed by all other code. Processes will in general execute mixed code, that is parts of their code is kernel code the rest isn't. When a process executes kernel code it is said to execute in kernel mode (sometimes called supervisor mode) otherwise it executes in user mode. Situations will arise where 2 or more processes wish to modify the same shared data structure. We say that these processes execute or wish to execute a critical section of code. We denote this type of critical section **Process - Process critical section** or for short **P-P critical section**. All critical sections are kernel code. The obvious way of protecting the data structures now is to let processes that enter the kernel code never give up the processor until they leave this kernel code again. This guarantees that only 1 process is modifying the kernel data structures at a time and, equally important, that once the modification started it is fully completed by that process. This is why the kernel is said to be non-preemptable. The reader will understand from the above that preempting the processor from a process executing in user mode is perfectly legal. In fact the Unix's process scheduler does so all the time.

One problem will arise when using the above technique and that is when an interrupt occurs while a process is in kernel mode. Such an interrupt could invoke an interrupt handler that also modifies the kernel data structures. A section of code critical in this sense is called a **Process - Interrupt Handler critical section** or **P-IH critical section**. This is where the second mutual exclusion technique is applied: When running kernel code interrupts are disabled just before entering a critical region and they are re-enabled immediately after leaving the critical region. In this way mutual exclusion is guaranteed.

Clearly these techniques are insufficient in a true multiprocessor system where more processors are allowed to run kernel code at a time and where more than one processor may be interrupted at the same time. The consequences of this will be explored in the next 4 chapters. One observation can already be made at this time. In the multiprocessor case there will also exist **Interrupt Handler - Interrupt Handler critical sections (IH-IH critical section)** when indeed more than one processor experiences interrupts. When the level of interrupt becomes important we will denote such a critical section as follows: IH(l)-IH(m) critical section.

3.2 Configurations

As said in the opening of this chapter, this chapter only serves to restrict in a coarse manner the range of possibilities we allow for the several aspects of our target multiprocessor. This is also true for its architecture. At this point we only pose the restriction that we will always use separate IO processors to do the IO. When a board contains an MP and global IO devices it should behave as an MP **and** an IOP to the rest of the system.

A number of architectures, all of which have separate IO processors, will be discussed in the next chapters. Finding architectures for a multiprocessor running the Unix operating system is strongly influenced by the way Unix protects its critical regions. We discussed the details in 3.1.5. One either stays close to single processor Unix's mutual exclusion methods or one uses completely different methods. The first road leads to the following two architectures:

Master-Slaves Architecture
Uni-Kernel Architecture

In these two architectures only one processor is able to run in kernel mode at a time.
The second road leads to the following two architectures:

Multi-Kernel Shared Memory Only (SMO) Architecture
Multi-Kernel Private and Shared Memory (PSM) Architecture

In these two architectures mutual exclusion is achieved through the use of semaphores. The difference of having or not having private and/or local memory is used to discuss which parts of the operating system could be conveniently and/or profitably transferred from shared to private memory and which parts better stay in shared memory.

4 Master-Slaves Architecture

4.1 Introduction

When converting the single processor Unix operating system to a multiprocessor version problems arise in different areas and at different levels.
C.H.Russell and P.J.Waterman [RUSS87] have identified a number of areas for the UNIX operating system where modifications must and/or may take place. These areas are:

a. Low-level mutual exclusion
b. Higher-level mutual exclusion
c. Multiprocessor task scheduling
d. Efficient inter process communications
e. Flexible memory management/cache coherency
f. Easy reconfigurability and expandability.

The areas a and d are discussed in detail in the sequel. In their discussion of area b Russell and Waterman already refer to the sleep/wakeup mechanism as a tool to achieve the Higher-Level mutual exclusion. We will rely on this method and will not detail area b. The task scheduling is detailed for the Multi-Kernel configuration in chapter 6. For the Master-Slaves configuration discussed in this chapter we only briefly look at it in paragraph 4.6. The area's e and f are left for further study: For the memory management we will, for the time being, use the single processor version as is. No caches will be applied in our system until decided otherwise (Caches must eventually be used to avoid the VMEbus of becoming a bottleneck):

Condition 4.1.1
No Caches are used (yet).

Reconfigurability and expandability will not be discussed as separate items in this thesis.
The considerations above leave us to discuss the problems encountered in area's a,c and d. For the case of the Master-Slaves architecture we do so in paragraphs 4.2, 4.6 and 4.3. We further discuss Inter Processor Communication in 4.4 and the System Startup in 4.5.

4.2 Mutual Exclusion.

The solitary reason for the existence of the Master-Slaves architecture is the large resemblance it has to a single processor system. The resemblance makes that a Master-Slaves system can often do with single processor constructs with hardly any modification. The modifications that have to be made tend to be less complicated than with 'Multi-Kernel' architectures. This particularly is true for the Unix-type mutual exclusion problem. By shifting all actions that suppose a single processor into the master processor (MMP) mutual exclusion gained through single processor methods is also sufficient for the Multiprocessor.
In the case of a Master-Slaves system running Unix, this means that all system calls must be handled by the MMP. The MMP is the only processor in the system that may run kernel code.

Since only the kernel will deal with IO, only the MMP will deal with the IOP's. The master processor therefore is the only one that needs to handle IO interrupts. This does not mean that a slave processor (SMP) is never interrupted or trapped. An example of an interrupt to a slave is the clock interrupt. An obvious trap to occur in a slave is the system call trap. Also exceptions like a divide overflow may occur in a slave. An SMP may also execute portions of kernel code that do not need special protection.

Before discussing the implications of the above let us first define a number of terms to ease the discussion. First of all let us make a distinction between the *actions performed by the kernel* on one hand and the *kernel mode* on the other. In single processor Unix these two terms are often used to denote the same namely that the 'kernel' is running. We will use the term *supervisor mode* to indicate the processor mode and *kernel mode* to indicate that the kernel is running.
Considering the above the situation experienced by the master processor hardly deviates from that experienced by a uniprocessor. When executing in kernel mode it is sure it is the only one and it is sure it will not be preempted. Also it knows that increasing the processor execution priority to block out interrupts does work since it is the only one receiving critical interrupts.
The slave processors never execute within critical regions so no mutual exclusion problem arises there at first sight.
Looking in greater detail a few problems become apparent. First of all slaves **do** run in supervisor mode in spite of the (first) impression one gets from a Master-Slaves architecture that they don't. Furthermore the master and the slaves interact via the process table. The slaves must request kernel mode services from the master. It does so by marking a process as requiring such service in the process table (placing it on the 'Master Queue'). The master will subsequently select it at some time for execution. The process table must therefore be protected via a special mutual exclusion mechanism (e.g. a semaphore). [Note that the processes can be kept on more than one queue so that the exclusion can be done per queue for efficiency].
On the other hand the master will have to 'dispose' of processes that return from kernel mode to user mode if other processes wish to run in kernel mode (placing it on the 'Slave queue'). The master will mark its process as runnable in user mode in the process table and select a new process from the Master Queue. Again it will manipulate the process table in parallel to other processors.
Possible race conditions are avoided through an intermediate *null* process on the slaves and *process 0* on the master as explained in 4.5.

Slaves sometimes run in supervisor mode. Supervisor mode is entered as the result of a number of events. The main events are:

- Interrupts
  - Clock
  - Page Faults
- Traps
  - System Calls
  - Illegal Instruction etc.

Most of these events will need kernel mode execution. A number will not. The code that will be executed by the slave when in supervisor mode must handle all events that do not need kernel mode processing and it must furnish a slave-master interface mechanism for those events that do need kernel mode processing.

The Master-Slaves architecture has been extensively treated by Annot and Janssens [ANNO85]. We refer to their work for further details about the topics just discussed.

Having achieved mutual exclusion easily one could rest in content. However the next paragraph will show that in some cases it is possible to do more. The Process-InterruptHandler (P-IH) critical regions can in part be eliminated.

4.3 Interrupts

Many engineers that have to find new ways of applying existing methods (or mechanisms) or that have to find new methods to solve old problems have difficulty in posing the fundamental questions. An excellent example is the interrupt mechanism. This mechanism has traditionally been applied for all kinds of (a)synchronous events on a uniprocessor. One may think of clock interrupts, IO interrupts, DMA interrupts, error interrupts, trap interrupts, exception interrupts etcetera. Let us define a new term to reference the events underlying the traditional interrupts: **i-events**. Given the existing interrupt mechanism and its traditional use tempts one to use it on a multiprocessor for exactly the same purposes. However at this point one should ask the fundamental question: 'Do I use interrupts justly for all these i-events ?'. This question should be asked to explore the extra possibilities a multiprocessor has in dealing with the events. These extra possibilities might be easily overlooked. We will find that in a uniprocessor with separate IOP's some of these extra possibilities exist too.

An event traditionally handled via an interrupt is the IO ready event. We would wish to deal with this event without the use of interrupts for two reasons. The first reason is that if we can do without the interrupt we don't have P-IH critical regions for this particular case anymore. As discussed in paragraph 3.1.5 this kind of critical region is somewhat awkward to protect. The second reason is that interrupt handlers nearly always run in the context of an unrelated process thereby obstructing a fair accounting. Further the method that we propose will prove to be more efficient than the traditional method using interrupts.

Knowing that elimination of interrupts could bring profit in mutual exclusion effort, accounting and efficiency, one could seek to eliminate all interrupts. This is not possible. For those interrupt types for which it is, it is not always wise. For those types of interrupts for which it is possible and wise, the elimination must be complete. A mere reduction of the rate of interrupts of that particular type will not be enough.

Let us now look at the way the multiprocessor may handle IO ready events in the case of canonical (= what you type is formatted before sent to process) terminal IO. Suppose a process on MP1 wishes to read a line of input from terminal tty4 controlled by IOP2. When no data is available the process will sleep. As soon as the user types a line at the terminal the process must wake up. Waking the process up can be done in a number of ways. When using interrupts, the interrupt handler may wake up the process. Or, the interrupt handler starts up a 'kernel mode only' process that in its turn wakes up the reading process. These are the two traditional ways of doing it. The next method can be applied in processors (uni- and multi- alike) that employ IOP's. This method lets the IOP wake up the reading process. Hence, interrupts are avoided and overhead reduced. This way of doing can also be applied for other forms of IO like for example

disk IO. Along the advantages there also exists a major drawback. The MP-IOP interface is not clear-cut anymore. An engineer who is adding an extra IOP must dig quite deeply into the operating system, reading and writing data structures that are critical. He or she must make the software running on the IOP specific for this type of operating system and is not, or less, able to use standard IOP software.
The last method is in essence a form of inter processor communication. For this reason we term this method *Implicit IPrC*. *Explicit IPrC* then is the communication between processors using an explicit universal communication mechanism in which there is no interference between the transport of the messages and their contents (Message Passing). In the next paragraph these forms of IPrC are discussed in greater detail.

It is hard to estimate the profit gained from the elimination of interrupts. We therefore consider a number of events, traditionally handled through interrupts, and look for ways to handle them on our multiprocessor possibly without the use of interrupts at the MP level.

The i-events can be classified as follows:

Hardware faults:	Bus Error
	Defective Components
	RAM ECC Errors
Exceptions:	Illegal Instruction
	Memory Protection
	Page fault
	System Call trap
	Divide by Zero
	Trace trap
	Breakpoint
IO:	IO-ready
	IO-error
Clock:	Clock
System Calls:	System Call trap
Miscellaneous:	Break-Switch

From the above we take a detailed look at the clock interrupt and the IO interrupts. The system call interface is studied in detail elsewhere (3.1.5.3).

4.3.1 IO Interrupts

There are two main types of IO in Unix. These are block type IO and character type IO.
The typical interrupt handler of the block IO type does the following essential actions:

In case of error: Mark error in *process table entry*.
Write error message to *console*.
Mark IO done on *buffer*.
Release *buffer* if IO is asynchronous.
Wakeup (in *process table*) anyone waiting for this IO operation.

Inspecting these actions we see that there are no problems in having them performed by the IOP. The only consideration is the familiar mutual exclusion theme for the IOP will have to access the *process table*, *console* and *buffer queue*.
Typically there are two interrupt handlers for character IO. One interrupt handler is the transmitter interrupt handler (*klxint, pcpint, lpint* or *dlxint* etc.). This handler wakes up processes that desire to write to the character device as soon as the number of characters in the output queue is less than a low water mark. Other actions are also initiated by the interrupt handler by calling a *{device}start* (e.g. *dlstart*) routine. In an MP-IOP configuration the actions performed by this start routine will be performed locally by the IOP. (That is, when the MP-IOP interface is formed by the *clists* as proposed by Bregman [BREG83] and Annot and Janssens [ANNO85]).
The above means that the only action crossing the MP-IOP border is, apart from reading and writing the clist queues, waking up processes. This can easily be done directly by the IOP as observed before.
The second interrupt handler for character IO is the receiver interrupt handler (not present for line printers and such.) This interrupt handler (*klrint, pcrint* etcetera) accepts a character from the hardware and puts it on the appropriate *clist* queue. It then wakes up all processes waiting for data to arrive on this queue. In certain cases, e.g. terminal IO, signals need to be sent to processes on receipt of special characters ("quit","delete" etc.).
Again all of these actions can be performed by the IOP directly.

Summarizing we may say that

- it is possible to avoid IO interrupts at the MP level by having the IOP's perform all IO concluding actions
- the main actions are to wake up processes and/or to send them signals. All data transfer is done through buffers in shared memory.

4.3.2 Clock Interrupt

The Clock Interrupt is extensively discussed in paragraph 5.2.2. The actions that are done by the SPUX Clock Interrupt Handler are divided there in a global and a local set. The global actions are pertinent to the entire system, the local actions are related to the current process on the local processor. In 5.2.2 the processor that performs the global actions is an arbitrary one from the set of processors. With the Master-Slaves architecture it is obvious that the Master will perform the global set of actions. For the actions contained in the sets see 5.2.2.

4.3.3 Summary

The major part of events handled through interrupts on a uniprocessor will be handled likewise on a multiprocessor. For some events (e.g. clock) actions must be taken that in part are related to the entire system and in part to a running process. One processor must be entrusted to perform the global actions. We propose this task to float, being assigned to a processor dynamically.
When IOP's exist in the system, it is possible to eliminate IO interrupts at the MP level. IOP's must then be allowed to manipulate kernel data structures. The advantages are a fairer accounting, less overhead and, even more important, the elimination of P-(IO)IH critical regions. A main disadvantage is the obscure MP-IOP interface, making it hard to maintain the software and to add IOP's.

4.4 Inter Processor Communication

As outlined in paragraph 2.6 Inter Processor Communication is an important item. Both speed and amount of traffic are key factors here.
Inter Processor Communication takes place among Main Processors on one hand and between Main Processors and IO Processors on the other. (Communication among IO Processors is rare. An example could be a bridge between two LAN IOP's. We will not look into this type of IPrC). These two types of IPrC have their own purposes.
For MP-MP IPrC we have the Inter Process Communication that must be supported. This includes pipes, signals, messages, shared memory regions and semaphores. (The Unix System V IPC package will be used. See 3.1.4). Other uses of MP-MP IPrC are in starting up the system (Who is in the system and what can he do for me?), and in Error Recovery where every MP must be made aware of any malfunctioning MP and where restarts or resets and the like should be possible.
MP-IOP Inter Processor Communication is used to initiate and conclude IO tasks. Above that it is also used during system startup to complete the configuration (See 4.5). Finally when hardware faults occur on an IOP the MP community has to be informed about it and vice versa.
Since more and more distributed computers are developed we should have an IPrC mechanism that can be used to that purpose also.

The requirements for IPrC are:

- One universal interface to rest of kernel.
- Fully Transparent to the user at the IPC level.
- Prioritized Communication (for support of communication channels with different speeds)
- Different *SubMechanisms* for local and remote traffic (efficiency)
- Global Name Spaces

Figure 4.1 clarifies the notion of *submechanisms*.

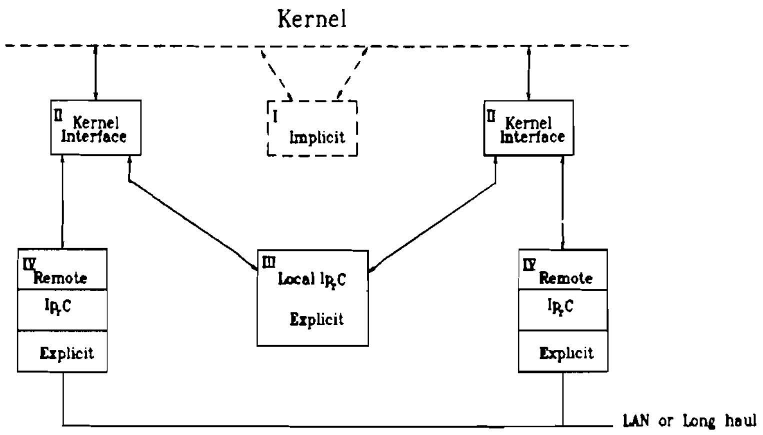


Figure 4.1 IPrC Submechanisms

The blocks II, III and IV are used for message passing. Blocks III contain the submechanisms for local traffic (internal to the multiprocessor), blocks IV contain the submechanisms for remote traffic (e.g. distributed computing) and blocks II contain the universal kernel interface as well as the necessary means to distinguish between local and remote addresses and to invoke the appropriate submechanism. To highlight the fact that two or more processors are involved, blocks II and IV have been drawn twice. In fact they are instances of one mechanism.

Finally block I. This block is somewhat special in that it provides IPrC through the shared use of (usually existing) kernel data structures. Since IPrC is only a secondary use of these data structures this form of IPrC is called **Implicit IPrC** as before.

We will not discuss the mechanisms used for remote IPrC (blocks IV) because they do not fall within the scope of this report. However since local communications in a multiprocessor can also be done through block IV mechanisms they can be important for a multiprocessor too. The interested reader may find more on this topic in [BERK83] which treats the BSD *sockets* from the University of California at Berkeley or in [RITC84] which treats system V *streams*.

In the next paragraph we take a look at the mechanisms used in blocks III which we termed **Explicit IPrC**. Finally we look at Implicit IPrC in paragraph 4.4.2.

4.4.1 Explicit IPrC

Explicit IPrC as we define it is more commonly known as Message Passing. This mechanism is characterized by a strict separation of the message transport function and the message contents as mentioned earlier. The message contents nor size (within system limits) are restricted. The beauty of this method is that it provides one universal framework for every communication need (Excluding shared memory and, for efficiency, semaphores and file transfer). The method is suitable for every hardware configuration.

Before discussing a number of implementations let us define a mailbox. A mailbox is a data structure and an associated set of operations, that has a built in synchronization mechanism to synchronize message writing and reading processes or processors. Optionally it can hold queues for writing and reading processes, a number of message slots, a priority field per message, a type field per message and more. As said the semantics of the contents of a message slot in the mailbox are the user's choice. It may be the message itself but it may also be a pointer to the actual message.
A mailbox that merely contains the synchronization and possibly the queues is no more than a semaphore. (This occurs for example when only 1 message slot exists with a fixed address).

We will consider two *by reference* implementations and three *by value* methods.
The first *by reference* method will be a 'Unix System V IPC'-like implementation. (In Unix data transfer between user and kernel is by value for protection reasons: Data is copied from user space to kernel space or vice versa. We do not consider this copying to be part of the IPrC mechanism. Once the data is in kernel space it is passed by reference.). Paragraph 4.4.1.1 gives a detailed description.
The discussion of *by value* implementations starts with the description of IPrC as it is used in our THE-KUNix system [BOKH83]. We named it *Private Mailboxes* because every processor has its own mailbox in which it receives messages. The method is discussed in paragraph 4.4.1.2. In paragraph 4.4.1.3 we briefly touch upon the VMSbus as transport mechanism. Finally paragraph 4.4.1.4 brings us to the IEEE P1296 specification, a newly evolving bus definition. This bus, the Parallel System Bus (PSB) has special **standard** hardware to perform IPrC. We will investigate if we may employ techniques from this standard in our environment too.

4.4.1.1 Unix System V IPC-like Implementation

When we stick to the Unix System V type of implementation the IPrC could be as described below. Before that we first quote the X/OPEN group commentary on Unix System V IPC (so not IPrC!):

"The kernel extension interfaces relating to shared memory, semaphores and message passing are included in "XVS INTER-PROCESS COMMUNICATION" as a short-term mechanism to satisfy the immediate requirements for Inter-Process Communication facilities. However, they are machine specific and cannot be supported on all hardware architectures. The Group believes that a more generalized approach to the whole subject of Inter-Process Communication is required." [XOPE87a]

And from the mentioned "XVS INTER-PROCESS COMMUNICATION":

> "X/OPEN are considering the problem of generali IPC, which is not fully addressed by the System V IPC. For example, the shared memory functionality is hard to implement efficiently on multiprocessor architectures or across networks."[XOPE87b]

From these remarks we learn that groups like X/OPEN (and POSIX ?) are not fully satisfied with the System V IPC and that they will come up at some time in the near future with newly defined IPC services. We will have to keep an eye on that.

Before we proceed to discuss Unix-like IPrC let us introduce some new data structures. First there is the **configuration table** data structure. This structure and those related are given for a system supporting dynamic configuration. No attention is paid to the efficiency of operations on these structures.

```
struct
{
ushort                  sys_id;        /* The system identification     */
(or:   char             *sys_id[8];    /* system identification         */)
struct      sem         configsem;     /* Mutual exclusion semaphore    */
struct      entity      *console;      /* Pointer to system console     */
struct      entity      *entities;     /* Pointer to entities queue     */
} conf_tbl;
```

The *sys_id* could be implemented as a character string which is more in line with the higher level in the hierarchy and thus closer to the 'human level'. Of course one has to consider the naming domain in which the system is used.
The semaphore *configsem* is used to mutually exclude processors from the entity queue.
An **entity** is a resource of some kind that can be associated to a processor in the system either because it resides on the same board or because it is controlled by it. Examples of entities are disks, terminals, network ports, operating systems etcetera. (Normally one would have used the word *device* instead of *entity*. However the word *device* is already in use in Unix for more limited purposes). The *console* entity is the terminal that is used by the system administrator. The system administrator uses it first of all to boot the system. It is further used to print all operating system error messages to.
The *entities* identifier is used to denote a pointer to the first entity in the so called **entity-queue**. This first entity usually is the boot processor. Entities are kept on a linked list to allow for easy system reconfigurations.

The *entity* structure is as follows:

```
struct entity
{
struct      lentity_id  e_id;           /* local entity id.              */
char                    e_status;       /* entity status                 */
struct      kmsg_qhdr *kmsg_q;          /* kmsg queue for this entity    */
long                    kmsg_type;      /* kmsg type for this entity     */
struct      entity      *next;          /* pointer to next queue memb    */
/*                                                                       */
/*  Other information may be contained in this structure:                */
/*          e.g. The capabilities of the entities such as:               */
/*                      Terminal IO; Hard Disk IO; Network IO;           */
/*                      Graphics IO; etc                                 */
/*                                                                       */
}
```

where *lentity_id* (local entity id) is a unique identifier for the entity within the system. To address from outside the system, the *sys_id* must be included. The *e_status* field provides information about the present condition of the entity. Some possible values are *operational* (E_OP), *out-of-order* (E_OOO), *busy* (E_BUSY) etcetera. Pointer *kmsg_q* points to a kernel message queue header. Kernel messages to this entity can be linked to this headers queue. A kernel message queue may be shared by several entities. To easily distinguish between messages for different entities on a shared queue a *kmsg_type* is added. Messages for this entity must have this type. Finally pointer *next* points to the next entity on the entity queue.
The *lentity_id* structure is defined as:

```
struct lentity_id
{
ushort      board_id;       /* board identification      */
ushort      proc_id;        /* processor identification  */
ushort      unit_id;        /* unit identification       */
};
```

When a board is in the system with one main processor and several IO devices, it is evident that we need the *number of devices plus 1* unit numbers (when counting from 0 the highest number will thus be equal to the number of devices present). Note that operating systems (global or local) may also figure as entities.

Remark: When we would conform to Unix 'standards' the *sys_id* and the entities pointer would become global variables while the *entities queue* would become an array.

Back to our description of Unix-like IPrC. The idea is to have a kernel message (kmsg) queue (= mailbox) either for each entity, for each processor, for each board or for the entire system. Let us assume queues per entity. Every queue is headed by a *queue header* set up at initialization time. This *queue header* holds values concerning the queue as a whole. Figure 4.2 shows the entity queue table and attached message queues.

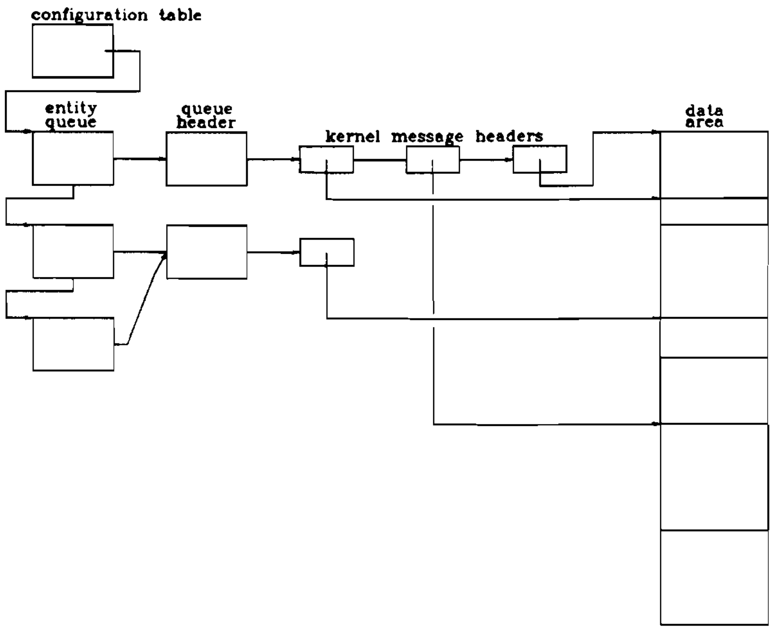


Figure 4.2 Kernel Message Queues

Below you see the possible declaration of the kernel message queue header. From it you may observe the structure of the queue headers (KMSG_QHDR). The queue headers are pointed to from the configuration table.

```
struct kmsg_qhdr
{
struct kmsg_hdr *kmsg_first;      /* pointer to first kernelmsg      */
struct kmsg_hdr *kmsg_last;       /* pointer to last kernel msg      */
ushort          kmsg_qnum;        /* nr of requests on queue         */
ushort          kmsg_qbytes;      /* nr of bytes on queue            */
ushort          kmsg_maxbytes;    /* max nr of bytes allowed         */
ushort          kmsg_lspid;       /* process id of last process      */
                                  /*    that sent a kmsg             */
ushort          kmsg_lrpid;       /* process id of last process      */
                                  /*    that received a kmsg         */
time_t          kmsg_stime;       /* time of last kmsgsnd            */
time_t          kmsg_rtime;       /* time of last kmsgrcv            */
time_t          kmsg_ctime;       /* time of last kmsgctl            */
long            kmsg_LM;          /* kmsg Location Monitor addr      */
}
```

In the kmsg_qhdr time stamps are kept for three out of the four associated kernel functions. The four functions are *kmsgget, kmsgsnd, kmsgrcv, kmsgctl.* They will be explained shortly. The stamps are kept for system monitoring purposes.

A kernel message header has the following structure:

```
struct kmsg_hdr
{
struct entity_id   kmsg_sentid;    /* sending entity id            */
ushort             kmsg_pid;       /* sending process id           */
time_t             kmsg_time;      /* time of sending              */
ushort             kmsg_prio;      /* kmessage priority            */
ushort             kmsg_state;     /* status: busy,new,scheduled   */
                                   /*          done,waiting        */
struct kmsg        *kmessage;      /* pointer to message           */
ushort             kmsg_size;      /* size of kmessage             */
struct kmsg_hdr    *next;          /* pointer to next message      */
}
```

The kernel message itself looks like:

```
struct kmsg
{
long               kmsg_type;      /* kernel message type      */
char               kmsg_text[];    /* kernel message text      */
}
```

The *kmsg_type* can be used to multiplex different types of messages onto one queue. This is for example necessary when only one system wide kernel message queue is implemented in order to address the correct entity. The actual message is *kmsg_text.*

The kernel will use a so called *map* to administer memory space for use by the kernel messages. A map is an array in shared memory. Each entry in the map registers a contiguous block of free space. An entry consists of two parts. The first part is the address of a block of memory, the second part gives the size of that block. When the map is created it holds only one valid entry of which the first part gives the starting address of the memory space while the second part gives the total amount of memory space available. Note that the size of memory can be measured in several units ranging from bytes to so called core clicks (256 to ... byte blocks dependent on the implementation). Allocation and de-allocation must also (de)allocate memory space in these units.

An example map is depicted in figure 4.3.

Suppose the available space is located at address 0 and consists of 100 units. Then we see from the map that the space from 0 to 5, 12 to 13 and 17 to 20 is in use somewhere. The entries in the map give the remaining free space.

6	6
14	3
21	79

Figure 4.3 Example Map

Four kernel functions are defined that operate on the kernel message data structures. The *kmsgget* function returns the kernel message queue identification of the kmsg queue used by a given entity. If such a queue does not yet exist, it creates one. When creating a queue it generates a new queue header using memory from the kmsg-map. The *kmsgget* routine is called at initialization time, in our setup once for every entity, to create queues. The memory used for the queue headers is thus drawn from the beginning of the map.
The syntax of the *kmsgget* function is

kmsg_qid = **kmsgget**(parms);

where *parms* represents the set of parameters needed to set up the queue, like kmsg_maxbytes etc.
The *kmsgget* function is typically called at system initiali time and in the future possibly during dynamic reconfigurations. If we opt for a fixed configuration, as is usual in Unix systems, we may hard code the creation into the kernel initiali routines. In this case the *kmsgget* function is obsolete.

The *kmsgsnd* function syntax is

kmsgsnd(kmsg_qid,message,count,flag);

where *kmsg_qid* is as before, *message* is a pointer to the actual kernel message, *count* is the length of the message in bytes and *flag* specifies the action the kernel should take if it runs out of internal buffer space.
The kernel checks that the message length does not exceed the system limit and that there are not too many bytes on the queue. When neither limit is exceeded the kernel allocates space from the kmsg map and copies the data from user space. Then a kmsg header is allocated and linked to the end of the headers queue. The kmsg header is updated to point to the message data and to contain the message size. Also a number of fields in the queue header are updated. The arrival of the message is signalled to the receiving processor through a Location Monitor. If one of the above mentioned limits are exceeded the process sleeps until other messages are removed. However if the *flag* parameter equals IPC_NOWAIT the function returns immediately with an error indication.

To receive requests *kmsgrcv* is called as follows:

```
count = kmsgrcv(kmsg_qid,message,maxcount,type,flag);
```

where *kmsg_qid* is the queue identification again, message is a pointer to a data array in the address space of the receiver (may be shared or private), *maxcount* gives the size of this data array and *type* indicates whether FIFO or prioritized fetching must be performed. The return value, *count*, will contain the number of bytes returned to the IOP's private memory and will be -1 in case of an error. If the IOP finds processes waiting to write requests onto the queue it will wake them up. If the request is bigger than *maxcount* the IOP removes the request from the queue, signals an error in the process table and makes the requesting process runnable.

Finally the *kmsgctl* function can be used to inquire after the status of a request queue. The syntax is

```
kmsgctl(kmsg_id,buf);
```

where *kmsg_id* is as usual and *buf* is the data array where the status should be copied to. This function may be called from system monitoring modules or could be made available to user programs as a system call. When the system is dynamically reconfigurable we must be able to delete kmsg queues from the table. This deletion possibility can be added to *kmsgctl* which should then accept a third parameter *cmd* which specifies the required action. Of course when deletion is possible through *kmsgctl*, it should not be 'callable' by ordinary users. Even the superuser should not be allowed to call it directly. An entity installation and de-installation utility should be incorporated into the kernel. This utility may check the legality of removal of the kmsg queue before calling *kmsgctl*.

In case the Unix System V IPC-like Implementation is selected for implementation we will at least use it for MP-IOP traffic. Then a mapping from entity id's to device numbers must exist. The simplest way is to make these the same. In that case the character device and block device switches become character and block entity switches. The inodes will have their device field changed into an entity field. An alternative way is to provide a device-entity mapping table that can be questioned by the device drivers to locate the entity the request is meant for. This latter possibility has two advantages. First of all the kernel hardly deviates from the SPUX kernel, most alterations are done in drivers. Second, because of the extra level of indirection, it is possible to move units geographically (e.g. a disk from one board to another) without affecting its logical device name. Because of these advantages the last method is favored over the first one.

4.4.1.2 Private Mailboxes

Private mailboxes are defined here as mailboxes belonging to 1 receiving processor or board. The mailboxes may be written into by any other processor. In the particular case treated here the mailboxes have only 1 message slot. The reason for discussing this method is that it is used in our THE-KUNix uniprocessor. We must therefore decide whether we keep using it and if so whether alterations are pertinent.

The implementation as used in THE-KUNix, is discussed in [BOKH83]. We will first coarsely describe his implementation and then use it as a foundation for subsequent discussions.

In the THE-KUNix implementation a mailbox logically consists of the following three items:

- a semaphore register
- a communication memory
- a Location Monitor

The **semaphore register** is a register with such associated circuitry that it behaves as a semaphore. It is therefore often termed *hardware semaphore*. Two operations are possible on this semaphore, a *read* and a *write* operation. The read operation returns the register value (0 or 1) while the associated circuitry loads the register with zero upon recognition of the read operation. (Compare with *test and set* (TAS) operation which reads and **sets** the location). The write operation sets the semaphore register to 1. Both operations require one single access and are therefore indivisible. This type of semaphore can even be used by processors that do not provide a TAS or LOCK instruction. The semaphore is used to control the access to the communication memory.
The **communication memory** is a piece of shared memory preferably located on the receivers board. It is the one and only message slot of the mailbox.
The **Location Monitor** is used to signal the receiving processor when a message has been placed in the communication memory. The Location Monitor is used without a value, e.g. a dummy value is sent.

The message passing protocol is as follows:

Sender	Receiver
request(communication memory)	
fill(communication memory)	
interrupt Receiver through LM	save(communication memory)
	release(communication memory)
	signal(message handler)

This protocol can be used for asynchronous message passing as well as for synchronous message passing. In the latter case the message arrival can be acknowledged by the receiver using an acknowledge message. This is not done in THE-KUNix, instead response messages are used. Response messages are sent after interpretation of the message and carrying out the requested actions. (To enlarge the flexibility the THE-KUNix implementation also supports the use of polling and the use of VME interrupts as alternatives to the Location Monitor for response messages.)

A number of disadvantages exist with this method. First of all there is the possibility of contention when multiple senders need to post a message with one receiver. The bottleneck is formed by the single slot that must be freed by copying its contents to private memory before it can accept a new message. A second problem is that the protocol is, in general, not deadlock free. For example suppose the VMEbus access is prioritized, several processors wish to send a message to receiver R, the communication area is assigned to sender X and a number of the waiting senders has higher priority than X. If in this case the waiting senders keep trying to acquire the communication memory, they could keep the VMEbus occupied to the extent that X is never allowed use of the bus. Then X cannot release the communication area and hence we have deadlock. In [BOKH83] several solutions are suggested of which one is in use in THE-KUNix. More solutions are given in [ANNO85] chapter 10. From these solutions we pick the one in which the 'round-robin' priority scheme is used to exclude bus starvation and which therefore excludes deadlocks of the above type. We choose this particular solution partly because of its conceptual simplicity and partly because it helps out in other cases too. Other solutions are left for further study.

Naturally there are also some advantages to the method. One of these advantages is the relative simplicity of the method. There is no need to find free space in shared memory to put the message in. There is no queue handling in shared memory, this is done locally at the receivers end. (By the way: this reduced overhead reduces the VME load). All routines used at the receiver to copy the message from the communication memory to private memory and to handle the private queue are completely independent. These routines need not be 'Unix-like' and can be different on every IOP. The only commonality is the format of the message and the use of the semaphore. The above mentioned independence eases the addition and replacement of IOP's.

4.4.1.3 VMSbus

The VMSbus has already been discussed in paragraph 3.1.1.5. Because this bus has been specifically designed to support multiprocessor systems it is 'by nature' suited to support inter processor communications.
For the sake of completeness we will give a short overview of its IPrC capabilities here.

The communicated information can be of the data or event type, where an event can be the transfer of a token, the setting or resetting of a semaphore or flip-flop.
The communication can be of the point-to point or the broadcast type. The latter type may be implemented to address a chosen set of receivers.
As we have seen in paragraph 3.1.1.5 all information is sent via frames. Every frame is headed by a **Header Frame**. This **Header Frame** contains among other fields a **Message Priority** field of 3 bits. Hence, when used for IPrC, IPrC messages can be prioritized. Above the priority mentioned here there should also be a priority difference between messages sent via the parallel bus and those sent via the serial bus as discussed before.

4.4.1.4 P1296 specification-like implementation

In [RAP86] Rap and Tetrick describe the P1296 specification, the specification of a 32-bit, synchronous standard bus that is inspecific to vendor or processor. This bus is called the Parallel System Bus (PSB). So far nothing much differs from the VMEbus specification. The specification of the PSB however also tackles inter processor communication. The inter processor communication in the PSB is a separate, hardware supported, subsystem that uses Message Passing as mechanism and that employs an entirely independent address space called **message space**. The message space is only accessible to specially specified modules on each board called **Message Passing Coprocessor** (MPC). Each MPC has its own piece of the message space in the form of send and receive FIFO buffers. Hence the message transfer is of the pass-by-value type.
The messages in the PSB have a standardized format. Each message has a field that carries the source's address in the message space and one that contains the destination's address. Further two fields contain the type and subtype of the message. To move an unsolicited message a processor writes the message into the transmit FIFO of the MPC and triggers transfer. The MPC transfers the message by copying it over the bus (in burst mode) into the receive FIFO of the destination. The MPC thereby decouples the local bus from the system bus, allowing the processor to continue work locally. The PMC may also retry the transfer a number of times when it did not succeed before alerting the processor. At the destination the MPC may interrupt the processor to signal the reception of the message. Unsolicited messages may carry a small amount of data. For large amounts of data special unsolicited messages are specified to request and grant (or reject) buffer space to hold the data. Buffer space is allocated by the local processor and transmission is straight from the buffer holding the data at the sender to the buffer to receive the data at the destination. This involves the local buses at both ends. To keep the advantage of decoupled buses the buffers within the MPC serve as pipeline buffers. For example the sending processor may fill one buffer via the local bus while at the same time the MPC transfers the contents of a second buffer over the system bus.

Rap and Tetrick highlight the decoupling of the buses as a main advantage of the PSB. It is, but it should be noted that similar decoupling has been achieved using the VMEbus (e.g. [FORC87]).

It seems very well possible to use the P1296 specification as the basis for an IPrC subsystem in the VMEbus environment too. The separate message space can be emulated by using one of the VME address spaces identified by a user definable address modifier code. (Eventually it could even be incorporated into the specification by using one of the reserved AM codes). The transfer of the messages could be done by value as in ~~P1296~~ but also by reference using shared memory. The mechanism of the AM codes provides almost the same protection against corruption as the message space in P1296 especially when here too an MPC is used. The greatest advantage that would result from basing IPrC on P1296 is the standardization that would result. Message formats can be chosen equal, message semantics could be chosen equal etcetera. Also since specifications of this kind are reviewed many times the chances of it being short of functionality are minimal.

We already have a project in the group that will give us the P1296 benefits too [VERS87]. However it has much wider scope and is therefore rather complex. I therefore suggest to start a study into a Message Passing Coprocessor for the VMEbus incorporating essentially the same features as the P1296 Parallel System Bus.

4.4.2 Implicit IPrC

Implicit IPrC uses the existing kernel data structures to communicate messages among processors. When implementing the operating system as a central resource with its data structures in shared memory, one could argue that processors that manipulate these data structures communicate.
Communicating in this way is what we call Implicit IPrC. An example may illustrate these MP-MP communications. Imagine a process A running on MP2 and a process B running on MP3. Suppose further that A and B have agreed on talking to each other via two unnamed pipes, one from A to B and one vice versa. On creation of the pipes two inodes have been allocated in shared memory to interface to the file system. In each inode a read and a write pointer point at the output and input side respectively. Through correct manipulation of the read and write pointers a circular data buffer, made up from the 'direct' disk blocks, is simulated. [A number of disk block pointers are present in every inode. Some of these point to disk blocks that contain data (direct blocks), others point to disk blocks that contain further pointers (indirect blocks). Pipes only use the direct blocks.] Synchronization (= IPrC in this case!) between A and B is achieved by putting their current process to sleep when they attempt to read from an empty pipe and when they try to write more into the pipe than it can hold. As you will know putting processes to sleep and waking them up is done through the process table, a kernel data structure in shared memory and hence is Implicit IPrC.

MP-IOP communication can also be done implicitly. The two directions of this communication (MP --> IOP; IOP --> MP) are treated separately in the sequel. Communication in both directions is usually implemented as a message passing scheme for control while data is passed by reference. It is possible to use Implicit IPrC for each direction independently. This is best discussed using an example.

Suppose a process has requested a file that is accessible through a long haul network. The request to get the file is submitted to the appropriate IOP (using IPrC of course) and the process goes to sleep until the file has arrived. Let us look at the IOP -> MP direction first. In Single Processor Unix the IOP would have interrupted the processor after receiving the last part of the file whereupon the Interrupt Handler would have updated some buffer cache fields and would have woken up the (or more than one) process by modifying its entry in the process table (a.o: p_stat = SRUN). With Implicit IPrC the IOP will not interrupt an MP but modify the process table *directly*. It performs the same actions the SPUX Interrupt Handler would have.
Next consider the MP -> IOP direction. We saw that the IO ready event was communicated using the existing process table. The IO request however cannot use such an existing structure. We may use Explicit IPrC to convey the requests or we may implement extra kernel data structures that are used by the IOP as an job queue. In the latter case these extra structures have their primary purpose in IOP algorithms. They may include scheduling parameter fields and such. Thus IPrC using these structures would be implicit too. The IOP job queue is the counterpart of the process table with respect to Implicit IPrC.

It will be clear that there are some pro's and con's using this Implicit IPrC method. An advantage already discussed in 4.3 is the removal of interrupts. No MP interrupts are used at all in the above example. A second advantage is an increase in efficiency. Since, in the MP-IOP example, the IOP

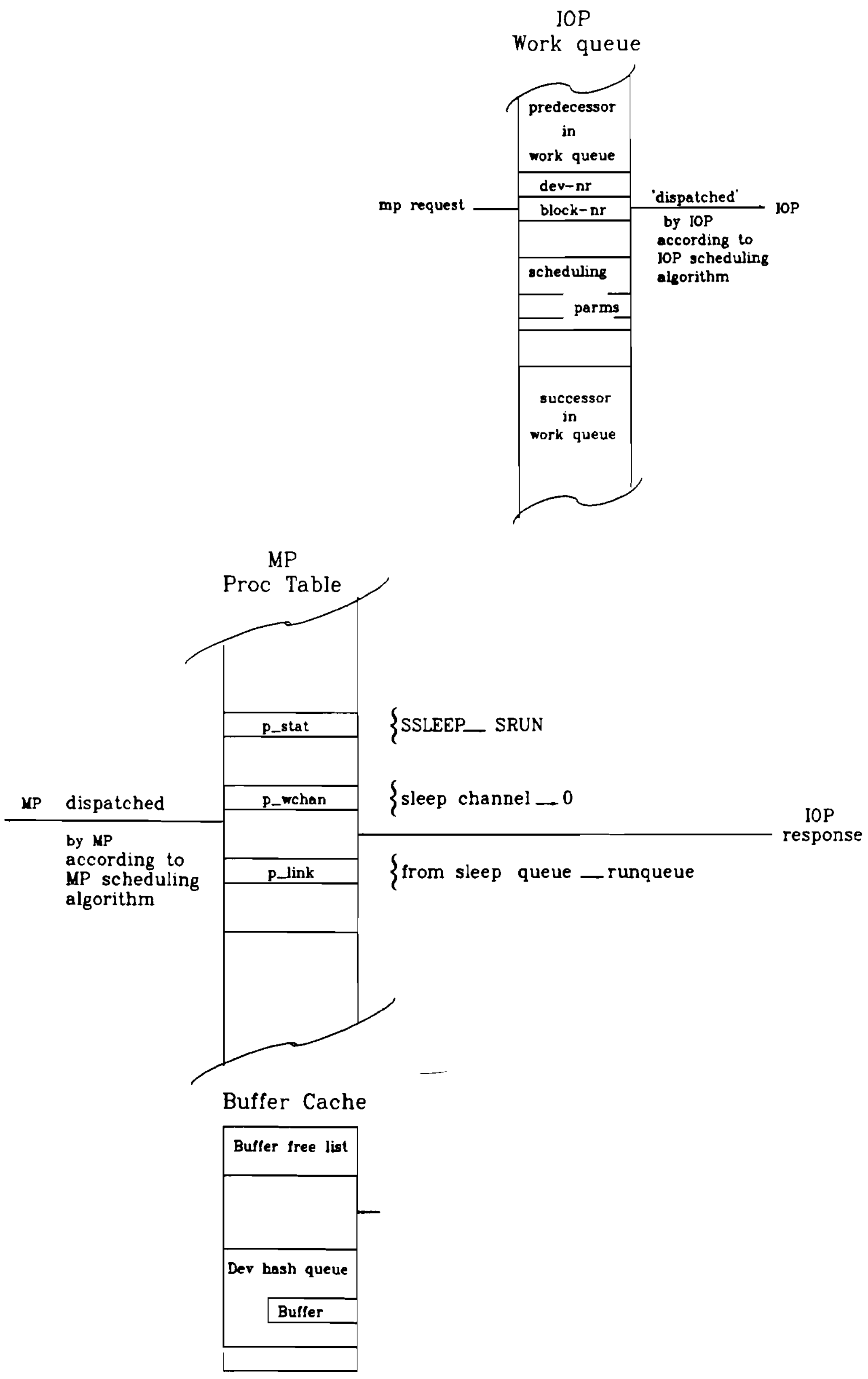


Figure 4.4 Implicit MP - IOP Communication

directly uses the *job queue* and directly uses the *process table* and *buffer cache*, no processing time is lost in handling a message protocol. A disadvantage of Implicit IPrC is that when the *job queue* holds no requests the IOP must poll or it must be signalled through an interrupt at the next arrival of a request. The latter method is more efficient since it allows the IOP to turn fully to the handling of requests for other devices it controls. A second drawback of Implicit IPrC is the dedicated nature of the data structures to be used. This means that for every purpose the need for a new data structure may arise. We should think of purposes like Error Recovery and System Startup here. A last remark that can be made is that a small difference will exist when this method is applied between SPUX and MPUX with respect to scheduling. As explained before SPUX schedules a process of higher priority immediately after leaving the Interrupt Handler that made it runnable when the process that was running at the time of the interrupt was in user mode. In MPUX this will not be so, since no processor and so no process will be interrupted at all. Also signals will, in average, be noticed later in time.

4.4.3 Conclusions

The distinction between implicit IPrC and explicit IPrc may be a bit artificial, it does however stress that communication between processors also takes place via kernel data structures. It opened our eyes for the possibility to have MP-IOP communication take place via data structures that have there main purpose elsewhere. We saw that this is much more efficient and frees us from some annoying P-IH critical sections. On the other hand it essentially means that we have operating system functionality in the IOP's, obliging them to use MMU's, and that we do not have a sharp MP-IOP interface. A clear MP-IOP interface is necessary in an environment where students work on a subject for a short period and are followed by others that have no prior knowledgeof the system. A clear-cut interface prevents them to study both sides of the line, how interesting that may be, but to restrict to the side their assignment lies too.

As systems become more and more communication intensive it is worthwile to see it as a separate subsystem. In such a subsystem all communication mechanisms are concentrated allowing combining of functionality, more efficient implementations, and broader functionality. Such a subsystem would also have to be operating system independent and the interface to the operating system should be independent of underlying hardware. (The ISO-OSI objective).
The independence of the underlying hardware allows it to be used in distributed as well as tightly-coupled environments. This does not mean that one should accept systems that do not fully utilize hardware capacity because they also have to run on hardware of less capacity. (So when shared memory exists, use it.)

We propose that such an IPrC subsystem must be developed. We also propose to first look at the P1296 specification in detail to see if conforming to it is useful. If it is not we still hink it is necessary to come to a specification of IPrC that is accepted by the entire VME community. In the meanwhile our "Unix-IPC"-like implementation can be used to serve immediate needs.

4.5 System Startup

'How do I get from scratch to a running system?' is an interesting question in the case of a uniprocessor. In the case of a multiprocessor it is even more so. The answer to the question has been given in 3.1.5.1 for the uniprocessor running Unix. For the Unix multiprocessor an answer will be given here.

As stressed several times we heavily weigh the symmetry of the system in all respects. However, we consider it sufficient to have a symmetric system when this system is operating in its normal environment. We do not regard the system startup environment as part of the normal environment. Therefore we do not try to make startup symmetrical but we will assign one particular processor with the startup sequence. This does not signify that we tolerate an 'after-startup' situation in which this particular processor has some special status. So the system startup has to be such that the processor that happens to perform the startup actions looses every affinity to the startup but will become identical to all other main processors in the system. We suppose in the following that the processor that will perform the startup (referred to as the 'boot processor') is made aware of its special role through the setting of a switch or jumper on the board where it resides. Naturally, in a Master-Slaves system the Master will boot. In the other three architectures any processor could be the 'boot processor'.

The actual startup and several data structures are influenced by the choice for a fixed or a dynamic configuration. Since we are in favor of the latter we try to implement new data structures and corresponding operations in such a way that this reconfigurability is supported. Unfortunately Unix itself uses a fixed configuration. This will lead to a dualistic nature in some parts of MPUX. However, it is not hard to return to the fixed configuration if one so chooses and on the other hand, if one desires a true dynamic configuration, part of the work is already done.

The startup sequence is as follows. After power up all processors in the system make the SYSFAIL line on the VME backplane active. They then perform there local initialization as required for the local hardware. After that the processors inactivate the SYSFAIL line. Since the SYSFAIL line is driven by open collector outputs, it will not return inactive until the last driver gets inactive. After local initialization all processors but the 'boot processor' will await a broadcast from the boot processor that the system is ready for them to use.

The boot processor's first actions are quite the same as in SPUX. The ROM resident bootstrap program loads a short boot program from block 0 of the system disk and transfers control to it. The boot program then loads the Unix operating system (kernel) into memory (usually from file "/unix"). Having done so it transfers control to the kernel startup code ('dstart' entry).
The kernel initializes all kernel data structures including those that contain configuration data. It then attaches the root file system.
Now the time has come to create the first process to live in the system. This process will have process identification number (PID) 0. Process 0 is manufactured by the kernel directly without the use of a fork call. This is necessary because process 0 is the first process and has no parent process. The kernel, now running as process 0, spawns a new process called the **init** process by calling the *newproc* function. (The function of the init process is discussed soon.) Process 0 may then spawn more processes. These are typically daemon processes operating in kernel mode only and providing system-wide services. An example of such a process is the *vhand* daemon that reclaims pages from main memory. Finally process 0 will turn to the task it will perform the rest of its life: the swapper task.

When process 0 forked it spawned process 1, the Init process. This init process 'exec's the "/etc/init" program as explained for SPUX. Now two things must be done. First, a special process (we call it a *null* process) must be spawned for every processor in the system. The necessity thereof will be explained shortly. Second, the processes mentioned in the file "/etc/inittab" must also be spawned. For a fixed configuration these two actions will pose little problems since the configuration is known on beforehand. When applying dynamic configuring the necessary information must yet be built up. A solution is to have the *init* process send a broadcast message that includes the address of the initialization text for 'non-boot' main processors as a parameter. This broadcast message is the one the other processors were waiting for in order to report themselves in the configuration table. Every 'non boot' processor initializes its MMU and creates the IPrC queues through which the rest of the system may communicate with every entity added to the configuration. Next it includes itself and its devices in the configuration table where it also sets pointers to the IPrC queues.

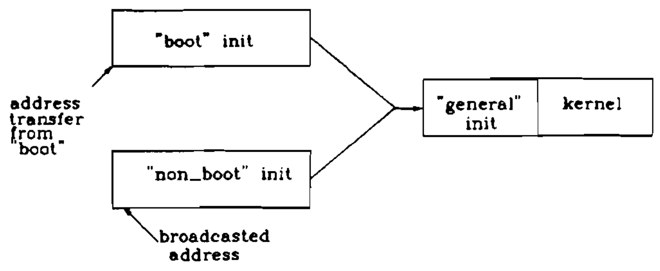


Figure 4.5 Initialization Code

It then signals the *init* process to take appropriate action and after that awaits a 'start through' message from *init*. *Init* has placed itself in a state in which it awaited the configuration after sending the broadcast message. When signalled that the configuration has changed it runs through the configuration table and spawns a *null* process for every new processor and *getty* processes for every terminal (if so indicated in "/etc/inittab"). The *null* processes are marked in such a way that they can only execute on 'their' processor. As a matter of course *init* also signals its child processes to exit when the processor or terminal they were related to has disappeared from the system. In case new processors entered the system *init* sends them the 'start through' message (via IPrC) they expect after it spawned the related *null* processes. This 'start-through' message contains a pointer to the *null* process context.

The *Init* process stays in this configuration monitoring state for the rest of its life. It still has the responsibility to take appropriate action (re-spawn f.e) when one of its child processes exits. Another of its responsibilities was to serve as a stepparent to processes whose parent process has exited. Probably it is best when it stays this way since *Init* is a global and always present process. A problem that arises using the above reconfiguration method is that init has to wake up on two different events. Namely the event of a reconfiguration and the event of a death of child. This needs to be solved in some way. A way is to send it a *death of child* signal and to design the signal-handler in the *init* process to determine the source of the signal and to take appropriate action.

Processors that received the 'start through' message now start executing in the context of their *null* process and call the *swtch* routine to see if real processes are eligible to run. As soon as the *swtch* routine is called the initialization has ended.

What is left for us to explain is why *null* processes are necessary on each processor. Bach [Bach86] explains the reason for this:

" In a multiprocessor system, the kernel cannot idle in the context of the process executed most recently on the processor *(As is normal in system V PWE)*. For if a process goes to sleep on processor A, consider what happens when the process wakes up: It is ready to run, but it does not execute immediately even though its context is already available on processor A. If processor B now chooses the process for execution, it would do a context switch and resume execution. When processor A emerges from its idle loop as the result of another interrupt, it executes in the context of process A again until it switches context. Thus, for a short period of time, the two processors could be writing the identical address space, particularly, the kernel stack."

The *null* processes need not be globally known to the rest of the system. In a system with only shared memory making them globally known seems the easiest way: they'll be normal processes with *init* as their parent. The only difference is that they are fixed to run on one particular processor. On a system with private memory they could be completely hidden from the rest of the system. However this will raise some obstructions in monitoring the performance of the system. Therefore we favor the first possibility.

The above discussion leads us to the following process tree for our multiprocessor.

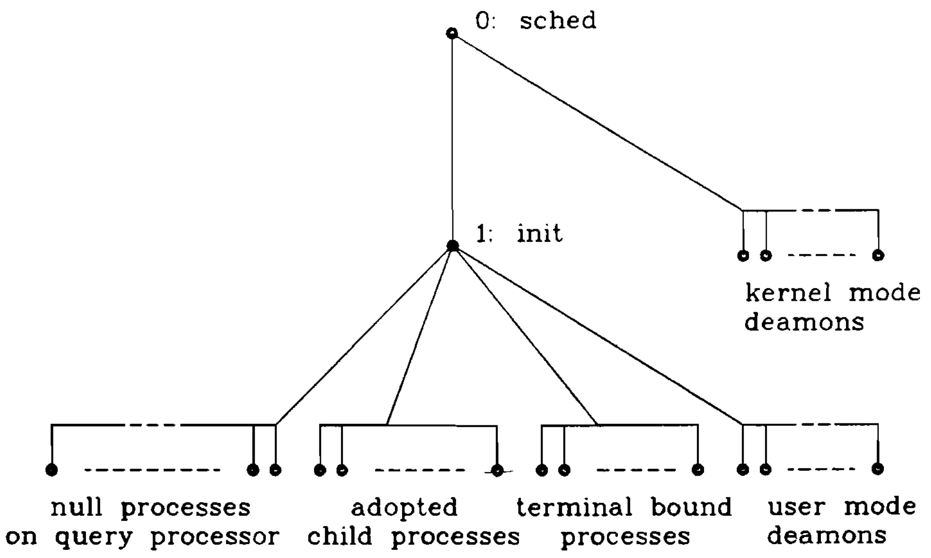


Figure 4.6 Multiprocessor Process Tree

5 Uni-Kernel Architecture

This architecture is a generalization of the Master-Slaves configuration. With this architecture it is possible for every MP to execute in kernel mode. However to keep it simple to mutually exclude processors from critical regions only one processor may execute in kernel mode at a time.
The advantage of this method is that no forced process switch need take place when a process does a system call (and usually one when the call is finished). Considering the high percentage of execution done in kernel mode by a typical process, it is very likely that another processor is in kernel mode at the time of the system call. In such a case a process switch needs to be done after all. The advantage may therefore be marginal. Nevertheless this architecture is advantageous. Looking at pure speed, a marginal decrease of VMEbus load may give rise to a relatively larger profit in system throughput. Especially when the VMEbus is near saturation this effect may occur. Furthermore there are some secondary advantages gained from this architecture. One of these is the increase in symmetry which gives (or at least can give) rise to a better fault tolerance. Another secondary advantage (which may prove important for practical reasons) is that this architecture provides a step-up to the next architecture that may be investigated. This next architecture is the Multi-Kernel architecture, where 'Multi' is used to denote the entire range from 2 to the highest number achievable. Starting from the Uni-Kernel configuration one could start chopping up the kernel in pieces and allow only one instance of such a piece to be executed at a time. This provides a very smooth development path towards the final multiprocessor.

An obvious problem to be dealt with when using this architecture is the handling of interrupts. The convenient master processor to send all interrupts to has now disappeared. In the next two paragraphs methods are discussed to tackle this problem.

5.1 Mutual Exclusion

This paragraph reduces to finding ways of telling every processor that a colleague has entered kernel mode and when it should hold off interrupts of a particular level. The rest of the mechanism is the same as for the Master-Slaves architecture.
Assume that there is an efficient way to inform all processors in the system that a processor owns the kernel resource. We will name it the 'kernel-processor'. Two situations will occur when a processor wishes to enter the kernel. The kernel is free or it is occupied. When it is free there are no problems: the processor enters the kernel and its process proceeds. However when the kernel is occupied the processor must put its process ~~to~~ sleep and put it on a 'kernel-queue' (Compare Master-queue from chapter 4). (Spin-locking would be very inefficient!). Then it should dispatch a process from the 'user-queue' (Compare Slave-queue from chapter 4), that is, it should start running a process that does not immediately request kernel mode processing. (It must behave as a slave). The moment the 'kernel-processor' releases the kernel it may find the 'kernel-queue' empty or not empty. In the first case it reschedules to start running the highest priority user mode process available. In the second case, the most efficient thing for it to do is to take the highest-priority process from the 'kernel-queue' and run it. The 'kernel-processor' behaves exactly as the master from the Master-Slaves configuration. We see that the 'kernel-processor' hogs the kernel resource when 'kernel-mode' processes are runnable. The chance of having such processes is high even for light system loads since processes spend approximately 50% of their lifetime in

kernel-mode. For high loads the Uni-kernel configuration behaves exactly as the Master-Slaves configuration (except for the interrupts). For low loads it behaves a little more efficient.
Above we assumed an efficient method for distribution of the 'kernel is occupied' information. Let us consider some methods in the next section.

5.1.1 Mode Broadcasting

One very familiar way of letting every processor know when some processor is in kernel mode is by considering the kernel mode as one big critical region and protecting it with a **semaphore**. Since about half of the processes wish to run in kernel mode a queue of processes that wish to run in kernel mode will build up. This means that we need to consider the way in which the next process is allowed access. There are several ways to determine the access order. Two obvious ways are to use FIFO ordering or prioritized ordering where the priority is derived from the process priority. Both cases are starvation free. Employing a semaphore is not the fastest method but it is a very flexible one. It also forms an ideal starting point for the 'chopping process' suggested above.

Another way of broadcasting that the 'kernel mode' resource is occupied is by using an **intelligent semaphore** as defined by the VMSbus specification. Using these, a process(or) that wishes to enter the 'kernel mode' resource inspects its local copy of it and, when free, sets it and continues. When already set the process is put to sleep and a 'user mode' process is dispatched. The functional characteristics of an intelligent semaphore are ideally suited for the broadcasting desired. Copies of the semaphore all over the system will be updated simultaneously, hence no inconsistency will arise. Unfortunately these intelligent semaphores are very slow since the underlying VMSbus is very slow. Especially for a resource as critical as the 'kernel mode' resource in a Uni-Kernel system this slowness can not be tolerated. We therefore renounce this implementation.

A third implementation makes use of the **broadcasting capabilities of the VMEbus**. (For an implementation of the broadcasting mechanism itself see 3.1.1 or [FORC87]). This broadcasting causes a local interrupt at every processor in the system thereby transferring a message of 8 bits. The 'kernel-processor' broadcasts a 'kernel-available' message when it finds the 'kernel-queue' empty. The 'kernel-processor' keeps ownership of the kernel. All other processors note the 'kernel-available' status and broadcast a 'kernel-wanted' message as soon as they wish to enter the kernel. The first request received by the 'kernel-processor' is honored. The 'kernel-processor' will broadcast the processor identification of the requesting processor in a 'kernel-granted' message, thereby seizing to be the 'kernel-processor'. Both 'kernel-wanted' as the 'kernel-granted' messages must contain a processor identification field. The handshake that takes place is necessary to avoid simultaneous access to the kernel by two or more processors. A total cycle, from the 'kernel-available' broadcast to the 'kernel-granted' broadcast inclusive, takes approximately 3 Ms on a 20 MHz 68030 based system. By incorporating the 'kernel-processor' id into the 'kernel available' message, the 'kernel wanted' broadcast can be replaced by a 'kernel-wanted' private message. It is not sure whether this is more efficient. A 'kernel-wanted' broadcast may cause other processors to refrain from sending such a message too. When a private message was used they would keep trying (loading the VMEbus and interrupting the 'kernel-processor') until a 'kernel-granted' broadcast was received.

An alternative way to inform a processor whether it may use the 'kernel mode' resource is to pass an access token round. When a processor has possession of the token it is allowed to perform kernel actions. There are three ways to pass the token, via the VMSbus, via Location Monitors or by broadcasting over the VMEbus. Because of the inherent delays incurred by passing the token by processors that do not need it this method is judged to be unfit on beforehand. Token passing is more suited for cases where a group of interchangeable resources exist.

5.1.2 Kernel Border

Once having a mechanism to lock, it is important to know where to lock. From figure 3.6 it appears that there are only a few ways to enter the kernel. These ways are, invoking a system call and the occurance of an interrupt/exception. Initially all of these ways can best be modified for locking of the kernel by placing the locks in the respective interrupt (trap) handlers. Later on performance can be improved by placing the locks further inwards. However since we aim at the Multi-Kernel architecture it would be better to directly start dividing the kernel in separately and simultaneously executable parts as discussed before.

5.2 Interrupts

5.2.1 General

In the Uni-Kernel configuration interrupts that cause manipulation of kernel data structures may occur on any processor. When they occur on the 'kernel-processor' there is no problem, the traditional solution of blocking them temporarily by increasing the processor priority may be used. When they occur on a 'non-kernel-processor' the situation is much more complex. When the 'kernel-processor' is in a P-IH(l) critical region (l = interrupt level of Interrupt Handler), interrupts of level l must also be held off in all 'non-kernel-processors'. Conversely, when a 'non-kernel-processor' is handling a critical interrupt **all** other processors are not allowed to service an interrupt of the same level. Therefore these critical regions must be protected in the same way for all processors alike. The only irregularity is that when the 'kernel-processor' wants to enter the critical region it must be granted access immediately after it becomes free while the others must compete on an equal basis for it. To avoid unnecessary busy-waiting a processor that occupies the critical region must notify all others to hold off interrupts of the appropriate level. Such an interrupt would cause busy-waiting. Holding it off lets processors perform useful work instead. A complicating factor is the leveling of the interrupts. To be efficient we want those interrupt requests that can be safely handled to be handled. Interrupts of a level n are allowed to interrupt Interrupt Handlers of level **n-1** and lower. This means that interrupt handlers of level n have no critical regions with respect to those of lower levels. (No IH(n)-IH(<n) critical regions). Hence interrupt handlers of a lower level than n may continue to be executed on other processors when a processor notes an interrupt request of level n. However we do not (yet) know whether the Unix kernel combines P-IH critical regions for different levels of interrupt as follows:

Note: l<h

```
spl(h);                     /* disable interrupts of level h and lower      */
P-IH(l) critical region     /* critical region for which spl(l)             */
                            /* would have been sufficient                   */
P-IH(h) critical region     /* critical region for which spl(h)             */
                            /* is necessary                                 */
spl(x);                     /* restore previous processor level             */
```

If the above is done we **do** have to consider the lower levels too.
A solution to the problem is as follows:

Assume a range of interrupt levels of 0 to r.
Define an array A[r,1]with length r+1 (0 - r)
and height 2 (0 - 1)
Initially A contains all zeroes.

Any P-IH critical region in kernel code or any Interrupt Handler code must be surrounded by extra code as follows (i = interrupt level):

```
BroadCast( spl(i) );                  /* disable interrupts of level i and lower on all*/
                                      /* CPU's not engaged in interrupt handling.  */
if (ProcID=KernelProcId)
   {
   for(j=0,j=j+1;j==i) A[j,0]=1;           /* Announce intention of kernel processor */
   for(j=0,j=j+1;j==i) while(T&S(A[j,1])); /* and seize resources                    */
   critical region;
   for(j=0;j=j+1;j==i) {A[j,0]=0;A[j,1]=0};
   }
else /* non-kernel-processor */
   {
   while(s!=0)
      {
        s=0;
        for(j=0;j=j+1;j==i) s=s+A[j,0];
        if (s!=0) continue;
        s=0;
        for(j=0;j=j+1;j==i)
           {
             B[j] = T&S(A[j,1]);
             s = s + B[j];
           };
        if (s!=0) for(j=0;j=j+1;j==i) if (B[j]==0) A[j,1]=0;         /* restore flags */
      }
   critical region;
   for(j=0;j=j+1;j==i) A[j,1]=0;
}
BroadCast( spl(x) );                  /* restore previous processor priority levels  */
```

If the kernel does not combine critical regions of different levels as shown above, the protection would be much simpler:

```
BroadCast( spl(i) );                 /* disable interrupts of level i and lower on all*/
                                     /* CPU's not engaged in interrupt handling.  */
if (ProcID=KernelProcId)
   {
   A[i,0]=1;                         /* Announce intention of kernel processor   */
   while(T&S(A[i,1]));               /* and seize resources                      */
   critical region;
   A[i,0]=0;
   A[i,1]=0;
   }
else /* non-kernel-processor */
   {
   while(A[i,0]);
   while(T&S(A[i,1]));
   critical region;
   A[i,1]=0;
}
BroadCast( spl(x) );                 /* restore previous processor priority levels  */
```

5.2.2 Clock Interrupt

The functions of the clock interrupt in SPUX System V are the following:

a. Restart clock
b. Invoke internal kernel functions from a callout table
c. Provide execution profiling for kernel and user processes
d. Gather system and process accounting statistics
e. Keep track of time
f. Send alarm signals to processes on request
g. Periodically wake up the swapper process
h. Control process scheduling

On a multiprocessor each processor will have its own clock interrupt or the clock event is broadcasted to them. When this clock event occurs all of the above actions must be performed. However some of these are global to the system while others have to take place locally and for every running process independently. Let us divide the above actions into a global and a local set.

First there is the action of restarting the clock. In a multiprocessor several possibilities exist as to the number of clock interrupt sources with respect to the number of processors. We distinguish two possibilities. It is possible to have one central clock. The event of timing out is then broadcasted to the processors as mentioned and restarting the clock is a global action. The second possibility is that every processor has its own private clock timer. Then the action of restarting the clock is a local action.

Some kernel operations, particularly device drivers and network protocols, require invocation of kernel functions on a real-time basis. For example, a process may put a terminal into raw mode. The kernel then satisfies user *read* requests at fixed intervals instead of waiting for the user to type a carriage return. The function to be called after the fixed interval expires is placed in the *callo* table. At every clock interrupt the clock interrupt handler checks the callout table and invokes those functions that are due. Calling out functions from a callout table is an action that is global to the system for as long as this callout table stays global. At least for the time being we do not consider local callout tables so for now the related actions are indeed included into the global set. In SPUX the clock interrupt handler issues a software interrupt to start an interrupt handler which in its turn calls out the functions from the table. To eliminate the P-IH critical regions that occur when accessing the *callo* table one could have a kernel mode only process performing the task of this interrupt handler. The P-IH critical regions then turn into P-P critical regions. Synchronizing the process to the clock interrupt can be done through a semaphore.

Profiling is in essence not much more than updating some locations dependent on the current value of the program counter. Since every processor has its own program counter this profiling clearly is a local action.

Under d. system statistics are gathered. The word *system* already indicates that this is a global action. Also under d. process statistics are accumulated that are used for accounting later on. These statistics are only accumulated for running processes. The actions involved thus belong in the set of local actions.

Keeping track of time means incrementing time and calendar values. Depending on the overhead allowed one can place these actions either in the global set or in the local set. In the first case the kernel maintains time and calendar values in shared memory locations, in the latter every processor could have local memory locations to that end. As usual in cases where multiple copies of the same data exist at physically distributed places one must take measures to keep all copies consistent. Here this means that at initialization time *all* local values must be initialized.

The sending of alarm signals is global as long as the process table is global. Implementing private process tables next to a global one is a very interesting topic left for chapter 7. For now we suppose one global process table only.

The swapper process needs to be woken up periodically. There is only one swapper process in the system classifying the wake up as a global action.

The SPUX clock interrupt handler controls process scheduling by manipulating the *CPU_usage* fields of every process. The CPU_usage is used in calculating the priority of a process: the higher the CPU_usage value the lower the priority. The interrupt handler increments the CPU_usage value for the running process on every invocation and halves the CPU_usage value for the other processes every second. Adjusting the CPU_usage fields for running processes will be done locally, for the other processes it will be done globally on a multiprocessor (again as long as the process table stays global).

Having divided the actions into the global and local set leaves us with the question who will perform the actions from the global set. It is evident that actions in the local set are performed by each processor experiencing the clock interrupt. Only one of these processors needs to do the global actions. A possible implementation resembles the scheme of token passing. It is a round robin scheme implemented by maintaining a value (in shared memory) that holds the processor identity of the processor that has to do the global actions. This processor will, after finishing the global actions, adjust the value to contain the processor identity of its successor. The interrupt handler algorithm may now look as follows:

```
algorithm clock
input: none
output: none
{
    /* local actions */
    restart clock;
    if (kernel profiling on)                          /* Note */
        note program counter at time of interrupt;
    if (user profiling on)
        note program counter at time of interrupt;
    adjust measure of CPU utilization for the current process;
    gather statistics per process;

    /* global actions  */
    if token == my proc_id;
    {
        if (callout table not empty)
        {
            adjust callout times;
            schedule callout function if time elapsed;   /* Note */
        }
        gather system statistics;
        if (1 second or more since last here and
            interrupt not in critical region of code)
        {
            for (all processes in the system)
            {
                adjust alarm time if active;
                adjust measure of CPU utilization;
                if (process to execute in user mode)
                    adjust process priority;
            }
            wakeup swapper process if necessary;
        }
    }
}
```

Note: In SPUX these events are dealt with by separate interrupt handlers invoked through a software interrupt. In MPUX the latter case could also be handled by a 'kernel mode only' process thereby replacing P-IH critical regions for P-P ones.

The solution with a token location takes some extra execution time at every clock interrupt in order to access the token and *proc_id* locations in shared memory. (Extra time may be lost in gaining access to these values). Furthermore a protection must be built in to detect a faulty processor. This is fairly easy since a faulty processor will not update the toke anymore. A simple time-out mechanism on every processor will do. The first processor to detect a faulty collegue automatically gains the responsibility of performing the global actions. More elegant than having a 'token-location' in shared memory is to make token passing a basic feature of the IPrC mechanism and to use that feature for the purpose at hand. An alternative is to assign a processor at initialization time to perform the global actions. This processor would be the only one performing these global actions until it either breaks down or the system is shut down. If it breaks down the recovery handler must select another processor to do this job. The selected processor could keep a private value to note its special status with respect to the clock interrupt.

6 Multi-Kernel SMO Architecture

The Multi-Kernel Shared Memory Only architecture as we propose it consists of a number of identical Main Processors and a number of heterogeneous Input-Output Processors all resident in a VMEbus based system. The Main Processors are allowed to execute the kernel code simultaneously. This feature can only be made possible when extensive modifications to the existing uniprocessor kernel are made. These extensions are, for the greater part, making the kernel preemptable and protecting shared resources. Because in the present kernel mutual exclusion methods are used that are sufficient in a uniprocessor but insufficient in a multiprocessor it is not immediately apparent in most cases where protection must take place. In order to learn what we deal with the critical sections as occurring in Unix are inspected in section 6.1.
Our exploration of this architecture then continues with the topic of scheduling. What methods of scheduling for multiple processors exist? What benefits can be gained from optimal scheduling? What hazards exist? These questions will be answered in section 6.2.

6.1 Unix Critical Regions

We already discussed the problems that occur when multiple processors run kernel code. It showed that in the uniprocessor kernel Process-Process (P-P) and Process-Interrupt Handler (P-IH) critical regions exist. Later we somewhat detailed the P-IH critical regions by discriminating them according to the level of interrupt: P-IH(l) critical regions. In treating the Uni-Kernel architecture, where interrupts may be fielded to any processor, we observed a third type, the IH-IH critical region.
In both Master-Slaves as the Uni-Kernel architecture the P-P critical regions were protected by allowing only one instance of the kernel at a time. The P-IH critical regions were easily protected in the Master-Slaves architecture. The Master is the only processor that experiences interrupt requests so it may temporarily disable them by raising its processor level as in a uniprocessor. More problems were encountered in the Uni-Kernel architecture where both P-IH and IH-IH critical regions are hard to protect because the process instruction stream and the IH instruction streams may be executed on different processors. This invalidates the 'processor-level raising' method as a complete protection. Solutions were discussed in the previous chapter.

In this chapter we not only allow IH instruction streams to be in execution on different processors, we also allow this to kernel mode process instruction streams. Furthermore we do not merely wish to protect critical regions, we wish to do so as efficient as possible. This means that we will have to find an optimal value to the granularity of our protection methods. That is, we will have to carefully balance the protection overhead against the benefit of less serialization by having more, smaller critical regions. We will also have to be apprehensive of possibilities to totally avoid critical regions.

To protect the critical regions two things must be done. A selection must be made from the possible synchronization mechanisms possibly on a per-critical-region basis. Second, because we deal with existing code, these critical regions must be detected and delimited in the present code. Paragraph 6.1.1 summarizes possible synchronization mechanisms and paragraph 6.1.2 deals with the detectability of critical sections in Unix code.

6.1.1 Synchronization Mechanisms

In this section we will discuss a number of synchronization primitives (SP's) that we may use to adapt the kernel to a true multiprocessor kernel. With the exception of the **Fetch&Add** (F&A) primitive we will be brief since literature seems to be an inexhaustible source of tutorials, descriptions etc. on this topic [DUBO88], [MUEH80] etcetera. The synchronization primitives are usually categorized in hardware and software primitives. Although more and more hardware is used to provide primitives that used to be provided by software, we will stick to the traditional Hardware-Software classification.

6.1.1.1 Hardware Synchronization Primitives

The hardware SP's are usually very simple and therefore often used as building blocks for complex Software SP's. In this light we may even judge atomic Load and Store actions as synchronization primitives as they have been used in many synchronization algorithms (e.g. [DIJK65]). Also inter processor interrupts may be considered as such primitives.
One outstanding case is formed by the **Test&Set** (T&S) primitive, used on its own as well as in more complex primitives. The semantics of Test&Set are:

```
Test&Set(flag)
        {
        temp = flag; flag = 1;
        return(temp);
        }
```

The software that uses the Test&Set will typically repeat it until a zero value is returned. A zero value indicates that no other processor is using the protected resource. The processor that performed the T&S primitive may then use the resource. The action of repeating the T&S operation until a zero is obtained is called *spin-locking*. When finished using the resource the flag can be reset using a normal Store(0) which translates to "flag = 0" in C.
Spinning on a flag causes a lot of bus load. This obstructs other processors, including the one that must reset the flag. To avoid spinning *suspend-locking* may be used. *Suspend-locking* functions as follows. A processor that finds a flag to be set (by doing a T&S on it) can note this in a data structure related to the flag, disable all interrupts to its processor except for the IPrC interrupts and then retire from the bus, effectively idling the processor. The processor that eventually resets the flag, signals this to all processors noted in the flag related data structure. These then retry to set the flag.
To implement the T&S primitive, processors perform a Read-Modify-Write (RMW) cycle. This cycle is composed of a Read and a Write cycle in such a way that it is guaranteed that no other processor will use the bus in between the Read and the Write. Hence a RMW cycle (and so a T&S primitive) is atomic.

A related primitive is the **Compare&Swap** (C&S) primitive also implemented as a RMW cycle. The semantics are:

```
Compare&Swap(r1,r2,w)
        {
        temp = w;
        if (temp == r1)
           {w = r2; z = 1;}
        else
           {r1 = temp; z = 0;}
        }
```

A variation, used for manipulating doubly linked queues, is:

```
Compare&Swap2(r1,r2,s1,s2,w,v)
        {
        temp1 = w; temp2 = v;
        if ((temp1 == r1) and (temp2 == s1))
           {w = r2; v = s2; z = 1;}
        else
           {r1 = temp1; s1 = temp2; z = 0;}
        }
```

There exist two problems with the **Test&Set** and the **Compare&Swap** primitives. The first is that they form small critical sections causing processors to serialize accesses to the flag. This effect is only noticeable in large multiprocessors. The second problem is that the aggregate information returned when N processors access a free flag is very low. For the case of the T&S it means that one processor obtains the zero value causing it to proceed while all others obtain the **same** value, 1, causing them not to proceed (at least not in the intended direction).
Through a suitable implementation it is possible to solve the first problem and provide critical section free T&S and C&S operations. We will show this for the next primitive the **Fetch&Operation** (F&O) which also solves the second problem (at least in the case that Operation = Add).
The F&O primitive is used in the IBM RP3 [EDLE85] and a special version, the **Fetch&Add** (F&A), in the NYU Ultracomputer [GOTT83a]. The semantics of the F&O primitive are:

```
Fetch&Operation(V,e)
        {
        temp = V; V = Operation(temp,e);
        return(temp);
        }
```

The parameters V and e are of type integer and the Operation must be associative and commutative. The F&O primitive must satisfy the serialization principle that when many F&O operations simultaneously update shared variable V, the effect is the same as if they occurred in some (unspecified) order. The F&O primitive contains the T&S, C&S and F&A primitives as special

cases. Simply substitute AND(V,e), PROJ(V,e)[2] and ADD(V,e) for Operation(V,e) respectively. The AND operation may be defined for integers or boolean parameters may be used. In the rest of this discussion we will only consider the F&A primitive.
This F&A primitive can be profitably used in a large number of cases, even to the extent that Edler et al [EDLE85] claim that an operating system "can be built that perform[s] the traditional functions of resource management, scheduling, and coordination using critical-section-free algorithms".
Let us consider two examples to illustrate the possibilities of F&A. The first example, due to [PADU80], is well-suited for fine-grain machines, the second, due to [GOTT83a and b], for course-grain machines.
In the first example the following high-level parallel FORTRAN statement can be executed in parallel by N processors if there is no dependency between iterations of the loop:

```
DOALL V = 1 to 100
   { code using V}
ENDDO
```

Each processor executes an F&A(V,e) with e set to 1, hence obtaining a unique value to be used in the body:

```
(initially V = = 1)
n = F&A(V,1);
while (n ≤ 100) do
   {
    { code using n}
    n = F&A(V,1);
   }
```

Observe that N may be smaller, equal or larger than 100.

The second example shows the possibility to manipulate a shared queue without any critical sections (assuming the F&A does not itself represent a critical section). The queue in the example implements a circular buffer.
Gottlieb et al write: "The procedures to be shown maintain the basic first-in-first-out property of a queue, whose proper formulation in the assumed environment of large numbers of simultaneous insertions and deletions is as follows: if insertion of a data item *p* is completed before insertion of another data item *q* is started, then it must not be possible for a deletion yielding *q* to complete before a deletion yielding *p* has started."

[2] The PROJ(V,e) operator returns e as the outcome.

In the procedures presented below the following variables and routines occur:

I	Points to the location in the queue for the next insertion (initially 0)
D	Points to the location in the queue for the next deletion (initially 0)
Size	Holds number of queue locations
#Ql	Holds the lower bound for the number of items in the queue. (Queue is empty when #Ql ≤ 0)
#Qu	Holds the upper bound for the number of items in the queue. (Queue is full when #Ql ≥ Size)
QueueUnderflow	True when a processor tries to delete from an empty queue, otherwise False
QueueOverflow	True when a processor tries to insert into a full queue, otherwise False
TIR	Test-Increment-Retest procedure. Checks if space for insertion exists and if so increments #Qu and returns True. If no space returns False.
TDR	Test-Decrement-Retest procedure. Checks if items are present and if so decrements #Ql and returns True. If no items present returns False.
Put	Put does the actual insertion of an item into the acquired Q cell. However other insert and/or delete operations may have been assigned this cell id. Therefore a mutual exclusion must be used to prevent corruption. This is done through semaphores[3].
Get	Get deletes an item from the queue. (See also Put).
P	P is the well known operation on a semaphore (See 6.1.1.2) that does not allow a process(or) to proceed when the associated resource is occupied. This P implementation (with F&A's of course) causes the processor to spin-lock until the resource becomes free.
V	V is the other well known semaphore operation.
InsertSem and DeleteSem	These are arrays to hold the semaphores to each queue cell. The array size is equal to Size.

The procedures are:

```
Procedure Insert (Data,Q,QueueOverflow)
    {
    If TIR(#Qu,1,Size)
        {
        MyI = Mod(F&A(I,1),Size);
        Put(Q,MyI,Data);
        F&A(#Ql,1)
        QueueOverflow = False;
        }
    else QueueOverflow = True;
    }
```

```
Procedure Delete (Data,Q,QueueUnderflow)
    {
    If TDR(#Ql,1)
        {
        MyD = Mod(F&A(D,1),Size)
        Get(Q,MyD,Data);
        F&A(#Qu,-1);
        QueueUnderflow = False
        }
    else QueueUnderflow = True;
    }
```

[3] When resources are scarce and mutual exclusive a critical section arises for the use of such a resource. Note that when we make the queue length n = N * M, with N is the number of processors and M the amount of multiprogramming, no multiple use of a queue cell occurs.

```
Boolean Procedure TIR(S,Delta,Bound)
    if (S + Delta ≤ Bound)
        if (F&A(S,Delta) ≤ Bound)
            TIR = True;
        else
            {
                F&A(S,-Delta);
                TIR = False;
            }
    else TIR = False;
```

```
Boolean Procedure TDR(S,Delta)
    if (S - Delta ≥ 0)
        if (F&A(S,-Delta) ≥ 0)
            TDR = True;
    else
            {
                F&A(S,Delta);
                TDR = False;
            }
    else TDR = False;
```

```
Procedure Put(Queue,Index,Item);
    {
        P(InsertSem[Index]);
        Queue[Index] = Item;
        V(DeleteSem[Index]);
    }
```

```
Procedure Get(Queue,Index,Item);
    {
        P(DeleteSem[Index]);
        Item = Queue[Index];
        V(DeleteSem[Index]);
    }
```

```
Procedure P(S)
    {
    OK = False;
    do
        {
         if (--S ≥ 0)
           if (F&A(S,-1) > 0) OK = True;
           else F&A(S,1);
        } while (OK);
    }
```

```
Procedure V(S)
    {
        F&A(S,1);

    }
```

When the queue is neither full nor empty the above procedures allow many deletions and many insertions to be done simultaneously. More F&A based software primitives are presented in [GOTT83b].

The above examples show the F&A to be very powerful for building software primitives that do not contain serial bottlenecks. In the case that the F&A constitutes a small critical region itself, very large multicomputers will suffer from it. The implementation of the NYU Ultracomputer is such that even this small critical section has been eliminated.
To not leave you wondering how true simultaneous execution of F&A primitives is made possible in the Ultracomputer its implementation is briefly discussed.

The secret lies in the Processors-Memory Network and the Memory-Network-Interface (MNI). The Network is an Omega Network. An example of a 4-input-4-output omega network and its environment is depicted below.

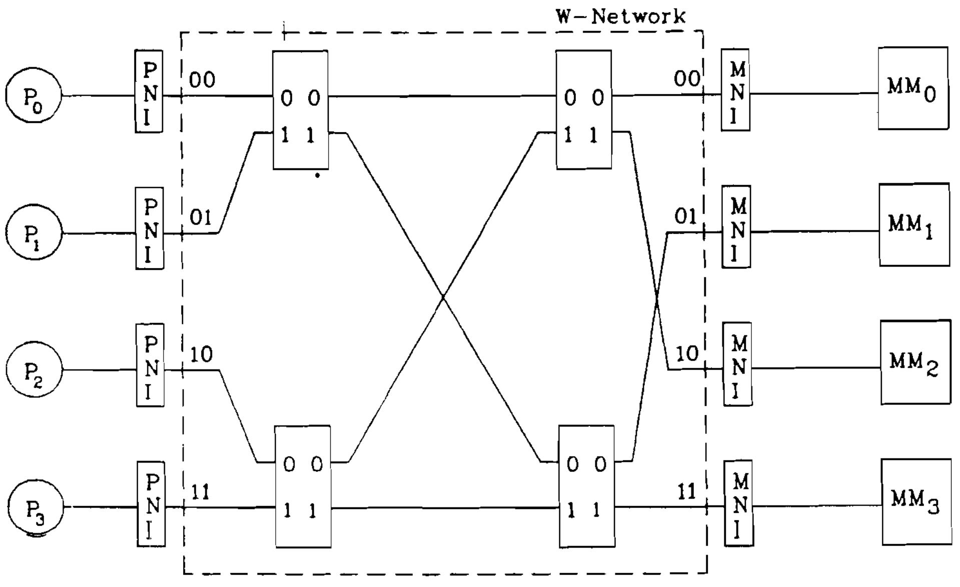


Figure 6.1 The Omega Network

Suppose V denotes location 10 in the above network. Suppose further that F&A(V,e) primitives are issued simultaneously at inputs 00 and 11. The two primitives will meet at switch 3. Switch 3 will combine the two requests as shown in figure 6.2 (suppose [V] = Y):

The switch will retain one of the operands to be added and will forward a single F&A primitive with the sum of **e** and **f** as its second parameter. This primitive will reach the MNI which will read value Y from location V, write Y+e+f to V and return Y to the Network. When this value Y reaches switch 3 (in our case immediately) the F&A requester for which the second parameter was retained is returned value Y, the other value Y+e.

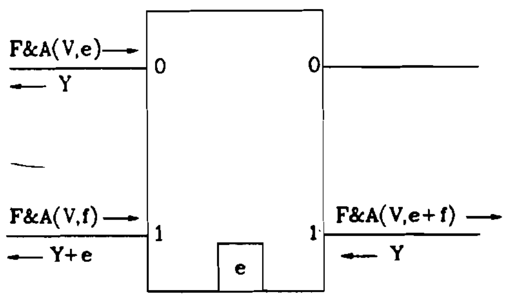


Figure 6.2 Combining Fetch&Add's

Much more can be said about this network, for example how pipelining is done and how switches deal with the situation where it must combine multiple request pairs. However this is not within the scope of this thesis. The interested reader is referred to [GOTT83a].

6.1.1.2 Software Synchronization Primitives

An outstanding software synchronization primitive is the semaphore. Its impact on computer based systems has been enormous. Semaphores were first introduced by Dijkstra [DIJK68]. Two operations are defined on a semaphore, the P(s) operation to request use of the associated resource and V(s) to free the resource. (P and V are the first letters of the Dutch words *passeren = to pass by* and *vrijgeven = to free*). The P(s) procedure puts the invoking process to sleep when the resource is already occupied. This process is then enqueued on the semaphore s and the processor is forced to switch to another process. This avoids busy-waiting for the associated resource at the cost of a larger overhead. The V(s) operation releases the first process enqueued if any. If no process is queued on the semaphore the free state of the associated resource is noted in the semaphores count field. Since processes may be put to sleep in the P operation a Unix implementation of P will have to deal with signals that may prematurely wakeup the process [Bach86 page 401]. (See also [BACH84]).
Several types of semaphores exist, for example binary, counting, additive and multiple semaphores. Other primitives exist such as the UP/DOWN [WODO72] and the INC/DEC set [SCHR76]. Detailed descriptions of these synchronization primitives can be found in [MUEH80]. All of these primitives rely on the simpler hardware SP's to become indivisible (a necessary property for correctness).
Yet another synchronization primitive is the Barrier, used to join a number of parallel processes belonging to a task force. The semantics are:

```
Barrier(N)
    {
    count = count + 1;      /* initially count = 0 */
    if (count ≥ N)
        {
        wake up all processes on barrier queue;
        count = 0;
        }
    else block process and place it in the barrier queue;
    }
```

Providing the Barrier as a separately implemented primitive may lower the overhead involved when compared to emulating it with, for example, semaphores. This is profitable in environments where many parallel, iterative algorithms are executed.

Finally it is possible to use message passing mechanisms for pure synchronization. This has been discussed in chapter 4.

6.1.1.3 Summary

Having the objective to eventually obtain a Unix kernel that is fit to be executed by more processors simultaneously, it is important to choose a set of synchronization primitives that allows to do this efficiently. From the primitives discussed the F&A seems to be very promising as it enables the implementation of a kernel that is free of (processor) critical sections [4]. Hence the efficiency of the processor community will be higher. The disadvantage is that the kernel needs to be largely rewritten which is a large task. On the other hand, using the traditional synchronization primitives such as semaphores for cases where F&A could be used, will also prove to be an enormous task. This tips the balance in favor of the F&A primitive. This conclusion does not mean that we can do without semaphores or the like, these still are necessary for protecting process critical regions.

6.1.2 Detectability of Critical Regions

In order to detect critical regions in existing code one has to have detailed knowledge of the forms in which such regions occur in the code. Fortunately, Janssens, Annot and v.d. Goor have already investigated the matter and presented their results in [JANS86]. We will summarize these results below.
Janssens et al recognize the following 5 types of critical region:

1. User Process	- User Process	
2. Kernel Process	- Kernel Process	(without embedded context switch)
3. Kernel Process	- Interrupt Handler	
4. Kernel Process	- Kernel Process	(with embedded context switch)
5. Kernel Process	- Kernel Process	(without switch but accessing same shared variables as type 4).

For completeness we add:

6. Interrupt Handler - Interrupt Handler.

Since Interrupt Handlers never allow a context switch and do not access shared variables also accessed in type 4 critical regions, they are treated as type 2 in the sequel.

[4] As opposed to processors, processes cannot, in general, be freed from critical regions. Think for example of Disk IO, printers etc.

Type 1

With the advent of the IPC package in Unix System V, which furnishes user-level semaphores, protecting User Process- User Process critical regions is principally done as follows:

```
semop(semid,P);
    critical section
semop(semid,V);
```

where *semid* is the system call that performs operations on the semaphore *semid*. (To allow for several extra functions the system V implementation of semaphores is more elaborate. To present the full code would only derogate the clarity achieved above. For an example, see [BACH86] pages 374-375.)

Type 2

As discussed before Kernel Process-Kernel Process critical regions occur in the kernel code with no explicit protection at all:

```
    critical section
```

The protection given by a non-preemptive kernel no longer suffices, forcing us to explicitly protect such a region of code.

Type 3

Type 3 critical regions (P-IH) are protected as follows in the kernel mode process code:

```
disable interrupt;
    critical section
enable interrupts;
```

Since the interrupt handler can only be interrupted by higher level interrupts, with whose handlers no variables are shared, it does no protection of the critical region:

```
    critical section
```

Type 4

These critical regions consist of two parts which are protected in SPUX as follows:

```
while(flag == 1)            A
    sleep(event);           A
flag = 1;                   A
    critical section part 1
swtch();                    /* dispatch! */
    critical section part 2
flag = 0;                   B
wakeup(event);              B
    noncritical section
swtch();
```

The first 3 lines of code emulate, in a way, a P operation on 'semaphore' *flag*. The lines marked B emulate the V operation, releasing all processes waiting for *flag* to be reset. The protection with *flag* is necessary since the process may be replaced by another one in the *swtch()* function. The new process might want to access the critical sections part 1 and/or 2. If so it must first check the flag and go to sleep when it is set. Therefore the new process must either execute the code given above or, when it does not call the dispatcher, the code given below:

```
while(flag == 1)
    sleep(event);
flag = 1;
    critical section
flag = 0;
wakeup(event);
    noncritical section
swtch();
```

This code contains what we have called a type 5 critical region.
Janssens et al make the following notes to the flag protection mechanism. First of all they note that setting the flag in the first piece of code can be done anywhere between the while statement and the *swtch()* call. Also resetting the flag can be done anywhere between the first and second *swtch()* call. Shifting these statements down respectively up is done to optimize the code. Due to the non-preemptiveness of the kernel on a uniprocessor the protection stays intact. Finally Janssens et al remark that in the second piece of code the flag need not be changed at all, again because of the (SPUX) guarantee that the process is not preempted in the critical section.

From the above it is evident that protecting the Unix critical regions will be a tough job, not because the choice of the right protection mechanism is that difficult, but because it is hard to find the critical regions. Finding the beginning of type 4 and 5 critical regions is a favorable exception to the rule, They can be easily recognized from the leading while statement.

In our group a start to this formidable task has been made by A. Kwaks [KWAK87]. We refer to his work for further details in this area.

6.2 Process Scheduling

Process scheduling on a uniprocessor is a well known topic. Equally well known is the fact that the ways to perform scheduling are numerous and that ample research has been carried out to find still more and better ways of doing it.
On a multiprocessor several degrees of freedom are added and the complexity of the scheduling problem is therefore even greater. In fact, the scheduling problem defined as the problem of finding an optimal schedule as to throughput and response time, belongs to the class of np-complete problems, even when all knowledge about processes and processors is available on beforehand.
For this reason many scheduling methods try to achieve local optima in the hope that this renders a solution that is close to the global optimum.
Below we will discuss a number of these methods and setup requirements to which scheduling methods for multiprocessor should in general conform. Having done so we will discuss the situation for Unix on a multiprocessor.

6.2.1 What is Scheduling?

Let us first give some definitions related to process scheduling.

Definition 6.1

Process Scheduling is assigning processors to runnable processes in such a way that some function of one or more quantities is optimized.

Note that processors and processes may have arbitrary characteristics. Processes for example have the following:

- execution time
- resource usage
- priority
- precedence relations with other processes
- etcetera

Processors exhibit, amongst others, the following properties:

- scalar processing speed
- vector processing speed
- IO speed
- private memory

Definition 6.2

A Schedule is an assignment of processors to a given set of processes such that both where and when any individual process is run is given.

Definition 6.3

A P_SET is a set of processes containing single processes SP and group processes GP. A GP is built up of processes between which precedence relations are present. A GP cannot be further divided in subsets without having any precedence relation between two processes belonging to different subsets of GP. Any SP is free of precedence relations both with respect to other SP's as to GP's.

Note: A group process GP as defined here has also been termed Crowd [LEBL87] and Task Force [JONE79] in literature.

Using the last two definitions one can also say that Process Scheduling is the act of making a schedule for a P_SET to optimize some quantity.

The quantity to be optimized can be dependant of the following:

- Optimum response time
- Optimum processor utilization
- Safe Schedule (No deadlocks)
- Fair Schedule (No starvation etc)

6.2.2 How to Schedule

Let us define the entity that performs the scheduling, be it a program or a human, the **scheduler**. Then we may define two different ways of scheduling depending on the viewpoint of the **scheduler**. First we have what we call **central scheduling**
where the **scheduler** is hierarchically placed above the processors. (This does not mean that the scheduler function is not carried out by these processors!). Associated with central scheduling is one single run queue. (This does not mean that this run queue has no internal structure as will be explained in paragraph 6.2.6). Second we have **distributed scheduling** where there are as many schedulers as there are processors and where the viewpoint of each scheduler is as if it was a processor. In this situation we have per processor run queues.

Both for central as for distributed scheduling we can define a number of policies used to generate a schedule. These are depicted in figure 6.3.

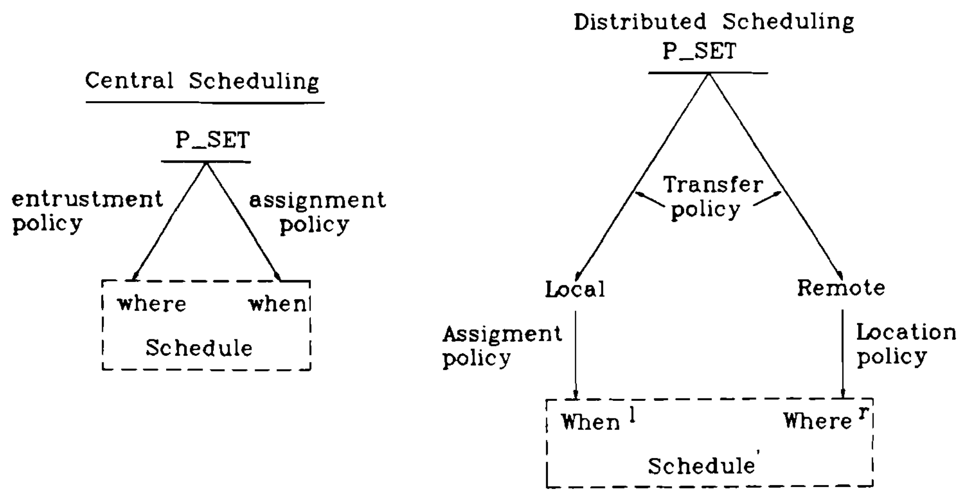


Figure 6.3 Scheduling Policies

For central scheduling we see a so called entrustment policy which decides which processor must run which processes. This policy may take precedence relations into account. The second policy for central scheduling is the assignment policy which determines when to schedule a process. In practice we meet scheduling algorithms that implement both policies in an interrelated form to obtain the optimum schedule, in other implementations we see that the entrustment policy is done first and then, independently, the assignment policy.

In case of distributed scheduling we find the transfer policy which decides whether a process must be executed on the local processor or must be transferred to a 'remote' processor. For processes that stay local the already known assignment policy determines when processes are executed. For processes to be transferred the location policy decides which processor they will be sent to. The apostrophe with the schedule indicates that this is the schedule for the local processor and that the total schedule must be obtained by combining all information from the local schedules. The terminology for the policies in this distributed case stem from [EAGE86]. Two remarks are due now. First observe that in the distributed case a global optimum for the entire system schedule is not practically achievable anymore. Even if such an optimum could be calculated in a reasonable time the calculation would cause a lot of communication between the processors (or better their schedulers) to gain access to all necessary information. But this effectively returns us to central scheduling for which better methods exist. Second we may see the combination of the transfer and location policy as the counterpart of the entrustment policy from the central case.

6.2.3 Types of Policies

Each of the policies as defined above uses some parameters to base its decisions on. Depending on the way these parameters are obtained we can classify the policies in each of the quadrants of the following matrix:

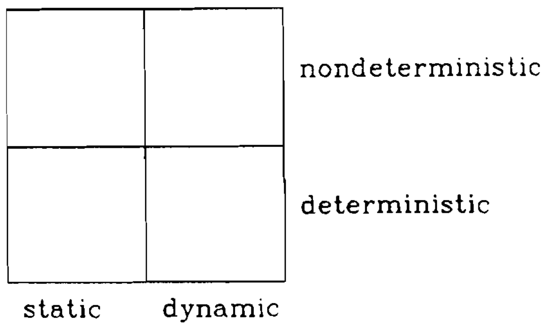


Figure 6.4 Scheduling Types

A policy is said to be static when the parameters it uses are determined once and fed to the policy at system startup time. A static policy will therefore never be able to adjust to any deviation of the system state from the one prevailing when the parameters were determined. Short-term deviations may be caused by work-load peak-hours or because department X is on holidays. Long-term deviations may be caused by an adjustment of the user community to the system given the set of parameters in use.

A dynamic policy on the other hand relies on parameters that are measured from the system at regular time intervals or that are requested when needed. A dynamic policy typically adjusts to medium-term and long-term changes in system state. If it would react to short-term changes unfavorable effects like continuous process transfers may occur.

The other dimension in the matrix above is whether the parameters are deterministic or non-deterministic. A deterministic policy uses deterministic parameters that may be either known on before-hand, estimated or measured. A non-deterministic policy uses stochastic parameters that are characterized by a probability density function (pdf). These parameters are in general either measured or calculated for certain types of processes. Most of the time however probabilistic models are used to calculate the (estimated) behavior of policies that are born from a different type of reasoning. Most of the methods we refer to are traditional assignment policies like round robin (RR), first come first served (FCFS), shortest job next (SJN) and priority scheduling.

Having said something about assignment policies we might as well add that these can be further divided along a third dimension, that of preemption. Some assignment policies choose a process to run which will then run until completion or until it is blocked. These policies are non-preemptive. Others preempt the processor(s) as soon as a higher priority process is detected in the set of runnable processes. (For reference: The Unix system basically uses round robin scheduling, allowing each process to execute for a certain amount of time (the time quantum). Processes that are not ready at the end of their time quantum are fed back to one out of many priority queues, dependent on its history. This assignment policy is known as round robin with multilevel feedback.).

6.2.4 Optimal Response Time

To obtain an optimum schedule a lot of information is necessary. Especially the scheduling of processes belonging to a GP is difficult. The information required to optimally schedule a GP comprises the precedence relations between the processes of the group and the execution times of these processes. The precedence information is often given as a graph while the resultant schedule is usually given as a so called Gantt chart. (See figure 6.5)

An example taken from [HWAN84] is used to clarify the type of reasoning underlying these schedule methods. The example uses a deterministic preemptive scheduling method due to [MUNT69]. As mentioned earlier this type of scheduling interrelates the entrustment policy and the assignment policy as we defined them.

It is assumed that there are p identical processors and a P_SET for which an optimal preemptive schedule must be derived. Furthermore it is assumed that all process execution times are multiples of some common value so that start- and endpoints of process executions are found at discrete points in time. The common value is called weight.

A schedule is derived for the GP as given in figure 6.5a and given in figure 6.5b.

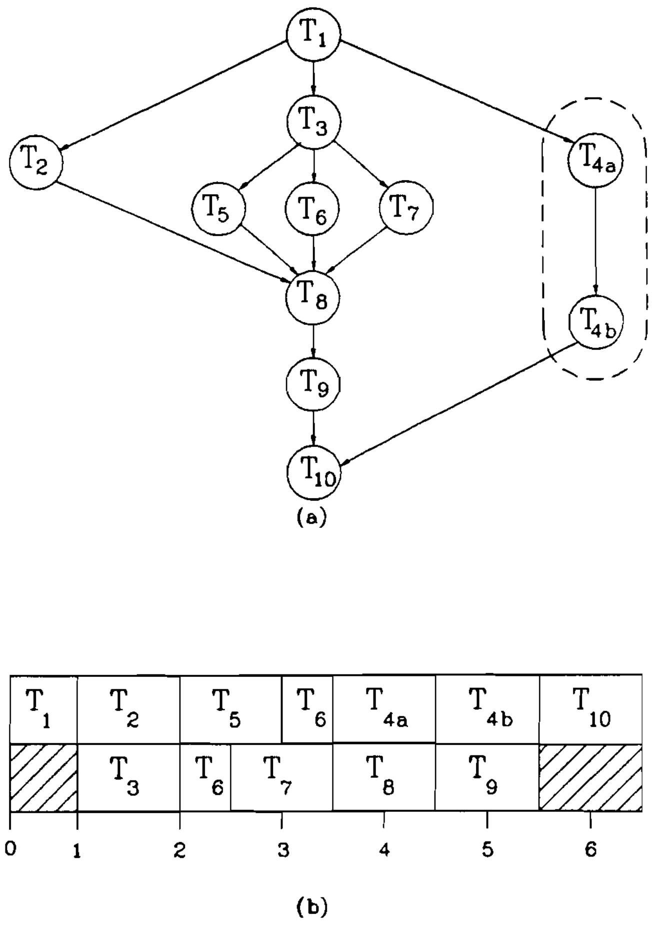


Figure 6.5 Optimal Schedule

The algorithm is as follows:

First transform the graph for the GP into one where all processes have unit-weight. That is transform all processes with a weight larger than 1 into strings of unit-weight processes. (See figure 6.5 process T4).

Second partition the graph into a sequence of disjoint subsets in such a way that all nodes in a subset are independent. Number the subsets, starting at the terminating node and working to the top, from 1 to N. The initial node thus obtains number N.

Entrust processes to the processors (in this example 2) starting at the highest numbered subset which contains the initial node. Every time a subset contains only one node a node from the next lower subset may be entrusted to the second processor unless prohibited by precedence relations (f.e. T2 belongs to subset 4 but is co-entrusted with T3 from subset 5). Every time a subset contains an uneven amount of nodes the method as used for T5, T6 and T7 can be used to reach minimum-time execution.

Since 1969 many other methods have been proposed for broader classes of processes. We will not discuss them here since this does not fall within the scope of this text. The sole purpose of presenting the above example here was to make the reader aware of the fact that when inherent parallelism in an algorithm is to be exploited on a multiprocessor, knowledge of the precedence relations between the processes that implement that algorithm is needed.

6.2.5 Optimal Processor Utilization

Processor utilization (or efficiency) is defined here as the cumulative amount of time a set of processors is executing processes divided by the cumulative amount of time they were available for execution of processes while runnable processes were present. On a uniprocessor processor utilization can be degraded by a large overhead due to process switching. On a multiprocessor another phenomenon may add to this degradation, namely load unbalance. Load unbalance is the situation where one processor has lots of work while others are idling.

Although optimal response time and optimal processor utilization are strongly related they are not quite the same. Examples exist where processor utilization is enlarged at the cost of the response time for the several users. Deriving a schedule for optimal response time is usually not practical, causing methods to be used that use other optimization criteria. One of these criteria may be processor utilization.
The situation with respect to achieving good processor utilization is somewhat different for central scheduling as for distributed scheduling as will be discussed next.

With central scheduling we have all information about all processes and all processors. This means that it will be relatively easy to obtain a good processor utilization. Unbalances in the load will not occur since the scheduler will assign a new process to every processor as long as there are runnable processes available.

With distributed scheduling it is quite different. Here every site-scheduler has all information about the local processes and only a subset of the total information about 'remote' sites. (If the information about remote sites were complete at every site we would effectively have returned to the central scheduling case. The central scheduler could then be considered to be composed of a number of cooperating subschedulers each having total knowledge.)
The partial availability of 'remote' information in the true distributed case gives rise to minor or major unbalances in the load of the respective processors.
To remove any unbalances processes can be either placed in a lightly loaded site upon their creation (initial placement) or they can be moved from one site to another even when they have been already executed for some time (process migration). Both initial placement strategies and process migration strategies belong to the transfer policy. If we model the creation of a new

process as a migration from an imaginary processor to a real one, we may include initial placement in process migration. A definition of process migration in our terminology, can then be as follows:

Definition 6.4

> **Process Migration** is the transfer of the duty to assign a process for execution, from one processor to another.

A process migration can only be successful when the destination processor has access to all relevant parts of the process's context. In order to be able to judge the consequences hereof we need to know how a process context looks like.

A process context is composed of a number of segments as depicted in figure 6.6 below. (We chose to present the context for a process in the Unix operating system since that is the operating system we are concerned with.)

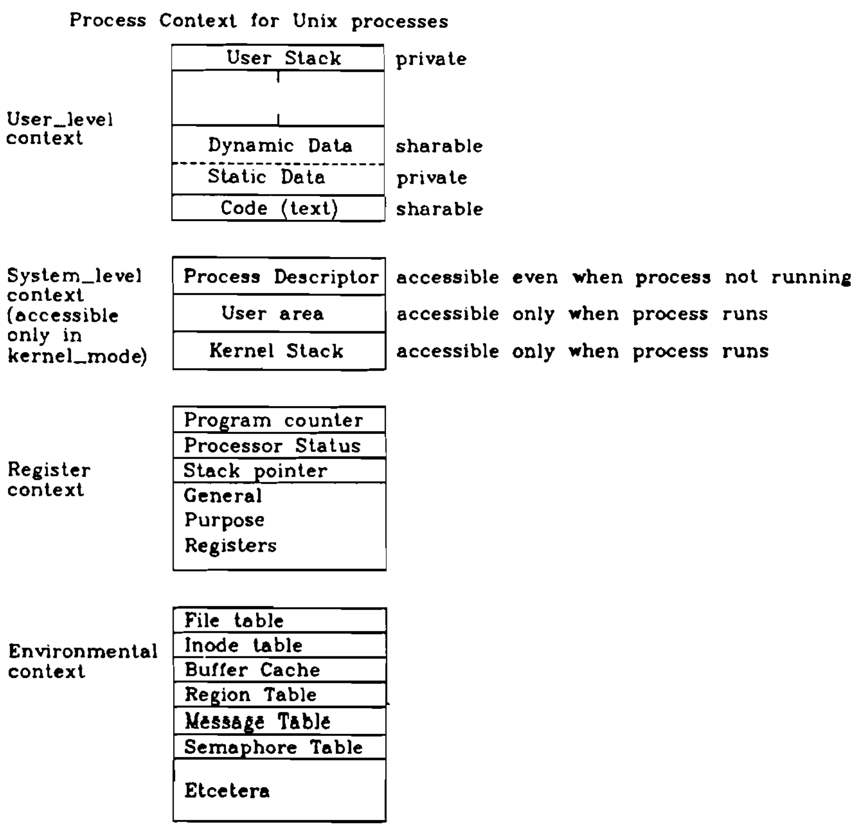


Figure 6.6 Unix Full Process Context

As you can observe the context is presented using a two-level hierarchy. This hierarchy is due to M.Bach [BACH86]. We also use his terminology to a large extent. The user-level context is, loosely spoken, the program. The system-level context contains all data-structures necessary to control (or manage) the process as long as it 'lives' in the system. The register-level context is

formed by the contents of all processor- or other registers used by the process when it is actually running. The environmental context is not especially set aside by M.Bach but does form an important part of the process's context, its importance becomes evident when trying to migrate processes in distributed computers [OUST88]. The environmental context is composed of bits and pieces of those system tables that relate to the process.
For the purpose of process migration we can ignore the register-level context since it will always be saved on the kernel stack before migration takes place.

In true distributed systems it will not be very easy to allow access to all these subcontexts after a process has migrated away. It is evident and one of the key advantages of multiprocessors satisfying our definition 2.2 that access can be guaranteed through the use of shared memory for appropriate parts of the process context. The severe problem of forwarding communication to a migrated process as encountered in distributed computers is also avoided in this way. Also no problems arise, at least not for migration, with shared segments of context.
A natural consequence of using private memory for parts of process contexts (as we intend to investigate) is that these problems return.

In the field of distributed computers initial placement and process migration have been thoroughly studied and still are. From these investigations a number of advantages gained through these methods are mentioned. The main ones are:

A. Load Balance. As discussed this can have a favorable effect on the performance of a system.

B. Fault Tolerance. Critical processes can be kept running. Some reasons for a process to be interrupted when running on some processor can be advertized in advance. A process that must stay running can then migrate to another processor.

C. Move small process to large data. In a distributed computer it is possible that an operation must be performed on large amounts of remote data. When the process that performs the operation and the result of the operation are small (or, when large, can be left at the remote site) , it is wiser to ship the process to the remote site than to fetch the data. When the operation finishes the result data are returned. An example of such a situation might be the gathering of statistical information from a remote database.

D. Remotely locking a resource may be impossible because remote locking primitives are not provided. In such a case sending the process to the remote site enables it to lock the requested resource there using local locking primitives.
Similar reasoning can be applied to file access mechanisms.

We may immediately conclude that advantages C and D above cannot be achieved on a multiprocessor since the underlying problems do not exist. Data will be at hand easily and locking primitives can be made of equal semantics for all processors. Advantage B has already been discussed in chapter 2 and is achievable in a multiprocessor too. As observed in the previous section load balancing (A) is also an item of discussion in a multiprocessor when distributed scheduling is applied.

A lot of load balancing strategies have been suggested in literature. Many of them try to control a certain quantity that has an intuitive relation to the balance of the load. Krueger and Finkel [KRUE84] for example try to keep the length of the processor run queues close to the system average queue length. Hoozemans measures all processor idle times to find the best site to move processes to [HOOZ86].
We will discuss two types of transfer and location policies explicitly targeted towards load balancing. The policies of the first type are known as Threshold policies, the other type uses process behavior to base decisions on.

6.2.5.1 Threshold Policies

The threshold policies discussed here are due to Eager et al [EAGE86]. They investigated three location policies with a fixed transfer policy in order to achieve insight into the complexity level needed to get near optimal results. This is also the main reason for treating them here. A multiprocessor like ours, employing a time-shared bus, suffers greatly from large bus loads. In order to minimize the load an impression of the complexity of load balancing policies is necessary to setup the right simulation experiments. These experiments should uncover the optimum compromise between the bus overhead caused by the policies and the effectiveness of the policies.

The system for which the policies are discussed is a distributed computer having K identical uniprocessor nodes. The nodes (or sites) are interconnected through a LAN with broadcast facilities.
A number of other assumptions are made mainly to simplify the system model. These assumptions are:

- The arrival rate L of new processes is identical for each site.
- There is only one type of process (average service time S).
- The cost of transferring a process from one site. to another is represented by an average processing cost C at the sending end.
- The processing cost of receiving a process is included in the service time S.
- The communication costs are negligible.
- The cost of obtaining state information from a remote site is negligible.
- The scheduling performed locally is independent of the actual process service times.
- The transferring of tasks has preemptive priority over processing tasks.

Eager discusses these assumptions thereby broadening the validity of his analysis to cases where site processors are not identical and where the arrival rate L is the same only over the long-term.

The transfer policy used is very simple. Any time the policy is invoked it examines the run queue, determines its length Q (= number of processes queued) and decides to transfer the process under consideration when Q>T, otherwise the process is executed locally. Eager et al have the transfer policy invoked only when new processes are created locally (and with one location policy when processes arrive from other sites). Processes that have been partially executed at some site will never be transferred to another site because of the problems that may potentially occur. (Think for example of the environmental context).

As soon as the transfer policy has decided to transfer a process the location policy is invoked to decide on the destination of the process. Three different location policies have been investigated by Eager et al. These policies have been selected for their increasing level of complexity. The level of complexity is strongly related to the amount of information about the state of remote sites that is necessary.
The three policies in order of increasing complexity are:

- Random
- Threshold
- Shortest

The **Random** policy chooses the destination site at random. A process arriving at the destination site due to a transfer is treated there as if it was created locally. This means that there is a risk that the process is further transferred. If the number of transfers allowed is unbounded the Random policy is unstable, that is after sufficient time all sites will be transferring processes to each other without executing any of them. (For a proof see [EAGE86]). For this reason a counter field is tagged to the process in transfer to indicate the number of transfers it has gone through since its birth. By imposing an upper bound L_t (the transfer limit) onto the value of the counter this undesired behavior is avoided. A process that arrives at a site with a counter value L_t must be processed (and not transferred) by that site.
Note that the Random policy does not need any state information from other sites.

The **Threshold** policy also chooses a destination at random. It does not however send the process immediately but first inspects if the chosen site is above threshold. If so it retries for a maximum of L_p times. L_p is called the probe limit since inspecting a remote site for the required information is called probing. When sites probed are all above threshold, L_p will be reached and the process is not sent at all but kept for local execution. When a process arrives at the destination site it will be processed there.
Note that the Threshold policy needs, at the most, L_p binary values representing the state of L_p remote sites. In general the process will be sent within less than L_p retries.

The **Shortest** policy randomly probes L_p sites and sends the process to the one having the least number of processes on queue. Of course the process is not sent when all L_p remote sites are above threshold. When a process arrives at the destination site it will be processed there.
Note that the Shortest policy always needs L_p remote state values. The values needed are integers in this case.

To give the reader some more background we summarize the modelling method used by Eager et al below.
The entire system has been modeled as a queueing network in order to calculate its behavior. The calculations were validated through simulation for a system employing 20 site-processors.
To simplify calculations the sites of the network are assumed to be stochastically independent. Eager et al state that this is an asymptotically exact simplification. This means that for finite systems the analysis is an approximation. Because of the 20-site simulation the analysis can be

applied to systems with 20 processors without difficulty. For systems with fewer sites an extra validation should be done.

Because of the simplification the system now is a collection of stochastically independent sites that feed each other with processes in transfer. These streams of transfer processes are modeled as an arrival process for each site with arrival rate L_t.
In equilibrium a site sends as much processes away as it receives. Because of this balance it is possible to calculate the behavior of the entire system (at equilibrium) from the behavior of one node. A node can be seen as a birth-death process as depicted in figure 6.7.

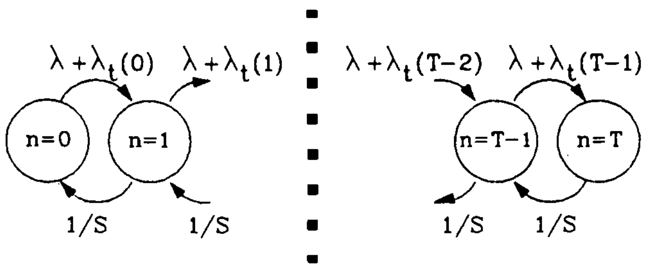


Figure 6.7 Birth-Death Model

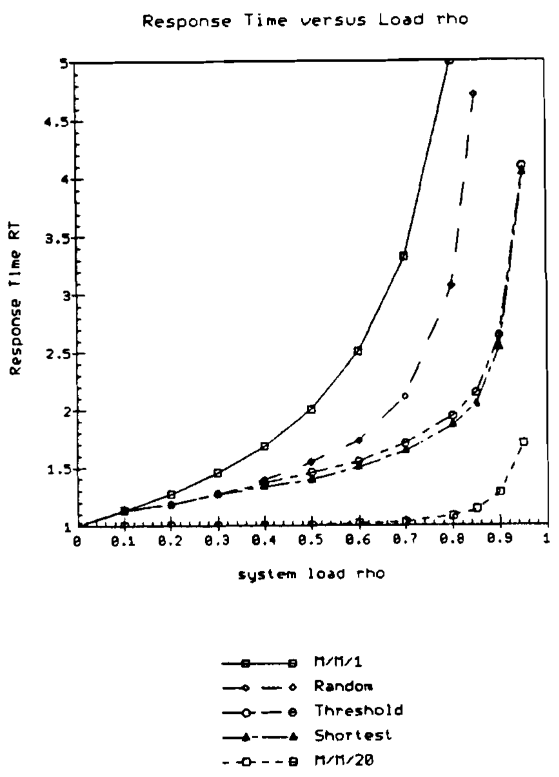
In this figure, two phases a site can be in, are indicated. A site is in the processing phase when its queue length is below or equal to the threshold T, otherwise it is in the transferring phase. In the processing phase the site-processor either idles ($n = 0$) or executes processes ($0 < n <= T$). In the transferring phase it either executes processes or transfers them ($n > T$).
The conduct during the processing phase is calculated using standard methods. The behavior of the transferring phase is identical to that exhibited by a two class, preemptive priority HOL M/M/1 queue in its busy period [KLEI76] and calculated as such.

Eager et al compared the three location policies with each other and with two bounding cases: K independent M/M/1 queues and one M/M/K queue. The former has no load sharing at all the latter has perfect load sharing. The overhead for the M/M/K queue is assumed to be zero. In a real system at least some overhead will be present so that the performance of some policies may be judged somewhat better than may appear from the graphs.
The quantity that is compared is the mean response time of processes.
We will only repeat the principal performance comparison done by Eager. The principal comparison is made using figure 6.8 as a guide. This figure gives average response time RT versus system load R (= site load at equilibrium). There are a number of parameters to the figure with values as indicated. Eager et al motivate these values in their article. We will discuss them later.

It appears from the figure that the **Random** policy substantially improves the response time. For example for a load of 0.7 response time decreases from 3.4 to 2.1 times the uniprocessor response time S.

The **Threshold** policy does even better due to the use of some remote state information. At a load of 0.7 the response time becomes 1.7S. Such a decrease of RT indeed is what one would expect.

The **Shortest** policy disappoints with respect to the expectations. Intuitively one would have expected another substantial increase over the Threshold policy. This is not so as the average response time reaches 1.6S when the Shortest policy is adhered to, only 0.1S below the Threshold value.



Eager et al have invested considerable effort in validating the above outcome. They investigated into the sensitivity of the policies to the transfer cost C, the choice of Threshold T and probe limit L_p. The network traffic caused by probing and transferring processes may turn out to create a bottleneck for the system. In a multiprocessor with a time-shared bus as the network and employing local memory, the amount of extra traffic induced by the load sharing policies also is a critical factor. Eager et al calculated the rates of transfer and probing for each of the policies. These are shown in figure 6.9.

S(task service time) = 1.
C(cost of task transfer) = 0.1
T(threshold) = 2
L_p(probelimit for Threshold and Shortest) = 3
L_t(transfer limit for Random) = 1

Figure 6.8 Threshold Policies Performances

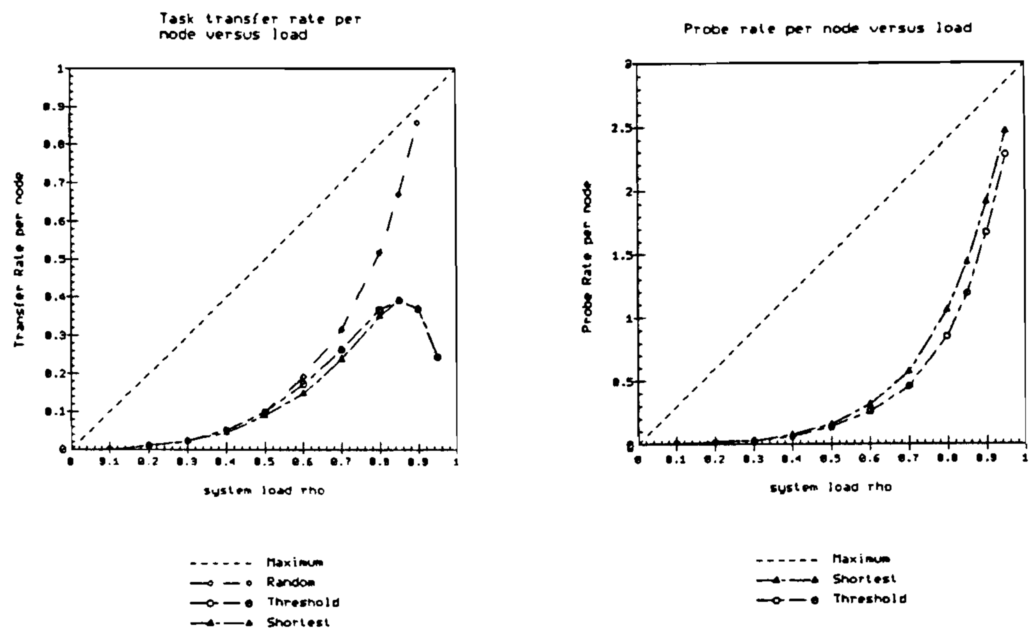


Figure 6.9 Process Transfer and Probe rates

The policies that use state information are much more efficient at high system loads in that they refrain from process transfers when they are not beneficial. This advantage is gained at the expense of some probing information. However the amount of traffic for probing is likely to be much less than that for a process transfer.

Eager et al conclude that their analysis predicts system behavior very well over a wide range of parameter values. This especially is true for the relative comparison of the policies. Their conclusion that the complexity level needed to achieve near optimal results can be very modest appears to be just. A modest complexity level has the advantage of a higher degree of stability, a better maintainability and less overhead in processing time and network load. As mentioned earlier especially this last aspect is of interest to us.

6.2.5.2 Process Behavior based Policies

Scheduling policies can also be derived starting from the **actual** behavior of processes. An argument for taking this approach is the observation that actual process behavior is far from exponential. For the Unix environment this observation is made in a.o [LELA86] and [CABR86]. In scheduling models based on queueing theory and often also in simulation models it is commonly assumed that a process has an exponential arrival rate, CPU usage, disk usage etcetera. The empirical data show otherwise. Leland and Ott [LELA86] observed from their data the following 3 classes:

1. Large CPU intensive processes with relatively few disk accesses.
2. Large disk intensive processes with relatively little CPU-usage.
3. Ordinary processes that need little of both.

Class 1 forms the 'tail' of the distribution of CPU-usage. This tail is much thicker than with the exponential probability distribution. The same goes for the disk access arrival rate. Leland and Ott also showed that the 98% smallest processes needed 35% of total CPU time while the 0.1% largest needed 50%. Intuitively, a lot of effort may be put in transfer and location policies for large CPU intensive processes for the overhead for these processes will be relatively low. Also an even distribution of these processes over the processors has largest effect on total system response times since they account for 50% of CPU time. Small processes will also benefit from evenly distributed large processes for then they will hardly ever have to run in the presence of more than one large process. The data from [CABR86] support the above reasoning. Cabrera found that 50% of all processes needed less than 0.4 seconds and between 78% and 95% less than 1.0 seconds of CPU time. Most processes should therefore not be moved between processors because of the relatively large transport overhead.
Two remarks are due. First of all when the vast majority of processes is small, a lot of user sessions will be composed of only such small processes. To provide such a user with improved response times, balancing the large processes helps, but is not enough. Crowds of small processes may exist on a particular processor. The expected exit time of one such process may therefore lie far into the future because many higher priority processes run first. It might be better to ship the process (or a group of processes) to another ,less loaded, processor and receive it back before the time it would have exited locally, in spite of large transport overheads. Second, balancing small processes becomes possible in the environment of a shared memory multiprocessor because of the high IPrC bandwidth. For example in a distributed environment (often used as the target environment in papers discussing migration policies) based on VAX750s the IPrC bandwidth is not higher than 100 kilobytes average at the user-level [LELA86]. In a shared memory environment with Unix operating system the bandwidth can easily reach 1 Megabytes per second and, with some effort, 3 Megabytes per second and more. Still, even in the shared memory case a number of processes use so little CPU time that it is not worth balancing them. For this reason we see that most migration policies based on actual behavior, alike the Threshold Policies, have a threshold (as was to be expected).
Hence threshold policies emerge from both the 'exponential camp' as the 'actual camp'. This makes one wonder how the actual behavior influences the policies. For the case of Leland and Ott the empirical data are used for two purposes. First it is used to justify a threshold and to estimate an efficient value for it. Second it is used to derive an estimation rule for the classification of an unfinished process above or below threshold. The rest of the policy (not discussed here) could have equally well been based on exponential behavior. To estimate whether a process's total CPU-usage lies below or above threshold the CPU usage so far has to be measured and the residual CPU-usage has to be estimated. Leland and OTT found the mean residual CPU-usage x_r of a process that already used x CPU seconds to be approximately:

$x_r \approx k_1 + k_2 x$ with k_1 very small
and $x > 3$ seconds.

Hence a process's lifetime can be estimated as

$$x + x_r \approx x + k_2x = (1 + k_2)x$$

for x larger than 3 seconds.

Since the class of processes that is of importance to migration in a shared memory multiprocessor has an average CPU-usage of less than 3 seconds the above estimation cannot be accepted without further study into x_r for x smaller 3.

6.2.6 Congestion Problems with Scheduling

A number of bottlenecks in the multiprocessor will prevent its performance to rise unlimited with the number of processors N. The main bottleneck will be the time-shared bus. By taking appropriate measures this bottleneck can be widened to allow for at least 16 processors [FREE84]. The first measure is to use caches with every processor which allows for somewhere between 4 and 8 processors. The second measure is the use of local memory for the process descriptor and process stacks. This last measure raises the number of effective processors above 16. (e.g Freeman and Kaplinsky measured a relative throughput of 0.89 on a multiprocessor employing 16 processors, using 8k of cache and 1k of local memory. The throughput is relative to a system of 16 independent ideal uniprocessors of the same type as the processors in the multiprocessor.).
As soon as the time-shared bus bottleneck is eliminated new bottlenecks will become apparent. These will mainly be caused by congestion problems when accessing shared data structures. An outstanding case is the run queue. This queue is heavily used by all processors in the system. Assuming for now that we stick to the single processor Unix scheduling mechanism, using the same time-slice of one second, this would mean that at least N (= the number of processors) times per second an access to the run queue occurs. This number will be higher because processes are created, get blocked, must be woken again and exit[1]. All of these actions require access to the run queue. For an arrival rate of 1 process per second and an IO rate of 2 per process (above the loading of the process code (exec)) this adds at least another 10 accesses. The total number of accesses could therefore easily reach 100 on a 10-processor system under a modest load. This means that when all these accesses are equidistant, run queue manipulation may not take more than 10 milliseconds. As indicated 100 is an underestimated number which, together with the effects of randomizing the accesses, will leave less time for the run queue manipulations, potentially causing performance degradation.
To eliminate this bottleneck it is possible to partition the run queue into more queues. The most obvious way to do this is to give each processor its own queue, containing the processes to be run by that processor only. This will necessitate load balancing for then we have distributed scheduling.
A solution also exists for the case of central scheduling as indicated by Cezzar and Klappholz [CEZZ83].

1. Cabrera measured a maximum of 670 processes being created in 5 minutes on a VAX 11/785. This means more than 2 creations per second, more during peaks.

Cezzar and Klappholz try to show the effect of multiple run queues when processors access these randomly and uniformly. Because of the manner in which the queues are used we effectively (or functionally) have 1 central queue.
Cezzar and Klappholz make the following assumptions about the system they declare their results valid for:

- Identical Processors
- Uniform access to common memory; where no processor is favored over others
- Uniform sharing of IO resources.

Our system fits in this class. They further recognize a functional entity that performs the following actions:

- Creating and Destroying processes
- Putting processes to sleep and waking them up
- Scheduling

This functional entity is referred to as the Program Management System (PMS). This PMS consists of executive code, one or more run queues and one or more blocked queues. Cezzar and Klappholz present their results using two measures, effective throughput and the number of effective processors K (= our Ne). We will not concern ourselves with the effective throughput but only present their results regarding the effectiveness of processors.
A queueing-analytic model is made to calculate the desired values. In this model a run queue is modeled as a server having a service rate of M.
Three cases are considered. The first case treats the situation where only one run queue exists. The second case treats a system with m run queues in which processors choose an unlocked run queue (e.g. to fetch a process) without any cost. If all queues are locked processors choose one at random and spin lock until it becomes free. The third case is like the second case except that a processor immediately chooses a run queue at random without even investigating whether it, or another one, is free. When the chosen run queue is locked the processor spin locks even if other queues are free. This third case is introduced to avoid as much as possible the overhead for choosing a free queue, that is assumed to be zero in case 2. It is this case that is finally proposed by Cezzar and Klappholz as tractable for practice. For the first two cases results have first been derived using deterministic values for arrival and service rates. These results give an upper bound on the effectiveness of the N processors in the system. Then the results using stochastic arrival and service rates have been calculated.
The 'probabilistic' results from the first and second case serve as a lower and a higher bound respectively for the third case.

The following symbols are used:

N	=	number of processors
Ne	=	number of effective processors
L	=	arrival rate of run queue accesses
M	=	service rate of run queue
R	=	run queue access traffic intensity (L/M)
A	=	PMS overhead in a 1 processor system (L/(L+M))
m	=	number of run queues

Case 1: One run queue (m=1):

The following graph presents the results for the deterministic and the probabilistic case. (All results are measured from the original graphs).

Cezzar and Klappholz conclude for the central queue system:

- there is an upper limit to the number of processors executing processes equal to 1/R independent on N. This upper limit is reached for N = (1+R)/R = 1/A in the deterministic case.

As an illustration of the probabilistic case consider the following numbers:

Increasing from 10 to 15 processors in the above graph increases the number of effective processors Ne with 2.5 .
Increasing from 15 to 20 increases Ne by 1.

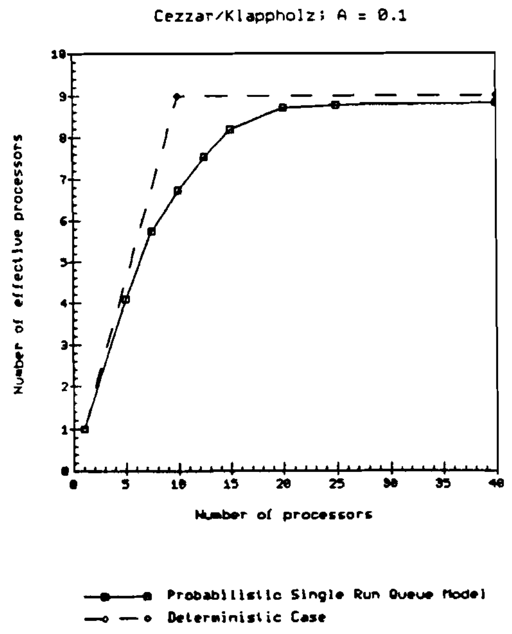


Figure 6.10 Single RunQueue Bottleneck

Case 2: Multiple run queues (m>1); Choose free one:

Processors choose free run queue if any, otherwise at random.
For this case the additional assumption is made that the time spent in a run queue does not depend on m!

Observe the graph:

This graph presents the results for m=AN and for m=N run queues. In the deterministic case the marginal gain of adding a processor becomes independent of N when AN run queues are used. The curve for this case is the same as the one for the probabilistic case shown in the graph, where m=N. In the probabilistic case m=AN is not enough, at least N run queues are needed to achieve constant utilization.

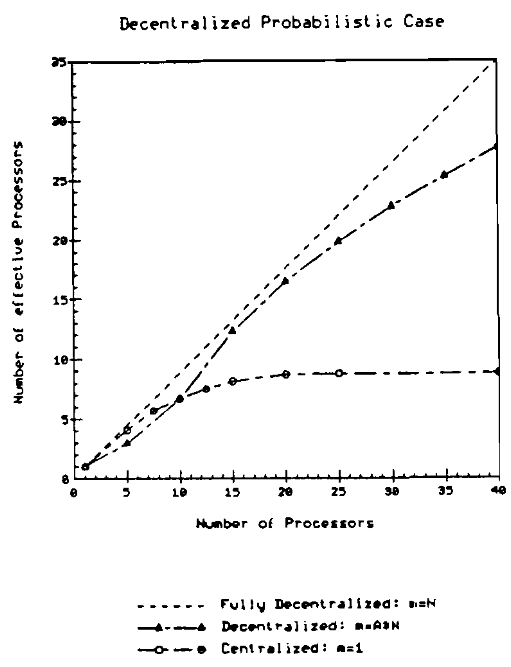


Figure 6.11 Decentralized Run Queues

Cezzar and Klappholz conclude:

- there is an upper limit to the number of processors executing processes equal to m/R independent on N. This upper limit is reached for N = m/A in the deterministic case.

- for constant utilization rates m must be at least equal to N in the stochastic case.

When also taking into account the centralized overhead for getting the information whether queues are unlocked the upper bound becomes:

M'/L for N >= 1+(M'/(N-1)A)(from [CEZZ83])

where 1/M' is the overhead time spent in the central server. Because this overhead cannot be avoided Cezzar and Klappholz proposed to access a run queue at random:

Case 3: Multiple run queues (m=N); Choose one at random:

For the deterministic case the results have been obtained through simulation. For the probabilistic case the results have been calculated using an approximation of a queueing network. The graph below gives all results.

Cezzar and Klappholz conclude:

- When m = N run queues with random access policy are employed, speedup factors very close to N are achievable.
- The random access policy is a viable method for avoiding centralized mechanisms and the resulting bottleneck conditions.

and they remark:

- The random routing policy as proposed does not take into account accesses to other then the run queue resources.
- Factors such as what happens when a run queue is found empty are not considered.

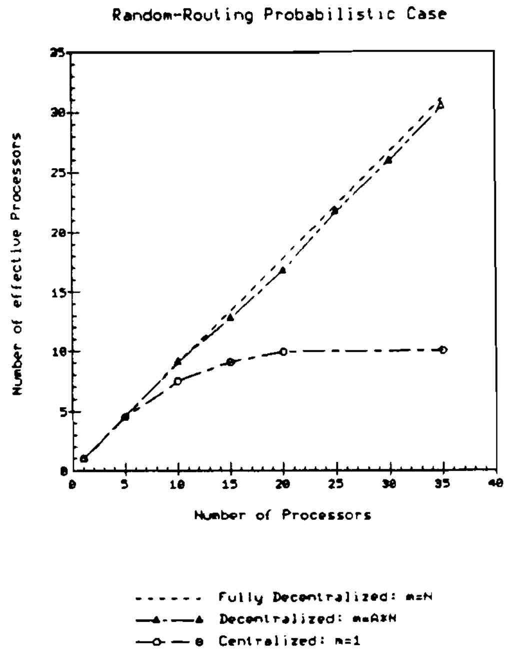


Figure 6.12 'Random-Routing' Run Queues

6.2.7 General Requirements for Scheduling

The scheduling of processes on a system's resources is done with a specific goal in mind. Multiprocessors too are developed with several goals in mind. One naturally hopes to achieve all possible benefits from ones ventures but practice learns that this is hardly ever the case. This is also the reason that some multiprocessors are designed to increase throughput while others are aimed at decreasing response-times. The first type of multiprocessor is called throughput-oriented, the latter speedup-oriented multiprocessor. The type of scheduling in these types of multiprocessor will therefore differ. In the speedup-oriented type it is necessary to know the parallel structure of a program in order to achieve the speedup, in the throughput-oriented type one seeks to balance the load, considering all processes to be independent.

The requirements for the scheduling are therefore also different for the two types. Furthermore there exist a number of requirements coming forth from pragmatic arguments, like for example the dedication of one or more processors to a specific task. There also are a number of requirements that must be fulfilled in all of the possible situations. We do not attempt to classify the requirements, this would lead to a bulky classification scheme, but we will present one unified list of requirements for scheduling.

Scheduling Requirements:

The scheduling mechanism must make it possible to,

- dedicate certain processors to certain (sets of) processes
- dictate the maximum number of processors to be used in parallel for a given application.
- (If special processors exist) make use of special processor characteristics.
- switch between preemptible and nonpreemptible scheduling (for research purposes only)
- prioritize processes and/or applications.
- dynamically adjust scheduling parameters to tune the scheduling to the work load (manual tuning of active system).

The scheduling mechanism must

- Schedule P-SET processes optimally.
- balance the system load.
- automatically adjust scheduling parameters (e.g. time slice) to the number of processors and to the load.
- log information pertinent to scheduling.
- minimize context switches.

6.3 Scheduling in a Unix based Multiprocessor

In this section we will investigate scheduling for our own special case, the Unix based multiprocessor. We will first discuss the Unix single processor scheduling. Based on this discussion we will drop some requirements from the set given in the previous paragraph. The remaining requirements are given in 6.3.2. In paragraph 6.3.3 the scheduling is extended to the multiprocessor case. Finally 6.3.4 summarizes.

6.3.1 Single Processor Unix Scheduling

Unix is a time-sharing system. This means that all users share the processor. The way in which this is achieved is to give each process a quantum of the processor time which is called time-slicing. (Users with many processes thus receive many processor time and users with few processes receive little.(See [HENR84] for a fair sharing of processor time).). Unix is also targeted towards the processing of largely interactive processes where response time must be low. For this reason processes are attributed a priority that is manipulated in such a way that interactive processes are favored over batch processes. The process of highest priority is selected to run at certain points in time. When several processes of the same priority exist they are served on a FCFS basis. Although al processes that are ready to run are kept on one run queue the behavior of the scheduler can be more easily discussed by placing all processes of equal priority in a separate (imaginary) queue.
The characteristics mentioned above place single processor Unix scheduling into the class of schedulers that is known as *round robin with multiple feedback*.

Explaining Unix scheduling is easiest done using the following figure. This figure gives an overview of the queues that play a role in the scheduling mechanism.

The queues are divided in two sets, the set of **kernel priority** queues and the set of **user priority** queues. Kernel priorities are higher than user priorities, processes of kernel priority will therefore run before any process having user priority. Processes normally operate in user mode where they have one of the possible user priority values. When a process wants to perform IO it does a system call. Because IO takes time the kernel puts the process to sleep until IO is done by placing it on a sleeping queue. As soon as the IO is done the processes awaiting this, wake up and are placed in one of the kernel priority queues visible in figure 6.13. The process has been given the kernel priority when it went to sleep. Kernel priorities are used to minimize the time that processes own valuable system resources. The kernel priorities are leveled such that processes performing the most critical system actions have highest priority.

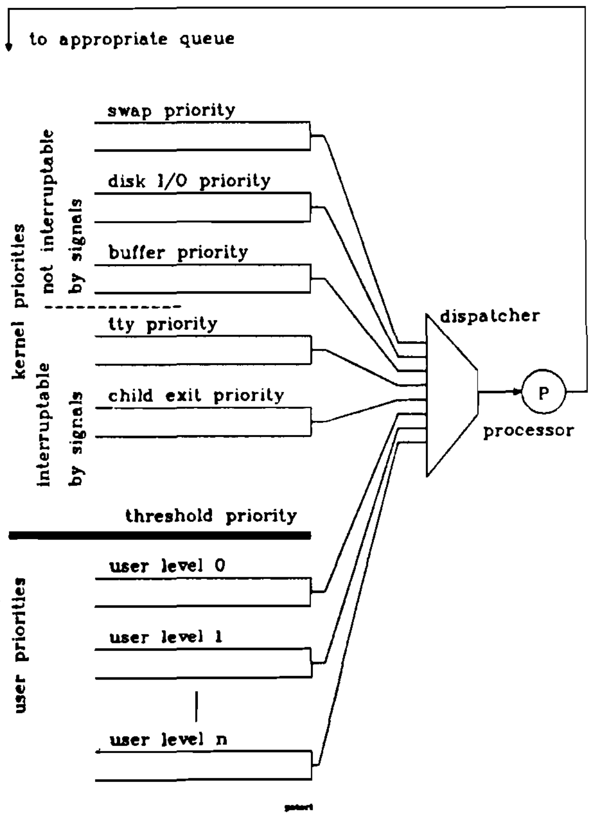


Figure 6.13 Unix Scheduling

When a process is running in user mode it may reach the end of a time quantum period. When at that moment a process of higher priority is present in one of the queues the current process is replaced by the highest priority process available. The replacement is done by the dispatcher (often named *swtch* in Unix sources) shown in the figure. The dispatcher's only function is to find the highest priority process available and make it running (after saving the context of the current process when one was running). The dispatcher does so on request. Requests are made when a process returns from kernel to user mode and a higher priority process is present, when a process is put to sleep and, as said, when a time quantum ended.

That the dispatcher is simple is nice but makes one wonder when and where the priorities are determined. The calculation of the user priorities is the essential scheduling that takes place. The priorities are calculated and/or assigned at the following occasions:

Event	Priority assigned
Process creation	Initial Priority
Process used up quantum	Preempted Priority
Timeslice event	Timesliced Priority
Process returns from kernel to user mode	User priority
Process is put to sleep	Kernel Priority
Process wakes up	Penalized Priority

The initial priority is a fixed value in system V apart from one component called the *nice priority* which is inherited from the parent process. The meaning of this *nice priority* is explained further on. The initial priority is given to a child process during the fork system call.
The timesliced priority is given by the clock interrupt handler. The clock interrupt handler does this every second just before it calls the dispatcher, but only for ready to run processes in user mode. The calculation of the priority is done according to the following two formula's which are evaluated in the order given:

CPU_usage = decay(CPU_usage)

Priority = f(CPU_usage,threshold priority,nice)

The CPU_usage is a field in each process descriptor. A process acquires a value in this field when it is in execution. The CPU_usage field is incremented for the current process by the clock interrupt handler on *every* interrupt and not only once per second. Clock interrupts usually occur at line frequency rate which is 50 or 60 Hertz. (sometimes even twice that rate). So processes in the run queue may have a CPU_usage that is not 0 when they have previously been in execution, but have been preempted from the processor. In Unix a low integer value represents a high priority and vice versa. The priority formula is such that when the CPU_usage is decayed the priority value decays and hence the priority rises. In this way processes that do not receive CPU time are migrating from low priority queues to high priority queues, finally obtaining the highest priority which will have it executed at the next process switch.
When a process has used up its time quantum its priority is calculated according to the same two formula's applied in exactly the same way.

Processes that return from kernel to user mode check a *switch flag* usually called *runrun* to see if a higher priority process has arrived in the time the process was not preemptable. If so the process's priority is recalculated according to the forementioned formula's and it is put in the run queue, giving way to the higher priority process.
We already explained how processes acquire kernel priority when they are put to sleep. When these processes wake up there priority is recalculated using the same formula for priority calculation again. Its CPU_usage has been decayed every second during the period that the process was blocked like for all ready-to-run processes and need not be adjusted at wake up time. One additional component is added to the priority value which is a penalty for using the valuable system resources.

Typical values for priorities are 0 to 50 for the kernel priorities (50 = threshold priority) and 127 for the level n user priority. The decay and priority formula's are different for every version of Unix. For example:

decay(CPU)

Version 6:	= max(threshold priority,CPU-10) + nice
Version 7:	= 0.8*CPU + nice
System V:	= CPU/2 + nice
BSD 4.2:	= ((2*avenrun)/(2*avenrun+1))*CPU + nice where avenrun = average nr of processes in the run queue

priority

Version 6+7:	= CPU/16 + threshold priority + nice
System V:	= CPU/2 + threshold priority + nice
BSD 4.2	= CPU/4 + threshold priority + 2*nice

Note: BSD 4.2 employs a scheduler that is more complex in the way it gathers and uses information. This makes straight comparison of the above formula's risky. It also makes the BSD 4.2 scheduler expensive: Up to 10% of CPU time may be used for priority calculation [LEFF84].

The *nice* value can be incremented by the user by calling the *nice* system call. It generally ranges from -20 to 20. Only the super user can use negative increments. This component of the process priority is meant to allow the user to be 'nice' to fellow users by lowering his or hers process priorities.

The characteristics of the scheduling discussed above makes Unix a friendly operating system for interactive processes. These processes have a high sleeptime-to-CPU_usage ratio which usually causes priorities to have risen to the threshold level by the time they are ready for execution. On the other hand batch processes will not starve from lack of processor time since they too will reach the highest priority level. In general this will take more time because their priority is lowered a lot due the high CPU usage at preemption time.
One peculiarity about Unix System V scheduling must be mentioned. It is that processes just dispatched are not given a full time quantum but only the remainder of the current one. This means that the successor to a process that goes to sleep at the very end of the current quantum

will have an extreme short quantum. Although it is highly probable that this successor will be awarded a second quantum immediately (because its priority can hardly have changed) it is not an elegant method that causes more overhead than necessary. Newer schedulers grant processes that enter the processor a full quantum.(e.g. [STRA86]).

6.3.2 Practical Requirements for Scheduling

It will be extremely difficult to provide a scheduler that will optimally schedule the P-SET processes as discussed earlier. Besides the problem of providing process characteristics on beforehand, the algorithms available today are not universal enough.
Therefore we drop the requirement that a P-SET must be scheduled optimally. In fact this means that we choose for a throughput-oriented multiprocessor.

Practical Scheduling Requirements:

The scheduling mechanism must make it possible to,

- dedicate certain processors to certain (sets of) processes
- dictate the maximum number of processors to be used in parallel for a given application.
- (If special processors exist) make use of special processor characteristics.
- switch between preemptible and nonpreemptible scheduling (for research purposes only)
- prioritize processes and/or applications.
- to dynamically adjust scheduling parameters to tune the scheduling to the work load.

The scheduling mechanism must

- balance the system load.
- automatically adjust scheduling parameters (e.g. time slice) to the number of processors and to the load.
- log information pertinent to scheduling.
- minimize context switches.

6.3.3 Multiprocessor Scheduling

A number of possibilities exist to perform multiprocessor scheduling as seen before. From these Central Scheduling seems to be the most favorable when it is combined with so called processor self-dispatching. In this case the process that is to run next is determined by a central mechanism while the switching of processes is carried out by a dispatcher per processor. In this way no process migration is necessary and load balancing is achieved inherently. Processors that cannot carry on with the current process fetch a new one from the central run queue preventing idling in the presence of ready-to-run processes.

Our first attempt will therefore be to employ central scheduling with decentralized dispatching. This very closely resembles the nature of the Unix System V uniprocessor scheduler which has a priority calculation that is easily kept global and which has a clearly identifiable dispatcher for the processor. For several reasons outlined below it would be nice to use this existing scheduler. However this scheduler has been criticized too. Therefore the main decision to be taken first is whether we indeed use the single processor Unix scheduler as it exists now as a starting point for the MPUX scheduler or that we develop or adopt a new one.
The advantages of the first option are that we will work with a well known scheduler. Its functions are known, its implementation is known and its behavior is largely known. Choosing this scheduler will also provide easier porting of future releases of single processor Unix onto the multiprocessor.

The SPUX scheduler has been criticized on the following points:

- Not highly tunable; available tuning (nice) gives unpredictable effects.[STRA86]
- Not very well suited for large programs.
- Not very well adapted to type of majority of processes.
- Costly priority assignments.[STRA86]
- Fluctuating quantum sizes.

Because of the disadvantages it will be of benefit to use an entirely new scheduler. In [STRA86] such a scheduler is discussed which, although new, still is in the spirit of Unix (simplicity, responsiveness). It has a much more process-behavior-related way of priority calculation causing better predictability and less overhead. Priorities are in- or decreased when certain events in the life of a process occur. For example the priority is lowered when a process consumes a full quantum while it is increased for terminal IO events. Furthermore each process is given a full time quantum when it is elected to run. Another scheduler can be found in [JACO86].

We will stick to the single processor Unix System V scheduler (at least for the time being) because of the ease of implementation. We aim to keep the scheduler fully intact, making only minor changes to make it more tunable. In this case dispatching of processes is the same as with the uniprocessor. The priority calculation and assignment is also done in the same manner as in the case of the uniprocessor. This is straightforwardly extended to the multiprocessor environment with the exception of the actions of the clock interrupt handler. The modifications to be made to allow for these (global) actions are presented in paragraph 5.2.2.
All data structures related to scheduling are kept the same except that they are protected against corruption. Suggested enhancements are to make the time quantum size dependent on the system load and the number of processors and to allow for a smaller quantum. Further the decay and priority formulas should be alterable (by the system administrator), possibly within hardcoded limits.

The above straightforward copying of the SPUX scheduler might cause a bottleneck to occur as indicated by Cezzar and Klappholz. In order to investigate the number of processors that will be efficient in our multiprocessor a GPSS model was made to simulate the behavior of the multiprocessor with respect to scheduling.

6.3.3.1 Simulation Model

The model that has been built is written using GPSS (General Purpose Simulation System) [GPSS88]. Our objective was to model the Unix Process Management System with a level of detail that would allow us to quantify the influence of the number of queues, queue manipulation times and arrival times for queue manipulation. The following figure presents the system that was modeled:

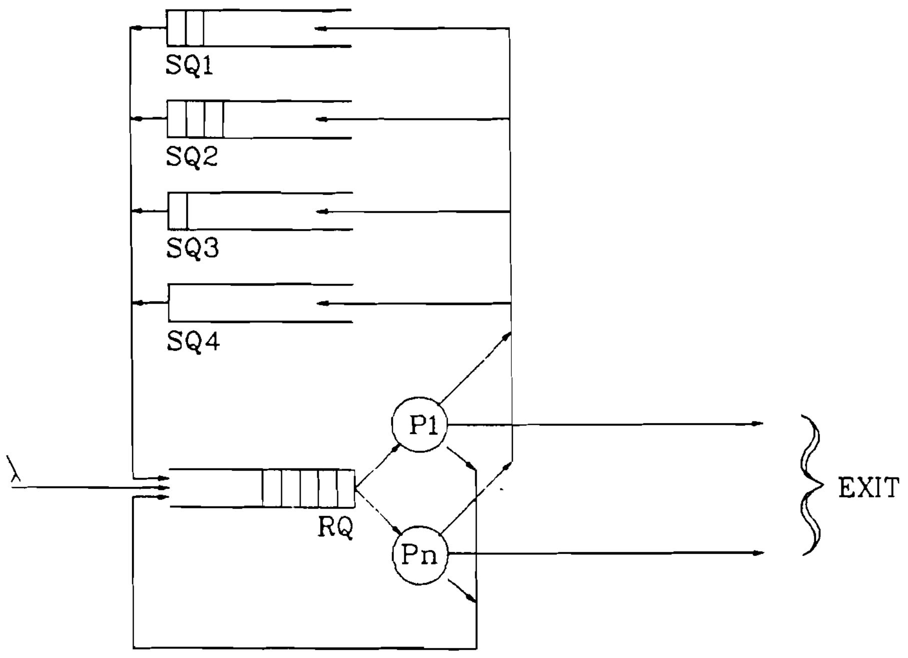


Figure 6.14 Version 7 Scheduling

In this figure we recognize the processors (Px) and the run queue (RQ). The other four queues (SQ1-4) are the queues in which sleeping processes reside. The channel (or kernel virtual address) on which a process sleeps is hashed on bits 5 and 6 to decide on the sleep queue to put a process in[2].

The way in which this system is modeled is somewhat different from what one would expect. One would expect the GPSS transactions to represent processes. We chose a transaction to represent a processor (in fact we have 11 transactions per processor of which only one is 'active'). The next figure gives the routes that a transaction may follow in the model:

2. We took the Unix version 7 situation in our THE KUNix system, for the system V source code had not yet arrived.

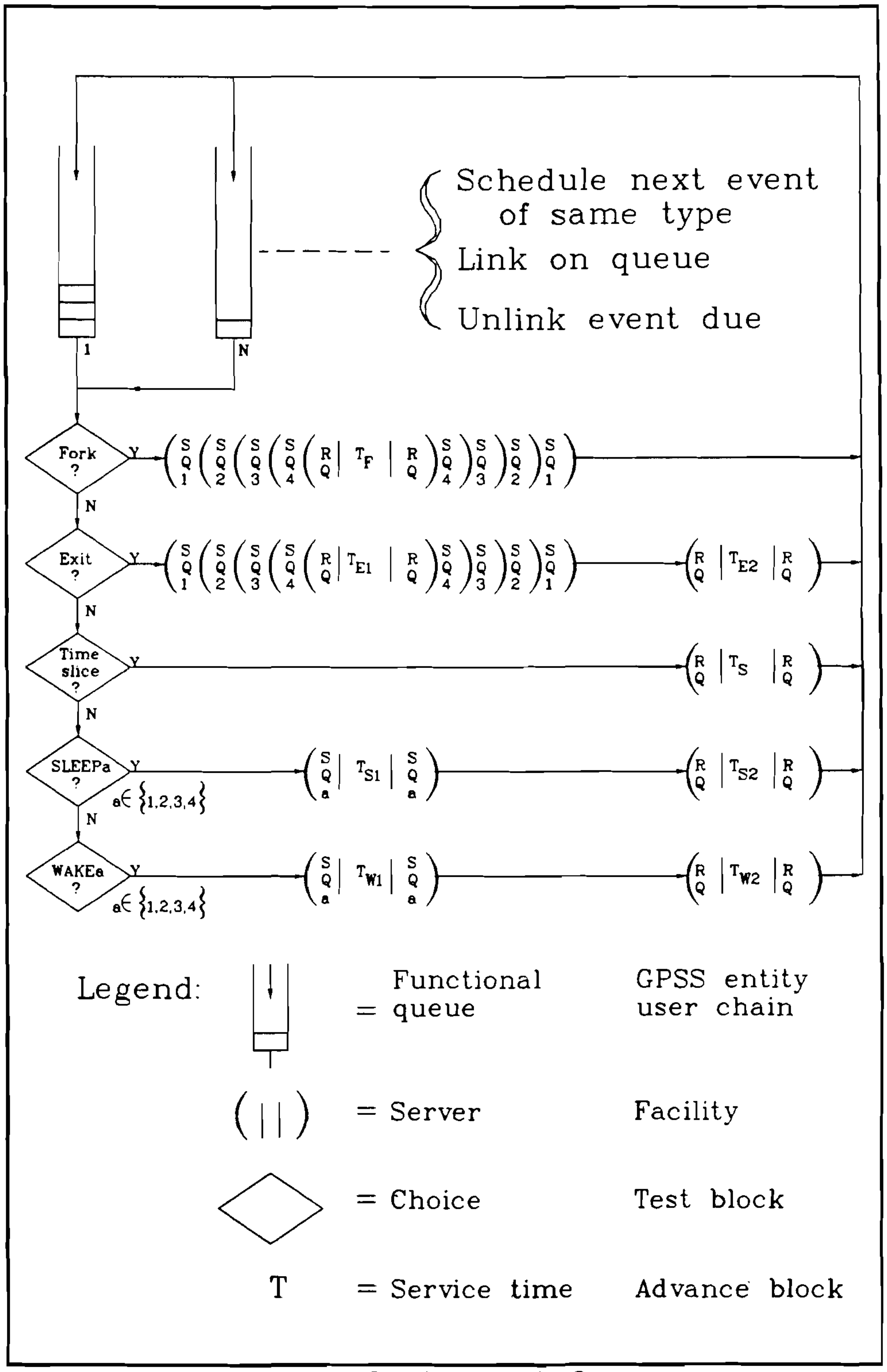


Figure 6.15 Simulation Model Structure

As said there are 11 transactions per processor. These transactions all represent one single event that may occur in the life of a processor. The 11 events are:

- Process Creation (FORK)
- Process Deletion (EXIT)
- Timeslice event (TIMESLICE)
- Process goes to sleep on queue **a** (SLEEP**a**); $1 \le a \le 4$
- Process wakes up from queue **a** (WAKE**a**)

All of these events cause the processor to manipulate one or more of the queues. From the 11 transactions 10 are queued on a *per processor queue* (*'chain'* in the sequel). The 11th transaction is walking one of the routes given in the figure dependent on its event type. In such a route the time that a processor exclusively needs a queue is modeled as a server. For example the TIMESLICE event causes the processor to access the run queue to place the current process in it and to remove a process for execution. The time this takes is Ts *units of measure* (1 millisecond in the model) which is the service time of the appropriate server (See figure). (The service time per queue is also referred to as the Queue Manipulation Time (QMT) in the sequel). The FORK and EXIT events both need access to the entire Unix process table which is modeled as the seizure of all queues. The actions performed for each event concerning the queues are:

FORK	:	Search process table for free slot Copy data from parent to child slot
EXIT	:	If process is group leader: send hangup to all members of group reset process group for all members to 0 Make process state zombie Assign parent ID of all child processes to be init process (1). Send death of child signal to parent. Fetch new process from run queue.
TIME SLICE	:	Switch process. Add current process to run queue Fetch new process from run queue
SLEEPa	:	Put a process to sleep. Add current process to sleep queue **a** Fetch new process from run queue
WAKEUPa	:	Wake up a process. Fetch process from sleep queue **a** Add this process to the run queue (Our Wakeup times do not depend on the number of processes woken up (yet)).

When a transaction has completed a route it will rejoin its processor chain after it has rescheduled itself on the system time axis. It then causes the event due next for its processor to be unlinked from the chain and sent on its way to complete its route.
The processor load in the model is 100%. The real world load of the processors can not be extracted from the model but has a strong correspondence with the inter arrival times for the

events. The inter arrival times are initially determined for a uniprocessor. As more processors cooperate their efficiency will decrease. A global system efficiency is calculated in the simulation and used to lengthen the IAT's proportionally as one would expect in a real system. The inter arrival times of TIMESLICE events are similarly derived.

A number of experiments have been carried out with this model. The first experiment was set up to validate the model. To do so, the model was used with all queue times (server times) set to 0, except for the SQ4 time which was set to 30 (= $1/M_{SQ4}$). The Inter Arrival Time (IAT) for this event was exponential with a mean value of 300 (= $1/L_{SQ4}$). The traffic intensity for SQ4 (R = L/M) therefore is 0.1. In the deterministic case this would allow for 10 effective processors (Ne = 10) in the system. The values above were chosen to arrive at an R also used by Cezzar&Klappholz. In the graph below our results are compared to those of Cezzar&Klappholz and the deterministic case. (Appendix C presents results in detail).

It appears that our results are trustworthy considering the fact that Cezzar&Klappholz calculated under nearly worst-case assumptions and for a system with exponential service and inter arrival times. The coefficient of variance is less than 1 in our case which causes our curve to be closer to deterministic. (See also next experiment).

In the following we will establish formula's to estimate the number of effective processors in the system. These formula's can then be used to limit the amount of simulation to be done. The following assumption underlies these formula's: The aggregate inter arrival rate is exponentially distributed.

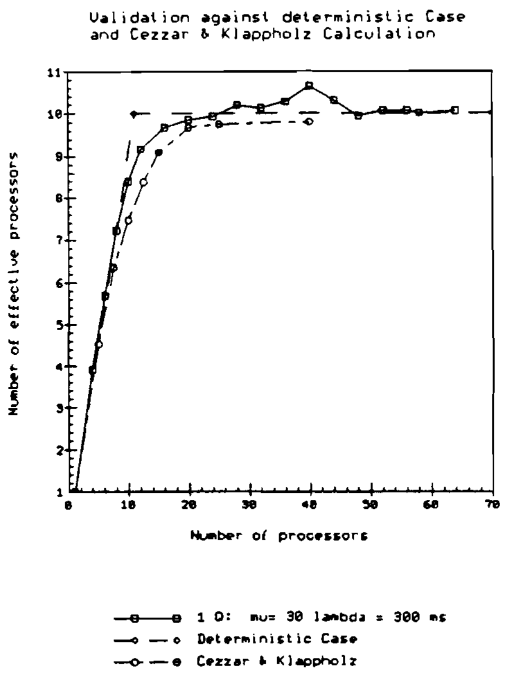


Figure 6.16 Validation of Model

The objective of the next experiment we did was to justify the following reasoning, in which we attempt to approximate part of our system by a G/G/1/N/N system with N very large.
We assume first that the server times for the Sleep Queues are zero. We then have a system with 1 queue, the run queue RQ. Arrivals at this queue are caused by all events. The next table shows two sets of IAT's and service times for the events that we used:

	Set 2		Set 4	
Fork	300	10	300	5
Exit	300	10	300	5
TimeSli	1000	7	1000	4
Sleep1	750	5	750	3
Sleep2	700	5	700	3
Sleep3	650	5	650	3
Sleep4	300	5	300	3
Wakeup1	750	2	750	1
Wakeup2	700	2	700	1
Wakeup3	650	2	650	1
Wakeup4	300	2	300	1

All arrivals are exponentially distributed except for the TimeSlice event which is deterministic. As long as the other distributions stay exponential we may neglect the influence of the TimeSlice event on the coefficient of variance. We assume it to be 1 for the arrivals.

We calculate the following quantities:

$$L_e = 1/IAT_e \qquad [1]$$

$$L = \mathrm{Sum}_e(L_e) \qquad [2]$$

$$E[t_s] = (\mathrm{Sum}_e(L_e.t_{se}))/L \qquad [3]$$

$$E[t_s^2] = (\mathrm{Sum}_e(L_e.t_{se}^2))/L \qquad [4]$$

$$C_{ts}^2 = (E[t_s^2]-E^2[t_s])/E^2[t_s] \qquad [5]$$

$$K = (C_i^2 + Ct_s^2)/2 \qquad [6]$$

$$R_{RQ} = L\,.\,E[t_s] \quad (= \mathrm{Sum}(L_e\,.\,t_{se})) \qquad [7]$$

$$Ne = 1/R_{RQ} \qquad [8]$$

where, IAT_e	is the inter arrival time of events of type **e**
L_e	is the arrival rate for event **e**
L	is the aggregate arrival rate for all events
t_{se}	is the service time of a queue for events of type **e**
$E[t_s]$	is the expectation of the service time
$E[t_s^2]$	is the expectation of the squared service time
C_{ts}^2	is the coefficient of Variance of the service time distribution
C_i^2	is the coefficient of Variance of the combined inter arrival times.
R_{RQ}	is the traffic intensity for queue RQ.
Ne	is the number of effective processors.

The results for the two sets are:

	Set 2	Set 4
L	0.0263	0.0263
C_i^2	1	1
$E[t_s]$	5.482	2.970
$E[t_s^2]$	38.842	11.005
C_{ts}^2	0.292	0.248
K	0.646	0.624
R_{RQ}	0.144	0.078
Ne	6.94	12.82

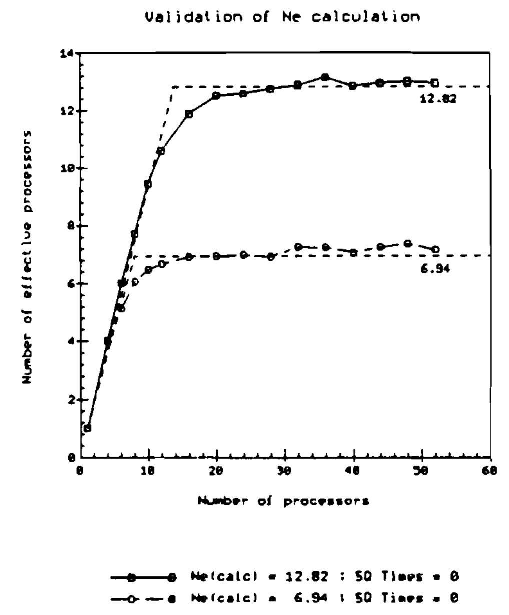


Figure 6.17 Calculated versus Measured

Note that K is less than 1 indicating that queue lengths, waiting times etc. in general are smaller than for the full exponential case. This was confirmed in our first experiment (figure 6.16) where our curve was closer to the deterministic case than the exponential one calculated by Cezzar&Klappholz.

To find out whether these calculations hold for our model we simulated for set2 and set4 values. The results are plotted in figure 6.17 above.

The results are indeed as expected.

The third experiment was aimed at discovering the model's (system's) sensitivity to the service times of the sleep queues. We thereto simulated using the following values for sleep queue service times:

	Set 1		Set 3	
(ms)	sleep	Wake	sleep	Wake
SQ1	2	2	1	2
SQ2	2	3	1	3
SQ3	2	4	1	4
SQ4	2	5	1	5

Note: The RQ service times are the same as those of sets 2 and 4 respectively.

The values were chosen equal for sleeps since this involves no more than adding a process at the head of the sleep queue. The values were chosen unequal for wakeups to find out about the influence of different sleep queue lengths on results. Using equations 1 through 8 mutatis mutandis for the sleep queues gives us (Ne only; Ne_{RQ} added for reference):

	Set 1	Set 3
Queue	Ne	Ne
RQ	6.94	12.82
SQ4	11.11	18.75
SQ3	13.18	24.4
SQ2	13.55	25.6
SQ1	13.89	26.7

The existence of more queues in a system which are not independent makes one expect a lower value for Ne. Intuitively one expects the influence of the SQ's to be larger when their Ne-value is closer to that of the queue used heaviest (RQ). For example sets 1 and 3 render an Ne_{SQ4} value that is approximately 1.5 as high as Ne_{RQ}. The SQ impact will therefore be clearly noticeable:

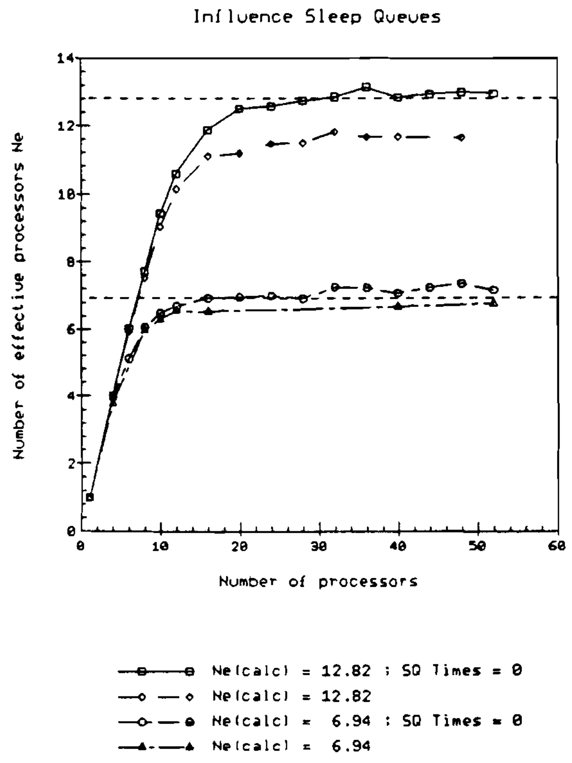


Figure 6.18 Sleep Queue Influence

In order to gain insight in the influence we did a number of experiments using 2,3 and 4 independent queues and an experiment using the run queue RQ with a fixed arrival rate and service time and one sleep queue with a variable service time.

Figure 6.19 on the next page presents three views on the results of the experiments with independent queues. Part a shows us the usual Ne versus N plots, part b plots Ne versus the Queue Manipulation Times (QMT) of the queues and part c does this for Ne versus the number of queues. It appears that Ne is reciprocally dependent on both the number of queues as the QMT's.

Results of the experiment with one fixed RQ (set 4 values) and a 'walking' Sleep Queue are presented in figure 6.20. The graph also contains a horizontal line representing the value of Ne when no sleep queue were present (In fact only valid on the Y-axis). The hyperbolic line represents the value of Ne for the sleep queue alone. (Ne here is the maximum number of effective processors, as in the deterministic case for large N).

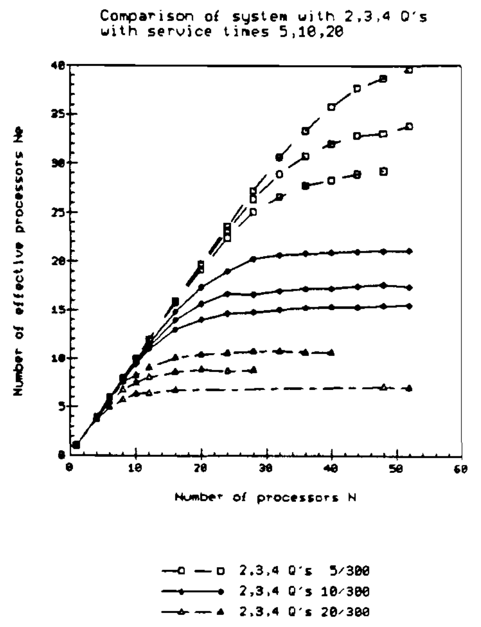


Figure 6.19a Ne=f(N) for independent queues

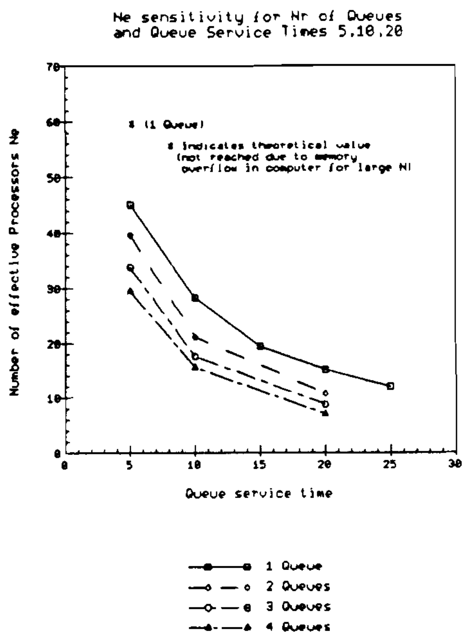


Figure 6.19b QMT Dependence

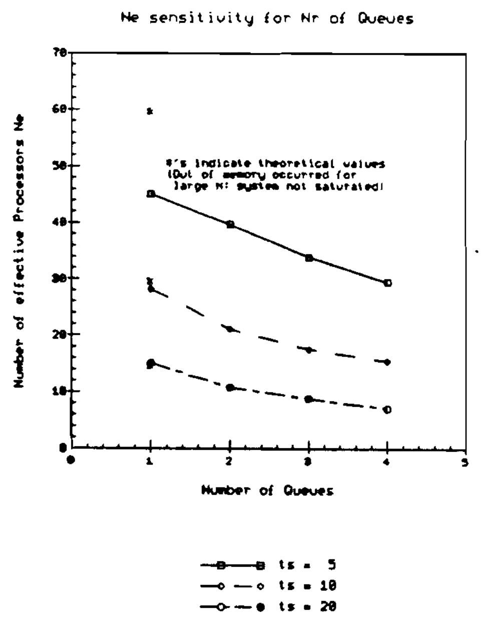


Figure 6.19c Nr of Queues Dependence

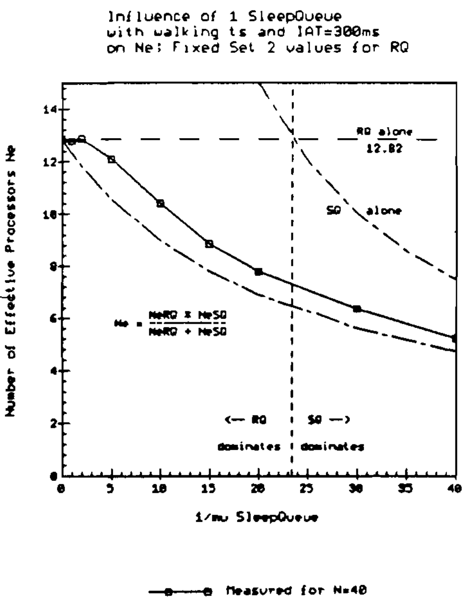


Figure 6.20 Walking Queue

The model enables to selectively measure the influence of 'per event/per Queue Queue Manipulation Times'. Figure 6.21 for example shows how the number of effective processors changes when the service time for forking is altered.

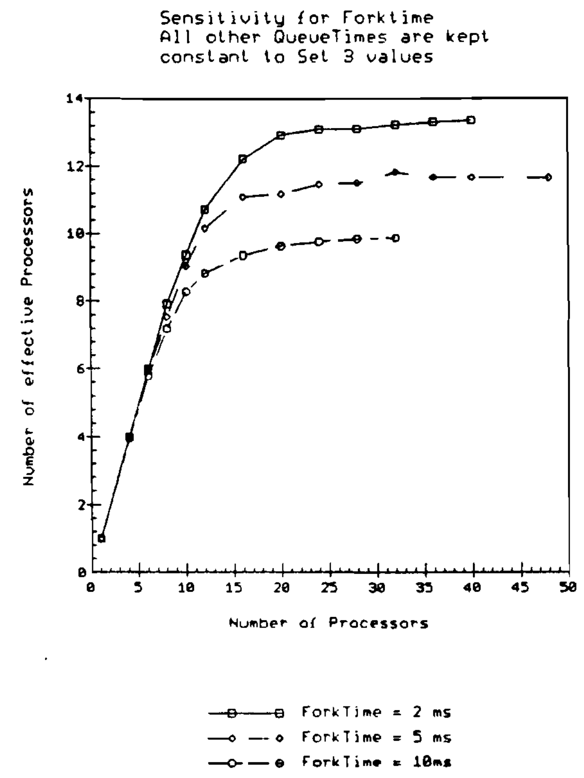


Figure 6.21 Forktime Sensitivity

6.3.3.2 Conclusions Simulation

The model we wrote has been validated against results from literature. The effects incurred by changing the number of queues or the QMT's and IAT's have an effect that is correct in nature. The model will generate results that are judged accurate with a high degree of confidence.
The model enables to independently vary all important quantities in the system, such as the IAT's and QMT's and their distributions, the number of hash queue's and , with a little effort, the number of Run Queues. Also the order of locking and unlocking the queues is easily controlled.

The next step to be taken is to quantify the different parameters to the model by measuring them from a real system. Once this has been done changes to the Process Management System can be easily evaluated through simulation.

Under set 3 assumptions we see that the Run Queue forms a bottleneck, restricting the number of effective processors to approximately 11.6, reached when N rises to about 20. We also have seen that when we diminish the fork QMT of the Run Queue from 5 to 2 ms, we gain 1.6 processor. Or when we allow this time to rise to 10 ms, we loose 1.6 processor. It is this kind of information that can be obtained.

For information related to individual processes, such as response times, queue lengthes etcetera, a model using processes as transactions will have to be used. Note however that when the distribution of queue lengthes is known and have an influence on Queue Manipulation Times, we may enter this into the current model by choosing an appropriate distribution for these QMT's.

In this paragraph we have derived some formula's to estimate the number of effective processors in a system. We warn here to use these formula's only for that purpose as we did not prove their global accuracy. The estimates made may depend on the mutual relations of the IAT's and QMT's.

Further simulation results can be found in appendix C.

6.3.3.3 Queue Manipulation Times

We do not consider run queue and sleep queue manipulation times to be prohibitive for a VME-based system with 20 or 30 processors[3]. However when the number of processors on a single VME board becomes 3 or more (if ever, but likely so) a bottleneck could arise. As seen we may then use multiple queues with random access to shift the bottleneck to a higher number of processors. A more fundamental approach would be to completely eliminate any critical section from the queue handling. The F&A synchronization primitive makes this possible[4]. Choosing this route would not only increase processor efficiency, but would also allow us to obtain experience in using this synchronization primitive.
The intention to use F&A primitives immediately raises the question how to implement it on a time-shared-bus based system. The following solutions exist; all of them represent small serial bottlenecks, degrading system performance only for very high processor numbers:

Software Solution 1:

```
F&A(V,e)
    {
    While (T&S(V.flag));
    temp = V.value;
    V.value = temp + e;
    V.flag = 0;
    return(temp);
    }
```

More complex schemes than spin-locking on V.flag can be used to diminish bus load. The length of the critical section is some 5 to 10 micro seconds dependent on the implementation. This is small enough for our purposes.

3. In [GOTL83b] fig. 1 shows a noticeable degradation of efficiency from 10 processors upwards due to a serial bottleneck in the queue handling of a program to calculate radiation transport.

4. Unfortunately no F&A-based algorithm is known today that may manipulate linked lists without critical sections. However solutions are available [EDLE85].

Software Solution 2:

Since we will probably use the 68030 microprocessor, a solution could be to use its Compare&Swap (CAS) instruction:

```
{Registers D0 and D1 have an arbitrary value, D2 contains e}

FNA MOVE.L  V,D0      ;Move V to D0
    MOVE.L  D0,D1     ;Move D0 to D1      (D1 used as accumulator for add)
    ADDX.L  D2,D1     ;Add e to D1 (= [V]+e)
    CAS.L   D0,D1,V   ;If [V] not altered, write [V]+e to V.
    BNE     FNA       ;else try again.
    RTS

{V and D1 contain [V]+e, D0 contains [V], D2 contains e}
```

Note: The [V] obtained when leaving the routine may be different from the value resident in V upon entering the routine. (due to other operations on V by different processors.)

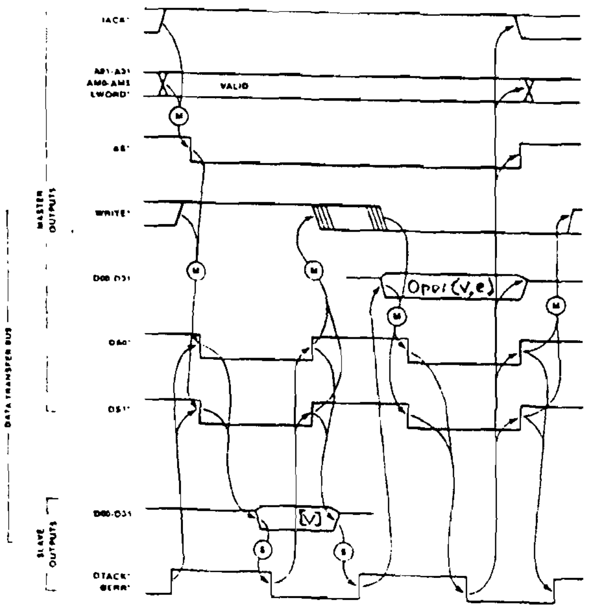
This is not different from the semantics of F&A as defined in the NYU Ultracomputer.

The true critical section consists solely by the CAS instruction which is a Read-Modify-Write cycle on the VMEbus, taking for example 600 ns. However due to accesses of other processors to V during the subroutine (ca 3Ms) a number of retries may be necessary, causing a larger F&A execution time and hence a degradation of efficiency.

For both solutions one can imagine a case of starvation when a set of fast processors of high bus priority continuously perform F&A's on the same location as a slow processor of lower bus priority. It is possible to deal with this unlikely situation for example by including delay times in the iteration loops (after performing the routine once without delay) or through other more sophisticated techniques.

Hardware Solution 1

Implementing the F&A primitive in hardware can best be done using a Read-Modify-Write cycle as for Test&Set and Compare&Swap. In fact all these primitives are special cases of the Fetch&Operation primitive as discussed before. It is possible to use the RMW cycle for all F&O cases. A RMW cycle with the values for a F&A(V,e) operation inserted looks as follows:



NOTE
ARROWS LABELED M SHOW TIMING RELATIONSHIPS GUARANTEED BY THE INTERNAL TIMING OF THE MASTER
ARROWS LABELED S SHOW TIMING RELATIONSHIPS GUARANTEED BY THE INTERNAL TIMING OF THE SLAVE
UNLABELED ARROWS SHOW TIMING GUARANTEED BY INTERLOCKED RELATIONSHIPS BETWEEN THE MASTER AND SLAVE

Read-Modify-Write Cycle Sequence

figure 6.22

Currently available processors do not provide the F&A primitive. For example the 68030 provides Test&Set and two versions of Compare&Swap but no Fetch&Add.

If we wish to use such a processor and the F&A primitive, we will have to add a separate F&A-unit. Below an example F&A-unit is given (principle only):

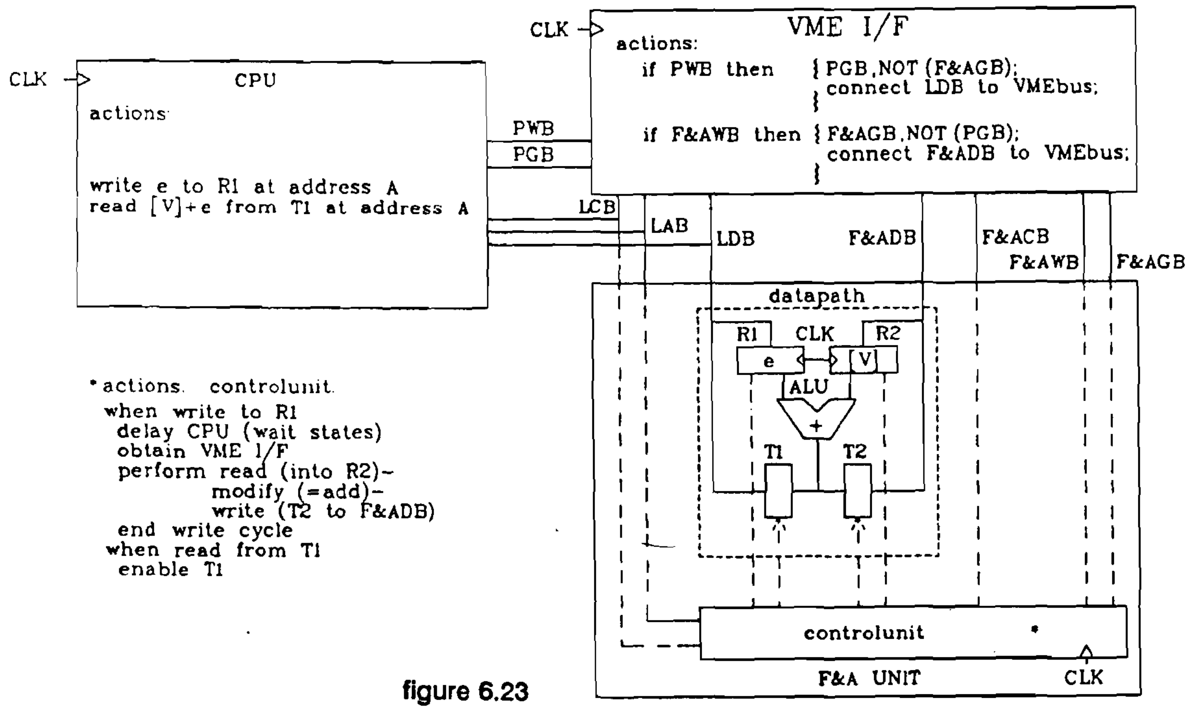


figure 6.23

We chose the F&A-unit to have a separate interface with the VME I/F unit. The F&A-unit can also be integrated into the VME interface.

For normal VME accesses the CPU requests the VMEbus by asserting PWB (Processor Wants Bus). When granted the bus normal VME accesses may proceed (including RMW cycles such as T&S and C&S).

When the CPU writes a location in the address space reserved for F&A locations (which must be distinct from the 'normal' VME address space) the F&A-unit will delay the processor by having it perform wait-states. Simultaneously it requests the VMEbus by making F&AWB active. After being granted the bus (F&AGB) it performs a RMW cycle on the bus, using the

address VMEmap(LAB). (VMEmap(LAB) represents the VME address corresponding to the LAB address. The mapping is done in the VME interface). The value read from the VMEbus ([V]) is placed in R2 via F&ADB. The subsequent write phase writes the outcome of the ALU operation (an add in our case) to the F&ADB and from there to the VMEbus. Finally it releases the CPU which ends its write cycle.
So, when the CPU writes value e into register R1, the sequence of actions mentioned above will provide value [V]+e at the output of the ALU when the write cycle ends. A subsequent CPU read anywhere in the F&A space will always return value [V]+e. (Tristate buffer T1 is enabled).
The VME I/F unit must decouple the LDB from the F&ADB, VMEbus pair during the write cycle since it (the LDB) will carry the value **e** most of the time.
Extension to the more general Fetch&Operation is straightforward. The required operator can either be included in **e** (this limits the range of **e**), or written to an F&A control unit register on beforehand [5]. Observe that multiple instructions from the local CPU, not forming an indivisible piece of code , cause one single indivisible RMW cycle on the VMEbus. The critical region has now reduced to 600 ns. Observe also that read and write occur reversed in the local CPU.

Hardware Solution 2

Incorporate the F&O primitive into the processor. This would allow for T&S, C&S, F&A and possibly other convenient primitives. This route is recommended for ongoing and future processor designs.

Hardware Solution 3

One could imagine the F&O primitives to be extra primitives to the memory module. This would place the responsibility for performing the operation with the memory modules. Extension of the RD/WR line (or DS0,1*/W* lines) would be necessary to request the memory cycle operation:

An example with 2 signal lines:

00	Load
01	Store
10	T&S
11	F&A

(Possibly a separate timing line may be used.)

To truly turn the RMW operations into 1 cycle we would have to split the databus into a separate (unidirectional) read and (unidirectional) write bus. Obviously this would cost a lot of extra pins and wiring. (As approximately 90% of memory cycles are read cycles the profit

5. The description given is not concerned with different operand lengths, it only shows one of the possible principles of operation

of parallel but independent read and write cycles is marginal. Furthermore a memory module should support simultaneous reads and writes.)
This solution is not only costly, it also requires modification of the time-shared bus, a possibility we reject (Condition 3.1). (When changing the Processor-Memory Network, we might as well go all the way and adopt for example an Omega Network).

Hardware Solution 4

This solution is meant for the situation where more than one processor is resident on a VME board. We could then build an Omega Network for the local processors to access F&A locations in the VME address space. The Omega Network could combine F&A operations as discussed before. Critical sections would still occur when processors from different boards access the same F&A location (Board serialization).
The Omega Network may be very small as it is unlikely to have a vast amount of processors on one VME board. For 4 processors a 4-input-4-output Network would suffice. Using today's technology this would comprise one chip. (See also [GOTT83a] where a 4-input-4-output switch is estimated to comprise 2 chips using today's technology).
We would obtain:

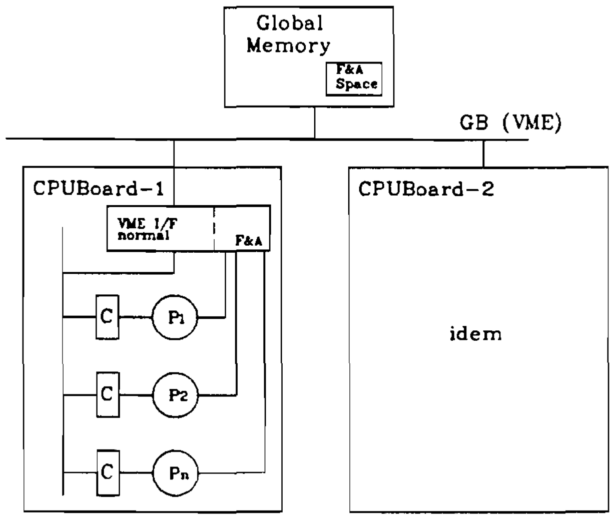


figure 6.24

The probability that the total (VME) system hosts a number of processors that is large enough to experience the serialization bottleneck caused by F&A's is negligible. This last method is therefore far too expensive, it will take the design and manufacture of a very complex chip while showing no benefit at all[6].

6. Lipovski and Vaughan have shown that networks can be built that implement the F&A and other primitives using binary trees. The complexity of such a network is several orders of magnitude lower than that of the NYU Ultracomputer [LIPO88].

6.3.4 Summary

In this section we have discussed the process scheduling for a multiprocessor based on Unix. We first considered the scheduling as it is commonly done in single processor Unix. The pro's and con's of this scheduler were disputed and the decision was taken to use it as the basis for the multiprocessor scheduler. The general requirements for scheduling from the previous chapter have been screened to give a set of practical requirements for the multiprocessor scheduler. The main requirement dropped was the requirement to optimally schedule P-SET groups. We then considered multiprocessor scheduling mainly from the angle of the congestion that can occur. To obtain measures on congestion we simulated the scheduler for several numbers of processors and for several queue manipulation times. The model was validated and fed with an estimation of queue manipulation times as they are likely to occur. Simulations will have to be done using empirical data to accurately predict congestion. With the estimated manipulation times the maximum amount of efficient processors is 12.8. We therefore looked into the possibility to diminish queue manipulation times. We found a method using the Fetch&Add primitive. The Fetch&Add primitive has some very beneficial characteristics. It makes possible the true simultaneous use of shared integer variables. It also returns more information to its 'invoker' than T&S, making possible more sophisticated software primitives that are essentially critical-section-free.

For a VME based system where the number of processors is low an implementation of F&A as a critical section will not hurt performance when the time to pass the critical section is kept small (for example < 50 us). This is easily accomplished by the methods presented in 6.3.2. Having disposal of a non-critical-free F&A primitive still leaves the advantage of the higher level of sophistication that can be built into more complex software primitives.

A heavy-weighing disadvantage of using the F&A primitive is the necessity to rewrite the kernel. For the access to resources that can only be used exclusively other synchronization types are available. Especially when resources are unavailable for a long time or provide service after a long time without main processor intervention, process-suspending-primitives such as semaphores should be used.

Our proposal is to use software method 1 (See 6.3.2) for implementation of the F&A primitive and to apply the F&A in selected area's of the kernel in order to gain experience[7]. (As an aside: For ongoing and future processor designs we recommend to incorporate the F&A primitive). One such a selected area clearly is the scheduling subsystem. Using F&A can eliminate the service time M of queue manipulation, removing the scheduling bottleneck.

Other synchronization can be dealt with using the traditional semaphores. For a first approach on Unix version 7 see [KWAK87].

7. This experience can also be gained by running multiple processes, cooperating through a shared memory region, on an existing System V.3 implementation.

7 Multi-Kernel PSM Architecture

Time was too short to fully investigate this architecture. The fact that private memory is available opens up a wide spectrum of optimization possibilities. The main guideline to decide whether an item should move from shared to private memory is that the move should diminish bus load. This measure depends on the number of processors in the system and should therefore not be used strictly.

One obvious use of private memory is caching data from remote memory, where 'remote' is defined as: only accessible via the VMEbus. We have sometimes assumed that no bus bottleneck existed for the numbers of processors we worked with, especially when we discussed scheduling. In a VME system such an assumption together with the use of more than 4 processors implies the use of a cache. A problem when using a cache, that also occurs in uniprocessors, is that when a process switch takes place, the cached data is largely invalidated. A solution is to add more private memory and use it to hold the context of all processes in execution on the local processor. This solves the cache invalidation, the cache only needs to cache operating system code and kernel data structures, but gives rise to a suite of problems among which the issue of Context Placement and possibly Process Migration.
A problem occurring in multiprocessors only is the possibility that cache data is not coherent. Coherency means that when more than one cache contains data from the same shared location and have it tagged valid, the data in these caches is the same. To enforce this cache coherence several methods are available. We refer to the literature for details on the subject.

Other uses of local memory could be to hold process contexts or parts thereof is indicated above. See figure 6.6 for a refresh on the different parts of a process context in Unix. From the different parts of a context the ones that are most heavily referenced when the process is running are the first to move to private memory. Also when referencing of process context parts by other processors is small such part is eligible for a move to private memory. The Register Context depicted in figure 6.6 is always moved when we consider the processors registers as local memory. Next the process stacks are used heaviest by the local processor so that they are the first to move. The parts of the environmental context of a process, these are the pieces of kernel tables used by the process, are the last to be moved. They are integral parts of global kernel tables. These kernel tables are, on the average, referenced equally by every processor.
A logical candidate for placement in private memory is the operating system code. It is read-only and never changes under normal operating conditions. However its size is quite large as opposed to the size of the stacks. Annot and Janssens [ANNO85] investigated the matter and found the following figures:

Approximate reduction of bus traffic

Local Stacks (+/- 2k):	75 %
Local OS code (128k) :	3 %

(A write-through cache is used in both cases with an assumed hit ratio of 95%)
(100% is the total of memory references in a single processor system)

The above percentages are derived for the situation where each of the measures is taken in the absence of the other. When we first arrange for local stacks we have a bus traffic of about 3.5% (using the figures also used by Annot and Janssens). Adding 128k of memory for OS code may then reduce traffic to about 3% which is a reduction of 15%[5] with respect to the previous situation. The amount of allowable processors then rises from 28 to 33. Note that a small change in these percentages changes the amount of allowable processors quite much. (e.g 2.9% gives 34 processors).

When process contexts can be local, it is possible to enhance the process dispatcher to favor processes of which the context is locally available.

It may be obvious that when more than 32 processors must function effectively in a VME based system this topic must be studied intensively. The amount of 32 processors for a VME based system is quite high as it already necessitates multiple processors on a board. The effort and cost to increase the number of processors above 32 will most likely prohibit such systems.

[5] We did not take into account the bus traffic for loading and flushing local stack memory when a context switch occurs.

8 Proposed Multiprocessor

When research capacity is limited a path of gradual growth is often followed to reach the final goal. This is what we propose for our case too. There are several pathes that will lead us to the final goal, the Multi-Kernel PSM multiprocessor. Four of these 'routes' are discussed below.

Route 1

Route 1 will provide us with a working multiprocessor soonest. The actions along the path are:

- Port Unix System V to the target hardware, however with only one processor.
- Add a processor and implement a Master-Slaves system.
- Make it a Uni-Kernel system.
- Gradually add protection mechanisms in the kernel for kernel data structures: Multi-Kernel SMO.
- Utilize private memory: Multi-Kernel PSM.
- Pursue research for further improvement.

The early availability of the multiprocessor enables to test multiprocessor mutual exclusion mechanisms in a realistic environment. It also enables students to acquire practical experience in working with multiprocessor systems both in the field of hardware is in software.

Route 2

Route 2 largely divides the transformation of the kernel software on one hand and its testing on the other in two phases that hardly overlap in time. The advantage thereof is that the transformation can be done on stable single processor hardware. The disadvantage is the postponement of testing the mutual exclusion primitives in a multiprocessor hardware environment. To a large extent this can be alleviated by simulating processors through processes on the single processor system V system (use shared memory regions). The actions along this route are:

- Port System V to the single processor target hardware .
- Build in Multiprocessor mutual exclusion mechanisms and test them
    a. in the single processor but 'multi-process' environment
    b. by emulating processors by processes.
- Add processors: Multi-Kernel SMO.
- Utilize private memory: Multi-Kernel PSM.
- Pursue research for further improvement.

Route 3

The third route would be to rewrite the kernel while not loosing Unix (binary?) compatibility. Rewriting the kernel frees us from the labour of incorporating mutual exclusion mechanisms in the existing kernel. It also enables to write a structured kernel and to adapt it to multiprocessor and distributed computing environments. Suggestions are to

- make process contexts less location dependent
- diminish process's volatile environments
- implement multithreading
- use Fetch&Add based algorithms to prepare for large numbers of processors (scalable OS).
- design a scheduler geared towards scheduling of median-lived processes (0.4 seconds).
- instrument the scheduler so that it may measure data needed for process migration in distributed environments.
- implement network IO as an independent IO class.
- make access to IO devices and file systems location independent (network transparency).
- provide 'copy on write' shared memory
- make it dynamically reconfigurable
- explore object-oriented programming style
- etcetera.

Route 4

Route 4 would be to acquire an already existing 'Unix-like' operating system that already supports multiprocessors. A number of these operating systems exist within the academic world ([OUST88],[ACCE86]) and can possibly be made available to us. Research could then start from that point onwards.

Which route will be followed is a matter of group policy. Personally I prefer route 3, giving priority to the creative act of designing an operating system over the early possesion of a working multiprocessor.

Below I will make some proposals that are applicable to route 1 and 2 and partly also for route 3.

- Do not use the Implicit IPrc for MP-IOP traffic as discussed in chapter 4. Use a separate IPrC subsystem instead. Give the IPrC subsystem the functionality to pass events, tokens, messages (little data) and data (large). Also provide broadcasting capability.
- Use VME interrupts for the signalling of Hardware Faults. Location Monitors will provide for all other inter process interrupts (a part of the IPrC subsystem).
- IO ready events are signalled to a main processor using the IPrC subsystem.
- All clock interrupt handlers perform the local actions discussed in 5.2.2. One performs the global set of actions. The responsibility for performing the global actions is passed from processor to processor is done using a token from the IPrC subsystem.
- Scheduling

  As with the entire multiprocessor we suggest to advance to the final goal step by step. The current scheduler has several disadvantages as discussed in chapter 6. To improve the scheduler in one step to its ultimate form is too much work along with all other activities. We propose to:

  - Start using the single processor scheduler with central run queue and sleep queues.
  - Protect the queues with only a few semaphores if one desires to have a working system in the near future.
  - Every processor self-dispatches processes.
  - Adjust the clock interrupt handler as discussed in 5.2.2. (The timeslice must depend on the number of processors N! (See appendix C)).
  - Start redesigning the scheduler using the Fetch&Add primitive as a building block and attacking existing problems by

    - optimizing it for median process lifetime (0.4 seconds)
    - making it tunable (automatic and manual)
    - instrument it for measuring data for performance monitoring, process migration in a distributed environment and automatic tuning.
    - improve behavior for large CPU intensive processes.
    - avoiding linear searches, rather hash tables.
    - lowering the overhead of priority calculations by using a new mechanism based on measured process behavior (The amount of data measured should be kept to a minimum, for example: lifetime, CPU-usage, IO-usage, CPU-time-Thinktime ratio).

9 Conclusions and Recommendations

In this chapter we will present our conclusions and recommendations. For easy reference we will do this in the form of a bullet list. To not overly duplicate text we sometimes refer to previous chapters.

- The Unix operating system as it is in use on single processor systems is not suited for use on multiprocessors. This is due to the mutual exclusion methods used by Unix. These do not mutually exclude processors.

- The amount of work to be done to transform Unix into a multiprocessor operating system, in the sense that every available processor may simultaneously run kernel code, is very large. It is possible to partly transform the kernel of Unix until a degree of parallelism is reached that does not pose a bottleneck for the target system. For example in a VME based system the amount of processors in the system will not easily surpass 32 processors (16 is more likely). A coarsely parallelized kernel will suffice in this case. However to be prepared for other architectures I recommend to fully transform the kernel.

- If one wishes to quickly own a working prototype of a multiprocessor one should follow route 1 as outlined in chapter 8. I prefer to follow route 3 as it allows to write an operating system free of inherited choices and better suited for nowadays (parallel) architectures.

- From literature we observe that more levels of abstraction are introduced into the computing model. The names for the different levels somewhat vary. For a Unix-like environment one usually recognizes the following abstractions:

    - process
    - thread
    - instruction

  The aim of introducing extra abstraction levels is to be better able to exploit small pieces of parallel code. The threads can be viewed as light-weight processes that run within a normal process. Threads use the resources owned by their process. Threads have a small volatile context and hence can be quickly switched.
  We recommend to investigate into the addition of abstraction levels.

- Communication is a main item of concern in a multiprocessor. A separate subsystem is necessary to provide for all communication needs. The subsystem can best be independent of the operating system used. It should also be implementable on several types of hardware. A multilayered approach as in the ISO-OSI system therefore seems best. See chapter 4 for details.
  We recommend to develop an IPrC subsystem as close to the ISO-OSI model as possible and to support it by special hardware. (We already have such a project [VERS87], this however does not make use of shared memory. The use of shared memory can be added though. A supplemental specification, design and implementation

would provide a uniform OS - IPrC-subsystem interface for both shared memory multiprocessors as distributed systems.

- In a multiprocessor distributed scheduling should be avoided. This avoids measures to keep the load balanced and hence the associated overhead. A central scheduler will not cause a bottleneck for small (10 - 30) numbers of processors, when existing mutual exclusion methods are used efficiently. For large numbers of processors one can use the Fetch&Add primitive. For even larger amounts one may use multiple run queues and have them randomly accessed by processors while other queues can be hashed. See the final paragraphs of chapters 6 and 8 for further conclusions on scheduling.

- Process dispatching may be made to favor processes whose context is in the processors private memory (Multi-Kernel PSM architecture).

- New Operating System Primitives (System Calls) will have to be provided that support parallelism. New primitives could be: n-way forks, n-way waits, forks with user controlled duplication of address space (Not, Partial, Complete), System Calls for the exclusive use of a number of processors, etcetera.

Some small conclusions for Unix and VME:

- Caches must be applied to push the bus bottleneck to approximately 6 to 7 processors.

- Local memory for stacks will further shift this bottleneck to between 16 and 32 processors with each processor having an efficiency of 80 to 90 percent.

- When less then N, the amount of main processors, processes are running, timeslicing can be switched of and a fork could warn an idling processor of the birth of the new process.

A lot of work still needs to be done. The impression exists that current bus technology is insufficient, even for industrial applications, in a few years time. The current efforts should therefore be targeted at an environment in which much more than 32 processors are able to cooperate. Especially the operating system should be scalable to run efficiently on multiprocessors with hundreds of processors.

Appendix AAbbreviations and Definitions

ACM	Association for Computing Machinery
adb	A DeBugger
F&AGB	Fetch&Add Granted Bus
ALU	Arithmetic Logic Unit
AM	Address Modifier
F&AWB	Fetch&Add Wants Bus
BSD	Berkely Software Department
CAE	Computer Aided Engineering
CAS	Compare And Swap
clist	Character (linked) LIST
DMA	Direct Memory Access
DTB	Data Transfer Bus
ECC	Error Correction Code
FCFS	First Come First Serve
FIFO	First In First Out
FLIH	First Level Interrupt Handler
FMB	Force Message Broadcast
FNA	Fetch aNd Add
GPSS	General Purpose Simulation System
IAT	Inter Arrival Time
IC	Integrated Circuit
IH	Interrupt Handler
inode	Index NODE
IO	Input Output
IOP	Input Output Processor
IPC	Inter Process Communication
IPrC	Inter ProcessoR Communication
ISO	International Standards Organization
LAB	Local Address Bus
LAN	Local Area Network
LCB	Local Control Bus
LDB	Local Data Bus
LM	Location Monitor
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMP	Master Main Processor
MMU	Memory Management Unit
MNI	Memory Network Interface
MPC	Message Passing Coprocessor
MPUX	Multi Processor UniX
MP	Multi Processor
OSF	Open Software Foundation
OSI	Open Systems Interconnect
pdf	Probability Distribution Function
PID	Process IDentification
PMS	Process Management System
PNI	Processor Network Interface
PSB	Parallel System Bus

PSM	Private and Shared Memory
PWB	Processor Wants Bus
RD	ReaD
RISC	Reduced Instruction Set Computer
RMW	Read Modify Write
RQ	Run Queue
SISD	Single Instruction Single Data
SJN	Shortest Job Next
SMO	Shared Memory Only
SMP	Slave Main Processor
SP	Slave Processor
spl	Set Processor Level
SPUX	Single Processor UniX
SQ	Sleep Queue
TAS	Test And Set
TDM	Time Division Multiplex
TOPLAS	Transactions On Programming Languages and Software
tty	TeleTYpe (terminal)
VME	VERSAtile Module Europe
VMS	VMe Serial bus
VSB	Vme Subsystem Bus
WR	WRite

Appendix BVMEbus - AM codes

AM codes occupy 6 signal lines on the bus. The range for these codes is therefore from 00 to 3F hexadecimal.

HEXADECIMAL CODE	FUNCTION	DEFINED BY
3F	Standard Supervisory Ascending Access	VMEbus Spec.
3E	Standard Supervisory Program Access	VMEbus Spec.
3D	Standard Supervisory Data Access	VMEbus Spec.
3C	Undefined	Reserved
3B	Standard Non-Privileged Ascending Access	VMEbus Spec.
3A	Standard Non-Privileged Program Access	VMEbus Spec.
39	Standard Non-Privileged Data Access	VMEbus Spec.
38	Undefined	Reserved
30-37	Undefined	Reserved
2F	Undefined	Reserved
2E	Undefined	Reserved
2D	Short Supervisory IO Access	Reserved
2C	Undefined	Reserved
2B	Undefined	Reserved
2A	Undefined	Reserved
29	Short Non-Privileged IO Access	VMEbus Spec.
28	Undefined	Reserved
20-27	Undefined	Reserved
10-1F	**Undefined**	**User**
0F	Extended Supervisory Ascending Access	VMEbus Spec.
0E	Extended Supervisory Program Access	VMEbus Spec.
0D	Extended Supervisory Data Access	VMEbus Spec.
0C	Undefined	Reserved
0B	Extended Non-Privileged Ascending Access	VMEbus Spec.
0A	Extended Non-Privileged Program Access	VMEbus Spec.
09	Extended Non-Privileged Data Access	VMEbus Spec.
08	Undefined	Reserved
00-07	Undefined	Reserved

NOTES:

Short address uses 15 address lines (A01-A15)
Standard address uses 23 address lines (A01-A23)
Extended address uses 31 address lines (A01-A31)

Codes defined by the "VMEbus Spec." should not be used for purposes other than those specified.

Codes defined by "User" may be used for any purpose which the VMEbus user (board vendor or customer) deems appropriate (page switching, memory protection, MASTER or task identification, privileged access to resources, etc.).

Codes defined by "Reserved" should not be used by the user; they are reserved for system use and future enhancements.

Appendix C Simulation Results

The Model:

```
; GPSS/PC Program File MPUX_MOD.GPS.  (V 2, # 38447)  01-11-1989 00:28:25
;
;*************************************************************************
;* EINDHOVEN UNIVERSITY OF TECHNOLOGY                                    *
;*                                                                       *
;*   Program Name   :                  Filename : MPUX_MOD.GPS           *
;*   Written by     : P.J.M. Welten    Date     : 01-11-1989             *
;*   Language       : GPSS                                               *
;*                                                                       *
;*   Description : This Model models the Scheduler of a Unix based       *
;*                 Multiprocessor. The Unix Version 7 Scheduling         *
;*                 system as implemented in our THE-KUNix machine        *
;*   served as a starting point. Transactions in this model represent    *
;*   events in the life of a processor. There are 11 events per          *
;*   processor. These are: FORK, EXIT, TIMESLICE, SLEEP1-4 and           *
;*   WAKEUP1-4. 10 ofthese are linked onto a per processor chain. The    *
;*   11th is 'walking' an event dependent model route. A 'walking'       *
;*   event seizes one or more facilities on its route. The facilities    *
;*   represent the RunQueue or SleepQueues1-4. The facilities occur in   *
;*   more than one route causing interaction between events from         *
;*   different processors. The resulting waiting times are summed and    *
;*   used to calculate the system efficiency V$Effi. This V$Effi         *
;*   multiplied by NrOfPro gives us our final result: The number of      *
;*   efficient processors.   Parameter 4 of the events holds the IAT    *
;*   of the event as an absolute time on the system time axe. When an    *
;*   event returns to the chain after its 'walk' it reschedules itself   *
;*   on the time-axis and links itself onto the chain. The first event   *
;*   due next is then released. Both mean and distribution of IAT's      *
;*   are independent per event. The times spent in the facilities,       *
;*   are also independent in both aspects.                               *
;*                                                                       *
;*   An attempt has been made to use (Unix) QueueLengthes to influence   *
;*   time spent in facilities. This was never finished. Residual         *
;*   blocks are still in the model. They are best recognizable by        *
;*   their use of parameter 3.                                           *
;*                                                                       *
;*      Time Unit = 1 millisecond                                        *
;*************************************************************************
;
;
;
;
;
;
```

```
10  *******************************************
20  *                                         *
30  *          GPSS/PC PROGRAM                *
40  *                                         *
50  *******************************************
60  *
70  *
80  ********************************************
90  *
100 * E Q U A T E S
110 *
120 ********************************************
130 *
140 *
150 *
160 FOMEAN1    EQU       300                 ;Fork Mean IAT in 1 msec Units
170 EXMEAN1    EQU       300                 ;Exit
180 TSMEAN1    EQU       1000                ;TimeSlice
190 SL1MEAN1   EQU       750                 ;Sleep on SQ1
200 SL2MEAN1   EQU       700                 ;Sleep on SQ2
210 SL3MEAN1   EQU       650                 ;Sleep on SQ3
220 SL4MEAN1   EQU       300                 ;Sleep on SQ4
230 WU1MEAN1   EQU       750                 ;WakeUp from SQ1
240 WU2MEAN1   EQU       700                 ;WakeUp from SQ2
250 WU3MEAN1   EQU       650                 ;WakeUp from SQ3
270 WU4MEAN1   EQU       300                 ;WakeUp from SQ4
280 PROCNR     EQU       20                  ;SaveValue Nr 20   Contains ProcNr
290 RQ         EQU       5                   ;Unix Queues have been numbered
300 SQ1        EQU       1                   ;  for easy reference
310 SQ2        EQU       2                   ;  and recognition
320 SQ3        EQU       3                   ;  (These Unix Queues are
330 SQ4        EQU       4                   ;   GPSS Facilities)
320 *
330 *
340 *
350 ********************************************
360 *
370 * V A R I A B L E S
380 *
390 ********************************************
400 *
410 *
420 *
430 NROFPRO    VARIABLE  16                  ;Number of Processors In System
440 *
450 HULPVAR    FVARIABLE P4-AC1              ;Used in RestTime Calculation
460 RESTTIME   FVARIABLE (V$HULPVAR'G'0)#V$HULPVAR+(V$HULPVAR'LE'0)#0
470 *                                        ;Time till return at Chain
480 *                                        ;  meas'd from Chain departure
490 WAITKJUS   FVARIABLE 5+P6                ;Q's that measure Wait Time for
```

```
500 *                                            ;  access to Facility P6
510 SUMOHT      FVARIABLE (QC6#QT6)+(QC7#QT7)+(QC8#QT8)+(QC9#QT9)+(QC10#QT10)
520 *                                            ;Sum of Overhead(Waiting) Times
530 C1A         FVARIABLE (C1'G'0)#C1+(C1'E'0)#1 ;C1 is time since CLEAR
540 *                                            ;C1A is C1 except when C1 = 0
550 *                                            ;  then C1A is 1 to avoid ovflw
560 EFFI        FVARIABLE (((V$C1A#V$NROFPRO)-V$SUMOHT)/(V$C1A#V$NROFPRO))#1000
570 *                                            ;System Efficiency
580 EFNROFP     FVARIABLE (TB$EFFICI#V$NROFPRO)      ;Eff. Numb. of Proc'rs
590 *
600 *
610 FOIAT       FVARIABLE P40+(FOMEAN1#FN$XPDIS1#1000)/V$EFFI
620 EXIAT       FVARIABLE P40+(EXMEAN1#FN$XPDIS1#1000)/V$EFFI
630 TSIAT       FVARIABLE P40+(TSMEAN1#1000)/V$EFFI
640 SL1IAT      FVARIABLE P40+(SL1MEAN1#FN$XPDIS1#1000)/V$EFFI
650 SL2IAT      FVARIABLE P40+(SL2MEAN1#FN$XPDIS1#1000)/V$EFFI
660 SL3IAT      FVARIABLE P40+(SL3MEAN1#FN$XPDIS1#1000)/V$EFFI
670 SL4IAT      FVARIABLE P40+(SL4MEAN1#FN$XPDIS1#1000)/V$EFFI
680 WU1IAT      FVARIABLE P40+(WU1MEAN1#FN$XPDIS1#1000)/V$EFFI
690 WU2IAT      FVARIABLE P40+(WU2MEAN1#FN$XPDIS1#1000)/V$EFFI
700 WU3IAT      FVARIABLE P40+(WU3MEAN1#FN$XPDIS1#1000)/V$EFFI
710 WU4IAT      FVARIABLE P40+(WU4MEAN1#FN$XPDIS1#1000)/V$EFFI
720 *
730 ADDPROC     FVARIABLE 1
740 GRABPROC    FVARIABLE 3
750 PROCSWTCH FVARIABLE V$ADDPROC+V$GRABPROC
760 MAKEZOMBI   FVARIABLE 5
770 FORKTIME    FVARIABLE 5
780 SLEEPTIM1   FVARIABLE V$ADDPROC
790 SLEEPTIM2   FVARIABLE V$ADDPROC
800 SLEEPTIM3   FVARIABLE V$ADDPROC
810 SLEEPTIM4   FVARIABLE V$ADDPROC
820 WAKETIM1    FVARIABLE 2
830 WAKETIM2    FVARIABLE 3
840 WAKETIM3    FVARIABLE 4
850 WAKETIM4    FVARIABLE 5
860 *
870 *
880 *
890 ********************************************
900 *
910 * M I S C E L L A N E O U S
920 *
930 ********************************************
940 *
950 *
960 *
970             INITIAL   X$PROCNR,0
980             RMULT     73145,96847,31043,28645,40187,36641,91239
990 QL          MATRIX    ,1,100
```

```
1000EFFICI        TABLE       V$EFFI,10,10,100
1010*
1020*
1030*
1040*******************************************
1050*
1060* F U N C T I O N S
1070*
1080*******************************************
1090*
1100*
1110* Exponential Function
1120*
1130XPDIS1        FUNCTION  RN1,C24
0.0,0.0/0.1,.104/.2,.222/.3,.335/.4,.509/.5,.69
0.6,.915/.7,1.2/.75,1.38/.8,1.6/.84,1.83/.88,2.12
0.9,2.3/.92,2.52/.94,2.81/.95,2.99/.96,3.2/.97,3.5
0.98,3.9/.99,4.6/.995,5.3/.998,6.2/.999,7.0/.9997,8.0
1140*
1150*
1160* Arrival Time per Event Type
1170*
1180IAT           FUNCTION  P2,M11
1,V$FOIAT/2,V$EXIAT/3,V$TSIAT
4,V$SL1IAT/5,V$SL2IAT/6,V$SL3IAT/7,V$SL4IAT
8,V$WU1IAT/9,V$WU2IAT/10,V$WU3IAT/11,V$WU4IAT
1190*
1200*
1210* 'Which Route' per Event Type
1220*
1230EVCONT        FUNCTION  P2,L11
1,FORK/2,EXIT/3,ALOT
4,ALOT/5,ALOT/6,ALOT/7,ALOT
8,ALOT/9,ALOT/10,ALOT/11,ALOT
1240*
1250*
1260* (Initial) Influence on (Unix) QueueLength per EventType
1270*
1280QLINFLU       FUNCTION  P2,L11
1,1/2,-1/3,0
4,1/5,1/6,1/7,1
8,-1/9,-1/10,-1/11,-1
1290*
1300*
1310* 'What Queue to seize' per EventType
1320*
1330EVKJU         FUNCTION  P2,L11
1,RQ/2,RQ/3,RQ
4,SQ1/5,SQ2/6,SQ3/7,SQ4
8,SQ1/9,SQ2/10,SQ3/11,SQ4
```

```
1340*
1350*
1360* 'Which Service Time' per EventType
1370*
1380EVTIME       FUNCTION  P2,M11
1,V$FORKTIME/2,V$GRABPROC/3,V$PROCSWTCH
4,V$SLEEPTIM1/5,V$SLEEPTIM2/6,V$SLEEPTIM3/7,V$SLEEPTIM4
8,V$WAKETIM1/9,V$WAKETIM2/10,V$WAKETIM3/11,V$WAKETIM4
1390*
1400*
1410* 'Where to go to' per EventType
1420*
1430EVJUMP       FUNCTION  P2,L11
1,RUN/2,RUN/3,RUN
4,SWITCH/5,SWITCH/6,SWITCH/7,SWITCH
8,WUSWTCH/9,WUSWTCH/10,WUSWTCH/11,WUSWTCH
1440*
1450*
2000*****************************************
2010*
2020* S T A R T   M O D E L
2030*
2040*****************************************
2050*
2060*
2070             GENERATE  ,,,V$NROFPRO          ;
2080             SAVEVALUE PROCNR+,1             ;
2090             ASSIGN    1,X$PROCNR            ;P1 = Processor Number
2100             ASSIGN    2,-1                  ;P2 = Event Number: FORK = 1 etc.
2110             ASSIGN    3,0                   ;{-1,0,1} Influence on UX Q Length
2120             ASSIGN    4,0                   ;Arrival Time
2130             ASSIGN    40,0                  ;Saved Arrival Time
2140             ASSIGN    5,0                   ;
2150             ASSIGN    6,0                   ;
2160             ADVANCE   1                     ;Allow For Contig. Proc. XACT Nrs
2170             SPLIT     11,RUN1,2             ;Make the 11 Event XActs
2180RESCHED      LOGIC S   P1                    ;The original XActs serve as
2190             GATE LR   P1                    ;  Gate Openers for Chains
2200             UNLINK    P1,ENTRY,1            ;Send next Event On its Way
2210             LOGIC S   P1                    ;Close Gate
2220             TRANSFER  ,RESCHED              ;Go and wait for Event return
2230*****************************************
2240RUN          LOGIC R   P1                    ;Trigger Gate Opener
2250RUN1         ASSIGN    4,FN$IAT              ;Reschedule on System Time-Axis
2260             ASSIGN    40,P4                 ;Remember this for next IAT calc.
2270             LINK      P1,P4,ENTRY           ;First Transaction Avoids Link
2280*****************************************
2290ENTRY        ASSIGN    4,V$RESTTIME          ;Entry Point for Event Route
2300             ADVANCE   P4                    ;Wait until due
2310             TRANSFER  ,FN$EVCONT            ;Transfer to Event Entry Point
```

```
2320*******************************************
2330FORK          ASSIGN    3,1                  ;FORK Entry Point; Incr. QueueL
2340              ASSIGN    DELTIM,V$FORKTIME    ;Parm DELTIM holds Q Manipul. Tim.
2350              TRANSFER  SBR,BLOCKED,RETADDR ;Call Subroutine BLOCKED
2360              TRANSFER  ,RUN                 ;Return to Chain
2370*******************************************
2380EXIT          ASSIGN    3,0                  ;EXIT Entry Point
2390              ASSIGN    DELTIM,V$MAKEZOMBI
2400              TRANSFER  SBR,BLOCKED,RETADDR
2410              TRANSFER  ,ALOT
2420*******************************************
2430ALOT          ASSIGN    3,FN$QLINFLU         ;Entry Point for rest Events
2440              ASSIGN    DELTIM,FN$EVTIME
2450              ASSIGN    6,FN$EVKJU
2460              TRANSFER  SBR,KJU,RETADDR
2470              TRANSFER  ,FN$EVJUMP           ;Sleeps/WakeUps Jump to (WU)SWITCH
2480*******************************************
2490*
2500SWITCH        ASSIGN    DELTIM,V$GRABPROC ;BreakOut Sleep Events
2510              ASSIGN    3,-1
2520              ASSIGN    6,RQ
2530              TRANSFER  SBR,KJU,RETADDR      ;"Grab a Process from RQ"
2540              ASSIGN    3,1
2550              TRANSFER  ,RUN
2560*******************************************
2570*
2580WUSWTCH       ASSIGN    DELTIM,V$ADDPROC     ;Breakout WakeUp Events
2590              ASSIGN    3,1
2600              ASSIGN    6,RQ
2610              TRANSFER  SBR,KJU,RETADDR      ;"Add a Process to RQ"
2620              ASSIGN    3,-1
2630              TRANSFER  ,RUN
2640*
2650*
3000**************************************************************************
3010*
3020* S U B R O U T I N E S
3030*
3040**************************************************************************
3050*
3060* Seize total Process Table:
3070*
3080BLOCKED       QUEUE     6
3090              SEIZE     SQ1                  ;you need entire
3100              DEPART    6
3110              QUEUE     SQ1
3120              QUEUE     7
3130              SEIZE     SQ2                  ;process table to do so
3140              DEPART    7
3150              QUEUE     SQ2
```

```
3160            QUEUE      8
3170            SEIZE      SQ3
3180            DEPART     8
3190            QUEUE      SQ3
3200            QUEUE      9
3210            SEIZE      SQ4
3220            DEPART     9
3230            QUEUE      SQ4
3240            QUEUE      10
3250            SEIZE      RQ
3260            DEPART     10
3270            QUEUE      RQ
3280            ADVANCE    P$DELTIM
3290            MSAVEVALUE QL+,1,RQ,P3
3300            DEPART     RQ
3310            RELEASE    RQ
3320            DEPART     SQ4
3330            RELEASE    SQ4
3340            DEPART     SQ3
3350            RELEASE    SQ3                  ;release in reverse order
3360            DEPART     SQ2
3370            RELEASE    SQ2
3380            DEPART     SQ1
3390            RELEASE    SQ1
3400            TRANSFER   P,RETADDR,1
3410*
3420* end of subroutine
3430*
4000*************************************************************************
4010*
4020* Seize one Queue
4030*
4040KJU         QUEUE      V$WAITKJUS
4050            SEIZE      P6
4060            DEPART     V$WAITKJUS
4070            QUEUE      P6
4080            ADVANCE    P$DELTIM
4090            MSAVEVALUE QL+,1,P6,P3
4100            DEPART     P6
4110            RELEASE    P6
4120            TRANSFER   P,RETADDR,1
4130*
4140* end of subroutine
4150*
5000*************************************************************************
5010*
5020* Measure Transactions
5030*
5040            GENERATE   100
5050            TABULATE   EFFICI
```

```
5060          SAVEVALUE FINAL,V$EFNROFP
5070          TERMINATE 1
5080*
5090*
6000*****************************************************************************
6010*
6020*    E N D   O F   M O D E L
6030*
6040*****************************************************************************
```

The measurement Series:

First Series:

Event Inter Arrival Times:

Event	Mean	Distribution	Event	Mean	Distribution			
FORK	300	Exponential	SLEEP1	750	Exponential	WAKEUP1	750	Exponential
EXIT	300	Exponential	SLEEP2	700	Exponential	WAKEUP2	700	Exponential
TIMESLICE	1000	Deterministic	SLEEP3	650	Exponential	WAKEUP3	650	Exponential
			SLEEP4	300	Exponential	WAKEUP4	300	Exponential

	Add	Grab	Switch	Fork	Make	Sleep	Sleep	Sleep	Sleep	Wake	Wake	Wake	Wake
Number	Proc	Proc	Tim	Tim	Zombie	Tim1	Tim2	Tim3	Tim4	Tim1	Tim2	Tim3	Tim4
84	2	5	7	10	10	2	2	2	2	2	3	4	5
78	2	5	7	10	10	0	0	0	0	0	0	0	0
72	2	6	8	10	10	0	0	0	0	0	0	0	0
71	1	3	4	5	5	0	0	0	0	0	0	0	0
65	1	3	4	5	5	1	1	1	1	2	3	4	5
64	1	3	4	10	5	1	1	1	1	2	3	4	5
63	1	3	4	2	5	1	1	1	1	2	3	4	5
192	1	3	4	5	5	0	0	0	0	0	0	0	1
191	1	3	4	5	5	0	0	0	0	0	0	0	2
190	1	3	4	5	5	0	0	0	0	0	0	0	5
189	1	3	4	5	5	0	0	0	0	0	0	0	10
188	1	3	4	5	5	0	0	0	0	0	0	0	15
187	1	3	4	5	5	0	0	0	0	0	0	0	20
186	1	3	4	5	5	0	0	0	0	0	0	0	30
185	1	3	4	5	5	0	0	0	0	0	0	0	40

Second Series:

Event Inter Arrival Times:

Event	Mean	Distribution	Event	Mean	Distribution			
FORK	300	Exponential	SLEEP1	750	Exponential	WAKEUP1	300	Exponential
EXIT	300	Exponential	SLEEP2	700	Exponential	WAKEUP2	300	Exponential
TIMESLICE	1000	Deterministic	SLEEP3	650	Exponential	WAKEUP3	300	Exponential

SLEEP4 300 Exponential WAKEUP4 300 Exponential

Number	Add Proc	Grab Proc	Switch Tim	Fork Tim	Make Zombie	Sleep Tim1	Sleep Tim2	Sleep Tim3	Sleep Tim4	Wake Tim1	Wake Tim2	Wake Tim3	Wake Tim4
70	0	0	0	0	0	0	0	0	0	0	0	10	10
69	0	0	0	0	0	0	0	0	0	0	10	10	10
68	0	0	0	0	0	0	0	0	0	10	10	10	10
67	0	0	0	0	0	0	0	0	0	0	0	5	10
66	0	0	0	0	0	0	0	0	0	0	0	15	10
62	0	0	0	0	0	0	0	0	0	0	0	12	10
58	0	0	0	0	0	0	0	0	0	0	0	5	5
57	0	0	0	0	0	0	0	0	0	0	5	5	5
56	0	0	0	0	0	0	0	0	0	5	5	5	5
55	0	0	0	0	0	0	0	0	0	0	0	20	20
54	0	0	0	0	0	0	0	0	0	0	20	20	20
53	0	0	0	0	0	0	0	0	0	20	20	20	20
52	0	0	0	0	0	0	0	0	0	0	0	1	10
51	0	0	0	0	0	0	0	0	0	0	0	3	10
199	0	0	0	0	0	0	0	0	0	0	0	20	10
198	0	0	0	0	0	0	0	0	0	0	0	25	10
197	0	0	0	0	0	0	0	0	0	0	0	0	10
196	0	0	0	0	0	0	0	0	0	0	0	0	5
195	0	0	0	0	0	0	0	0	0	0	0	0	20
194	0	0	0	0	0	0	0	0	0	0	0	0	15
193	0	0	0	0	0	0	0	0	0	0	0	0	25
157	1	3	4	5	5	0	0	0	0	0	0	0	1
156	1	3	4	5	5	0	0	0	0	0	0	0	2
155	1	3	4	5	5	0	0	0	0	0	0	0	5
154	1	3	4	5	5	0	0	0	0	0	0	0	10
153	1	3	4	5	5	0	0	0	0	0	0	0	15
152	1	3	4	5	5	0	0	0	0	0	0	0	20
151	1	3	4	5	5	0	0	0	0	0	0	0	30
150	1	3	4	5	5	0	0	0	0	0	0	0	40
184	1	3	4	5	5	0	0	0	0	0	0	1	1
183	1	3	4	5	5	0	0	0	0	0	0	2	2
182	1	3	4	5	5	0	0	0	0	0	0	5	5
181	1	3	4	5	5	0	0	0	0	0	0	10	10
180	1	3	4	5	5	0	0	0	0	0	0	15	15
179	1	3	4	5	5	0	0	0	0	0	0	20	20
178	1	3	4	5	5	0	0	0	0	0	0	30	30
177	1	3	4	5	5	0	0	0	0	0	0	40	40
176	1	3	4	5	5	0	0	0	0	0	1	1	1
175	1	3	4	5	5	0	0	0	0	0	2	2	2
174	1	3	4	5	5	0	0	0	0	0	5	5	5
173	1	3	4	5	5	0	0	0	0	0	10	10	10
172	1	3	4	5	5	0	0	0	0	0	15	15	15
171	1	3	4	5	5	0	0	0	0	0	20	20	20
170	1	3	4	5	5	0	0	0	0	0	30	30	30
169	1	3	4	5	5	0	0	0	0	0	40	40	40
168	1	3	4	5	5	0	0	0	0	0	1	1	1
167	1	3	4	5	5	0	0	0	0	0	2	2	2
166	1	3	4	5	5	0	0	0	0	0	5	5	5
165	1	3	4	5	5	0	0	0	0	0	10	10	10
164	1	3	4	5	5	0	0	0	0	0	15	15	15
163	1	3	4	5	5	0	0	0	0	0	20	20	20
162	1	3	4	5	5	0	0	0	0	0	30	30	30
161	1	3	4	5	5	0	0	0	0	0	40	40	40
160	0	0	0	0	0	0	0	0	0	0	0	0	30
159	1	3	4	5	5	0	0	0	0	0	0	0	0

Third Series:

Event Inter Arrival Times:

Event	Mean	Distribution	Event	Mean	Distribution			
FORK	300	Exponential	SLEEP1	750	Exponential	WAKEUP1	750	Exponential
EXIT	300	Exponential	SLEEP2	700	Exponential	WAKEUP2	700	Exponential
TIMESLICE	1000*	Deterministic	SLEEP3	650	Exponential	WAKEUP3	650	Exponential
			SLEEP4	300	Exponential	WAKEUP4	300	Exponential

*) Here the mean value is independent of N. (So not multiplied by V$EFFI)

Number	Add Proc	Grab Proc	Switch Tim	Fork Tim	Make Zombie	Sleep Tim1	Sleep Tim2	Sleep Tim3	Sleep Tim4	Wake Tim1	Wake Tim2	Wake Tim3	Wake Tim4
59	2	5	7	10	10	2	2	2	2	2	3	4	5

Fourth Series:

Event Inter Arrival Times:

Event	Mean	Distribution	Event	Mean	Distribution			
FORK	300	Exponential	SLEEP1	infinite	Exponential	WAKEUP1	infinite	Exponential
EXIT	infinite	Exponential	SLEEP2	infinite	Exponential	WAKEUP2	infinite	Exponential
TIMESLICE	infinite	Deterministic	SLEEP3	infinite	Exponential	WAKEUP3	infinite	Exponential
			SLEEP4	infinite	Exponential	WAKEUP4	infinite	Exponential

Number	Add Proc	Grab Proc	Switch Tim	Fork Tim	Make Zombie	Sleep Tim1	Sleep Tim2	Sleep Tim3	Sleep Tim4	Wake Tim1	Wake Tim2	Wake Tim3	Wake Tim4
74	0	0	0	30	0	0	0	0	0	0	0	0	0

Fifth Series:

Event Inter Arrival Times:

Event	Mean	Distribution	Event	Mean	Distribution			
FORK	300	Exponential	SLEEP1	infinite	Exponential	WAKEUP1	infinite	Exponential
EXIT	infinite	Exponential	SLEEP2	infinite	Exponential	WAKEUP2	infinite	Exponential
TIMESLICE	infinite	Deterministic	SLEEP3	infinite	Exponential	WAKEUP3	infinite	Exponential
			SLEEP4	infinite	Exponential	WAKEUP4	300	Exponential

Number	Add Proc	Grab Proc	Switch Tim	Fork Tim	Make Zombie	Sleep Tim1	Sleep Tim2	Sleep Tim3	Sleep Tim4	Wake Tim1	Wake Tim2	Wake Tim3	Wake Tim4
73	0	0	0	30	0	0	0	0	0	0	0	0	30

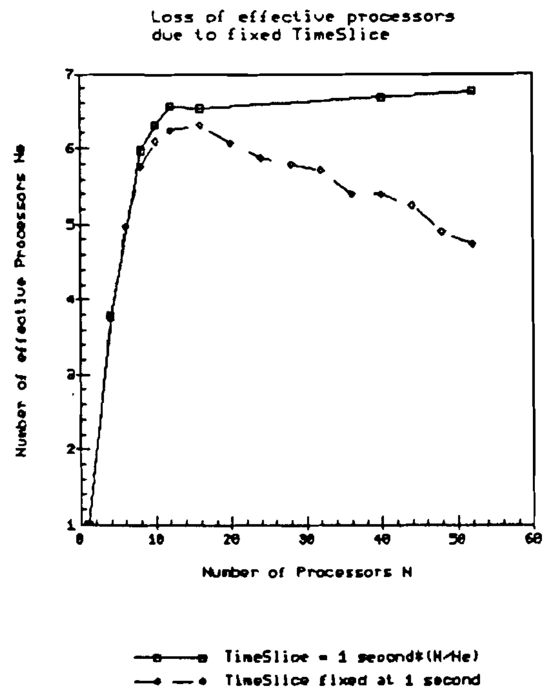
135g

Measurement Data:

Measurement Number	4	6	8	10	12	16	20	24	28	32	36	40	44	48	52	56	58	64
51											25.78	26.01	26.93	26.78	27.32			
52											26.29	27.36	27.35	28.33	28.51			
53	3.67	4.94	5.73	6.34	6.40	6.75								7.06	7.02			
54	3.85	5.49	6.77	7.49	8.06	8.59	8.85	8.71	8.82									
55	3.88	5.63	7.61	8.25	9.07	10.08	10.41	10.59	10.78	10.78	10.66	10.64						
56	4.00	6.00	8.00	10.00	12.00	15.76	19.23	22.43	25.11	26.60	27.78	28.36	28.98	29.23				
57	4.00	6.00	8.00	10.00	12.00	15.89	19.60	23.19	26.39	28.98	30.75	32.06	32.89	33.08	33.82			
58	4.00	6.00	8.00	10.00	12.00	16.00	19.72	23.58	27.24	30.64	33.36	35.81	37.73	38.71	39.62			
59	3.78	4.97	5.76	6.09	6.23	6.30	6.07	5.88	5.79	5.72	5.40	5.40	5.25	4.90	4.73			
62	3.99	5.92	7.84	9.71	11.40	14.40	16.53	17.50	18.20	18.42	18.59	18.86	19.07	18.96				
63	4.00	5.99	7.91	9.37	10.73	12.24	12.94	13.11	13.12	13.24	13.32	13.36						
64	3.96	5.81	7.19	8.28	8.83	9.37	9.65	9.77	9.85	9.88								
65	4.00	5.94	7.54	9.05	10.15	11.10	11.19	11.48	11.51	11.83	11.68	11.68		11.66				
66	3.97	5.91	7.76	9.52	11.03	13.62	15.03	15.78	15.89	16.01	16.61	16.33	16.55	16.60	16.68			
67	4.00	5.99	7.94	9.88	11.81	15.44	18.55	21.29	22.94	24.19	24.75	25.27	25.52	25.72	26.01			
68	4.00	5.84	7.73	9.43	10.98	13.01	14.03	14.69	14.83	15.11	15.28	15.37	15.37	15.50	15.52			
69	4.00	5.91	7.83	9.64	11.24	13.97	15.67	16.69	16.67	17.03	17.22	17.29	17.51	17.67	17.42			
70		5.94		9.80	11.58	14.83	17.39	18.98	20.27	20.68	20.87	20.91	21.02	21.07	21.11			
71	4.00	6.00	7.72	9.44	10.60	11.89	12.52	12.59	12.75	12.86	13.15	12.84	12.95	13.00	12.96			
72	3.73	4.93	5.67	6.03	6.26													
73	3.54	4.39	4.77	5.03	5.02	5.08	5.01	5.01	5.18	5.06	5.38	5.16	5.41		5.54			
74	3.91	5.69	7.23	8.39	9.17	9.67	9.84	9.93	10.19	10.13	10.27	10.64	10.31	9.94	10.06	10.06	10.01	10.05
78	3.96	5.11	6.06	6.49	6.69	6.93	6.95	6.99	6.93	7.26	7.24	7.08	7.25	7.36	7.16			
84	3.78		5.97	6.30	6.55	6.53						6.68			6.76			
158	3.68		5.41		5.89	6.02	6.08	6.10				6.20						
159	4.00		7.66		10.15	11.20	11.60	11.76	11.96						11.96			
160	3.79		6.35		7.73	8.74	9.06								9.93			
161												2.84						
162												3.60						
163												4.80						
164												5.96						
165												7.68						
166												10.24						
167												11.72						
168												11.84						
169												3.20						
170												4.08						
171												5.32						
172												6.52						
173												8.24						
174												10.64						
175												11.72						
176												12.04						
177												3.76						
178												4.64						
179												6.24						
180												7.40						
181												9.12						
182												10.96						
183												11.84						
184												11.29						
185												5.24						
186												6.36						
187												7.80						
188												8.84						
189												10.40						
190												12.08						
191												12.84						

192												12.76			
193	3.84		6.82		8.65	9.94	10.48	11.04	11.23	11.46				11.95	
194	3.94		7.69		10.79		14.88		17.14	18.08	18.57			19.34	
195	3.89		7.34		9.77	11.33	12.46	13.27						15.07	
196	4.00				12.00				27.30		34.24		40.44	43.01	45.08
197	4.00	5.95	7.88	9.86	11.68	15.04	18.06	20.42	22.46		25.13	25.80	26.97	27.36	28.24
198											10.97	11.40	11.31	11.01	11.18
199											13.59	13.31	13.61	13.83	13.51

Several combinations of the above results are possible. Presenting these in graphs would lead to excessive amounts of paper. One exception is made for the measurements 59 and 84. The graph below speaks for itself:



Appendix D Chapter 2 in Tables

2.1 Memory Map

- Shared Memory Only
- Private plus Shared Memory *

2.2 Interconnections

- Processors-Memory Modules
    - Circuit Switching Techniques
        - Time Shared Bus *
        - Crossbar
        - Multiport
    - Packet Switching Techniques

- Processor-Processor
    - Point to Point
    - Separate Time Shared Bus
    - Using Processors-Memory datapath *

2.3 Symmetry

- Full Symmetry
- Main Processors are unequal to IO processors *
- Main Processors are of different types
- ...
- ...
- Totally asymmetric

2.4 Input/Output Structure

- All IO done by uniprocessor
- IO done by IO processors (no OS tasks in IO)
- IO done by IO processors (including OS tasks) *
- IO requests via 1 main processor
- IO requests via any main processor *

2.5 Interrupts

- Internal Interrupts Local
    - All others on 1 (master) main processor.
    - Hardware Fault interrupts on separate proc.
    - Rest also Local
    - Interrupter looks for handling processor according to some searching scheme. (f.e Daisy Chain implementation)
    - Others are handled by main processor that handles that particular level of interrupt.

2.6 Inter Processor Communication

- Local/Fast

  - Shared Memory *
    . Data block in Memory
    . Signal server
    . Server reads block
    . Server answers in same block or sends new separate block
    . Server signals requester.

- Remote/Slow

  - Communication Lines

    - Message Handling Protocols (X400)
    - Actor

- Software Methods

  - Shared Memory/Polling
    . Slow
    . Deadlock (possibly when processor runs failing job that holds locks and other processors have no way of knowing this)

- Hardware Methods

  - Location Monitors (VME) *
  - SIGP (chapter 2 [SATY80])

2.7 Reserved Storage Areas

- Interrupt Vectors
- IO status and control
- Diagnostics
- Process Relocation Address Spaces

Every main could have such areas: possible clash of reserved areas

Solutions:
- Put these areas on the same addresses for every processor: physically in parallel
- Put these reserved areas on the same addresses for every processor: 1 physical memory(mutual exclusion mechanisms necessary)
- Put these reserved areas on sequential addresses
- Reserved Areas in private memory.

2.8 The Operating System

- OS in shared memory/every main processor uses it.
  . re-entrant
  . locks around critical actions

- 1 master handles critical OS code/n slaves do rest

- Every main processor has same OS code to handle interrupts & traps. Rest of OS functions is mapped at system startup time onto a set of main processors.

- Protection
- Scheduling
- Memory Management
- File Handling

2.9 User Facilities for Parallel Processing

- System Calls to manage processes
  (create, sync, communicate, delete)
- Languages that provide 'parallel' constructs
  (PARBEGIN {statement} PAREND;)
- Parallelizing Compilers and Libraries

2.10 Error Recovery

- Watchdog timer
- Hardware-generated malfunction alert
- Reset; restart and diagnostic capabilities from one main processor towards another.
- Corruption Check of Critical Data & Recovery
- Deadlock
- Restart system from other hard disk instance of OS
- Backtrack failed process until safe restart reached or restart completely.

2.11 Performance

- Memory Contention
- Cache Consistence Delay
- Longer Scheduling than on single processor system
- Waiting in semaphore queues
- More complex device handling
- Load Balancing

Appendix E Implementation Conditions

Condition 3.1
The implementation must be based on the VMEbus.

Condition 3.2
The Operating System to be used is Unix^R System V.

Condition 3.3
VME interrupts shall only be used for the handling of hardware faults.

Condition 4.1
No Caches are used (yet).

Literature

[ACCE86] Accetta M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation For Unix Development," *Proc. 1986 Summer Usenix Techn. Conf.*, Atlanta, Georgia, p446-458, 1986.

[ANNO85] Annot J.K. and M.D. Janssens, *Multi Processor Unix*, M.Sc. Thesis, Code 1-68340-28(1985)07, Delft University of Technology, Dep. of Electrical Engineering, Delft, june 1985.

[BACH84] Bach M.J. and S.J. Buroff, "Multiprocessor Unix Operating Systems," *AT&T Bell Laboratories Techn. Journal 63*, No 8, Oct. 1984.

[BACH86] Bach M.J, *The Design of the Unix*^{R} *Operating System*, Prentice Hall, Englewood Cliffs, NJ, 1986.

[BERK83] *Unix Programmer's Manual*, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley, August 1983.

[BOKH82] vanBokhoven L., *De bouw van de 'THE KUNix Machine'*, Internal Report, Digital Systems Division, Department of Electrical Engineering, Eindhoven University of Technology, Netherlands, feb. 1982.

[BOKH83] vanBokhoven L., *De Interprocessor Communicatie van de THE KUNix Machine*, Internal Report, Digital Systems Division, Department of Electrical Engineering, Eindhoven University of Technology, Netherlands, 1983.

[BREG83] Bregman P., *Ontwikkeling van Ondersteunende Systeemsoftware voor een Serie I/O Controller*, Digital Systems Division, Department of Electrical Engineering, Eindhoven University of Technology, Netherlands, 1983.

[CABR86] Cabrera L.F., "The Influence of Workload on Load Balancing Strategies," *Proc. 1986 Summer Usenix Techn. Conf.*, Atlanta, Georgia, p446-458, 1986.

[CEZZ83] Cezzar K. and D. Klappholz, "Process Management in a Speedup-Oriented MIMD System," *Proc. of Int. Conf. on Parallel Processing*, 1983.

[CONT85] Conte G. and D. DelCorso, *Multi-Microprocessor Systems for Real-Time Applications*, D.Reidel Publishing Company, Dordrecht, ISBN 90-277-2054-1, 1985.

[DIJK65] Dijkstra E.W., "Solution of a Problem in Concurrent Programming Control," *Comm. of the ACM*, Vol 8, No 9, p569, 1965.

[DIJK68] Dijkstra E.W., "The Structure of the 'THE'-Multi-Programming System," *Comm. of the ACM*, Vol 11, No 5, p341-346, 1968.

[DUBO88] Dubois M., C. Scheurich and F.A. Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors," *Computer*, p9-21, Feb. 1988.

[EAGE86] Eager D.L., E.D Lazowska and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems", *IEEE Trans. on Software. Eng. 12*, No 5, p662-675, May 1986.

[EDLE85] Edler J., A. Gottlieb, J. Lipkis, "Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3," *Proc. of the USENIX Techn. Conf.*, Denver, Colorado, p193-210, Winter 1986.

[FLYN72] Flynn M.J., "Some Computer Organizations and Their Effectiveness," *IEEE Trans. on Comp.*, C-21, 9, p948-960, Sep. 1972.

[FORC87] *FORCE Computers Data Book 1988*, FORCE Computers, Oct. 1987.

[FREE84] Freeman M. and C. Kaplinsky, "On a model for Multiprocessor Systems with Private Caches", Signetics Corporation, Sunnyvale California, 1984.

[GOTT83a] Gottlieb A., R. Grisman, C.P Kruskal, K.P. McAuliffe, L. Rudolph and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. on Comp.*, p175-189, Feb. 1983.

[GOTT83b] Gottlieb A., B. Lubachevsky and L. Rudolph, "Basic Techniques for the Efficient Coordination of very large numbers of Cooperating Sequential Processors," *ACM TOPLAS 5*, Jan. 1983.

[GPSS88] *GPSS/PC^{TM} general purpose simulation Reference Manual*, Minuteman Software, Stow Massachusetts, 1988.

[HEAT87] Heath S., "New Multiprocessing Techniques for VME Bus," *New Electronics*, Vol 20, No 16, p23-24, aug. 1987.

[HENR84] Henry G.J., "The Fair Share Scheduler," *AT&T Bell Laboratories Techn. Journal*, Vol 63, No 8, Part 2, p1845-1858, Oct. 1984.

[HILL86] Hillis W.D, *The Connection Machine*, MIT Press, London, ISBN 0-262-08157-1, 1986.

[JACO86] Jacobs H., "A User-tunable Multiple Processor Scheduler," *Proc. of the USENIX Techn. Conf.*, Denver, Colorado, p183-191, Winter 1986.

[JANS86] Janssens M.D., J.K.Annot and A.J. van de Goor, "Adapting Unix for a Multiprocessor Environment," *Comm. of the ACM*, Vol 29, No 9, p895-901, sept. 1986.

[JONE79] Jones A.K., R.J. Chansler, I.E. Durham, K. Schwans and S. Vegdahl, "StarOS, A Multiprocessor Operating System for the Support of Task Forces," *Proc. 7th Symp. Operating System Princ. ACM*, p117-129, Dec. 1979.

[KAPL81] Kaplinsky C., "Decentralizing uP Bus grows easily from 16 to 32 Bits," *Electronic Design*, p173-179, nov. 1981.

[LEBL87] LeBlanc T.J. and S. Jain, "Crowd Control: Coordinating Processes in Parallel," *Proc. of the 1987 Intern. Conf. on Parallel Processing*, p81-84, 1987.

[LEFF84] Leffler S., M. Karels and M.K. McKusick, "Measuring and Improving the Performance of 4.2BSD", *Proc. of the USENIX Conf.*, Salt Lake City, p228-236, June 1984.

[LELA86] Leland W.E. and T.J. Ott, "Load-balancing heuristics and process behavior," *Proc. ACM SigMetrics Performance 1986 Conf.*, p54-69, 1986.

[LION77] Lions J., *A Commentary on the Unix Operating System* and *UNIX Operating System Source Code, Level 6*, Dep. of Comp. Sciences, University of New South Wales, 1977.

[LIPO88] Lipovsky G.J. and P. Vaughan, "A Fetch-And-Op Implementation for Parallel Computers," *IEEE Proc. Computer Architecture*, p384-392, 1988.

[LIST75] Lister A.M., *Fundamentals of Operating Systems*, The Macmillan Press Ltd., London, 1975.

[MUEH80] Muehlemann K., *Eln Beitrag zur Synchronisation in Mehrprozessorsystemen und Computernetzwerken*, Ph.D. Thesis, Diss. ETH Nr. 6520, Eidgenoessischen Technischen Hochschule Zuerich, Zuerich, 1980.

[MUNT69] Muntz R. R. and E.G. Coffman, "Optimal Preemptive Scheduling on Two Processor Systems," *IEEE Trans. on Comp.*, C-18, p1014-1020, Nov. 1969.

[OUST88] Ousterhout J.K., A.R. Cherenson, F. Douglis, M.N. Nelson and B.B Welch, "The Sprite Network Operating System," *IEEE Computer 21(2)*, p23-36, feb. 1988.

[PADU80] Padua D.A, D.J. Kuck and D.H. Lawrie, "High-Speed Multiprocessors and Compilation Techniques," *IEEE Trans. on Comp.*, p763-776, Sept. 1980.

[RAP86] Rap M.D. and R.S. Tetrick, "P1296: The Interprocessor Communication Standard," *IEEE Micro*, p72-77, june 1986.

[RAMA87] Ramachandran U., M. Solomon and M. Vernon, "Hardware Support for Interprocess Communication," *Proc. of the ACM on Comp. Arch.*, p178-188, 1987.

[RITC84] Ritchie D.M., "A Stream Input Output System," *AT&T Bell Laboratories Techn. Journal*, Vol 63, No 8, Part 2, p1897-1910, Oct. 1984.

[RUSS87] Russell C.H. and P.J. Waterman, "Variations on Unix for Parallel-Processing Computers," *Comm. of the ACM*, Vol 30, No 12, p1048-1055, Dec. 1987.

[SATY80] Satyanarayanan M., *Multiprocessors; A Comparative Study*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1980.

[SCHR76] Schrott G., "Common Elementary Synchronization Functions for an Operating System Kernel," *European Workshop on Industrial Computer Systems*, OP/SYS II-3-1, Jan. 1976.

[SHAR87] Sharp J.A., *An Introduction to Distributed and Parallel Processing*, Blackwell Scientific Publications; Oxford UK, 1987.

[SMIT88] Smith J.M., "A Survey of Process Migration Mechanisms," *ACM Operating Systems Review* 22 (3), p28-40, July 1988.

[STRA86] Straathof J.H., A.K. Thareja and A.K. Agrawala, "UNIX Scheduling for Large Systems," *Proc. of the USENIX Techn. Conf.*, Denver, Colorado, p111-139, Winter 1986.

[TANE81] Tanenbaum A.S., *Computer Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[TEST86] Test J.A., "Multi-Processor Management in the Concentrix Operating System," *Proc. of the USENIX Techn. Conf.*, Denver, Colorado, p173-182, Winter 1986.

[TIMM87] Timmerman M., "The VMEbus User Handbook," regular chapter in: *VMEbus Applications*, European VMEbus Usergroups, since 1987.

[VERS87] Verschueren A., *A Coprocessor for Hardware Multitasking Support*, M.Sc. Thesis, Digital Systems Division, Department of Electrical Engineering, Eindhoven University of Technology, Netherlands, 1983.

[VMS83] *VMSbus (VME Serial bus) Specification Manual (Preliminary)* Revision A4, VMEbus Manufacturers' Group, Nov. 1983.

[WODO72] Wodon P.L., *Still Another Tool for Synchronizing Cooperating Processes*, Report, AD-750538, Carnegie-Mellon University, Dept. Comp. Science, Aug. 1972.

[XOPE87a] The X/OPEN Group Members, *X/OPEN Portability Guide, "The Common Applications Environment"*, p2.5, 1987.

[XOPE87b] The X/OPEN Group Members, *X/OPEN Portability Guide, "XVS Inter-Process Communication"*, p1.1., 1987.

Index

Application Layer 3
Architecture . . . . i, iii, iv, 1, 3, 10, 12, 31, 36-39, 41, 61, 63, 69, 121, 127, 142
    Master-Slaves . . . . i, iii, 1, 10, 15, 36-39, 41, 57, 61, 62, 69, 123
    Multi-Kernel . . . . i, iv, 1, 36, 37, 61, 63, 69, 121, 123, 127
    Uni-Kernel . . . . i, iv, 1, 36, 61-63, 69, 123
Bandwidth 9, 29, 93
Cache 7, 18, 31, 37, 54, 56, 94, 121, 138
Critical Section 35, 71-74, 78, 79, 115, 116, 120
    IH-IH 35, 69
    P-IH 35, 39, 56, 63, 64, 66, 67, 69, 78
    P-P 35, 66, 67, 69
Distributed Computing 3, 43, 124
Economic 5
Error Recovery . . . . iii, 6, 12, 13, 17, 19, 42, 56, 138
Fault Tolerance 10, 29, 61, 87
IEEE 20, 24, 44, 141, 142
    P1014 . . . . 20, 142
    P1296 . . . . iv, 44, 53, 56, 142
Inter Process Communication 12, 16, 32, 42, 128
    IPC . . . . iv, 12, 30, 32, 42, 44, 45, 49, 50, 56, 78, 128
Inter Processor Communication . . . . i, iii, 9-13, 22, 24, 32, 37, 40, 42, 53, 128, 137
    Explicit . . . . iii, 40, 43, 44, 54, 56
    Implicit . . . . iv, 40, 43, 54, 56, 125
    IPrC. . . . . iii, iv, 11-13, 17, 24, 28, 29, 32, 40, 42-45, 47, 52-54, 56, 58, 68, 70
        93, 125, 126, 128
Interconnection
    Circuit-Switching 7
    Crossbar 8, 9, 136
    Message-Switching 7
    Omega Network 75, 119
    Packet-Switching 7
    Time Division Multiplex 8, 129
Interrupt . . . . . iii, iv, 9-11, 13, 14, 21, 22, 25, 27, 28, 32, 34, 35, 38-41, 51, 53, 54, 56, 59
        62-69, 77, 78, 101, 104, 125, 128, 136, 137
    Clock . . . . iii, iv, 38, 40, 41, 65-68, 101, 104, 125
ISO-OSI 3, 56, 126
    Application Layer 3
    Network Layer 3
Kernel Mode 34-36, 38, 39, 57, 61-63, 66, 67, 69, 78
Load Balancing . . . . i, 4, 12, 87, 88, 90, 91, 94, 103, 138, 140, 141
Memory Map . . . . iii, 6, 14, 19, 23, 29, 30, 136
Miniaturization 3
Multiprocessing 4, 141
Multiprocessori, iii, iv, 1, 4-9, 11, 12, 14-20, 26, 29, 30, 32-37, 39, 40, 42, 43, 45, 52, 57, 59,
    61, 65, 66, 69, 70, 80, 85, 87, 88, 91, 93, 94, 98, 99, 103, 104, 120, 123-127, 140, 141
    tightly coupled i
Multiprocessoring . . . . iii, 4

Multiprogramming 4, 18, 73
Mutual Exclusion . . . i, iii, iv, 13-15, 33, 35-39, 41, 45, 61, 69, 73, 123, 124, 126, 127, 137
Network Computing 3
Operating System i, iii, 4, 6, 7, 10, 13-17, 19, 20, 23, 26, 30, 32-37, 40, 45, 54, 56, 57, 72, 86, 93, 102, 121, 124, 126, 127, 137, 139-143
Optical 3, 9
Parallel Processing . . . . . i, iii, 5, 6, 16, 19, 138, 140-142
Performance . iii, 1, 3, 6, 10, 12, 15, 17-19, 31, 59, 63, 87, 90, 94, 115, 120, 125, 138, 142
Pipelining 3, 75
Process Management 105, 114, 128, 140
Process Migration . . . . . i, 85-87, 103, 121, 124, 125, 143
PUX 16
Scheduling 1, 15, 16, 18, 37, 54, 56, 65, 66, 69, 72, 80-85, 87, 88, 92, 94, 95, 98-105, 120, 121, 124, 125, 127, 138, 142, 143
Simulation . . . . . i, ii, iv, v, 1, 88, 89, 92, 98, 105, 108, 114, 115, 128, 132-135, 141
Stack 30, 34, 59, 87, 122
Supervisor Mode 15, 34, 35, 38
Symmetry . . . . . iii, 6, 9, 10, 19, 32, 57, 61, 136
Synchronization
    Barrier 76
    Compare&Swap 71, 116-118, 128
    Fetch&Add 70-75, 77, 115-120, 124, 125, 127, 128, 144
    Fetch&Operation 71, 116, 118
    Mechanisms . . . . . iv, 69, 70
    Semaphore 26-28, 38, 44, 45, 51, 52, 62, 66, 73, 76, 78, 79, 138
    Test&Set 51, 64, 65, 70, 71, 115-118, 120, 129
    UP/DOWN 76
System Startup . . . . . i, iii, iv, 12, 33, 34, 37, 42, 56, 57, 83, 138
Technology 1, 3, 119, 127, 140, 143
Uniprocessor 6, 12, 15-18, 20, 26, 38, 39, 42, 51, 57, 69, 79, 80, 85, 88, 91, 104, 108, 136
Unix iv, 1, 20, 30, 32-38, 41, 42, 44-47, 49, 50, 52, 54, 56, 57, 63, 69, 76-80, 83, 86, 92-94, 99-102, 104, 105, 107, 120, 121, 123, 124, 126-129, 140-144
    Multiprocessor 56, 57, 67, 104, 128
    Single Processor 41, 50, 54, 56-58, 65-67, 79, 104, 129
    System V . . iii, iv, 20, 30, 32, 42-45, 50, 59, 65, 78, 101, 102, 104, 123, 139, 144
    Version 6 . . . . . 33, 102
    Version 7 . . . . . 20, 102, 105, 120, 144
User Mode 15, 35, 38, 56, 61, 62, 67, 100-102
VME. 20-26, 29, 32, 33, 37, 52, 53, 56, 57, 61-63, 69, 115-122, 125-127, 129-131, 137, 139, 141, 143
    Arbitration . . . . . iii, 8, 9, 21, 25, 27
    DTB . . . . . iii, 21, 22, 25, 26, 128
    Interrupt 32
    Utility Bus . . . . . iii, 21, 22, 26
VMS . . . . . iii, iv, 21, 26-29, 44, 52, 62, 63, 129, 143
VSB 21, 29, 129