TR 90-65

A STUDY OF REAL-TIME OPERATING SYSTEMS FOR MICROCOMPUTERS

THESIS
Submitted in Fulfilment of the
Requirements for the Degree of
MASTER OF SCIENCE
of Rhodes University

by

GEORGE CLIFFORD WELLS

January 1990

Table of Contents

1. Introduction . . . 1
   - 1.1. Terminology . . . 1
   - 1.2. Criteria for the Evaluation . . . 2
   - 1.3. The Hardware Configuration . . . 3
   - 1.4. An Overview of the Operating Systems . . . 4

2. The Evaluation Process . . . 6
   - 2.1. The Benchmarks . . . 6
   - 2.2. The Simulation . . . 11
   - 2.3. Analysis of the Results . . . 20

3. XENIX System V . . . 22
   - 3.1. Introduction . . . 22
   - 3.2. The Benchmarks . . . 25
   - 3.3. The Process Control System and Plant Simulation . . . 32
   - 3.4. Qualitative Assessment . . . 33
   - 3.5. Conclusions . . . 34

4. OS/2 . . . 35
   - 4.1. Introduction . . . 35
   - 4.2. The Benchmarks . . . 39
   - 4.3. The Process Control System and Plant Simulation . . . 43
   - 4.4. Qualitative Assessment . . . 44
   - 4.5. Conclusions . . . 45

5. QNX . . . 47
   - 5.1. Introduction . . . 47
   - 5.2. The Benchmarks . . . 49
   - 5.3. The Process Control System and Plant Simulation . . . 53
   - 5.4. Qualitative Assessment . . . 54
   - 5.5. Conclusions . . . 55

6. FlexOS . . . 56
   - 6.1. Introduction . . . 56
   - 6.2. The Benchmarks . . . 57
   - 6.3. The Process Control System and Plant Simulation . . . 61
   - 6.4. Qualitative Assessment . . . 62
   - 6.5. Conclusions . . . 63

7. Comparison of the Operating Systems . . . 64
7.1. The Benchmarks . . . 65
7.2. The Process Control System and Plant Simulation . . . 72
7.3. Qualitative Comparison . . . 72
7.4. Discussion . . . 74

8. Conclusions . . . 76

9. References . . . 78

Appendices

A. Algorithms for Benchmark Programs . . . 81
A.1. Introduction . . . 81
A.2. Data Types and System Calls Used in Algorithms . . . 82
A.3. Concurrency . . . 84
A.4. Interprocess Communication . . . 87
A.5. Synchronisation . . . 92
A.6. Exception Handling . . . 93

B. Results of Benchmarks . . . 96
B.1. XENIX System V . . . 96
B.2. OS/2 . . . 104
B.3. QNX . . . 117
B.4. FlexOS . . . 123

List of Figures

1. Widget Manufacturing Plant 12
2. Context Schema - User's View 13
3. Context Schema - Process Control System View 13
4. Context Schema - Plant Simulation View 14
5. Detailed Context Schema - Process Control Subsystem 15
6. Poor Grouping of Transformations (Not According to Simulated Structures) 16
7. Good Grouping of Transformations (According to Simulated Structures) 17
8. Poor Choice of Transformations (Non-Subject Matter Based) 17
9. Good Choice of Transformations (Subject Matter Related Naming) 18
10. Behavioural Model 19
11. Example Distribution 20
12. Process Creation Times (XENIX System V) 27
13. Message Passing - Shared Memory (XENIX System V) 29
14. Interrupt Handling Times (XENIX System V) 31
15. Implementation Model (XENIX) 32
16. Implementation Model (OS/2) 44
17. Frequency Distribution with Ten Background Processes 52
18. Implementation Model (QNX) 54
19. Implementation Model (FlexOS) 62
20. Sieve of Eratosthenes (Compiler Efficiency) 64
21. Multiprocessing Overhead (Disk Operation) 66
22. Process Creation - Memory Operation 67
23. Process Creation - Disk Operation 67
24. Interprocess Communication (100 Byte Messages) 68
25. Interprocess Communication (Circular Messages) 69
26. Synchronisation 71

Acknowledgments

I would like to thank my supervisors, Peter Clayton and Pat Terry for their guidance and encouragement.

Without the financial support and donations of equipment and software from AECI Process Computing this project could never have been undertaken. Many thanks are due to Tony Heher and John Ballinger of APC, in particular, for their time and help.

I am also most grateful for the financial support of the Council of Rhodes University.

Thanks are also due to my colleagues and fellow Master's students for making Rhodes such a pleasant place to work and study.

Finally, my appreciation of my family and friends for their love, support and encouragement cannot really be expressed in words, but thank you, all the same.

Abstract

This thesis describes the evaluation of four operating systems for microcomputers. The emphasis of the study is on the suitability of the operating systems for use in real-time applications, such as process control. The evaluation was performed in two sections. The first section was a quantitative assessment of the performance of the real-time features of the operating system. This was performed using benchmarks. The criteria for the benchmarks and their design are discussed. The second section was a qualitative assessment of the suitability of the operating systems for the development and implementation of real-time systems. This was assessed through the implementation of a small simulation of a manufacturing process and its associated control system. The simulation was designed using the Ward and Mellor real-time design method which was extended to handle the special case of a real-time simulation.

The operating systems which were selected for the study covered a spectrum from general purpose operating systems to small, specialised real-time operating systems. From the quantitative assessment it emerged that QNX (from Quantum Software Systems) had the best overall performance. Qualitatively, UNIX was found to offer the best system development environment, but it does not have the performance and the characteristics required for real-time applications. This suggests that versions of UNIX that are adapted for real-time applications are worth careful consideration for use both as development systems and implementation systems.

1. Introduction

This study is an evaluation of operating systems for microcomputers, with special regard to the suitability of the operating systems for real-time processing. The project was motivated by the increasing capabilities of microcomputers which have resulted in their utilisation in areas such as process control which were once the domain of minicomputers or mainframes. With this increase in the power of the hardware has come an increase in the capabilities and performance of the operating systems for microcomputers. There are now many such operating systems available, and a survey such as this would be intractable if all the possible candidates were to be considered. Accordingly, this study concentrates on four operating systems, namely UNIX System V, Microsoft Operating System/2 (better known as OS/2), QNX and FlexOS, which are representative of the spectrum of systems available. Each of these systems is discussed in more detail in section 1.4.

The rest of this introduction covers the terminology used in this document, the criteria used for the evaluation and the configuration of the hardware that was used. The next chapter discusses the evaluation procedure in some detail, outlining the tests that were performed. This leads into the following chapters which discuss the results of the assessment of each of the operating systems in detail. The systems are then compared side by side in chapter seven, and the last chapter presents the conclusions of the evaluation.

1.1. Terminology

As in many areas of Computer Science, real-time programming has certain terms that are used very often, and which it is important to define at the outset, as various authors may use different terminology. Accordingly this section gives a few definitions which will be used in this report.

Process: A process is a program that is eligible for execution by the hardware of a computer. It may be located in primary or secondary storage, and may be suspended by the operating system (for example, because it requires some resource currently being used by another process, such as the CPU, or an area of shared memory).

Task: This term is used as a synonym for the term **process**. In particular, QNX uses the term task rather than process, and this convention has been adopted in the discussion of QNX.

System Call: This term is used to describe the interface between a program written in a high level language (such as C) and the operating system. This interface is usually provided by means of what appear to be function or procedure calls, but which access the operating system, either directly or indirectly.

Supervisor Call (SVC): This term is used as a synonym for the term **system call**. In particular, FlexOS uses the term supervisor call rather than system call, and this

convention has been adopted for the discussion of FlexOS.

1.2. Criteria for the Evaluation

The survey of the four operating systems named above has concentrated on those facets which are relevant to real-time applications. This raises the issue of what is meant by the term "real-time", and what constitutes a real-time operating system. The following definition by Young [1982, p15] serves as a useful starting point for discussion.

> *The term "real time" can be used to describe any information processing activity or system which has to respond to externally generated input stimuli within a finite and specifiable delay.*

This brings to light what is probably the fundamental factor in real-time systems, namely **time**. Hehner [1989] makes the point that almost all computer systems operate within some sort of time constraints, but that the consequences of not reacting within a given time limit are greater for a real-time system. Thus, real-time systems need to be able to respond rapidly and predictably to external events, and this has obvious implications for the operating systems used. Such operating systems must allow for external events to be monitored (that is, external both to a given process or task, and to the computer system). The usual way in which this is achieved is through the use of multitasking. When an external event occurs the currently executing task is suspended, and another task invoked to deal with the event. In a multiprocessor environment a processor could be dedicated to monitoring a specific event, and so there would be no need for task switching. A task invoked (or resumed) to handle an event will usually need to inform other tasks of the fact that the event has occurred. Very often there is also the need to share some form of database between the tasks as well. In addition, the provision of networking facilities to allow for cooperation between multiple computers is of increasing importance. This is especially true in the case of microcomputers, where it may be necessary to use several processors where a single minicomputer or mainframe would suffice.

Furht *et al* [1989] also discuss the criteria to be used in judging a real-time operating system. They maintain that the two categories that are important are **performance** and **determinism**. Both of these factors are related to the subject of time discussed above: performance being a measure of the time taken to perform a particular task, and determinism being a reflection of the predictability of the time which it takes for the operating system to react to an event.

These considerations lead to the conclusion that a real-time operating system must support both **concurrency** (either genuine concurrency using multiple processors, or pseudo-concurrency using time-slicing) and **interprocess communication** (IPC), and that these facilities must be both efficient and predictable in their performance. Two important features of concurrency that must be present are mechanisms which allow for tasks to be created and controlled by other tasks. In order to support the existence of shared data it also becomes necessary to provide facilities for mutual exclusion and synchronisation of processes.

With these factors in mind, the evaluation of the operating systems concentrated on the efficiency

and determinism of the concurrency and interprocess communication facilities of the operating systems. The assessment was divided into four sections: the first focused on the creation and use of multiple processes, the second on interprocess communication, the third on synchronisation, and the last on interrupt (or exception) handling. These areas are each discussed in more detail in section 2.1. In addition to these factors which can be measured quantitatively, it was felt that a qualitative appraisal of the operating systems would also form a useful part of the assessment, and, in particular, would provide an evaluation of their suitability for use as system development environments. In order to meet this criterion a simulation of a manufacturing plant and its associated process control system was implemented under each of the operating systems. This provided the experience of developing a complete system that was much larger than the benchmark programs. The design of the simulation and control system was undertaken using the Ward and Mellor approach to the design of real-time systems [Ward and Mellor, 1985, 1986]. During the course of the design several extensions were made to the basic Ward and Mellor method, and this formed an important subtopic within the study.

1.3. The Hardware Configuration

As early as 1978 people were beginning to realise that microprocessors could successfully be used for real-time applications [Musstopf, 1979]. Prior to this, real-time systems had largely been the domain of minicomputers, such as the PDP-11. Currently, there is a large range of microprocessors available, and their performance has improved such that they are widely used for process control and other such real-time applications ([Bunnel and Bunnel, 1989], [Stewart, 1987], [Bemmerl and Schöder, 1987], [van Zyl, 1986] [Mackay, 1986], [Laboratory Technologies Corp., 1985] and [Weiss, 1985]). With the increasing interest in the use of microprocessors for real-time applications, it became apparent that they have considerable price-performance advantages over minicomputers.

In order to standardise the evaluation of the operating systems as far as possible, they were all evaluated using essentially the same hardware configuration. The first step was to select the processor technology that was to be used. In view of the large-scale acceptance of the Intel 80x86 family of processors it was decided that the 80286 would form a suitable platform for the research. The Motorola 680x0 range of processors was also considered at one stage, but it lacks the standardised hardware environment that the IBM-PC has brought to the Intel range. The 80386 was also considered, but it was still a relatively new development when the study commenced and there was not a lot of software and information available for it. Accordingly, the hardware consisted of an IBM PC-AT compatible computer (a SAPEC 286).

The 80286 processor is a 32 bit microprocessor designed as a successor to the 8086 [Wells, 1984]. It provides sophisticated hardware support for operating systems, such as virtual memory management, task switching and protection. One of the main features of the 80286 is that it operates in one of two modes, called **real** and **protected** modes. In real mode the processor acts as a high speed 8086 chip, with a one megabyte address space (20 bit addressing). In protected mode the 80286 supports a physical address space of 16 megabytes (24 bit addressing) and a

virtual address space of one gigabyte (32 bit addressing). The protection facilities of the 80286 offer data-type checking, system software isolation (using a four level, ring-type system) and task isolation. A floating point arithmetic coprocessor is available for the 80286 (the 80287) but was not used for the benchmarks. The use of a floating point unit would probably have very little effect on the results of the tests, as very little real arithmetic was employed.

The 80386 and 80486 processors are the successors of the 80286. These are both supersets of the 80286 architecture. The 80386 expands the capabilities of the 80286 to provide full 32 bit functionality, together with cleaner support for the 8086 modes of operation. The 80486 essentially combines certain support functions (such as floating point capability) with the central processing functions of the 80386, on a single chip. Both of these processors would provide an equally good basis for an evaluation such as this. In fact, several of the operating systems looked at in this study have been extended to provide versions that are tailored to make full use of the facilities of these newer processors.

The clock speed of the computer used for the tests was 10MHz. The computer was equipped with one and a half megabytes of memory. This was used with one wait state. For the evaluation of XENIX, OS/2 and QNX a 42.8 megabyte hard disk was used for those tests that required disk access. This had a data transfer rate of 160.6 kilobytes per second, and an average seek time of 35.2ms. As funding did not allow the purchase of a copy of FlexOS, the evaluation of FlexOS was performed during a visit to AECI Process Computing. The computer which was made available for this evaluation was identical to that used for the other operating systems, except for the hard disk. This was a 21.4 megabyte hard disk with a data transfer rate of 160.5 kilobytes per second, and an average seek time of 70.7ms.

1.4. An Overview of the Operating Systems

The choice of operating systems for the study represents a cross section of the spectrum of available systems from general purpose operating systems to real-time operating systems. The inclusion of general purpose operating systems is due to the fact that, very often, they are easier to use than the more specialised real-time operating systems. McQuaid [1985] makes the point that many users may have to work with operating systems such as MS-DOS, CP/M and UNIX, which may not even have facilities for multitasking. Specialised operating systems are sold within a smaller market than general purpose systems and so may be prohibitively expensive in some situations.

One of the problems with undertaking a study such as this is that, for commercial reasons, much of the detailed information about the operating systems is unpublished. This can make it difficult to draw conclusions about the reasons for the performance of the operating systems. For example, one operating system may do something exceptionally well compared with the others. Without any recourse to the code for the operating systems, or even detailed algorithms or explanations, it is almost impossible to state why there is a large difference in the performance of the systems in that area.

The programming language which was used for the programs written for the purposes of the evaluation was C.

UNIX is a general purpose operating system, and its designers did not intend it for real-time use [Ritchie and Thompson, 1974]. UNIX is both multi-user and multitasking, and dates originally from the early 1970's, although the System V version dates from 1983. The version of UNIX evaluated was XENIX System version 2.2.1 from the Santa Cruz Operation Inc. (SCO). SCO XENIX System V complies with the AT&T System V Interface Definition (the SVID, 1985) with a few minor exceptions, mainly due to the architecture of the Intel 80286 processor, which it utilises in protected mode. XENIX System V is also available for the 8086 processor (IBM XT), the 80386 processor, the AT&T 6300+ computer and the Hewlett-Packard Vectra computer (both 80286-based machines). XENIX was provided with its own C compiler. It would appear from the XENIX documentation that this compiler is a port of the Microsoft C compiler for MS-DOS.

Operating System/2 (OS/2) is also a general purpose system but has a mode of operation which is suited to real-time processing. It is the most recent of the systems considered in this evaluation as it was released in 1987. It was developed jointly by IBM and Microsoft. It is a multitasking operating system, but it is not a multi-user system. It is available for 80286 and 80386 processors. It uses the 80286 processor in protected mode, but does not use the extended capabilities of the 80386 processor at all. The version used for the evaluation was version 1.00. OS/2 is not supplied with a compiler, and so the Microsoft C compiler (version 5.1) was used for the testing procedure.

QNX is a relatively modern system, dating from 1982. It is a multi-user, multitasking real-time operating system. It was developed by Quantum Software Systems Ltd. of Ontario, Canada. The design of QNX is based on message passing. The version used for the evaluation was 3.05 release 5B. This version utilises the 80286 processor in protected mode. Versions of QNX are available for the 8086 processor (IBM PC or XT) and for the 80286 processor in real mode. QNX was supplied with its own C compiler.

Together with QNX, FlexOS represents the high end of the scale, and is also a multi-user, multitasking real-time operating system. It is a relatively new system, first released in 1986. It was developed by Digital Research, and there are versions available for the 80186, 80286, 80386 and 680x0 processors. One of the main strengths of FlexOS is that the file system used is completely compatible with the MS-DOS file system, and many of the operating system utilities work in the same way and have the same names as their MS-DOS counterparts. The version used for the evaluation was FlexOS 286 release 1.42. FlexOS was supplied with the MetaWare High C compiler. This is produced by MetaWare (a third party company), but is marketed by Digital Research as part of the FlexOS Developer Kit.

2. The Evaluation Process

There were two main aspects to the assessment of the operating systems. The first involved the development of a set of benchmarks that could be implemented on each of the operating systems in order to obtain quantitative results of the performance of the real-time facilities. The second aspect was to perform a more qualitative appraisal of the operating systems. This involved the use of the development facilities of the operating systems to implement a real-time simulation of a process control system. This was aimed at evaluating the suitability of each of the systems as a development environment, as this is an important factor in assessing the practical effectiveness of the systems. The last section in this chapter discusses how the results were analysed.

2.1. The Benchmarks

As discussed in the introduction, the benchmarks concentrated on those aspects of the operating systems that are relevant to real-time systems. These fall into four categories, namely concurrency, interprocess communication, synchronisation and exception or interrupt handling. The choice of the benchmark programs was one of the more difficult aspects of the assessment of the operating systems. Atkinson [1986] discusses some aspects of benchmarking multiprocessor systems that have some relevance to multitasking, single processor systems. He proposes four benchmarks: a **message passing** benchmark (communication between two processes); a **double buffer** benchmark (pipelined applications, such as Conway's Problem [Conway, 1963]); a **multiplex** benchmark (combining two communication channels into one); and an **interrupt** benchmark (using timer interrupts to assess the service time and latency of interrupts). These benchmarks are rather concentrated on the area of interprocess communication in various forms, presumably since this is often a bottle neck in the performance of multiprocessor systems. Kar and Porter [1989] discuss some of the problems inherent in benchmarking real-time systems. This they take to mean the combination of the hardware and software that is under consideration. They then identify six areas of performance that are important for real-time performance: **task switching** time, **preemption** time, **interrupt latency** time, **semaphore shuffling** time, **deadlock breaking** time and **datagram throughput** time. This is a slightly lower level approach than that which was taken by Atkinson, and that which was adopted for this study. This is reflected by the fact that specialised tools (such as in-circuit emulators, statistical profilers, or simulators) are required to evaluate performance using their approach.

The four categories of benchmark that were selected for this evaluation (concurrency, interprocess communication, synchronisation and exception handling) allow the performance of the operating systems to be measured without the use of sophisticated tools. In addition, they provide a more general assessment of the performance of the operating systems than the benchmarks suggested by Atkinson, by covering areas such as synchronisation and concurrency.

Aburto [1988] discusses some of the problems that arise in performing benchmarks (dealing specifically with single tasks, but the points he raises are just as applicable to multitasking

situations). The first point he raises is that the environment in which the benchmarks are run needs to be kept as constant as possible. He highlights the rôle played by compilers, optimisation, cache memories, buffer sizes, numbers of buffers, *et cetera.* He also warns against benchmark programs that can be drastically altered by modern optimising compilers, and cites some benchmarks that can be optimised to an empty program. Vose and Weil [1989] also make the point that benchmarks are often very small programs and may not give an accurate reflection of the performance of the system when running full sized applications.

In order to keep the environment as consistent as possible the operating system benchmarks were run using the same hardware configuration wherever possible (see section 1.3). However, the operating systems considered in this study all present very different software environments. In order to standardise on the approach taken to this aspect it was decided that the systems would be used in as standard a way as possible. This meant that no system constants (such as buffer sizes) were altered, but the default values were used. Similarly, no compiler or linker options were set. Some exceptions had to made to these rules in order to run some of the benchmark programs. For example, the size of the process table under XENIX System V had to be increased to allow the process creation benchmark to be run. The decision to use the compilers in their default configuration (that is, with no command line options set) resulted from the facts that (1) this meant that the compilers would be operating in the state intended by their creators, and (2) the benchmarks themselves would be relatively immune to the optimisation effects that plague most benchmarking procedures. The reason for this last fact is that the benchmark programs are typically small loops calling on just the system facilities that are being examined in each case. This means that the compiler should not perform any of the common optimisations (such as deletion of redundant code, removal of common subexpressions, in-line implementation of functions, *et cetera*) as it generally cannot be sure of the effect of the system call. For example, in OS/2 a call of the form `DosSleep(0)` may appear to be redundant, but can in fact cause the operating system to schedule another process.

In all cases the operating systems were run in as favourable a state as possible. This meant that only essential system tasks were enabled and that the multi-user capabilities were disabled where appropriate. In order to time the tests as accurately as possible the computer's hardware clock was used. This raised the issue of the resolution used by each of the operating systems for reporting the system time. For example, XENIX has a resolution of 20ms and QNX has a resolution of 50ms. Another factor is the overhead introduced by the calls to the system timer in order to measure the start and end time of each of the tests. This is difficult to assess, since in measuring the overhead of the timer calls one would have to use the timer calls themselves. The approach that has been taken is to repeatedly exercise a particular function of the operating system until the time being measured is significantly greater than the expected overhead for the timer calls. For this reason, the smallest intervals measured were generally of the order of a few seconds. Since the overhead due to the timer calls would be of the order of a few milliseconds or tens of milliseconds (certainly less than the resolution of the system timers used), it would then, clearly, be of little significance. In order to allow for transient events (such as the operating systems flushing internal buffers periodically) each of the benchmarks was repeated ten times. The treatment of this information is discussed in section 2.3. In presenting the results, the time taken per operation is presented for each of the operating system features that

was studied. Smith [1988] makes the point that this is a less ambiguous method of presenting the results of benchmarks than generating an overall value for the series of tests, and expressed as a rate (rather than as a time per operation).

2.1.1. Concurrency

The benchmarks performed in this category assessed the efficiency of the task switching performed by the operating system, and the time taken to create a new task. In addition, the effects of background processing on the benchmarks in some of the other categories have been measured.

The first benchmark was aimed at evaluating the efficiency of the multitasking performed by the operating system. It consisted of the ubiquitous Sieve of Eratosthenes which was executed singly, then in parallel with one other instance of itself, and then in parallel with nine other instances of itself. Analysis of these results allowed for the overhead due to process switching to be assessed. Aburto discusses the use of the Sieve of Eratosthenes as a benchmark in his article on the problems of benchmarking [1988]. He makes the point that the Sieve is a good benchmark in that it is simple and yet it does something useful and verifiable (this prevents optimising compilers from deleting large sections of useless code). However, the Sieve is sensitive to the use of register variables (an optimising technique used by some compilers) and to the size of the variables used (which may differ from compiler to compiler). The use of the Sieve of Eratosthenes for this study is useful on two counts. Firstly, it gives a fairly good indication of the efficiency of the compiler and the hardware, and hence, as the hardware configuration has been kept constant, allows the compilers used with the different operating systems to be compared. The second advantage presented by the Sieve of Eratosthenes is that it is relatively independent of the operating system, as it makes no use of system calls. This makes it very useful as a standard background process, since it is guaranteed not to interact at all with the benchmark programs.

The process creation benchmark assessed the time taken to create a new child task. In order to standardise this, the child task in all cases was an null process that terminated as soon as it was created. In order to obtain measurable results this test was performed by creating 100 child tasks and measuring the entire time taken. This test was performed for creation directly from memory and for creation from a disk file, wherever applicable. This also gives an indication of the relative efficiency of the file access facilities of the systems.

Finally, the appraisal of the effects of background processing on the other benchmarks was performed by executing a number (either one or ten) of instances of the Sieve of Eratosthenes in the background. This was repeated for priority values that were equal, higher and lower than that of the process being tested. This brought to light some interesting phenomena. For example, a common finding was that synchronisation suffered a severe degradation in the presence of such background processing. It must be noted that the use of the Sieve of Eratosthenes probably represents the worst case in this type of testing, as the program is extremely computationally intensive and makes no use of system calls or I/O routines (as

mentioned above). This means that it will fully utilise the entire duration of any time slice allocated to it. In general, processes will tend to perform system calls and I/O operations, thus often causing the operating system to suspend them before a complete time slice is utilised.

2.1.2. Interprocess Communication

There were two tests performed in this category. The first utilised two processes - a sender and a receiver. The receiver simply accepted the messages (which contained 100 bytes of information) and discarded them, acknowledging the receipt where necessary. The test was repeated for one, ten, one hundred and one thousand messages, and with and without background processing (as discussed in section 2.1.1). In addition, tests were run for 10 000, 20 000, 30 000 and 40 000 messages in order to assess whether the time taken by the message passing facilities followed a linear pattern as the number of messages increased. Where an operating system offered several alternative message passing mechanisms (for example, XENIX offered pipes, asynchronous messages and shared memory segments) the tests were performed for all of them.

The second test utilised three processes, connected in a circular fashion and using messages with one byte of information. The first task sent the first message to the second task, which acknowledged it if necessary, then passed the message on to the third task. This again performed any acknowledgment required then passed the message back to the first task. The first task then sent the second message, and so on. This test was performed for multiples of 26 messages (the information sent in each message was one of the letters A to Z). The main series of tests and the effects of background processing were measured for 100 iterations of this process (that is, 2600 messages). Further tests were performed for 50, 150, 200, 250 and 300 iterations in order to assess the linearity of the behaviour of the message passing facility. Of course, this test also provides an indirect measure of the efficiency of the process switching performed by the operating system, as each process sends a message and is then suspended waiting for a message to be received.

2.1.3. Synchronisation

This test made use of two processes. The first repeatedly wrote the string "Hi" to the screen of the computer and the second repeatedly wrote "Ho". This was done 1000 times. To measure the effects of the synchronisation facilities the processes were first run without synchronisation. This resulted in output of the form "HiHiHi...HoHoHo...HiHiHi...HoHoHo...". The processes were then executed with synchronisation, producing the output "HiHoHiHoHiHoHiHo...". The degradation in the execution time between the two versions allowed the time taken for the synchronisation to be measured. This was repeated in the presence of background tasks, as described in section 2.1.1. A comparison of the times taken for the different systems also allows some assessment of the efficiency of the screen handling facilities, although this is not of direct interest.

2.1.4. Exception or Interrupt Handling

This category of tests did not assess the use of low level (hardware) interrupts as provided by the 80286 processor, but concentrated on the high level facilities made available by the operating systems to resume suspended processes. Most of the systems studied had provision for hardware interrupts to generate an interrupt/exception, if necessary. However, it is difficult to design a generalised benchmark for this type of feature. The approach taken here allowed the calculation of the time taken by the operating systems to resume a suspended task in response to some external stimulus (that is, a stimulus generated either by another task or the operating system itself).

The features found in the operating systems fell into two categories: facilities to voluntarily suspend a task for a specified period, whereafter the operating system would resume the task, and facilities for a task to send some form of signal to a suspended task to resume it. In all cases the action taken by the resumed task was simply to reenable the interrupt handling mechanism (if necessary) and then suspend itself again. As this would, in general, require only a very short amount of time it would have been difficult, if not impossible, to measure it directly. In addition, direct measurement would have made it impossible to draw meaningful comparisons between the systems, as the system overhead due to the timing calls would vary from one operating system to the next, and, as the interval being measured is very small, this overhead would have become appreciable. Accordingly, an indirect approach was adopted. This involved a timer process (the Sieve of Eratosthenes) which executed in an infinite loop, reporting on the time taken for a given number of iterations of the Sieve algorithm. This was initially performed alone. The interrupt handling task was then started and the degradation in the processing time for the Sieve was measured. This allowed the interrupt service time to be assessed. Naturally, this includes the task switching time as well as the time taken to generate and handle the interrupt. This was repeated for various intervals between interrupts, where applicable.

One complication was in assessing the number of interrupts that occurred during the iteration of the Sieve task. One approach might have been to get the interrupt handler to count these, but this would then have required extra synchronisation and communication between the timer process (the Sieve of Eratosthenes) and the interrupt handling process or processes. It would have been difficult, if not impossible, to account for this extra overhead. Accordingly, the number of interrupts was *estimated* from the time taken by the iteration of the Sieve of Eratosthenes and the interrupt interval. This can only be an estimation, since the operating systems cannot guarantee that the various timer facilities offered by them will operate exactly within the specified interval (especially when this interval is of the same order as the system clock tick interval, or possibly even less than the system clock tick interval). In analysing the results of this test, it became apparent that the standard error in the calculated values was extremely large. This is due to the fact that it was the difference between two readings that was used to calculate the result. This difference is of the same order as the standard deviation in the two measured values themselves and so the overall effect is to produce results that are subject to a great deal of uncertainty. In view of this, and the fact that the number of interrupts can only be estimated and not known exactly, the results found in this section are extremely

unreliable. They have been included to give a general indication of the performance of the operating systems in this area. In particular, as all the measurements are subject to the same uncertainty, the results may have some value as a relative indication of the efficiency of the operating systems.

2.2. The Simulation

The simulation consists of a widget manufacturing plant and its associated process control system. In designing the simulation, cognisance was taken of the same criteria that were used for the choice of benchmarks. Thus, the first criterion was that the simulation would have to make use of several tasks. This then implied the need for interprocess communication. To test the facilities available for protecting critical resources and process synchronisation it was desirable to have some form of critical region or resource. Exception handling was also included in the simulation. In addition to these fundamental areas, there are several more mundane factors that need to be assessed. These include facilities for efficient disk I/O, calculations, data logging and the presentation of data on a terminal and the acceptance of operator commands.

During the course of this study a suggestion was made that an overall performance index be calculated by implementing a background task at a low priority and measuring the amount of processor time that it received. This was done for XENIX and for OS/2. The approach was to implement a low priority, performance monitoring task which performed the Sieve of Eratosthenes. The time taken for each iteration of the Sieve algorithm could be measured by this performance monitoring task and then compared with the time found during the benchmarking operation. The results are given in the format x/y, where x is the time measured by the performance monitor, and y is the time measured by the benchmarks, and as a performance index figure. The performance index is calculated from the value of y/x. The closer the value of the performance index to the value 1.00 the better the performance of the operating system in executing the simulation.

2.2.1. The Structure of the Plant

The structure of the plant is illustrated in Figure 1. The plant consists of a reaction vessel with two inlets - one for each reactant A and B. The vessel is fitted with metering devices for both fluid level and temperature. The temperature is controlled by means of a heating element. The output of the vessel (fluid C) is fed through a moulding device which deposits moulded widgets on a conveyor system. This has a maximum capacity of ten widgets, and transports the widgets to one of two polishing machines which perform the final processing of the widgets before they leave the system. The process control system has control of the valves (two inlet and one outlet), the current to the heater element, the release switch of the moulding device and the selection and starting of the polishing machines. The plant produces level and temperature readings from the reaction vessel and the mass of the fluid in the mould. The polishing machines produce termination signals when the processing of a widget is completed. This system

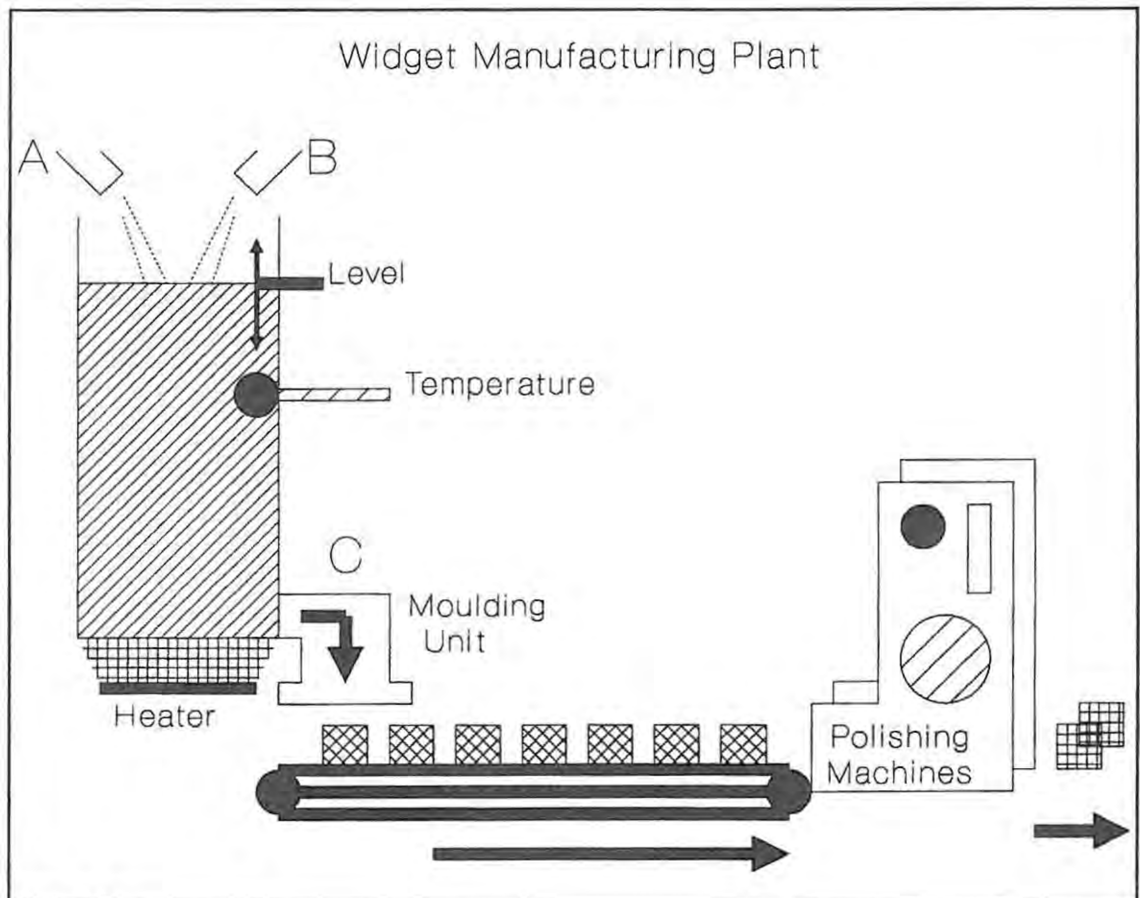

Figure 1

contains most of the features of general simulations, such as discrete and continuous data.

2.2.2. The Design of the Simulation

A first attempt was made to design the simulation using an *ad hoc* approach, as the design itself was initially considered to be of little importance. However, using this approach proved to be extremely difficult and little progress was made. Accordingly, the Ward and Mellor real-time design method was adopted for the development of the system. This development approach involves the use of various tools and heuristics as a guide to the design of real-time systems, and is described in the three volumes of "Structured Development for Real-Time Systems" [Ward and Mellor, 1985, 1986]. In the course of designing the simulation it was found that some extensions to the basic Ward and Mellor approach were desirable to deal with some of the aspects peculiar to simulations. This resulted in the design of the simulation forming an important subtopic within the area of the research.

This section gives a brief overview of Ward and Mellor's method, outlining the extensions that were made for the special case of simulations. This subject is discussed in slightly greater detail in [Wells, 1989].

2.2.2.1. The Environmental Model

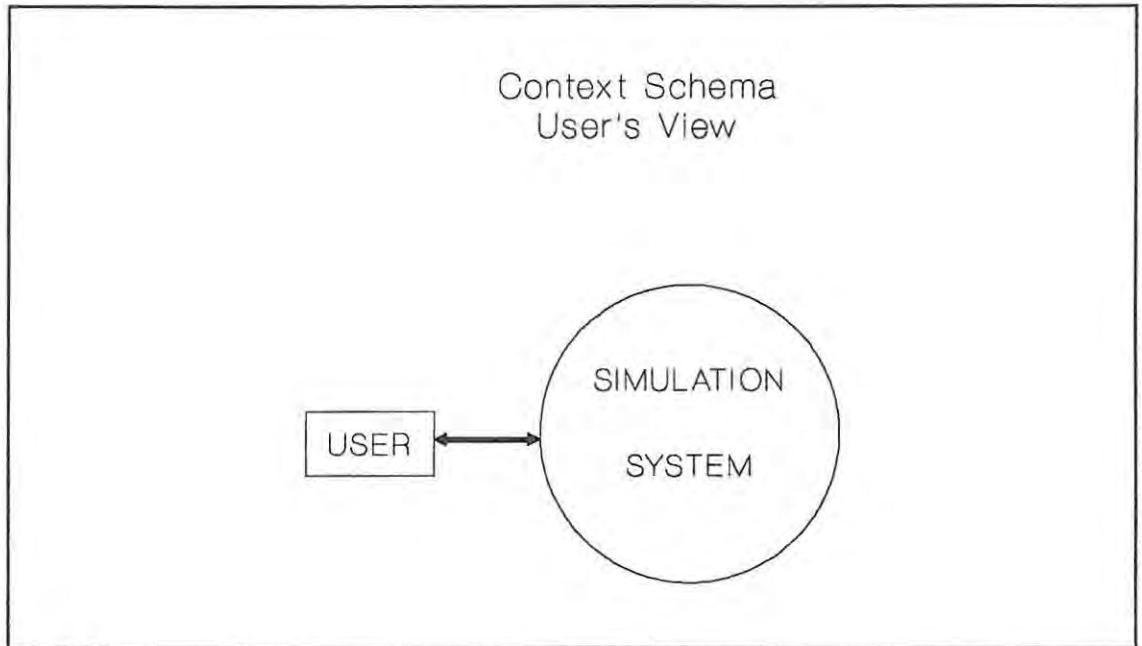

Figure 2

In the case of a simulation, the environment consists only of the user of the system. This follows the standard approach of Ward and Mellor [Vol 2, pp 14, 16], and is illustrated in Figure 2. However, it was found that a more useful approach was to introduce an extra step by partitioning the simulation into two subsystems. For the widget plant process control system these components are the process control system and the widget plant. (In the general case,

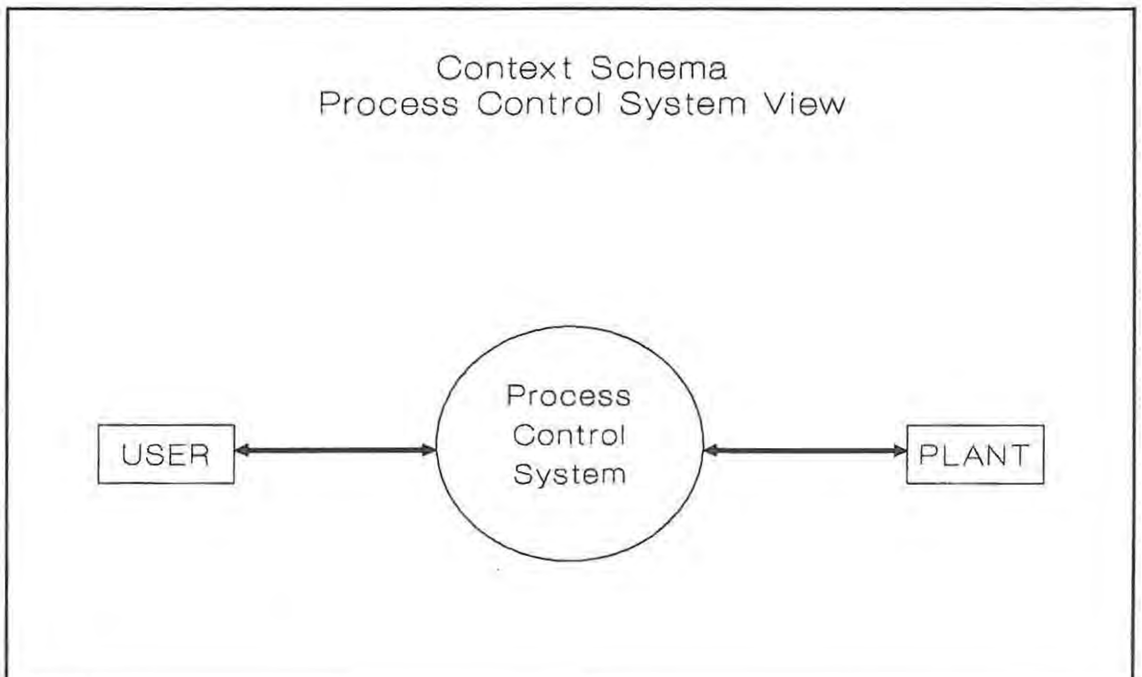

Figure 3

there would be the simulated system, and a control/data-analysis system.) This partitioning of the overall system leads to three points of view: 1) The user view (as shown in Figure 2), which is the representation of the actual system; 2) The process control subsystem view (Figure 3), which embodies the simulated plant as a terminator; and 3) The plant subsystem view (Figure 4), which has the process control system as a terminator. It is important to bear in mind that the final system has the configuration given in the user view, but that the two subsystems can (and should) be designed independently, using the view for that subsystem.

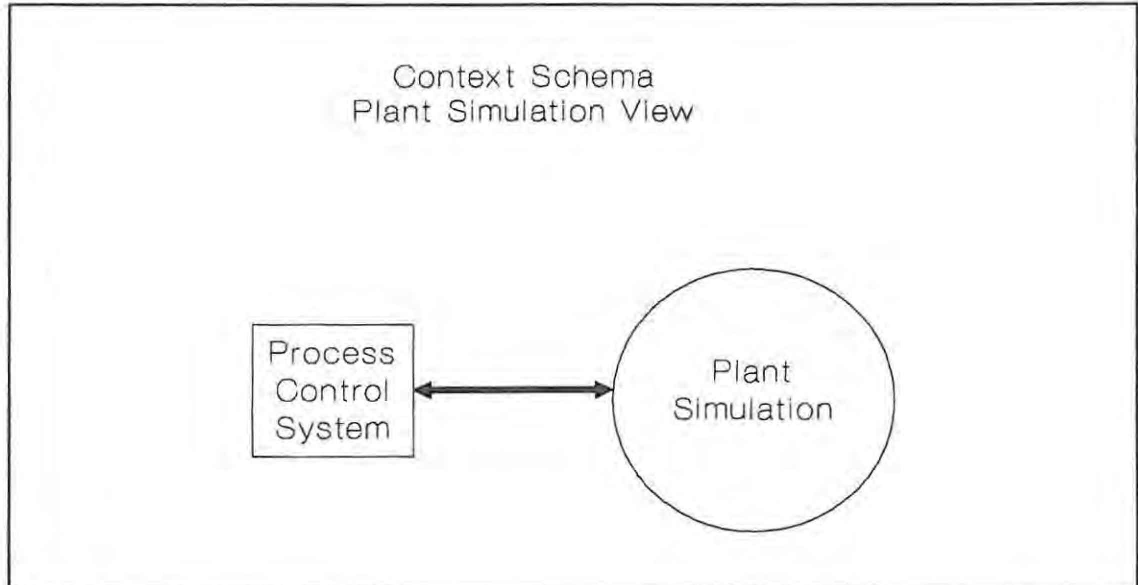

Figure 4

In the decomposition of the process control subsystem the underlying assumption should be that it is immaterial whether the system is to control a simulated or actual plant, as is implied by the status of the plant as a terminator in Figure 3. The terminators chosen should come from a physical description of the plant, and not from any preconceived ideas of how a simulation will eventually be implemented. In the example of the widget plant control system, this leads to the choice of the following terminators: the tank, the moulding line and the polishers. This choice is independent of the fact that the plant is actually simulated - if a physical widget manufacturing plant were to be connected to the control system (using suitable interface devices), the system should still function correctly. The context schema for the process control subsystem is shown in Figure 5. Note that the operator is a "real" terminator, and the others are simulated - they do not exist in the environment of the system, but are in fact a part of the overall system. Similarly, in the decomposition of the plant simulation subsystem, the fact that the operator is the only terminator for the overall system can be ignored, as the only "terminator" that the plant perceives, is the control system. (In a physical plant this would not necessarily apply, as the operator might interact directly with the plant - for example, to close valves manually, or to switch equipment on and off.) This leads to the context schema for the plant simulation subsystem as shown in Figure 4, in which the operator does not appear.

Having developed the context schemata for the simulation, the next step is the construction of event lists (or event-response lists) for each of the subsystems. This can be done following the

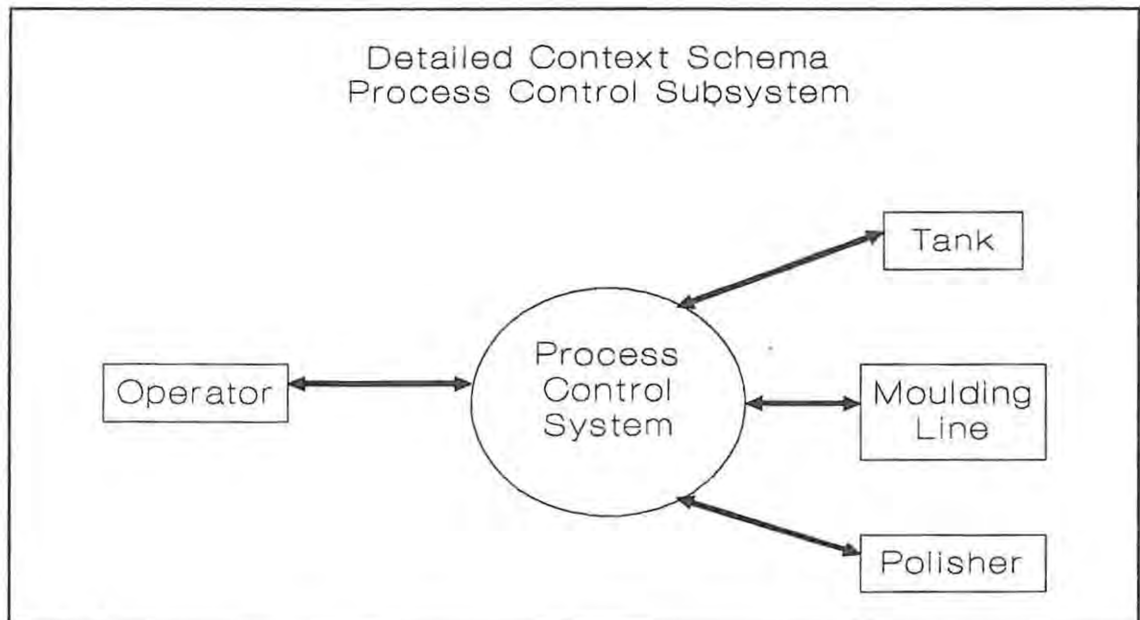

Figure 5

principles outlined by Ward and Mellor, looking at each of the subsystems independently. Again, when choosing the events for the process control subsystem it is important to refrain from making any assumptions about the nature of the plant (that is, whether physical or simulated). As an example, the event list for the widget plant process control system is as follows.

* Operator requests shutdown of plant
* Operator sets new temperature setpoint
* Operator sets new ratio of reactants
* Temperature setpoint is reached
* Widget mould becomes full
* Widget polishing completed

This completes the environmental model, which forms the analysis phase of the development of the overall system. The next step is the first part of the design phase - constructing the behavioural model.

2.2.2.2. The Behavioural Model

As mentioned above, this comprises a specification of the intended behaviour of the system. It should not be influenced at all by implementation or technological concerns - that is the subject of the implementation model. The construction of the behavioural model starts with the context schema of the environmental model and refines it by specifying the content of the central transform. The first step is to use the event list to generate detailed transformation and data schemata. The result of this is a complete specification, but is often unusable due to the amount of detail included. This leads to the next step - that of upward levelling. This is a process of combining the detailed transformations to form higher level groupings. The upward levelling procedure is repeated until a reasonably high level representation is obtained. This operation results in a hierarchical structure of transformation schemata, with the context schema conceptually forming the highest level and the detailed behavioural model forming the lowest

level. The levelling process is guided by the application of several heuristics [Ward and Mellor, Vol 2]. In designing a simulation it is useful to add some further heuristics. These new heuristics follow from the general principles that form the foundation of the heuristics given by Ward and Mellor.

Heuristic 1: Grouping According to Simulated Structures

In grouping lower level transformations to form higher level representations, the groupings should take into account the structure of the system being simulated. In our example, it is possible to conceive of a system in which one transform is controlling the simulated polishing of the widgets

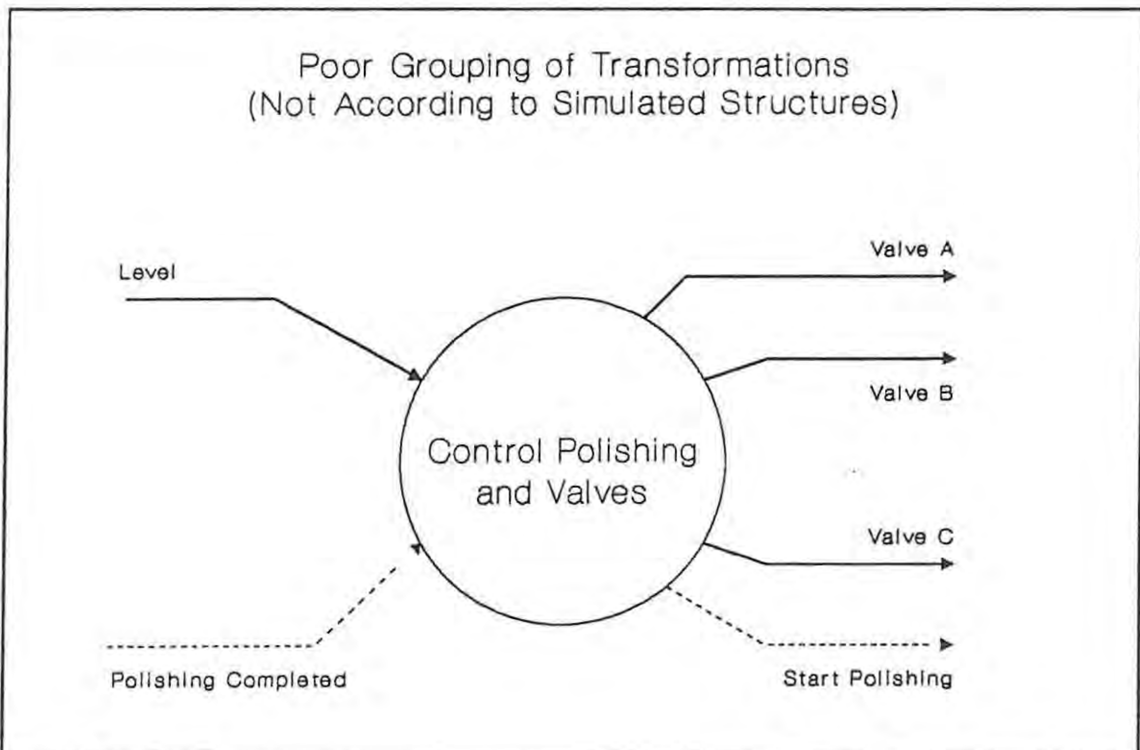

Figure 6

and the flow of reactants into the vessel (as shown in Figure 6). However this leads to an unnatural view of the system. In the case of a physical plant, the inlet valves and polishers would be distinct terminators and thus the principle of using terminator-related groupings [Ward and Mellor, Vol 2, p69] would apply - the grouping according to simulated structures is a specialisation of this more general principle.

It should be born in mind that the eventual implementation model may distort the behavioural model to include a transform such as that mentioned above. (For example, if the widget polishing control transformation spends most of its time waiting for a signal to say that the polishing of a widget is complete, it may be desirable to utilise this waiting time in calculating the settings of the inlet valves, but this is an implementation consideration and has no place in

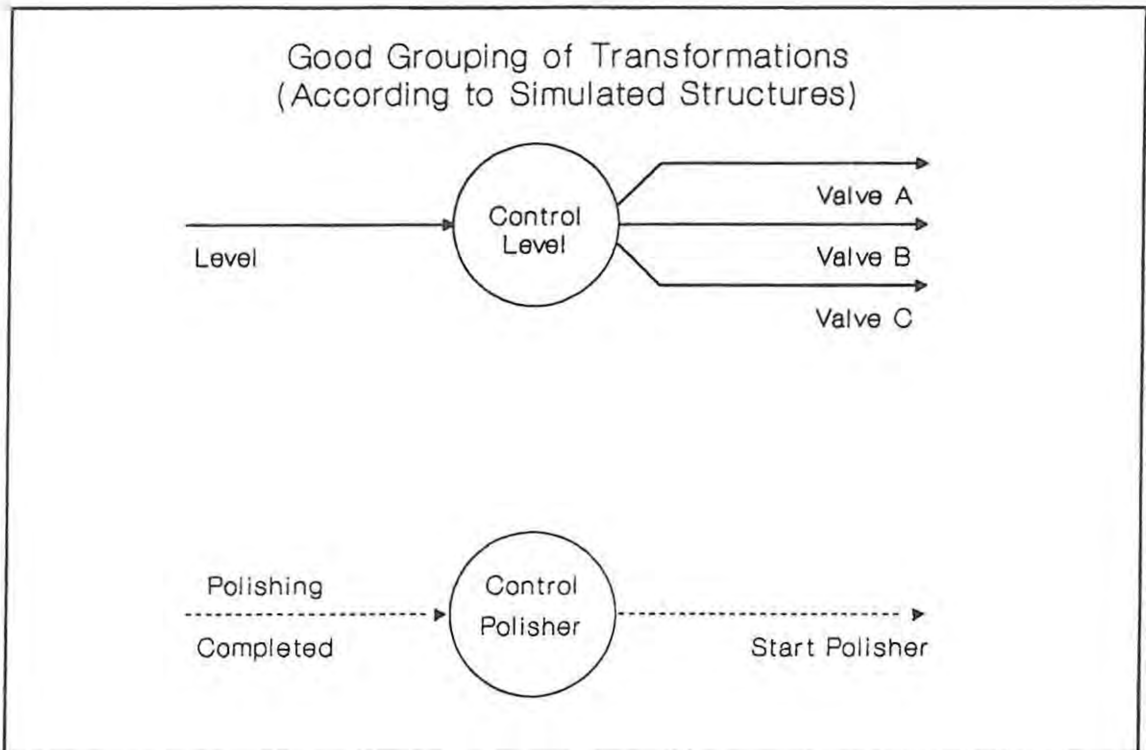

Figure 7

the behavioural model). A better grouping of the transformations is shown in Figure 7.

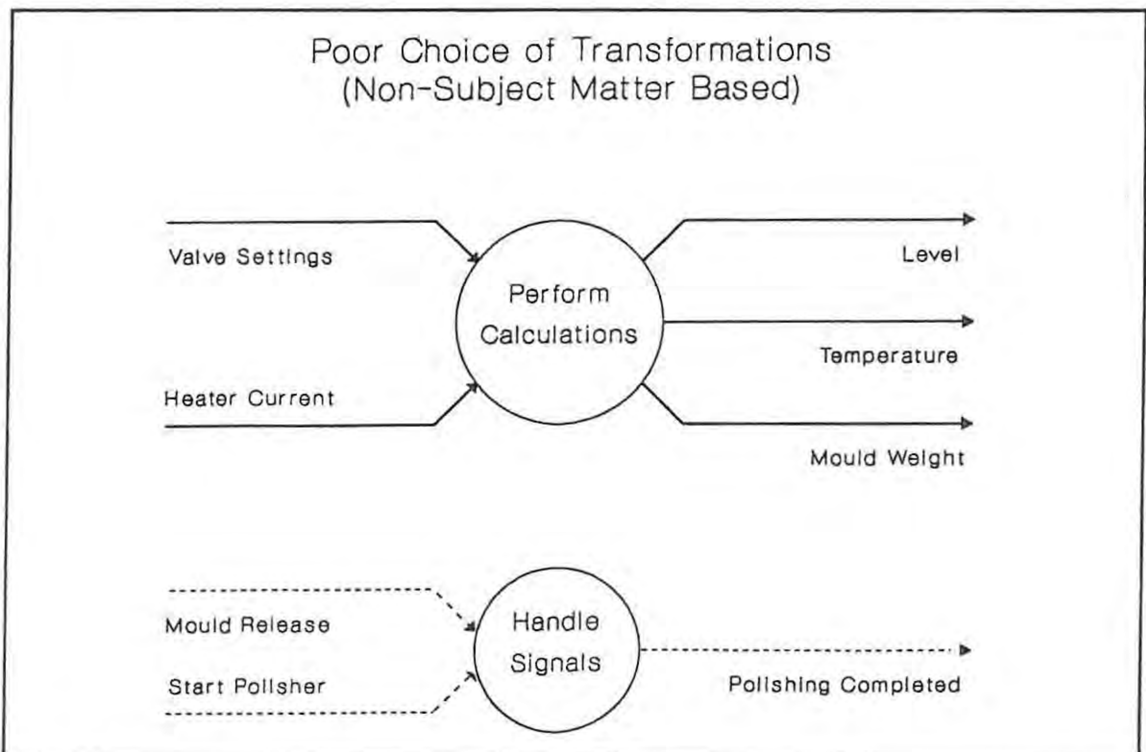

Figure 8

Heuristic 2: Naming According to Subject Matter Terminology

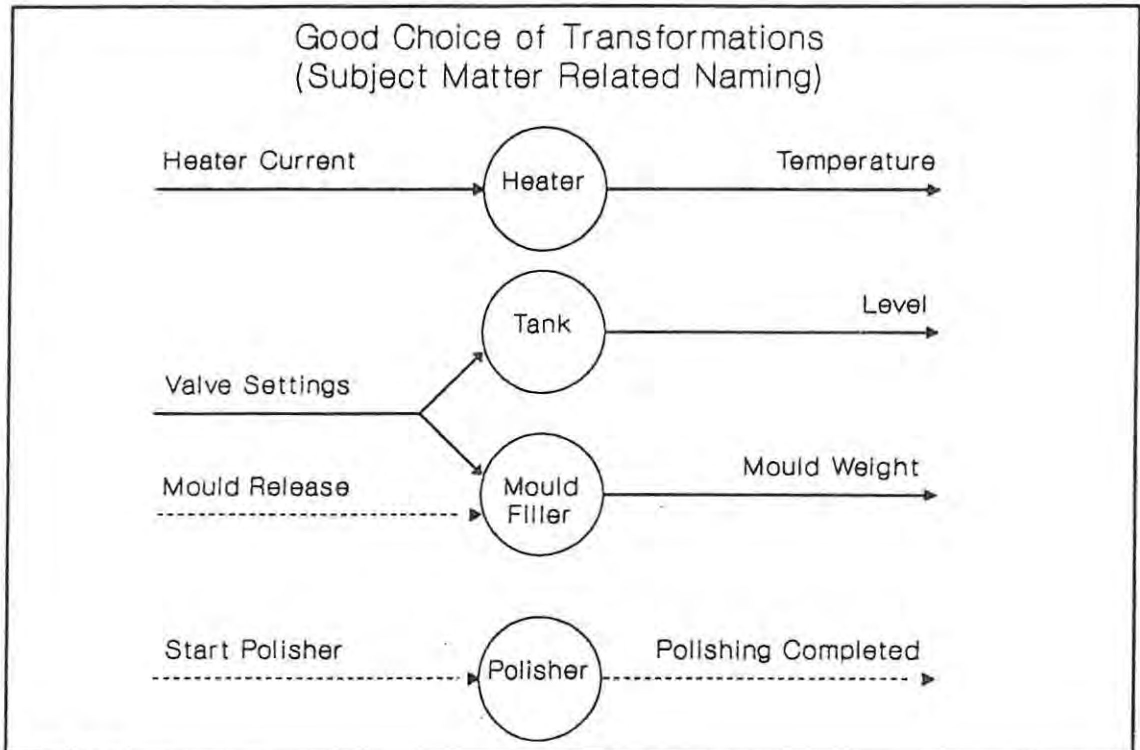

Figure 9

This heuristic arises from the principle of subject-matter-centered modelling [Ward and Mellor, Vol 2, p6]. It states that the transformations in a simulation should be named according to the subject matter being simulated, and not according to the mechanism of the simulation. This allows for a logical grouping of transformations, and simplifies the interface with the control system (if that has been designed using the principles discussed above). It is possible that a grouping could be proposed that would violate this heuristic by concentrating on the implementation strategy for the simulation. For example, in partitioning the upper levels of the widget plant simulation we might propose a transformation for performing calculations and one for handling interrupts, as shown in Figure 8. However, this is again a rather unnatural grouping and does not follow the principle of naming transformations using subject-matter terminology. A better partitioning is shown in Figure 9.

The overall behavioural model for the system is shown in Figure 10. Having completed the behavioural model the analysis of the intended system is complete, and the fundamental concepts required for the design (that is, the ideal structure of the solution) are in place. The next step is to produce the implementation model before embarking on the implementation itself.

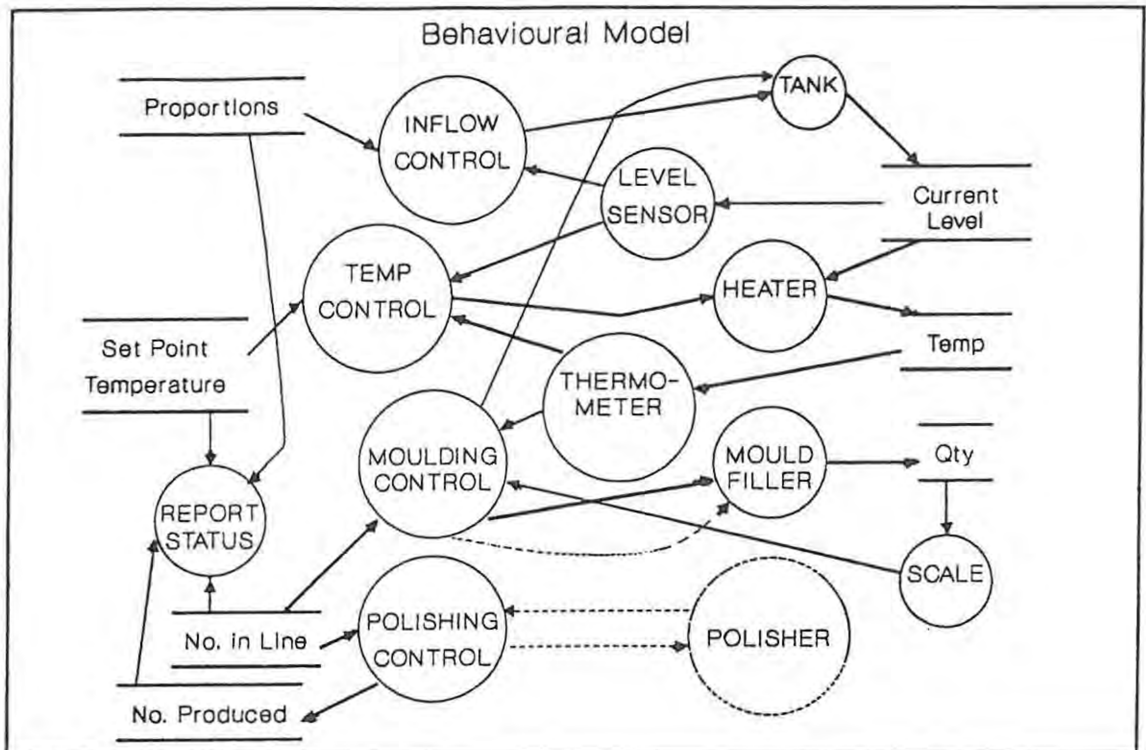

Figure 10

2.2.2.3. The Implementation Model

The implementation model seeks to structure the system in the best possible way for implementation as a computerised system. This may involve distortion of the behavioural model, as discussed above. The implementation model embodies restrictions specific to the hardware and software available for the implementation and so it is difficult to formulate general principles. The main factor when working on a simulation is the implementation of the interface between the two major subsystems (the control/analysis system and the simulation itself). This is one area where the fact that the systems are being simulated can be an advantage and may be allowed to influence the design of the overall system. The reason for this is that it may be possible to disregard the need for physical communications. For example, in a physical plant a temperature reading might be made by a thermocouple. This analogue value would need to be converted to a digital format, transmitted to the process control system and then converted and scaled (from a voltage value to a temperature). In a simulation all the details involved in this can be overlooked, as the simulation and the control system both work in terms of digital values and can both be made to work in units of temperature. This considerably simplifies the decisions that would have to made for a physical system. Of course, if communications problems of this nature are significant among the reasons for performing the simulation, then allowance for them would have to be made.

Typically, when implementing communication between two software systems, there is a variety of possible techniques available. These include shared storage (memory or secondary storage), message passing (which can be either synchronous or asynchronous), semaphores and signals (if no data transmission is involved, only event recognition), and other structures such as pipes. The

choice of mechanism for linking a simulation to the control/analysis subsystem depends on the implementation technology available. An important factor which must be born in mind is that the communication should be asynchronous. There is no reason why the two subsystems should run synchronously if they are independent of each other, as they would be in a physical situation (a plant does not synchronise itself with its control system, rather the control system must react to changes within the plant within the time constraints imposed by the nature of the plant). Another factor arising from this is that the simulated system should run at a faster rate than the control system in order that it may appear to be a continuous process, to the control system.

That ends the discussion of the extensions required to the Ward and Mellor methodology, and an overview of the design of the widget simulation. The implementation schemata for each of the operating systems will be presented in the chapters dealing with the systems individually.

2.3. Analysis of the Results

The results obtained from the benchmarks consisted of a large number of readings which had to be analysed. As mentioned in section 2.1, the benchmark programs were all run ten times. This almost invariably resulted in a range of results for each test that had to be interpreted. Spreadsheet analysis was used for the results, and the tables given in Appendix B are the output from the spreadsheet program. For each of the sets of results the mean and the standard deviation were calculated.

2.3.1. The Use of the Mean and the Standard Deviation

The use of the mean, rather than the mode, is a possible cause for debate. Lyon [1970, p94] suggests that the mean should be used as an estimator of the true value when the distribution of the random errors in the data is a well-behaved one with a bell-shaped form (that is, symmetrical, unimodal and tending to zero for large errors of either sign). In the case of the benchmark results the distribution is not a well formed bell shape, but rather a form of Poisson distribution, where the random errors tend more towards positive values. This results in a form of distribution as shown in Figure 11. Lyon states that in these cases it may be more reasonable to quote the mode as the true value.

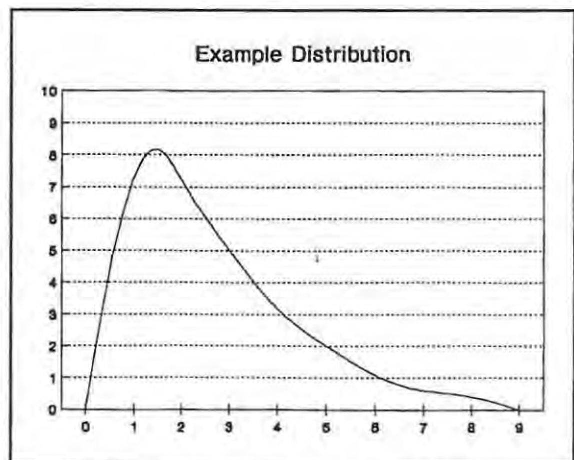

Figure 11

In the case of the benchmarks the mean of each set of readings has been used as the true value of the result. The reason for this is that the strict meaning of true value (as

used by Lyon) does not really apply to the measurement of the performance of the operating systems. Lyon is dealing mainly with physical experimentation in which it is expected that there is one true result and that any deviation in the measured results is due to experimental error of one form or another. In the case of the assessment of operating systems the deviation of the results is due to the implementation of the software, which is subject to periodic interference (such as the need for flushing buffers, or for compacting memory segments). This is what causes the distribution of the results to spread more in the direction of positive "error". This corresponds to the case of random fluctuations due to the nature of the system being studied, which Lyon discusses briefly [1970, pp19-21]. He makes the point that for these situations the standard deviation is often of more interest than the standard error. Accordingly, the standard deviation has been adopted in the evaluation of the operating systems as a measure of the uncertainty in the results.

The use of the mode for the benchmark results would lead to the most common result, that is, the response that could be expected most of the time. However, this ignores the fact that occasionally a larger response time will occur. The mean thus gives a better indication of the average response that can be expected, and has been adopted for this reason.

2.3.2. The Effect of Clock Resolution

In addition to this source of "error" (that is, the factor introduced by the software), the resolution of the measuring device used (that is, the hardware clock) also gives rise to some uncertainty as to the true value. The standard deviation due to this factor is well understood and is given by the following formula:

$\sigma_R = a / (2 \sqrt{3})$ Where a is the resolution of the clock [Lyon, 1970, p75].

This factor should always be borne in mind when studying the benchmark results.

2.3.3. Combination of Uncertainties

In most cases the result, and hence the uncertainty in the result, had only to be scaled in order to calculate the quantity under scrutiny for a particular benchmark. For example, if the time taken to send 100 messages of 100 bytes was measured to be 1234 ± 56 ms, then the time taken per message is 12.34 ± 0.56 ms. However, in the case of the synchronisation and the exception handling benchmarks the final result had to be calculated from the difference between two measured values. In this case the uncertainties in the two measured values must be added to find the uncertainty in the calculated result. Lyon makes the point that if the calculated result in this type of situation is very small the percentage uncertainty in the result becomes very large [1970, p41]. This was found to be especially true in the case of the results of the exception handling benchmarks.

3. XENIX System V

This chapter discusses the results of the investigation of XENIX System V. This is an implementation of UNIX System V for microcomputers, and the names XENIX and UNIX are used interchangeably in this report.

3.1. Introduction

All the results were obtained using the `ftime()` system call available under XENIX. This returns time information in a structure which allows the elapsed time to be calculated in units of milliseconds. The resolution of the timer is 20ms. This resolution gives rise to a standard deviation of 5.8ms due to the accuracy of the clock (as discussed in section 2.3.2). To give an indication of the relative efficiency of the XENIX C compiler, the time taken for the Sieve of Eratosthenes program was 3.0420 ± 0.0063 s.

3.1.1. Concurrency

UNIX provides for the creation of child processes through the `fork()` system call. When a process performs a fork operation a second copy of the process is created in memory which is almost identical to the first. The main distinguishing factor between the two processes is the value returned by the `fork()` system call. This is zero in the case of the child process and the value of the process identifier (PID) of the child process in the case of the parent process. This can be illustrated by the following code segment:

```
if (fork() == 0)
  { /* Child process executes from here */
  }
else
  { /* Parent process executes from here */
  }
```

It must be stressed that the two processes are copies of each other and hence this system call is often not particularly useful in itself. The most common use of `fork()` is in conjunction with the `exec()` system call (or calls, as there are several variations in the use of `exec()`). The `exec()` system call is used to replace a process with another program loaded from secondary storage. Due to its implementation by replacing the calling process, this function effectively terminates the calling process and hence never returns (unless an error occurs during the call). It is because of this that the `fork()` and `exec()` calls are often combined as shown in the following code segment:

```
if (fork() == 0)
  { exec( /* Some other program */ )
  }
else
  { /* Parent process executes from here */
  }
```

In this case the child process is replaced by the program named in the exec() system call. The wait() system call is provided to allow parent processes to wait for the termination of a child process and to receive information about the cause of the termination (such as error values, *et cetera*). Accordingly, the following variation on the example given above is often found in programs written under UNIX:

```
if (fork() == 0)
  { exec( /* Some other program */ )
  }
else
  { wait(&status); /* status is the exit value of the child process */
  }
```

An important factor in any real-time operating system is the scheduling policy used when allocating the processor to tasks. The following overview of the UNIX scheduling mechanism is based on the excellent discussion by Bach [1986]. The process scheduler in the UNIX operating system uses a **round robin with multilevel feedback** algorithm. This means that the kernel allocates the CPU to a process for a fixed time period and when it completes that period (or quantum) it is placed back in one of several priority queues. When rescheduling the CPU, the kernel chooses the process with the highest priority that is ready to run and loaded in memory, or has been preempted. The system distinguishes two levels of priority: user mode priority and kernel mode priority. The latter is further classified into interruptible and non-interruptible levels. The priority of a process is calculated according to several criteria:

- Processes that are waiting for I/O to complete, or for kernel resources (such as buffers) to become available are assigned a fixed priority according to the reason for their suspension. Essentially, processes waiting on low level facilities are more likely to cause bottlenecks and so receive higher priorities.

- When a process returns from a system call its priority is adjusted. The first adjustment is to reset the priority of the process to a user mode priority. The second is to decrease the priority of the process as it has just made use of valuable kernel resources. This ensures fair access to the kernel for all processes.

- Finally, the clock interrupt handler adjusts the priorities of all user mode processes approximately once a second and then causes a reschedule to take place. The recalculation of the priority takes into account the following two factors.

  i) The amount of CPU usage is used to penalise CPU intensive processes. This results in highly interactive tasks such as editors receiving high priorities, giving good response times.

  ii) The **nice** value set by a process is added into the priority calculation. This can be used by a process to increase or decrease its priority in a crude way.

As can be seen from this discussion, the scheduling of processes is very fair. This not really suitable for real-time processing where a critical task must be handled timeously, irrespective of

whether that is "fair" to the other tasks in the system. As well as this, Bach points out that the kernel is nonpreemptive. That is, the kernel cannot schedule a user mode task if another task is currently executing in kernel mode. This means that real-time tasks which must respond to external events must be coded into the kernel (which is not a very satisfactory approach).

3.1.2. Interprocess Communication

The interprocess communications facilities provided by UNIX System V fall into two categories. There is the classic `pipe()` system call which has always been closely associated with UNIX (and adopted by many other operating systems), and then there are the new message passing and shared memory facilities introduced in UNIX System V Release 2. The `pipe()` facility is designed to work in conjunction with the `fork()` system call, as it relies on the fact that the processes that wish to communicate using a pipe must share a common ancestor which initially created the pipe. (Note that this is no longer strictly necessary as **named pipes** were introduced in UNIX System III. This facility allows independent processes to access a common pipe.) Pipes allow for asynchronous communication of data. The data is sent as a sequence of bytes with no structure imposed by the system. A process reading from an empty pipe will be suspended until another process writes some data to the pipe. Processes writing to a pipe are not usually suspended unless the pipe is full (a pipe has a typical capacity of around five kilobytes). The communication is usually unidirectional, but, with the use of a suitable protocol, can be bidirectional.

The newer message passing and shared memory facilities (introduced in UNIX System V Release 2) are somewhat more sophisticated than this, but they suffer from an extremely complicated interface from the perspective of the programmer using the facilities. The message passing calls allow for the asynchronous communication of data using **message queues**. A process may gain access to a message queue by using a **key** (an integer selected by the programmer that identifies the message queue to the operating system). Having gained access to a message queue a process may place messages on the queue, or read messages from the queue, provided that it has the correct access permissions. The access permissions are specified by the process which created the message queue. The messages are essentially unstructured sequences of bytes, but do have a header field which can be used to read messages selectively from the queue.

The shared memory facilities allow processes to access a common area of primary storage. Again access to the shared memory is controlled by the use of keys and access permissions, as for message queues. A process which has access to a shared memory segment may **attach** the segment to its logical address space and manipulate the data stored in that segment. This obviously raises the need for synchronisation of the access to the shared memory segment if two or more processes attempt to access the same piece of data concurrently. This synchronisation is usually provided by semaphores (see section 3.1.3).

3.1.3. Synchronisation

The older versions of UNIX had no real need for synchronisation as processes executed almost independently. Parent processes could synchronise with the termination of a child process using the `wait()` system call, and pipes provided for communication between processes. The property of pipes that readers will be suspended until there is some data to read, and that writers will (in general) not be suspended allows pipes to be used to simulate semaphores [Wells, 1987]. In addition to this, the latest versions of UNIX System V (since Release 2) have included semaphores. These semaphores have the same complex programming interface as the message passing and shared memory calls. A set of semaphores is accessed using a key and then may be signalled or waited upon in the usual fashion. One uncommon feature of UNIX semaphores is that they are in fact *sets* of semaphores and a series of operations can be applied to an entire set at once. If any one operation in the set of operations results in the suspension of the process then none of the operations is performed. There are some other unusual features of UNIX semaphores, but they will not be considered here.

3.1.4. Exception Handling

As already mentioned, these facilities are not directly related to hardware interrupts, or error handling, but are the mechanisms provided for the suspension and resumption of processes. Under UNIX there are two approaches that can be taken. The first is the voluntary suspension of a process by calling on the operating system to suspend it for a given period of time. XENIX System V provides a system call called `nap()` which provides this facility, allowing a process to be suspended for a specified number of milliseconds. The second mechanism is where a process suspends itself awaiting a signal from another process. This is provided for by UNIX through the use of the `kill()` and `signal()` system calls. The action of `kill()` is to send a signal to a specified process. There are several different types of signals defined, some used by the operating system and some for general use by programs. The default action of a signal is to terminate the receiving process. However, the receiving process can control the actions to be taken by using the `signal()` system call to specify a function to be executed when a particular signal is received. This is the approach that was used for the tests discussed below.

3.2. The Benchmarks

3.2.1. Concurrency

The first test performed in this category was to measure the effect of multiple tasks executing concurrently. Two versions of the programs were written. The first used the `fork()` system call to create child processes which then executed the Sieve algorithm. The second used the `fork()` system call again, but the child processes then used the `exec()` system call to execute a program that performed the Sieve algorithm.

Within the limits of the experimental uncertainty it was found that the execution time increased linearly with the number of processes. Looked at from another perspective, the average execution time per process remained constant as the number of processes increased. The actual results for the overhead due to the processes were subject to experimental uncertainty which was of the same order as the results themselves, and so no definite values could be calculated. The only exception was in the case of the exec() system call when repeated ten times. This gave a result of 268 ± 84 ms for the overhead due to the concurrent execution of ten copies of the program. As could be expected, the first program (using fork() alone) was more efficient than the second (by a relatively small factor of approximately 7%). This can be attributed to the extra system call required and the disk access in the second case.

Again, the second test in this category (that is, the process creation benchmark) was performed for both the fork() system call and the fork()-exec() combination. The first version was more efficient once more, taking between 10% and 15% of the time for the second version. For the fork() system call the creation time for each child process was found to be almost constant (at 19.1 ± 3.9 ms) in relation to the number of processes created. The variation for the fork()-exec() combination showed a sharp "elbow" between 20 and 30 child processes. This is probably due to the execution time reaching a critical level where the operating system starts to interrupt the processing to perform "house-keeping" tasks. The values for process creation times in this case ranged from 25 ± 49 ms (obviously, one would not expect negative results - this high standard deviation is a result of the indeterminacy of the scheduling algorithms) per process for 20 processes, to 175 ± 15 ms per process for 100 processes. The value of 175ms appears to be asymptotic (see Figure 12).

The effects of background processing were assessed for the second benchmark in this category. For a single background process, the performance of the fork()-exec() version was unaffected, but the performance of the fork() version deteriorated by 38%. For ten background processes degradation factors of 120% and 1500% were measured, respectively. This was for background processes executing at the same priority level. Background processes at a higher priority level effectively blocked the benchmark processes completely. The results with background processes at lower priority are summarised in the following table.

	1 Background Process	10 Background Processes
Fork()	13%	87%
Fork()-Exec()	0%	4%

While these factors are much less than for background processes at the same priority it is obvious from this that any background process at any priority is going to impact the performance of any other process. It should also be noted that although the fork() system call is more efficient than the fork()-exec() combination, it is impacted more severely by background processing. This is probably due to the fact that in the second case the background processes can be scheduled during the delays for disk access and so have less impact on the foreground process.

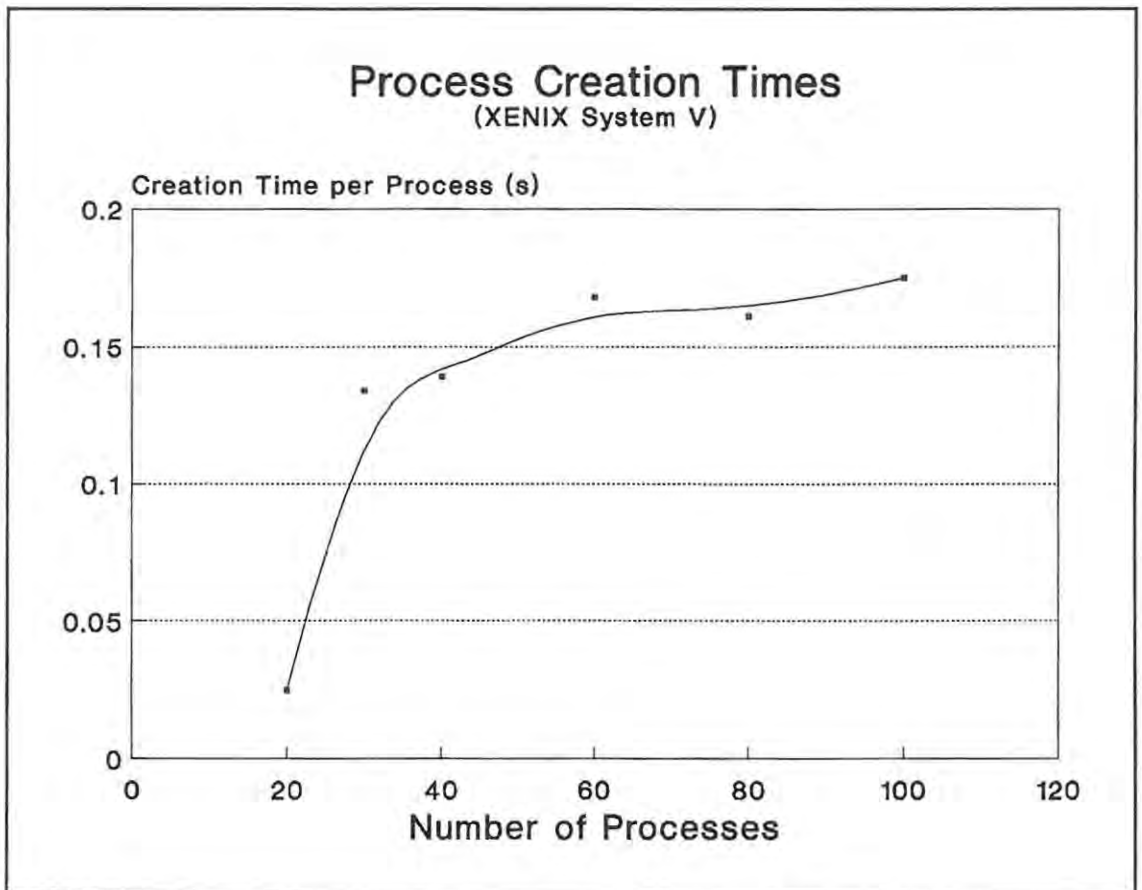

Figure 12

3.2.2. Interprocess Communication

This test was performed using pipes as well as the more modern message passing and shared memory system calls. The results of the benchmarks are summarised in the following table. The ratio given in the table below is the ratio between the most efficient of the communication mechanisms (message passing) and the mechanism used in the test. This gives an indication of the relative performance of the different methods for interprocess communication.

	Time per Message (ms)	Ratio
100 Byte Messages		
Pipe	3.06	1.6
Message	1.89	1.0
Shared Memory	8.68	4.6
Circular Messages (1 Byte)		
Pipe	3.85	1.3
Message	2.96	1.0

From this it can be seen that the message facilities available in UNIX System V are the most efficient means of IPC (they take 62% and 77% of the time that the pipe call does in the 100 byte test and the circular test respectively). It is expected that the shared memory method would be faster if it were not for the fact that synchronisation is required to control access to the shared memory segment. In the tests performed here synchronisation was provided using System V semaphores. One anomaly in these results is that the time taken for the circular (1 byte)

messages (3.8515 ± 0.0021 ms for pipes, and 2.959 ± 0.016 ms for messages) was greater than the time for the 100 byte messages (3.06 ± 0.46 ms for pipes, and 1.89 ± 0.53 ms for messages). A possible explanation of this is that it is due to the fact that there were three processes executing for the circular message test, and they were all performing similar processing. In the case of the 100 byte message test there were only two tasks executing, and the processing performed by one of them (the receiver) was almost negligible.

The pipe() system call and the message passing calls were found to have a very linear relationship to the number of messages sent. (A very slight decrease in the times was noted as the number of messages grew very large.) However, the shared memory calls showed an extremely marked decrease in the time taken per message as the number of messages increases, as shown in Figure 13. This is also highlighted by the ratio between the values given in the table below. The ratio between the message passing and pipe programs remains at around 1 : 1.6, but the ratio between the message passing and shared memory versions drops from 1 : 4.59 to 1 : 3.23 as the number of messages passed increases from 1000 to 40000.

Ratio	message passing	:	pipe	:	shared memory
1000 Messages	1	:	1.62	:	4.59
40 000 Messages	1	:	1.65	:	3.23

It is impossible to isolate the cause of this phenomenon with any certainty, but one possible explanation is that the overhead due to the synchronisation becomes less significant as the number of messages increases.

The effect of background processing was assessed for all the tests discussed above. For background processes at the same priority as the benchmark process the results were as follows. The values shown are the percentage increase in the execution time of the benchmark programs.

	One Background Process	Ten Background Processes
100 Byte Messages		
Pipe	54%	970%
Message	49%	1020%
Shared Memory	83%	110 000%
Circular Messages (1 Byte)		
Pipe	19%	890%
Message	47%	810%

The main point to note from this is the severe degradation suffered by the benchmark using shared memory. This can be attributed largely to the synchronisation required in this case (see the section 3.2.3 for further discussion of this point). For the other IPC facilities the degradation is of the same order of magnitude in each case.

For background processes at a higher priority it was found that in almost every case the presence of the background processes blocked the benchmarks almost completely. However, for some of the tests using 100 byte messages with only one background process the effects were measurable. For message passing it was found that for 10 and 100 messages the benchmark program executed so quickly that the background process had no effect at all. For 1000 messages the benchmark was blocked. For pipes and shared memory the degradation for 10 and

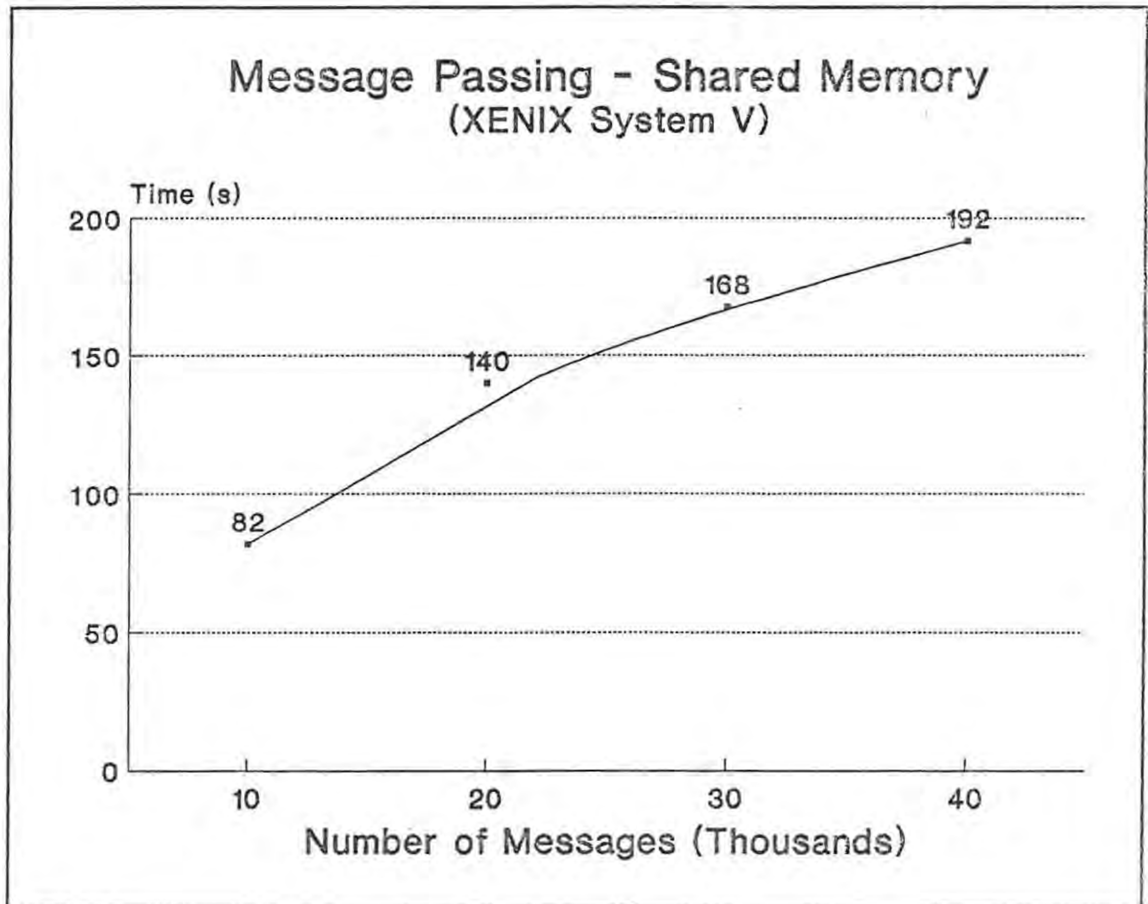

Figure 13

100 messages was difficult to assess due to the very small timing intervals involved, but there appeared to be a slight degradation. For 1000 messages the performance of the version of the benchmark using a pipe deteriorated by 45%. For the shared memory version the deterioration was found to be 133%. It is difficult to account for these relatively low values. A possible explanation is that the penalty factor applied to the background processes due to their CPU intensive nature partially overcomes the effect of their higher priority.

For background processes at a lower priority the results summarised in the following table were found.

	One Background Process	Ten Background Processes
100 Byte Messages		
Pipe	23%	60%
Message	32%	52%
Shared Memory	48%	330%
Circular Messages (1 Byte)		
Pipe	12%	60%
Message	30%	32%

Again, this follows much the same pattern as in the case were the priorities are equal. However, the effects are much less severe, as should be expected. Ideally, for real-time applications, they should all have no effect, of course.

3.2.3. Synchronisation

Two versions of the synchronisation benchmarks were performed. The first used System V semaphores, and the second used pipes (as discussed in section 3.1.3). In the latter case the pipes were used to send a one byte message from one task to the other as a signal that the other could proceed.

It was found that adding the synchronisation caused the execution time of the unsynchronised program to experience a degradation of 550 times (for the semaphore version) and 810 times (for the pipe version). The results allow the calculation of the actual time taken per synchronisation operation. For System V semaphores each operation took 5.006 ± 0.021 ms. For pipes 8.01 ± 0.17 ms were taken per synchronisation operation.

The effect of background processing on the time taken for synchronisation was assessed and produced very interesting results. The program using System V semaphores was extremely badly affected by the presence of background processes. For one background process the time per synchronisation operation rose to 9.35ms (an increase of 90%). For ten background processes this rose to 9.97s (an increase of 1600%). The effect on the version of the benchmark that used pipes was not as severe. The results were 12.2ms (that is, an increase of 59%) and 70.0ms (that is, an increase of 900%) for one and ten background processes respectively. This is probably due to the fact that the UNIX kernel uses various factors to determine the dynamic priority of processes. This approach associates a penalty with particular kernel function in order to prevent processes from hogging expensive kernel resources. In this case it would appear that the penalty associated with the use of semaphores is greater than that associated with pipes.

For background processes at higher and lower priorities the following results for the degradation were found.

	One Background Process	Ten Background Processes
Higher Priority		
Unsynchronised	147%	
Pipe	∞ %	
Semaphore	130%	
Lower Priority		
Unsynchronised	52%	170%
Pipe	42%	93%
Semaphore	47%	350%

The only fact that can be deduced from this data is that it follows no logical pattern at all! For higher priority background processes the semaphore version suffered less degradation than the unsynchronised version. Only the pipe version behaved as one would expect and was blocked completely. For lower priority background processes the pipe version suffered the least degradation while the semaphore version was now affected even more than the unsynchronised version. This highlights the non-deterministic nature of the UNIX scheduling algorithms, and the unsuitability of UNIX for serious real-time applications.

3.2.4. Exception Handling

Again, two versions of this benchmark were performed. The first used the `nap()` system call available under XENIX System V to cause the operating system to generate an exception. The second used UNIX signals, thus testing the generation of software exceptions. The task which sent the signals in this case used the `nap()` system call to suspend itself between successive exceptions. These tests were performed for various intervals between exceptions, ranging from 1ms to 60ms. The results are summarised in Figure 14. It can be seen that the `nap()` system call provides shorter service times. This is to be expected, as this case is handled by the operating system alone. The `kill()-signal()` version has the extra overhead of a third user process executing to generate the exceptions.

The service time for the `nap()` system call has a clear asymptotic value of 8.6ms. The service times for the `kill()-signal()` version do not follow such a clear pattern, but appear to tend towards a value of around 15ms. For an interval of 40ms, exact results of 8.7 ± 6.6 ms and 10.37 ± 0.79 ms were obtained for the `nap()` version and the `kill()-signal()` version respectively.

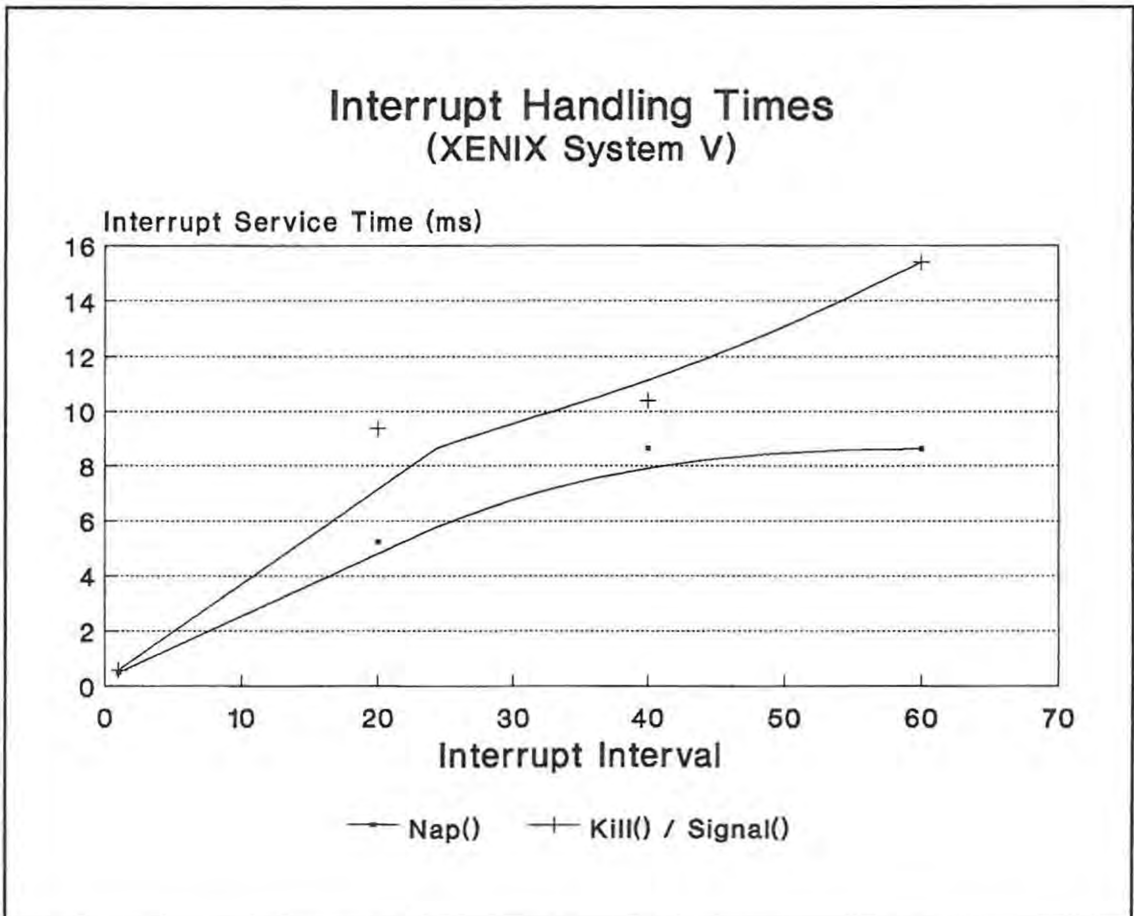

Figure 14

3.3. The Process Control System and Plant Simulation

The implementation model for the simulation is shown in Figure 15. The simulation comprised a set of twelve tasks under XENIX System V. These were subdivided as follows: three tasks performed the simulation itself; four tasks made up the control system; and five tasks formed the user interface and data logging system. This represents a total of over 1200 lines of C code. As is very often the case, this implementation model reflects a distortion of the (ideal) structure depicted by the behavioural model. The functions of the Tank, Mould Filler, Heater, Level Sensor, Thermometer and Scale transformations shown in the behavioural model have been combined into a single Tank-and-Mould task. This consolidation distorts the ideal structure suggested by the heuristic of grouping according to simulated structures, but is much simpler in practice, as the calculations performed are very simple. In addition, there is a need for data to be shared among the tasks, and had they been implemented as separate tasks there would have been a need for communication and possibly synchronisation. The other major distortion has been to centralise the communication of information between the subsystems.

The communication between the subsystems was implemented using shared memory segments to form a passive communication system (*cf.* QNX). This was particularly useful in this case, as many of the tasks are independent of each other. This meant that there was little need for synchronisation, and so the shared memory segments could be used without the overhead of synchronisation. Where synchronisation was required (both between the two tasks controlling the final polishing steps, and between these control tasks and the simulated polishers) it was provided using semaphores. The communication between the Display Handler and the Status and Keyboard Handlers was implemented using a message queue. In this way all the UNIX System V interprocess communication facilities have been utilised in the simulation (the older facility of pipes was not used).

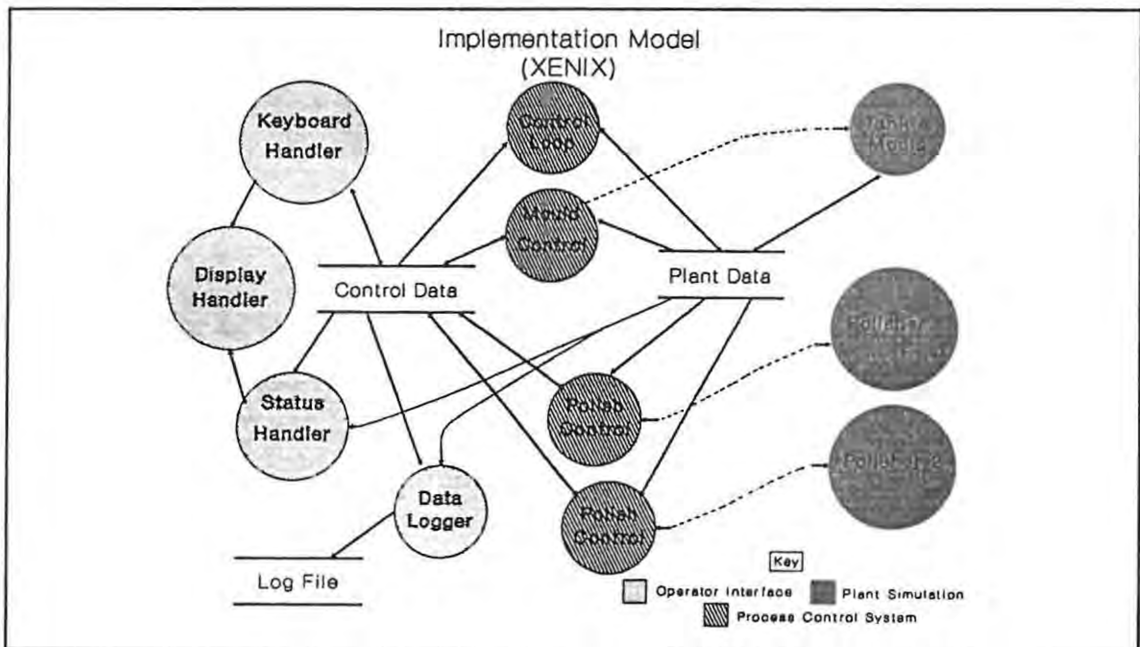

Figure 15

The simulation ran quite adequately, but was not entirely satisfactory. One of the main problem areas was the user interface, specifically in the handling of access to the terminal. UNIX does not allow for more than one task to access a terminal very easily. The approach that has been taken is to place one task in charge of all screen output and it receives messages from the two other tasks that need to perform output, as shown in the implementation model. The first of these tasks periodically updates the status of the plant and control settings, while the second accepts input from the operator of the system and sends it to the screen handler for echoing.

Another weak area was the scheduling algorithms of UNIX which are not designed to support real-time systems, as discussed in section 3.1.1. This means that there is always the possibility of a vital task not receiving processor time when it is required. This was not too much of a problem in the plant simulation as minimum response times are of the order of 250ms to 500ms. However, in a more heavily loaded system with tighter requirements this could well happen. Real-time variations of UNIX have been developed (such as REAL/IX from MODCOMP [Furht *et al*, 1989]) that allow for genuine real-time processing. In fact, efforts are in progress to produce standards for real-time versions of UNIX [van Halm, 1989].

On a more positive aspect, the rich set of IPC facilities available under UNIX System V (including pipes, signals, semaphores, shared memory and asynchronous message passing) were found to be extremely useful in the implementation of the simulation, and are one of the great strengths of System V. The only flaw is the complexity of the interface to the more modern facilities. However, this complexity is a result of the generality and flexibility of the functions, which is highly desirable.

The performance of the simulation was monitored. The average performance found was 4.74s/3.04s, providing an overall performance index of 0.64.

3.4. Qualitative Assessment

This assessment of UNIX as a development environment is based on the experience gained in implementing the simulation described above, and also from several years experience of UNIX systems. UNIX was developed in the late 1960's and early 1970's and is thus the oldest of the operating systems considered in this study. As a result, it has had almost all its defects ironed out. The power and flexibility of the system is considerable, especially in terms of the capabilities of the command interpreters or **shells** that are available. In addition there is a wide variety of useful utility programs available. A common complaint is that because UNIX is old, it is primitive compared to the more modern systems. This is true of "pure" UNIX; however new facilities (such as shells which use mice, touchscreens, *et cetera*) are available to overcome these objections. XENIX System V includes such a menu-driven shell (called the visual shell, or `vsh`).

There is a large range of tools available for program development, including debuggers (`sdb` and `adb`) and utilities such as `make` (for handling large projects with many files) and the Source Code

Control System (SCCS - for handling multiple versions of programs and/or documents). The UNIX documentation is comprehensive and well indexed and cross referenced for easy access, but is extremely bulky. The volume of information can sometimes make locating specific information difficult. There are also numerous third party books available on the subject of UNIX and these cover all levels from novice user to system programmer.

3.5. Conclusions

UNIX is not intended for use as a pure real-time system, but it is suitable for real-time systems with relatively low response requirements. The effects of background processing that have been discussed above highlight the non-deterministic scheduling performed by UNIX and show that there is no clear relationship between CPU usage, priority and use of system calls for determining the rate of execution for a given process in the presence of other processes. However, the large set of IPC facilities available is very useful in developing real-time applications. The powerful development environment and the large set of utilities that are available are also extremely useful in this regard. Thus, it seems that those versions of UNIX that have been adapted for real-time processing would be suitable both for the development and implementation of real-time systems.

4. OS/2

This chapter discusses the results of the investigation of Microsoft Operating System/2, better known as OS/2. This is a commercial multitasking, single-user operating system developed jointly by Microsoft and IBM.

4.1. Introduction

The results were obtained using the `DosGetDateTime()` system call. This fills a structure with the date and time, down to units of milliseconds. The accuracy of the milliseconds value is dependent on the interval of the system timer, which, in this case, operated at with a period of 31ms. This resolution gives rise to a standard deviation of 8.95ms due to the accuracy of the clock (as discussed in section 2.3.2). To give an indication of the relative efficiency of the Microsoft C (version 5.1) compiler used, the time taken for the Sieve of Eratosthenes benchmark was 2.186 ± 0.018 s.

4.1.1. Concurrency

There are three levels of concurrency supported by OS/2. The highest level is a **session**. Each session has a separate virtual screen, keyboard, mouse, *et cetera*. The user of the computer can switch between sessions using a hot-key. New sessions are created by means of the `DosStartSession()` system call. There are various options that allow the new session to be completely separated from the parent, or to allow the parent to detect the termination of the child; to start the new session in the foreground, or in the background; and to provide the parent with trace information. Within a particular session there will be at least one (but possibly more) **processes**. These are instances of programs loaded from disk files. The `DosExecPgm()` system call is used to create new processes. Again the system offers several options: the child can run to completion before the `DosExecPgm()` call returns with information about the termination code from the child; the child can execute concurrently with the parent process, with the option of retaining the child's termination code for examination by the parent; the child can be detached (that is, it executes in the background, without access to a screen group); the child can be loaded and then executed under the control of the parent (useful for tracing and debugging); and the child can be loaded and suspended, pending its release by the parent process (using the `DosSystemService()` call).

The lowest level of concurrency is the **thread**. This is the fundamental form of executable entity recognised by the operating system. Each process is animated by at least one thread. In addition, processes can dynamically create other threads. Each thread executes in the context provided by the process. That is, each thread has access to the global variables, files, *et cetera* of the process. Threads are created using the `DosCreateThread()` system call. This specifies the start address of the thread (usually a subroutine of the process) and an area of memory that the new thread can use as its stack.

To prevent any confusion, the term **task** will be used to refer to any of the executable entities supported by OS/2 (that is, sessions, processes and threads) throughout this chapter and whenever referring to OS/2. Similarly, the term **process** will be used in the strict sense of an OS/2 process when used in the context of OS/2.

The scheduling mechanism of OS/2 is extremely configurable. Scheduling priorities are associated with threads; thus even the threads in a single process can have differing priorities. The scheduling priority of a thread consists of a **dispatching class** and a **priority**. The dispatching class is one of three: **idle-time** class, **regular** class and **time-critical** class (with the obvious hierarchical precedence). Within each class a thread has a priority value between 0 and 31, where higher values denote higher priority. The benchmark programs were all run at a priority of 15 in the time-critical class.

Both processes and threads were used as background tasks to assess their impact on the various system calls under examination. In general, it was found that background tasks executing at a lower priority (10, time-critical) had a negligible effect and that background tasks at a higher priority (20, time-critical) resulted in the foreground task being blocked completely. Accordingly, the discussion in the sections of this chapter dealing with background tasks concentrates on those at an equal priority.

In addition to controlling the behaviour of the system using the priority value and class, the operating system can also be switched between two scheduling algorithms. In the first (**dynamic** scheduling) the operating system can adjust the priority of processes to ensure that low priority tasks are not excluded from executing completely by higher priority tasks. In addition, the foreground process receives preference over the background processes, if any. The adjustment of the priority values can be further controlled by user specified parameters. Minasi [1988, p147] gives a good discussion of the various parameters and their effects. The second scheduling algorithm (**absolute** scheduling) ensures that priorities are not adjusted by the operating system at all. The latter is obviously of greater interest for real-time applications. Consequently, all the benchmarks (except as discussed in section 4.2.6) and the simulation were performed using absolute scheduling.

In addition to these parameters, the user of the system can control the action of the memory manager. This involves both the swapping of program segments from memory to disk to provide virtual memory capabilities, and also moving segments within memory to make optimum use of the available memory. These factors will obviously have an effect on the performance of the system. The benchmarks were all run with both swapping and moving enabled. However, at no time was any sign of disk activity noted during the testing and so it would appear that swapping did not occur.

4.1.2. Interprocess Communication

OS/2 supports a wide range of interprocess communication facilities. Duncan [1989] provides a

detailed description of all the facilities and some results of benchmarks (unfortunately, he gives no details of the hardware platform utilised for the testing, rendering it impossible to make a fair comparison between his results and those reported below). The simplest of the interprocess communication facilities is the **pipe** mechanism, which is essentially the same as that found under the UNIX operating system. The `DosMakePipe()` system call creates the pipe and returns two file handles (one for reading and one for writing) for the pipe. These handles can be passed to child tasks to allow them to communicate through the pipe. Information is written to/read from the pipe in exactly the same way as for OS/2 files.

Queues provide for the communication of relatively small amounts of data between tasks. A queue is given a name which is of the same form as a file name, appearing within an imaginary directory `\QUEUES` (for example, `\QUEUES\TEST1.Q`). A queue is created using the `DosCreateQueue()` system call. This system call also specifies whether the queue is to be handled as a LIFO queue, a FIFO queue or a prioritised queue. Other tasks can gain access to the queue by making use of the `DosOpenQueue()` system call. Queue items can then be placed in the queue using the `DosWriteQueue()` system call. A queue item consists of four words of information. These may be used by the tasks in any way, although they are often taken to be a **description**, a **size** and a **far pointer** (a two word quantity). In this case, the queue allows tasks to communicate blocks of data of practically any size. The description field specifies the nature of the block of data, the size specifies its size, and the pointer allows the task receiving the queue item to access the data. The queue items may be examined by using the `DosPeekQueue()` call to copy one of the queue items. Items can be removed from the queue by calling `DosReadQueue()`. The operating system also provides facilities for querying the number of items in the queue and purging the queue (that is, deleting all items from the queue).

The third type of interprocess communication supported by OS/2 is shared memory segments. There are three differing methods of sharing segments: **giving, getting** and **naming**. The first of these involves a process explicitly placing the segment selector of a given memory segment into the selector table used by another process (often a child of the process giving the segment). Getting involves a process requesting access to a segment belonging to another process - this requires the process getting the segment to know the value of the selector for the segment, which it must obtain from the process that allocated the segment.

The last method (naming) was the one used for the shared memory operations in the benchmarks and the simulation. In this method the process creating the shared segment gives it a name when it allocates it with the `DosAllocShrSeg()` system call. The name is of the form of a file name within an imaginary directory `\SHAREMEM`. Other processes can then gain access to this memory segment by calling on `DosGetShrSeg()` and specifying the same name.

4.1.3. Synchronisation

Synchronisation is provided for under OS/2 by means of semaphores and critical sections. The latter are supported by means of the `DosEnterCritSec()` and `DosExitCritSec()` system calls. These provide for exclusive execution of a particular thread within a process, but do not provide

protection between the threads of different processes. This can be provided using semaphores. There are two types of semaphores supported by OS/2 - **RAM semaphores** and **system semaphores.** Both of these are simple binary semaphores (as opposed to counting semaphores). RAM semaphores are double-words allocated within the memory of a particular process. Only tasks that can access the semaphore can make use of it, and hence this generally restricts the use of RAM semaphores to the threads within one process. System semaphores, on the other hand, are named like files within an imaginary directory `\SEM`. One process must use `DosCreateSem()` to create the semaphore. Other processes can then use `DosOpenSem()` to access the semaphore.

The semaphore operations supported by OS/2 can be applied to both types of semaphore. There are three fundamental operations provided: **setting** a semaphore, **waiting** on a semaphore and **clearing** a semaphore. When a semaphore is set by a thread is said to be owned by that thread. Another thread that waits on the semaphore will be suspended as long as it is owned by some other thread. Ownership of a semaphore is released by clearing the semaphore. The operations that can be performed on semaphores are implemented by a set of system calls that provide the three fundamental operations and several composite operations, and also simultaneous operations on sets of semaphores.

Mutual exclusion is supported by the composite operation `DosSemRequest()`, together with the fundamental operation `DosSemClear()`. The request system call examines a semaphore and, if it is clear, sets it, otherwise it suspends the calling thread. This is illustrated by the following algorithm, which provides for exclusive access to some critical region using a semaphore named `MutEx`.

```
DosSemRequest(MutEx)
  Critical Region
DosSemClear(MutEx)
```

Synchronisation is provided by the three fundamental operations. Again, this is illustrated by the following algorithm for the classic producer-consumer problem. This makes use of two semaphores (`CanStore` which must initially be clear, and `CanTake` which must initially be set) to control access to a central buffer.

Producer

```
repeat
  Produce Item
  DosSemWait(CanStore)
  DosSemSet(CanStore)
  Place Item in Buffer
  DosSemClear(CanTake)
forever
```

Consumer

```
repeat
  DosSemWait(CanTake)
  DosSemSet(CanTake)
  Take Item from Buffer
  DosSemClear(CanStore)
  Consume Item
forever
```

This approach differs slightly from the classic use of semaphores in that the usual wait operation is provided by two separate operations (`DosSemWait()` and `DosSemSet()`) under OS/2.

4.1.4. Exception Handling

In common with most of the facilities provided by OS/2 there is a rich set of interrupt handling

and timer operations. Programs running under OS/2 can set up signal handlers to deal with three system signals (control-C and BREAK key interrupts, and program termination) and three flag interrupts (named signals A, B and C). Signals may be **held** (that is, ignored temporarily), **ignored**, **blocked** (that is, the signal is ignored and the signalling task receives an error indication) and **handled** (that is, a specified interrupt handling routine will be invoked to deal with the signal).

The timer operations provide for the suspension of a thread for a given number of milliseconds (using the `DosSleep()` system call), and two forms of interval timers - one-off and periodic. Both of these timer services make use of system semaphores to notify a task that the requested time interval has expired. The one-off timer service (provided by `DosTimerAsync()`) clears the given semaphore once only, when the requested time has elapsed. The periodic timer service (provided by `DosTimerStart()`) regularly clears the given semaphore at the requested interval. Both forms of timer service can be cancelled by calling on `DosTimerStop()`.

4.2. The Benchmarks

4.2.1. Concurrency

The multiprocessing benchmark was performed for processes, threads and sessions. In all cases it was found that there was a slight decrease in the execution time per task as the number of concurrent tasks was increased. Presumably, this is due to decreasing system overhead, but may be related to the anomaly discussed in section 4.2.5. This effect was most clearly noticed in the case of threads where the execution time per thread fell from 3.72s to 2.85s (a decrease of 23%) as the number of threads was increased from one to ten. For processes the time dropped from 2.90s to 2.76s (4.8%), and for sessions from 2.97s to 2.86s (3.7%). Comparing the times for a single concurrent task with the time taken for the sequential program (2.186s) provides an indication of the overhead due to setting up a concurrent task. This ranged from an increase of 32% (for processes) to 70% (for threads). Sessions showed an increase of 36%.

Turning to process creation times, it was found that the system constraints did not allow results to be obtained for threads or sessions. In general, allocating memory for threads to use as a stack proved to be rather awkward using Microsoft C. This was the cause of the failure of the process creation benchmark - insufficient memory was available for thread stacks. The reason for the failure of the session tests is a little less clear, but was probably due to exceeding the capacity of various system tables. It was found that the time taken to create a new process was of the order of 310ms. There was a slight increase in the time taken as the number of processes was increased. For 20 processes the creation time per process was 310.5 $\pm$ 1.9 ms, increasing to 315.24 $\pm$ 0.53 ms for 100 processes.

The effect of background tasks on process creation time was also assessed. It was found that one background process resulted in a degradation of 2.96 times and one background thread resulted in a degradation of 2.92 times (a difference of only 1.4%). Interestingly, when the

number of background tasks was increased to ten, the impact on the performance of the process creation benchmark was not impaired by a factor of ten as was found for most of the other benchmarks. The degradation rose to 24.1 times for processes and 24.5 times for threads (a difference of only 1.7%). That is, the performance dropped by a factor of 8.1 times for processes and 8.4 times for threads. In most other cases the performance dropped by ten times (the more expected result). The only other test program which showed this kind of behaviour was the unsynchronised HiHo program (see section 4.2.3). This leads to the conclusion that I/O intensive programs are less likely to be affected by increasing the number of background tasks, presumably since the operating system can take advantage of natural delays in the execution of the I/O intensive task to allow the background tasks to execute.

4.2.2. Interprocess Communication

The first test in this category (using 100 byte messages) was performed for both pipes and shared memory segments (with access synchronised by system semaphores). Queues were not used as they have the limitation that only four words of information can be passed through the queue (although, as discussed in section 4.1.2, they can be combined with the use of shared memory segments to pass larger amounts of information). Considering pipes first, for 1000 messages, the time taken per message was found to be 2.232 ± 0.021 ms (or 22.32μs per byte). This result did depend to some extent on the number of messages sent. For 100 messages the time per message was 2.22 ± 0.19 ms, rising to 2.3346 ± 0.0021 ms for 10 000 messages. Increasing the number of messages from 10 000 to 40 000 produced no further increase in the message passing time, and so this may be an asymptotic value. When the effect of background tasks at a lower priority was assessed, a slight impact (of up to 12% for ten background threads) was noticed. At an equal priority each background task produced a degradation of 112 times.

The use of shared memory produced less efficient message passing due to the overhead of the synchronisation required (for 1000 messages the shared memory version took 73% longer than the pipe version). In addition, there was no predictable pattern to the dependency between the number of messages and the time taken per message, as shown in the following table.

Number of messages	Time taken per message (ms)
100	5.27
1000	3.86
10 000	3.91
40 000	3.90

Clearly, there is a tendency towards an asymptotic value of about 3.9ms, but there is no clear relationship between the number of messages and the time taken. For 1000 messages the exact result was 3.858 ± 0.024 ms. As was the case for pipes, background tasks at a lower priority produced a slight degradation (of up to 9%). However, the effect of background tasks at an equal priority was markedly less than for the pipe case. The degradation due to such tasks was a factor of 65 times (nearly half of that found for pipes). This is probably due to the fact that the sending and receiving processes in the shared memory test are closely synchronised, and presumably the tightly controlled switching between the two processes allows the operating system to schedule the background tasks more favourably.

The second test performed in this category was the circular message passing benchmark. This was performed for both pipes and queues. In the case of pipes and sending 2600 messages, it took 6.4831 ± 0.0089 ms for a message to pass around the circle of processes, or 2.16ms per message. This agrees closely with the results found for the first test above. The effect of background tasks was also very similar - at a lower priority degradations of up to 8% were found, and at an equal priority there was a degradation by a factor of 116 per background process. The time taken was found to increase linearly with an increasing number of messages.

When queues were used for this benchmark the message passing time increased to 12.9654 ± 0.0086 ms (for a complete traversal of the circle of processes; 4.32ms per message). This is exactly double the time taken by sending the data using a pipe. The effects of background tasks were almost identical to those for the pipe version (up to 11% degradation caused by lower priority tasks and a degradation of 116 times for equal priority tasks). As for pipes, the time taken increased linearly with the number of messages.

4.2.3. Synchronisation

This test was performed for three cases: (1) using threads and RAM semaphores, (2) using threads and system semaphores and (3) using processes and system semaphores. The time taken per synchronisation operation (in milliseconds) and the degradation factor due to background tasks at an equal priority are summarised in the following table for each of the three cases.

Case	Synchronisation Time	Degradation
1	1.241 ± 0.041	41
2	1.532 ± 0.036	39
3	1.474 ± 0.041	40

As would be expected, RAM semaphores are the most efficient. However, it is surprising that system semaphores are slightly more efficient (by 4%) when used with processes than threads. This behaviour is counter-intuitive and there is no obvious reason for it. As can be seen from the table, the effects of background tasks are somewhat less than for the other benchmarks. This is especially true of the unsynchronised version of the program which only suffered a degradation of 2 times for one background task, rising to 12 times for ten background tasks. This is presumably due to the I/O intensive nature of the test, as was commented on in section 4.2.1.

4.2.4. Exception Handling

This test was performed using five different mechanisms: (1) `DosSleep()` was used to suspend the process, (2) the repetitive interval timer (`DosTimerStart()`) was used to signal a semaphore, (3) the one-off interval timer (`DosTimerAsync()`) was used to signal a semaphore, (4) a thread was used to send a signal to its parent thread and (5) a process was used to send a signal to its parent process. As usual, it was difficult to get definite results from this test. Approximate service times (in microseconds) for intervals of 20ms, 40ms and 60ms for each of the five tests

are summarised in the following table.

Test	40ms	60ms
1	360	280
2	360	540
3	890	1400
4	180	--
5	180	280

Note that all of these values are of the same order as the standard error in the results for the reasons discussed in chapter two. However, although definite service times cannot be derived from these results it is clear that the various timer operations are less efficient than the signal handling mechanisms. The third case (using the one-off timer service) is clearly less efficient than the others, and this is probably due to the extra steps needed to restart the timer each time an interval expires. Of the results given, the last set are probably the most accurate (since 60ms is of the order of two clock ticks).

4.2.5. The Effect of the Print Spooler on Execution Times

Early on in the testing program the invocation of the print spooler in the system configuration file was accidentally disabled. It was noticed that this resulted in an *increase* in the execution times of the benchmark programs. Accordingly, the main testing program was performed with the print spooler enabled. In order to investigate this phenomenon further, a second series of tests was run with the print spooler disabled. There was an average increase in the execution time of all the tests of 7.4%. Only two tests proved to be more efficient without the spooler. These were the multiprocessing test using sessions, and the process creation test, which showed improvements (decreases in execution time) of 1% and 3% respectively. The increases in execution times ranged from 2% (for the multiprocessing test using processes) to 27% (for the multiprocessing test using threads). Marked increases were also noted in the standard (sequential) Sieve of Eratosthenes (21%) and the interprocess communication test using shared memory (12%).

The reason for this behaviour is far from clear. Intuitively, one would expect the presence of the print spooler (even although inactive) to cause an increase in execution times, as it must make use of some of the resources of the operating system (in the form of entries in tables, *et cetera*). The phenomenon is clearly related to the multiprocessing facilities of the operating system, since the four tests in the concurrency category all provide the extreme cases. One possible explanation is that the algorithms used by OS/2 for the manipulation of the system tables are such that their efficiency is enhanced by the presence of extra processes. Without any knowledge of the exact nature of the algorithms and the structure of the tables it is impossible to draw any firm conclusions, but the effects may be similar to those achieved by the use of a "header" node (or sentinel, as it is sometimes called [Riley, 1987, p238]) in linked list structures [Knuth, 1973, p272]. That is, the presence of an item in the list (or table) may simplify the processing of the list as it gets around the potential special case that arises when the list is empty. As already mentioned, this is extremely dependent on the structures and the algorithms

that are employed.

4.2.6. The Dynamic Scheduling Algorithm

In order to assess the effectiveness of the absolute scheduling mechanism used for the benchmarking process, a second series of tests was run using the dynamic scheduling algorithm. The print spooler was enabled during this set of tests. The change in execution time ranged from a decrease of 6% (for the process creation test) to an increase of 9% (for the multi-processing test using threads and the interprocess communication test using shared memory). Overall, there was an average increase in execution time of 3.1%.

Of more interest is the change in the standard deviation for the tests, as this gives an indication of the determinacy of the scheduling algorithm. This is of paramount importance in real-time applications, of course. In only one case did the standard deviation decrease when the dynamic scheduling mechanism was used. This was for the circular message passing test using pipes, where the standard deviation was 0.01s with absolute scheduling and 0.00s with dynamic scheduling (hardly a great change). Four of the thirteen tests carried out in this series showed exactly the same standard deviations. Of the remaining eight tests which showed increases in the standard deviation, the greatest was for the process creation benchmark (which increased from 0.04s to 2.27s). This was also the test which showed the greatest decrease in execution time, as already mentioned. However, the execution time under the absolute scheduling algorithm is still within the error margin of the lower time, due to the large increase in the standard deviation. The next largest increases in the standard deviation were for the unsynchronised HiHo program (which went from 0.01s under absolute scheduling to 0.06s under dynamic scheduling, with no change in execution time) and the synchronised version using threads and RAM semaphores (from 0.01s to 0.07s, with a two percent increase in execution time).

From these results it is clear that, in general, the dynamic scheduling algorithm is not as deterministic as the absolute scheduling algorithm. This is especially true of I/O intensive processes. The absolute scheduling algorithm is both more efficient on the whole, and provides a far better level of determinacy. Thus, it is to be preferred for real-time applications.

4.3. The Process Control System and Plant Simulation

The OS/2 implementation model for the widget manufacturing plant simulation and the control system is shown in Figure 16. This implementation consisted of a set of eleven tasks. The plant simulation was made up of three threads in a single process - one running the simulation of the reaction vessel and mould filler, and two for the polishing machines. The control subsystem was comprised of four processes - one controlling the reaction vessel, one the mould filler, and two the polishers. The operator interface and logging facilities were provided by four tasks - one process with two threads provided the operator input and status screen handlers; two other processes provided the logging and performance monitoring facilities. Communication

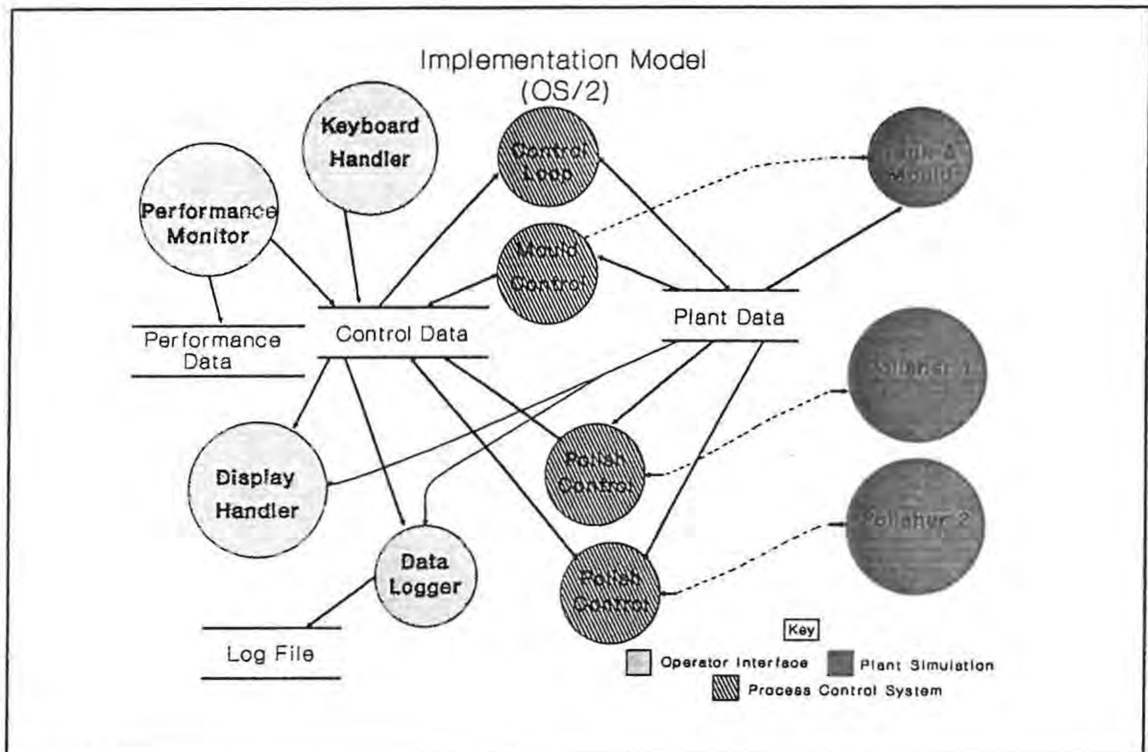

Figure 16

between the three subsystems was provided by means of two shared memory segments, with system semaphores providing exclusive access where necessary. The synchronisation of the polishers with their respective control systems was also provided by system semaphores. The mould release was implemented by using flag interrupt A. The plant simulation was run at a priority of 15 in the time-critical class. The control system ran at a priority of 0 in the same class. The operator interface and data logger were placed in the regular dispatching class with a priority of 0, and the performance monitor was run at a priority of 15 in the idle-time class. The absolute scheduling algorithm was used, and the print spooler was enabled.

The simulation ran very well, and the facilities provided by OS/2 were well suited to writing such an application. The performance monitor produced an average result of 2.41s/2.19s. This gives a performance index of 0.91, which is quite impressive.

4.4. Qualitative Assessment

OS/2 is a large operating system and attempts to be very general, in providing many options that can be controlled by the user of the system (such as alternative scheduling algorithms, configurable memory management, *et cetera* - Minasi reports a comment made to him that "OS/2 sounds good, but I'm afraid it's going to force me to become a systems programmer in my spare time" [1988, p147]!). The result of this size and generality is, as is often the case, increased complexity. This is evident in the large number of system calls available (over 120 basic system calls and a further 82 for screen, keyboard and mouse handling in version 1.00). In addition to the sheer number of system calls, the interface to many of the calls is complicated by various

options that can be set and parameters that must be supplied. A further source of complexity from the programmer's viewpoint is that the naming of the system calls tends to be rather haphazard. For example, most of the semaphore system calls are named `DosSemXXX()`, but there are also `DosCreateSem()`, `DosCloseSem()`, `DosOpenSem()` and `DosMuxSemWait()`. This may seem a minor issue, but makes remembering the names more difficult (for example, one instinctively tends to write `DosSemCreate()` as this is the general naming pattern) and complicates locating the system calls in an alphabetically ordered manual, or list of system calls. In a similar vein, the system calls for instantiating semaphores, queues and files are named `DosOpenSem()`, `DosOpenQueue()` and `DosOpen()` respectively. On the other hand one sets up a pipe by calling `DosMakePipe()` (in fact, this is the only system facility that is made as opposed to created or opened!).

The recognised approach to large scale software development for OS/2 is to obtain the Microsoft Software Developers Kit (SDK) for OS/2. This is a fairly expensive package, and so this evaluation was performed by purchasing OS/2 version 1.00, and Microsoft C version 5.1 separately. The drawback to this approach is that the C compiler documentation contains no information about the operating system calls. This necessitates purchasing some extra documentation. "The Programmer's Essential OS/2 Handbook" by Cortesi [1988] was used for this evaluation and proved to be quite adequate in lieu of the SDK documentation. A brief comparison of these facilities with the Microsoft SDK appeared to indicate that it was just as satisfactory for most development work. In fact, reports from people working with the SDK suggest that extra sources, such as Cortesi's book, are extremely desirable anyway, as the SDK does not give much in the way of discussion or examples.

Turning to the utilities available under OS/2, the Microsoft C compiler came with its own editor and the CodeView debugger. The editor was used throughout the program development and was found to be quite powerful, although it is not easy to use, especially at first. Another consideration to be borne in mind is that OS/2 is being marketed as a general purpose operating system and is aimed at a very large market. If it reaches widespread acceptance then there should be an increasing proliferation of editors, language systems (already Modula-2 compilers are available, and Ada compilers are being announced) and utilities. The debugger was not used during the evaluation at all. However, it is a port of the widely used MS-DOS version of CodeView and should be as acceptable as that version.

OS/2 also suffers from its position as the successor to the MS-DOS operating system. The result of this has been that the facilities offered are, in most cases, simply modelled directly on the equivalent MS-DOS facilities, with a few minor enhancements. This is especially noticeable in the user interface, where the system falls considerably short of the flexibility and power offered by the UNIX shell and other advanced command interpreters. There is also a lack of utilities for monitoring and controlling processes executing in the background.

4.5. Conclusions

OS/2 is quite suitable for real-time applications. Its provision of alternative scheduling algorithms

and configurable system parameters make it possible (although not necessarily easy) to tailor the system to exhibit the desired characteristics. In particular, the absolute scheduling mechanism provides the determinacy necessary for real-time programs. The system calls provide all the necessary facilities for the construction of multitasking applications with the need for interprocess communications, synchronisation, event handling, *et cetera*. However, the complexity of the system means that there is a fairly steep learning curve that needs to be overcome when first using the system. Hopefully, as the use of OS/2 spreads, an increasing number of languages and utilities will also become available.

5. QNX

This chapter discusses the results of the investigation of QNX. This is a small multitasking, multi-user executive from Quantum Software Systems.

5.1. Introduction

The results were obtained using the `get_ticks()` system call available under QNX. This reports time in units of system clock ticks, where each clock tick represents an interval of fifty milliseconds. This resolution gives rise to a standard deviation of 14.4ms due to the accuracy of the clock (as discussed in section 2.3.2). To give an indication of the relative efficiency of the QNX C compiler the time taken for the Sieve of Eratosthenes program was 2.380 ± 0.039 s. The following sections provide a brief overview of the facilities offered by QNX. Johnstone [1988], Hildebrand [1988] and Elfring [1987] all give slightly more detailed discussions of the operating system.

5.1.1. Concurrency

The `create()` system call used to spawn child tasks under QNX has three options. The first option allows a child task to be created and the call returns when the child terminates. The second allows both tasks to continue executing concurrently, but the child task will be terminated if and when the parent dies. The last option allows the two tasks to continue executing concurrently, but independently, and in this case the death of the parent will *not* result in the termination of the child task. This allows for considerable flexibility in constructing systems with multiple processes. The `create()` system call invokes a named program (on secondary storage) as the child task. (Note that this corresponds to the `fork()`-`exec()` combination commonly used under UNIX.) There is also a `fork()` system call that works in much the same way as in UNIX (that is, by creating a duplicate of the parent task from memory).

Finally, QNX makes use of a prioritised round-robin task scheduling mechanism. The highest priority task available is always chosen to run. Priorities are not changed by the operating system, but a task may change its own priority dynamically. If several tasks are ready to run at a given priority level then round-robin scheduling is used to give each task an equal portion of processing time. If a higher priority task becomes ready to run it will preempt a lower priority task that is currently executing. This was noted during the assessment of the system, as background tasks at a higher priority resulted in the benchmark task(s) being blocked completely, and background tasks at a lower priority had no effect on the benchmark times. Accordingly, the discussion in the following sections concentrates on the effects of background processing at an equal priority.

This approach to task scheduling is ideal for real-time applications as it allows high priority interrupt handling tasks to respond to interrupts as soon as they occur.

5.1.2. Interprocess Communication

The main facility provided for interprocess communication under QNX is synchronous message passing, with non-blocking reply. In this model of communications, the sending task is blocked until the receiving task is ready to receive, or *vice versa*. When both tasks are ready to communicate the data is transferred and the receiving task returns from the `receive()` system call. It then continues to execute and must call the `reply()` system call in order to transfer a reply back to the sending task whereupon the sending task returns from the `send()` system call. In this way both communication and synchronisation are provided for by the operating system. If several tasks send messages to a single receiving task the messages will in general be received in the order that they were sent. However, provision exists for the receiving task to specify the task identifier of the sending task, in which case only a message from that task will be received irrespective of its position in the queue. This approach is very similar to that used by the Thoth operating system [Cheriton *et al*, 1979] developed at the University of Waterloo in the mid to late 1970's. (QNX is reported to have been developed from work done at the University of Waterloo [Hildebrand, 1988, p36], suggesting that it is, in fact, a descendant of Thoth.)

In addition to synchronous message passing QNX has so-called **ports** which provide for task identification, non-blocking communication and simple semaphore facilities. QNX supports task identification through the operations `attach()` and `detach()` for ports. The action of `attach()` is to claim the port for a task, if no other task has already attached it. If another task has already attached the port then the `attach()` system call returns the task identifier (TID) of the attached task. The action of `detach()` when called by the task that has attached the port is to free the port for use by another task. If a task which has not attached a port attempts to detach the port then the attached task is identified, just as for the `attach()` system call. This provides for task identification, since a task that needs to make its identity known to other tasks, such as a database server, can attach a specified port. Any other tasks that need to use the services provided by the database server can then attempt to detach the port and will be informed of the TID of the database server. They can then send messages to the database server in the usual way.

Non-blocking communication is provided for by **signalling** a port. The `signal()` system call causes the specified port to send a message (with no data content) to the task that has attached the port. The task making the `signal()` system call is not suspended. The receiving task will receive the message as soon as it performs a `receive()` or an `await()` system call. The message sent by the port is queued ahead of any other messages in the message queue and so will be received first by the receiving task. Multiple signals can be sent using the same port and they will all be queued (the `csignal()` system call can be used to send a signal only if there is no signal already queued). These features allow ports to be used to perform the function of semaphores. The final point about signals is that they may be performed by interrupt handlers, allowing hardware events to be monitored by tasks.

5.1.3. Synchronisation

The implementation of the `attach()` and `detach()` system calls for ports also provides for simple, binary semaphore facilities. If many tasks need to access a single resource, such as a printer, a convention can be introduced that a task must attach a particular port before attempting to access the resource. When the task is finished with the resource it can detach the port allowing any other task that requires the resource to attach the port and use the resource.

5.1.4. Exception Handling

QNX defines 32 exceptions that may be set on a task. The first 16 are reserved for use by the operating system, but the second 16 are available for general use. These exceptions are very similar to UNIX signals. The implementation of exceptions is quite flexible and makes use of four fields, called the `permit`, `allow`, `pending` and `function` fields. If an exception occurs that is not set in the `permit` field then the task is simply terminated (this is usually the default for all exceptions). If the exception has been permitted then it is checked against the `allow` field. If the exception is disallowed then it placed in the `pending` field, which can be examined by the task to see if any exceptions have occurred (this allows exceptions to be polled). Finally, if the exception has been allowed then the function whose address is found in the `function` field is called (if no exception handling function has been specified then the task is terminated at this point). More than one exception can be set on a task at once, in which case the above steps are carried out in parallel. Exceptions may arise from hardware events (such as the BREAK key being pressed), or may be set by one task on another (or on itself).

The `set_timer()` system call is used to perform several timer functions. The simplest is the use of the call to suspend the process for a given interval. In addition the calling process can request that it be forced into the ready scheduling state, have an exception set upon it, or have a port signalled. The time interval is specified in terms of clock ticks (units of 50ms). In addition, an absolute time at which the requested event will take place can be specified.

5.2. The Benchmarks

5.2.1. Concurrency

Two versions of this benchmark were implemented. The first used the fork construct, and the second used the create construct available under QNX. Comparing the results from these tests with the time taken by the sequential Sieve of Eratosthenes algorithm, the overhead due to the multiprocessing facilities can be measured. For the `fork()` system call the overhead was subject to a large uncertainty and so could not be measured accurately. The large uncertainty was due to the fact the overhead was extremely small, and, in fact, the reported execution times are less than those for the sequential form of the algorithm. The overhead that was measured for the `create()` system call was 268 ± 84 ms.

For both versions of the program, it was found that the execution time increased linearly as the number of tasks increased. The version using `fork()` was found to be marginally more efficient. This is not surprising, as the `fork()` operation is carried out entirely in memory, whereas the `create()` function involves accessing the disk. However, the difference is almost negligible (about 3 to 4%).

The second benchmark (that is, the process creation test) was also performed using both the `fork()` and `create()` functions, and again, the `fork()` version was found to be more efficient (in this instance 180% faster than `create()`). However, the version using `fork()` showed a slightly steeper increase with the number of child tasks created, than the create version. The results were that it took 33.85 ± 0.35 ms to create a child task using `create()` and 12.15 ± 0.38 ms using `fork()`. (See section 5.2.5, also.)

For the second test the effects of background processing were assessed. A single background process (the Sieve of Eratosthenes) was found to degrade the performance of the `fork()` version by a factor of 720%. The degradation in the `create()` version was approximately 210%. This version showed very strange behaviour when background tasks were present (see section 5.2.5). In both cases the degradation was found to increase almost linearly as the number of background processes increased. For ten background processes the results were degradations of 8100% and 2900% respectively.

5.2.2. Interprocess Communication

For the 100 byte message benchmark the time taken to send a message was found to be 0.735 ± 0.037 ms (or 7.35μs per byte). The time taken was found to be independent of the number of messages sent. For the circular (one byte) message passing test it was found that it took 2.0385 ± 0.0055 ms to send a message around the circle. This corresponds to 0.68ms per message, which agrees quite well with the results of the first test.

The effect of executing simultaneous background tasks was measured for both tests. For the first benchmark the degradation was found to depend on the number of messages sent. The degradation was as low as 146 times for ten messages. (For one message a degradation of 80 times was measured, but the uncertainty in the results at this level make it impossible to state this result with any certainty.) An asymptotic value for the degradation factor of 270 times was found as the number of messages increased. In the second benchmark the presence of simultaneous background tasks was found to cause a degradation of 147 times for 2600 messages, which agrees with the value found in the first test for 1000 messages. A degradation of 131 times was measured for 260 messages.

5.2.3. Synchronisation

In this case the `await()` and `signal()` functions were used to synchronise the two tasks. It was

found that the synchronisation caused a degradation of 1.000 ± 0.068 ms per synchronisation operation (the time taken by the tasks increased by 26% compared to the unsynchronised version).

When the effect of background tasks was assessed it was found that the unsynchronised version suffered a degradation of 25.8 times, and the synchronised version 81.9 times. This corresponds to a time of 300ms per synchronisation operation per background task. This suggests that each of the background tasks was given a time slice of 300ms every time there was a task switch between the benchmark tasks.

5.2.4. Exception Handling

Two different versions of the exception handling test were used. The first used the `set_timer()` call of QNX to produce a **signal** at a specified interval, and the second used the `set_timer()` function to generate an **exception**. In both cases the exception handler was given a higher priority than the timer task, and the interval between exceptions was one system clock tick (50ms). An exception service time of 1.7 ± 1.5 ms was measured for both versions of the test.

5.2.5. The Effect of Disk Activity on Task Creation

For almost all of the tests described above the results were very consistent (shown by the fact that the standard deviations were very small). The one exception was the task creation benchmark which used the `create()` system call to create empty child tasks from secondary storage, when executed together with background tasks. The results (in system clock ticks) for the initial series of ten tests are given below, together with the corresponding figures for the `fork()` system call for comparison.

	create	+b/g task	fork	+b/g task
	67	201	24	200
	68	201	24	200
	67	201	25	200
	68	228	24	200
	68	218	24	200
	68	201	25	200
	67	201	24	201
	68	201	24	201
	68	201	25	200
	68	257	24	200
Mean:	67.7	211.0	24.3	200.2
Std Dev'n:	0.42	18.39	0.47	0.42

Note the extremely high value for the standard deviation for the `create()` system call with one background task (at the same priority). To gain further insight into this phenomenon, a further series of 500 tests was run for three different cases: `create()` with no background tasks, with one background task and with ten background tasks. The results were as follows:

a) The `create()` system call with no background tasks: The values (in ticks) ranged from 68 to

70 (a range of 3 ticks, or 150ms, or 4.4%). The frequency distribution was as follows:

68 ticks	130
69 ticks	367
70 ticks	3

b) The `create()` system call with one background task: The results ranged from 200 to 261 (a range of 62 ticks, or 3.1s, or 30%). The frequency distribution had a clear peak around at 200 ticks (365). This fell immediately to a frequency of 5 for 201 ticks, and then varied from frequencies of 0 to 11 (recorded for 249 ticks). At the upper result of 261 ticks the frequency was 1.

c) The `create()` system call with ten background tasks: The values ranged from 2007 to 2022 (a range of 16 ticks, or 800ms, or 0.8%). Although the range of results was smaller, the frequency distribution had a greater spread, as shown by the graph in Figure 17.

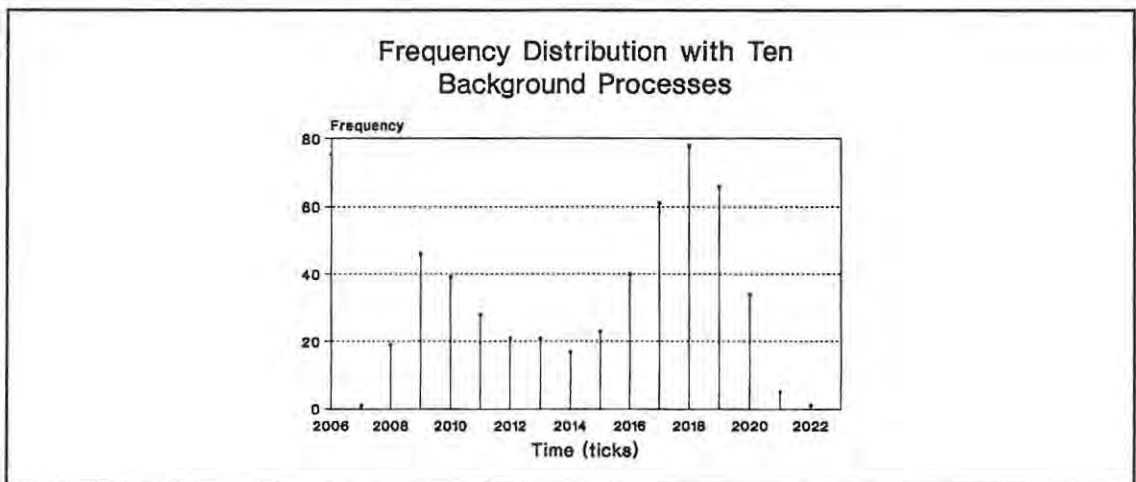

Figure 17

In addition, the effect of using a virtual disk (rather than the hard disk used in the other cases) was studied. It was found that there was no significant deviation from the mean in this case. The anomaly noted here is related to the use of the hard disk.

This shows that there is some interaction between the presence of background tasks and the disk activity associated with the benchmark. The most likely explanation is that the operating system is performing a task switch when there is a delay in disk access, allowing one of the background tasks to proceed. The amount of the delay would appear to be dependent on the position of the disk heads, and at certain times this results in the background tasks getting a larger amount of processing time than at others.

5.2.6. Degradation Functions

It was noticed when studying the results for some of the tests performed with background tasking

that there were distinct functions that described the behaviour of the tests. This was apparently due to the regular nature of the functions when the results were expressed in units of system clock ticks. The functions depend on the number of iterations of the test performed (**n**, for example, the number of messages sent) and also on the number of background tasks (**b**). As the benchmarks have little direct application to the real world these functions are of limited value in themselves. However, they do provide a very good indication of the effect of background tasks on the various system calls studied. The functions are given in the following table, where **t** is the time taken for the test in units of ticks.

	b > 0	b = 0
100 byte messages:	t = b(4n + 4)	t = 0.015n
Circular messages:	t = b(6n + 12.5)	t = 0.043n
HiHo's (unsynch):	t = b(2n + 10)	t = 0.08n
HiHo's (synch):	t = b(8n + 12)	t = 0.10n

Note: In the case of the circular message passing test the constant part of the background tasking function was not as clearly defined as for the other functions, and ranged from 12 (for 10 background tasks) to 13 (for 1 background task).

The unsynchronised version of the HiHo program probably represents the best case for background tasking as it only made use of system calls for screen output. In each of the other tests there were system calls that resulted in tasks being suspended (waiting for messages, etc) and so there were more opportunities for the operating system to allocate the CPU to a background task. In all cases the mere presence of a background task resulted in a fairly severe degradation (of 10 to 100 times), but as mentioned previously, the background tasks used here are extremely computationally intensive and make no use of system calls. In a more realistic situation (such as a process control system) this would almost certainly not be the case.

5.3. The Process Control System and Plant Simulation

The implementation model for the widget manufacturing plant simulation and its associated control system under QNX is shown in Figure 18. This implementation consisted of a set of twelve tasks (three for the simulation, four for the control system, three for the user interface and two for communications buffering) comprising over 1400 lines of C code. Again, the ideal decomposition of the system has been distorted somewhat. As in the case of the other operating systems a subsystem has been introduced to handle the interaction with the operator. The functions of the Tank, Mould Filler, Heater, Level Sensor, Thermometer and Scale transformations have also been combined into a single Tank-and-Mould task. (The justification for this approach was given in the discussion of the implementation of the simulation under XENIX.) The communication between the subsystems is one of the more interesting aspects of this system. As stated in section 2.2.2.3, the communication should be asynchronous, but the main communication facility available under QNX is synchronous message passing. In order to implement the communication in an asynchronous manner an active communication subsystem was adopted. This makes use of tasks which buffer the data, and respond to messages requiring data to be updated, or messages requesting data values. This is a fairly standard technique for handling such situations, and corresponds to the general concept of a proprietor task discussed by Gentleman [1981]. In the case of the polishers and their control tasks there is a need for

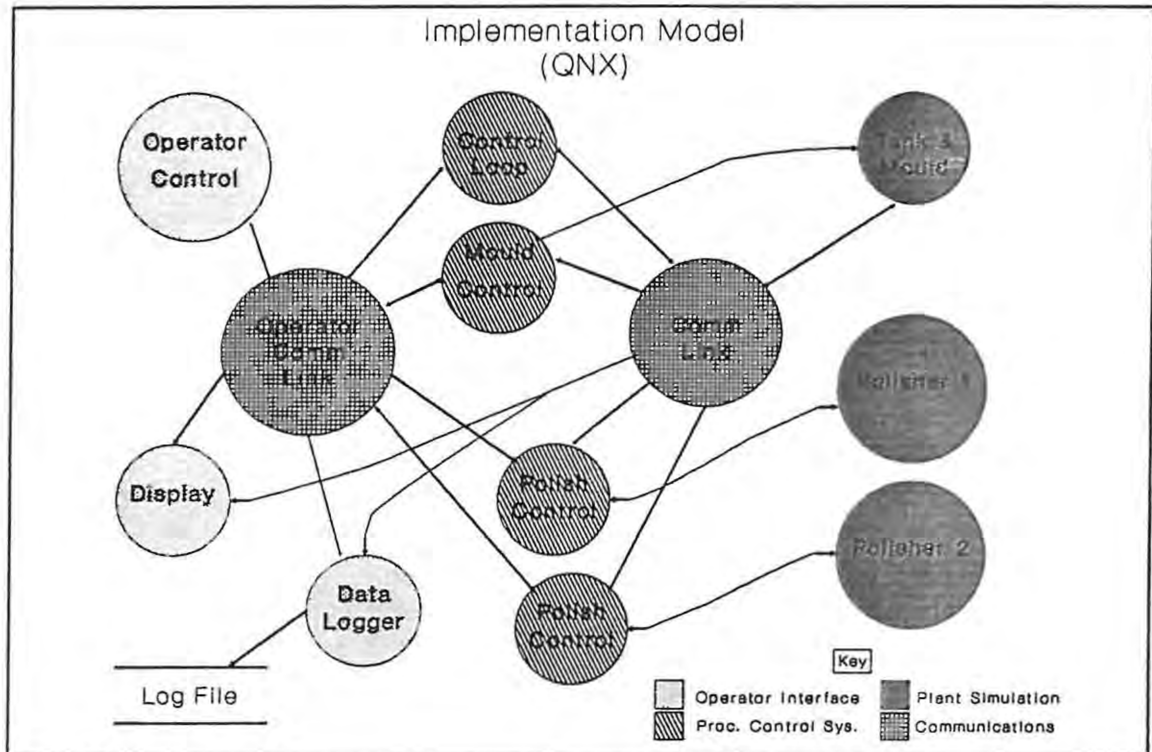

Figure 18

synchronisation, and this was provided by using messages with no significant data content.

The simulation ran well, and it was found that QNX provided some very useful facilities for use in the user interface (for example, simple box drawing and bar graphing routines). The multitasking and interprocess communications facilities available under QNX were found to be well suited for this kind of application. In the course of implementing this system considerable experience was gained in the use of QNX, and the available utilities. The qualitative evaluation of the operating system based on this experience is presented in the next section.

5.4. Qualitative Assessment

QNX performed very well throughout the benchmarking process and the implementation of the simulation. With the exception of the disk activity factor discussed in section 5.2.5 of this report, the facilities offered by the system performed efficiently and predictably. However, QNX is a relatively recent system and lacks the solid "feel" of more established operating systems. In particular the utilities such as the editor and debugger do not match the sophistication of those found under UNIX, or under MS-DOS. Also, the editor (which is partially line oriented) is far from the state of the art. Another factor arising from the relative youth of this product is the lack of certain useful (if trivial) utilities found on systems such as UNIX or MS-DOS (for example, `more` and `cls`).

The documentation was found to be weak in several areas. Notably there is no index, and the

level of detail is lacking in several areas. The general organisation of the manuals also leaves much to be desired. For example, the entry in the Libraries manual (the reference manual for the system calls and library routines) for the `set_exception()` system call informs one that "Exceptions are complex to explain. They are described in some detail in the QNX manual"! The QNX manual to which this refers is a reference guide which describes setting up the system, terminal handling, files and directories, the use of the shell, tasking (including the section on the use of `set_exception()`), networking, "Tips on Using QNX" and the use of the on-line update system. This manual appears to be a conglomeration of information that the authors overlooked in other sections, or for which there was no more specific location. The manual entries for the system calls and library routines are also rather brief, especially in their description of the header files used. Finally, nowhere is there any attempt to summarise the system's error messages or to expand on the meaning and/or possible cause of errors which may arise.

5.5. Conclusions

QNX can be classed as rather a rough diamond. It has all the necessary features and the performance required for serious real-time systems, and adds facilities which greatly simplify the construction of user interfaces, *et cetera*. However, the utilities and documentation leave a lot to be desired, and much further refinement is required in these areas. The system is usable in its present form, but considerable effort must be made to master it. In addition, the use of the system for large scale development work will probably require some time devoted to writing utilities to overcome the deficiencies inherent in the system.

6. FlexOS

This chapter discusses the results of the investigation of FlexOS. This is a commercial multi-tasking, multi-user real-time operating system developed by Digital Research Inc.

6.1. Introduction

The results were obtained using the `s_get()` supervisor call to obtain the time and date table data, which includes the time since midnight in milliseconds. The documentation does not specify the resolution of the system clock, but it is clear from the results that the resolution is 31ms to 32ms. Taking 31.5ms as the effective resolution gives rise to a standard deviation of 9.1ms due to the accuracy of the clock. To give an indication of the relative efficiency of the MetaWare High C compiler used, the time taken for the Sieve of Eratosthenes was 1.9690 ± 0.0091 s.

6.1.1. Concurrency

The `command()` supervisor call used to create child processes under FlexOS has two forms. Both of these forms execute a program from secondary storage; there is no provision for replicating a process in memory, as in the UNIX `fork()` system call. The first form of this supervisor call is the synchronous `s_command()` form, which can be used to chain programs together, or to run one program as a subroutine of another. Neither of these approaches was used in the evaluation of FlexOS. The second form of the supervisor call is the asynchronous `e_command()` version. This allows an independent child process to be created. The parent process has two options for handling the termination of the child process. The first is to specify a software interrupt handler (a function) which will be executed when the child dies, and the second is to use an **event mask** (returned by the `e_command()` supervisor call) with the `s_wait()` supervisor call to wait for the death of the child process.

The FlexOS scheduling mechanism is claimed to use prioritised, round-robin approach. The documentation states that the highest priority task that is ready to execute will be selected to run. If there are several processes with the same priority then each process is given an equal portion of processing time. However certain of the tests performed showed that this was not always true (see section 6.2.5). The default priority value for a process is 200. Background processes at a higher priority were assigned a priority of 190 and those of a lower priority were assigned a value of 210 for the purposes of this evaluation. It was found that executing background tasks at a higher priority resulted in the main process being blocked completely, as was expected. Accordingly, the discussion in the subsequent sections concentrates on the effects of background processes at lower and equal priorities.

6.1.2. Interprocess Communication

FlexOS provides both pipes and shared memory areas for use in interprocess communication. FlexOS pipes are created by the `s_create()` supervisor call. This is the same supervisor call that is used to create files, the difference being that pipes are created on a pseudo-device called `pi:`. A pipe is then accessed in exactly the same way as a file. Shared memory segments are also created using the `s_create()` supervisor call (on the `sm:` pseudo-device). Once a shared memory segment has been created multiple processes can access it by calling on the `s_get()` supervisor call to obtain a logical address for the area in the process address space. In most instances it is necessary to enforce mutual exclusion on the contents of a shared memory segment, and this can be done under FlexOS by using semaphores (see section 6.1.3).

6.1.3. Synchronisation

Synchronisation under FlexOS is provided by means of pipes with no capacity for data, which act as simple binary semaphores. These are referred to as semaphore pipes. Performing a read operation on a semaphore pipe obtains the semaphore if no other process owns it. Similarly, performing a write operation on a semaphore pipe releases it for another process to obtain. If a process attempts to obtain a semaphore that is owned by another process it has the option of being suspended until the semaphore is released, or continuing. In this way semaphore pipes provide for synchronisation and mutual exclusion.

6.1.4. Exception Handling

FlexOS provides both synchronous and asynchronous timer functions. The synchronous form (the `s_timer()` supervisor call) allows a process to suspend itself for a given number of milliseconds. The asynchronous `e_timer()` supervisor call can be used to invoke a software interrupt handler after a given number of milliseconds (this is useful for providing a timeout facility), and also returns an event mask which can be used with the `s_wait()` supervisor call to wait for the time period to elapse. Both the synchronous and asynchronous forms were used for the evaluation, the latter with the use of the event mask, rather than the software interrupt mechanism.

6.2. The Benchmarks

6.2.1. Concurrency

The first benchmark (running multiple copies of the Sieve of Eratosthenes) produced some interesting results. A single iteration of the Sieve, using a purely sequential program took 1.969s. However, a concurrent version using only a single process (that is, a parent process which created a single iteration of the Sieve as a child process and waited for it to terminate) took

2.541s, a 29% increase. This suggests that there is a fairly large overhead (572 ± 45 ms) associated with the creation of a child process and the termination handling. As the number of child processes was increased the execution time per child process fell slightly, reaching 2.380s for the case of ten child processes. This still represents an increase of 21% over the sequential execution of the Sieve algorithm, and an overhead of 411 ± 14 ms.

The second benchmark performed in this section (to measure process creation times for 100 child processes) gave a result of 289 ± 12 ms per process. When comparing this result with the other operating systems, it must be remembered that the disk drive used for the evaluation of FlexOS had a longer seek time than that used for the other tests. The effect of varying the number of child processes was assessed and showed that the process creation time *increases* as the number of child processes increases. For the creation of twenty child processes the process creation time dropped to 189 ± 19 ms. The effects of background processing were also assessed and it was found that a single background process at the same priority as the benchmark program caused a degradation in the execution times of 30%. A single background process at a lower priority caused a negligible degradation of 2%. When the number of background processes was increased to ten it was found that the benchmark process was blocked for the case of background processes of equal priority. In the case of background processes at a lower priority the benchmark ran for seven iterations (out of the usual ten) and on the eighth iteration the operating system crashed with a PANIC error. The results for the first seven iterations were extremely low (of the order of 16% of the execution times with no background processes), suggesting that the operating system was terminating the benchmark program (with no diagnostic messages). This kind of problem is extremely undesirable in a real-time operating system, as it suggests that there may be severe problems with the scheduling algorithms. This subject is discussed further in section 6.2.5.

6.2.2. Interprocess Communication

For the hundred byte message benchmark using pipes the time taken to send a message was found to be 3.484 ± 0.067 ms (or 34.84μs per byte), when 1000 messages were sent. When the effect of increasing the number of messages was assessed it was found that the time per message decreased slightly, as shown in the following table.

Number of Messages	Message Time (ms)
10 000	2.395
20 000	2.334
30 000	2.314
40 000	2.304

Note too how the time has dropped considerably from the value found for 1000 messages. From the figures in the table, the time taken appears to be approaching an asymptotic value of just less than 2.3ms. This test was repeated using a shared memory segment and semaphore pipes for synchronisation. As could be expected the time taken to pass a message was somewhat longer, at 5.331 ± 0.056 ms (or 53.31μs per byte) for 1000 messages. In this case too, a steady decrease in the time taken was noted as the number of messages was increased, as shown in the following table.

Number of Messages	Message Time (ms)
10 000	3.989
20 000	3.914
30 000	3.889
40 000	3.877

Again, these values appear to tend towards an asymptotic value (of about 3.85ms).

The second test performed in this section was the circular message passing benchmark. This was performed using pipes only and produced a result of 2.1222 ± 0.0037 ms. This agrees closely with the result for the 100 byte messages. As in the case of the 100 byte messages it was found that the time taken to send a message dropped off slightly as the number of messages was increased.

For all three series of tests performed, the effects of the presence of background tasks was measured. The following table shows the percentage by which the execution time was increased for a single background process at the same priority as the benchmark process and at a lower priority.

	Equal Priority	Lower Priority
Pipe Messages	598%	1%
Shared Memory Messages	463%	1%
Circular Messages	366%	1%

From this it can be seen that the presence of background tasks at an equal priority causes a fairly severe degradation in execution time. Background processes at a lower priority have a consistent, but negligible, impact on execution times. The results that were obtained for ten background processes will be discussed in section 6.2.5.

6.2.3. Synchronisation

The time taken per synchronisation operation, using pipe semaphores, was found to be 2.78 ± 0.16 ms. The addition of a single background process at an equal priority increased the execution time by 27%, and resulted in a time of 2.82 ± 0.93 ms per synchronisation operation. Except in this one case, the synchronisation produced very strange results in the presence of background processes, as is discussed in section 6.2.5. In fact, this result is extremely low, when compared to the results obtained for the other operating systems, and may indicate that the operating system was not working as expected in this case either.

6.2.4. Exception Handling

FlexOS, like the other operating systems, did not provide clear results for this test. For small intervals the apparent times were very low (of the order of 20 to 30 μs). For larger intervals (of the order of the resolution of the system clock) the service times were found to be 560 ± 570 μs and 920 ± 850 μs for the synchronous and asynchronous versions respectively.

6.2.5. Problems with the Scheduling Mechanism

When investigating the performance of the benchmark programs in the presence of background processes it became apparent that the claimed round-robin scheduling mechanism does not always produce the expected results. For almost all the benchmarks it was found that the presence of ten background processes, executing at the same priority as the benchmark program, caused the benchmark program to receive no share of the available processor time. (In one or two instances it was even found that background processes at a *lower* priority caused the benchmark process(es) to be ignored by the operating system). As has already been mentioned, the use of the Sieve of Eratosthenes reflects a worst case scenario, as far as background tasks are concerned, since it makes no use of system calls or I/O facilities, and thus will use the maximum amount of CPU time made available to it by the operating system. In order to investigate this problem in more detail, the background Sieve program was altered to include some I/O. Specifically, the following line was added in the outer most loop of the Sieve:

```
putchar('*');
```

This resulted in the background processes all writing out an asterisk approximately once every two seconds, allowing the assessment of the effect of background processes with I/O. The assessment of the effects of background processes was then repeated for some of the benchmark programs, using the case of the priorities being equal. It was found that this change allowed the benchmark tasks to execute in the presence of ten background tasks, where they had been blocked before (that is, when the background processes performed no I/O). The results are summarised in the following table (all times are in units of seconds).

Benchmark	1 B/g Process		10 B/g Processes
	No I/O	With I/O	With I/O
Process Creation	37.533	39.881	125.987
100 Byte Messages (pipe)	24.310	26.278	279.438
100 Byte Messages (shared memory)	30.006	29.953	297.384
Circular Messages	77.119	76.594	743.825

As can be seen, the results for a single background task agree very closely (to within 8%), whether the background tasks performed I/O or not. With the exception of the process creation benchmark, the times for ten background processes are all of the order of ten times the time for one background process, which is much as expected. The result for the process creation benchmark with ten background processes is of the order of three times the result for a single background process. Even the degradation noted for a single background process, when compared with the result with no background processes, is much less than for the other benchmarks. The probable cause of this is that the process creation benchmark is diskbound, whereas the other tests are more CPU-bound. This would result in the background processes taking up time while the benchmark process is waiting for disk I/O to complete, which is necessary even in the absence of background processes.

The fact that predictable results were obtained when the background processes made use of I/O, but not when the normal background processes were used, shows that the round-robin scheduling

mechanism that FlexOS is claimed to employ does not really work correctly. It is clear that CPU-bound tasks can "hog" the processor, excluding other processes executing at the same priority. While this is definitely undesirable, it may not necessarily be as serious as it first appears. This situation (that is, having completely CPU-bound background tasks) is unlikely to arise in practice, since most processes will perform I/O to display or store results, or will make use of system calls to communicate or synchronise with other processes. In this case the scheduler should allocate the processor fairly.

6.2.6. The Effects of Postlinking

FlexOS provides a utility which allows programs to be postlinked. This operation involves the resolution of certain segment and offset addresses which are otherwise calculated only when a program is loaded for execution. Making use of postlinking thus decreases the time taken by the operating system to load a program. In order to assess the gains to be had from this feature the process creation benchmark and the concurrent execution of several Sieves of Eratosthenes benchmark were repeated using postlinked child programs. The results of this investigation are summarised in the following table (the results are expressed in seconds).

Benchmark	Normal	Postlinked	Percentage
10 Concurrent Sieves	23.797	21.679	91%
Process Creation (100 processes)	28.900	14.494	50%

From this it is clear that the postlinking process does have an appreciable effect on the load times for new processes. If a particular program is to be loaded often it would be beneficial to postlink it; however, for programs that are loaded once and then run for a long period of time the effect would be almost negligible.

6.3. The Process Control System and Plant Simulation

The implementation model for the widget manufacturing plant simulation and its associated control system under FlexOS is shown in Figure 19. This implementation consisted of 10 tasks (three for the simulation, four for the control system and three for the user interface and data logging system). This reflects a total of over 1500 lines of C code. The overall structure of the system is very similar to that adopted for UNIX, as shared memory segments have been used to provide the communication between the three subsystems. The synchronisation of the polishers and the polisher control tasks was provided using semaphore pipes, as was the mutual exclusion required for sections of the shared memory segment.

The facilities provided by FlexOS for formatted screen output and cursor control are very powerful, but also extremely complicated. For this reason a simple **curses**-like header file was developed to make formatted screen output a little easier. The `make` utility available under FlexOS is very similar to that found under UNIX, and proved to be useful in developing the simulation.

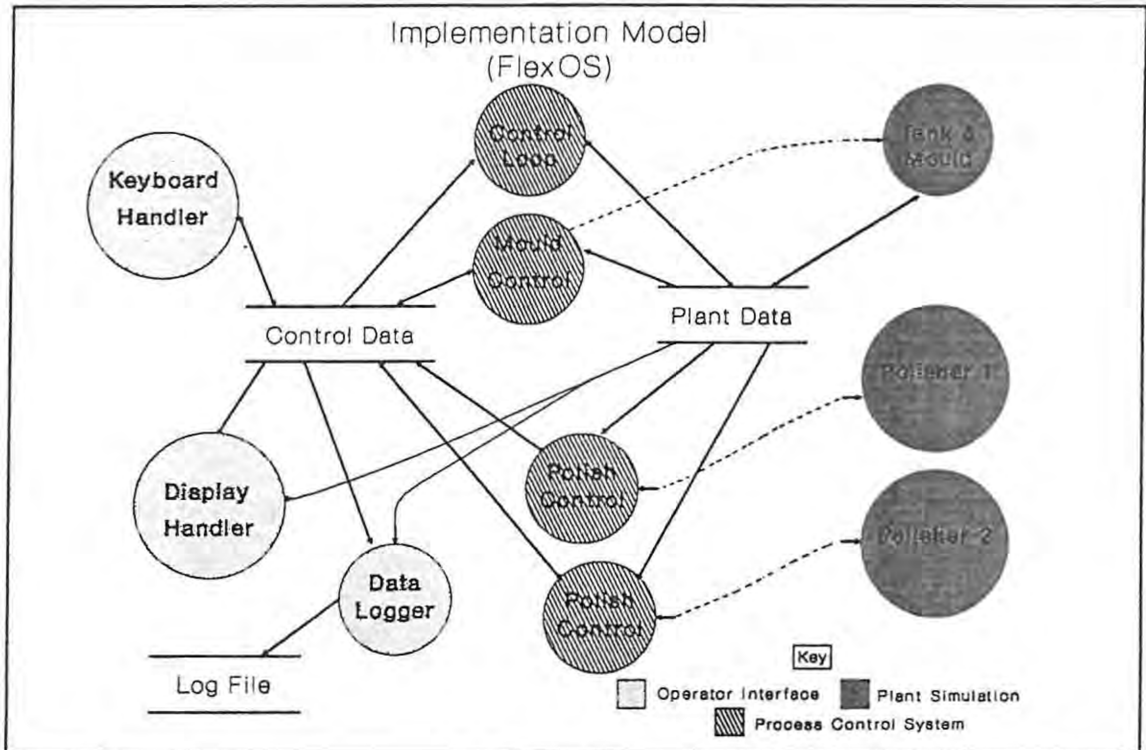

Figure 19

The simulation ran well under FlexOS and the facilities available for multitasking and interprocess communication were found to be quite adequate for this sort of application. However, during the development and testing of the simulation the operating system crashed several times with a "PANIC: Divide by zero" error message, which necessitated rebooting the computer. The error in the simulation programs which caused this problem was the incorrect specification of a parameter to one of the screen formatting functions, and not any major misuse of the system. The operating system should, of course, do something to report such an error, but to crash the entire system because of a screen format error is a rather extreme approach. This lack of robustness in the operating system is not a very desirable feature of a real-time operating system, and points to the need for very thorough testing of any production system to be implemented under FlexOS.

6.4. Qualitative Assessment

This assessment is based on the experience gained in implementing the benchmark programs and the simulation over a period of two weeks of intensive work. FlexOS was found to provide a fairly good environment for system development. The operating system is very similar in appearance to MS-DOS and makes use of exactly the same file system structure. This makes the transition to FlexOS from an MS-DOS environment relatively straight-forward. However, while the similarity to MS-DOS may be considered an advantage, it also brings with it some of the limitations imposed by MS-DOS. In particular the facilities available for batch file processing are extremely primitive when compared with those provided with UNIX. For example, in order

to terminate background processes it was necessary to write a program that took as input the list of currently executing processes and produced as output a batch file that could be executed to terminate the background processes.

Some of the commands suffered from overly verbose parameterisation. For example, the manipulation of background processes is performed by the process command, which uses the following syntax to view the currently executing processes and then terminate a process:

```
process view
process cancel id=<process_number>
```

For comparison, the equivalent UNIX commands are as follows:

```
ps -e
kill <process_number>
```

In addition the process cancel command can be used to terminate a single task at a time, whereas the UNIX kill command can be used to terminate an indefinite number of tasks. This unnecessary verbosity can make interacting with the operating system rather tedious at times.

The editor provided with the system is relatively good, and follows the widely accepted WordStar standard for the use of control keys. The operating system documentation was a rather weak area. The system calls all have rather complicated interfaces (in terms of the numbers of parameters and the possible options that may be set for flags) and examples of their use, and especially the most common usages, would have been very useful. However, there were almost no examples at all. Finally, the MetaWare High C compiler proved to be a very sound and powerful product, with excellent warnings for possible misuse of the C language. The speed of compilation was very good, although linking times were rather long. The only weak area was in regard to the library interface to the operating system facilities which was not well supported in terms of header files or documentation.

6.5. Conclusions

FlexOS was found to be rather disappointing. The high level of compatibility with MS-DOS is a very strong feature, and it has all the necessary facilities for the development and implementation of real-time systems. However, it suffers from the problems found in the scheduling algorithms and from a general lack of robustness, characterised by the system crashes that occurred during the implementation of the simulation. The weak documentation, overly complex supervisor calls, overly verbose commands and the poor batch file facilities are also somewhat of a drawback. If these areas were to be rectified, FlexOS would be an excellent product.

7. Comparison of the Operating Systems

Having studied each of the operating systems in detail individually, they are now considered together, comparing their strengths and weaknesses in each of the areas assessed.

Before comparing the systems it is worth repeating the resolution of the system clock for each of the operating systems and giving the time taken for the Sieve of Eratosthenes. The results

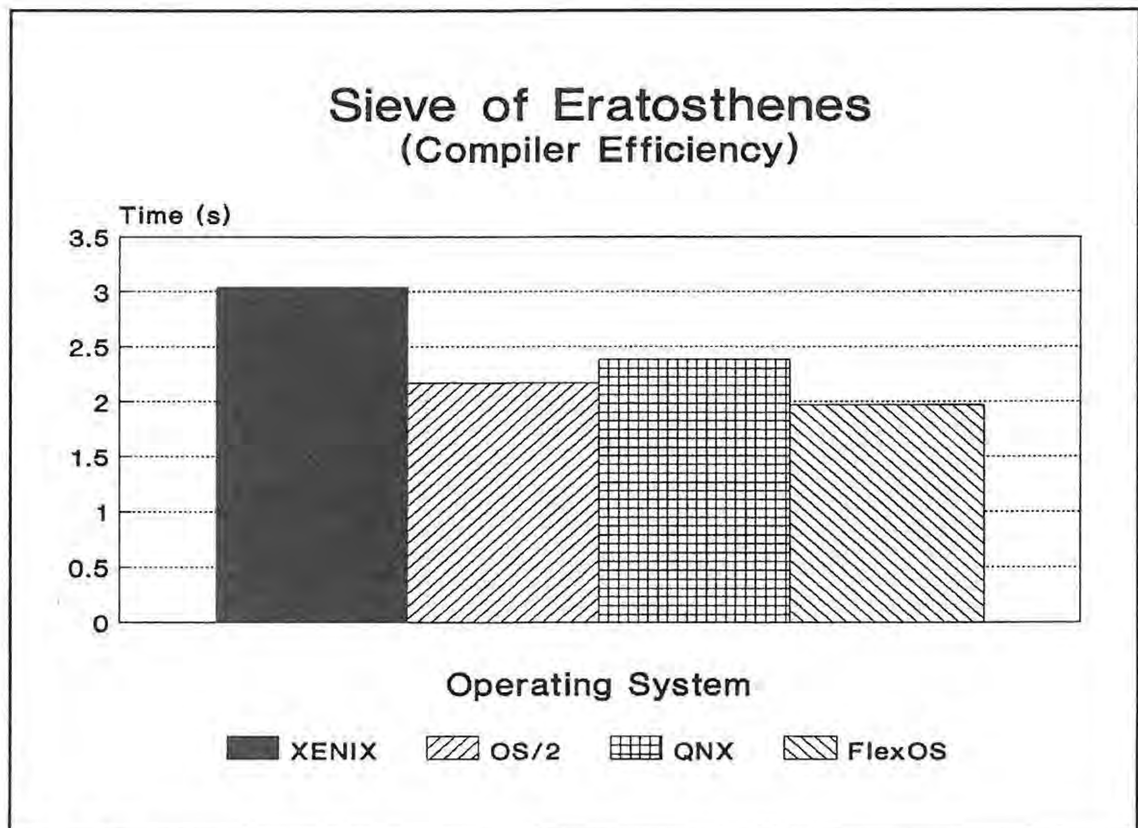

Figure 20

are given in the following table. Figure 20 gives the results for the Sieve of Eratosthenes in graphical form. As can be seen, there is not a lot of difference between the results for the different systems, although XENIX is somewhat less efficient than the others.

	Clock Resolution (ms)	Sieve of Eratosthenes (s)
XENIX	20	3.04 ± 0.01
OS/2	31	2.19 ± 0.02
QNX	50	2.38 ± 0.03
FlexOS	31.5	1.97 ± 0.01

In the following sections, the performance of each of the operating systems in relation to the others will be considered in the same order as the individual results in the previous chapters.

7.1. The Benchmarks

7.1.1. Concurrency

The facilities offered by each of the operating systems for process creation generally fell into one of two categories: a memory operation (creating a new process from part or all of the memory image of the parent process), or a disk operation (creating a new process from an executable file found on disk). XENIX, OS/2 and QNX offered both of these types of facility (with OS/2 offering two forms of disk operations - one for processes and one for sessions), while FlexOS offered only the disk-based form of process creation operation.

The first set of measurements taken in this section assessed the overhead due to multiprocessing and the linearity of the multiprocessing features. The only operating system that produced totally linear values was QNX. XENIX returned results that were linear within the bounds of the standard error, but which appeared to show a tendency to decrease as the number of processes increased. On the other hand, both OS/2 and FlexOS showed a clear decrease in execution time per process as the number of processes increased. Quantitative results for the overhead due to multiprocessing are given in the following table.

XENIX The results of the memory operation (fork) were subject to a large standard error (a result of the nondeterministic scheduling policy employed) and so could not be determined accurately.
The disk operation (fork/exec) produced an overhead of 268 ± 84 ms (measured for ten concurrent processes).

OS/2 The memory operation (thread) produced an overhead which ranged from 1533 ± 30 ms (for one concurrent process) to 665 ± 21 ms (for ten concurrent processes).
The first disk operation (process) produced an overhead which ranged from 717 ± 45 ms (for one concurrent process) to 571 ± 21 ms (for ten concurrent processes).
The second disk operation (session) produced an overhead which ranged from 780 ± 71 ms (for one concurrent process) to 678 ± 26 ms (for ten concurrent processes).

QNX The memory operation produced no measurable overhead, in fact, the results were better than for the sequential form of the test program.
The disk operation (create) produced an overhead of 70 ± 54 ms (for any number of concurrent processes).

FlexOS The disk operation (e_command) produced an overhead which ranged from 572 ± 45 ms (for one concurrent process) to 411 ± 14 ms (for ten concurrent processes).

The results for the disk operation are summarised in Figure 21. This data clearly shows that QNX is the most efficient of the operating systems, with unmeasurable overhead for the memory operation and a result for the disk based operation that is an order of magnitude better than for any of the other operating systems.

The second benchmark performed in this category was the measurement of the process creation time for the various facilities provided by each of the operating systems. Under OS/2 it was impossible to measure the process creation times for threads and sessions using the same form of test program as for the other operating systems and so no results are available for these

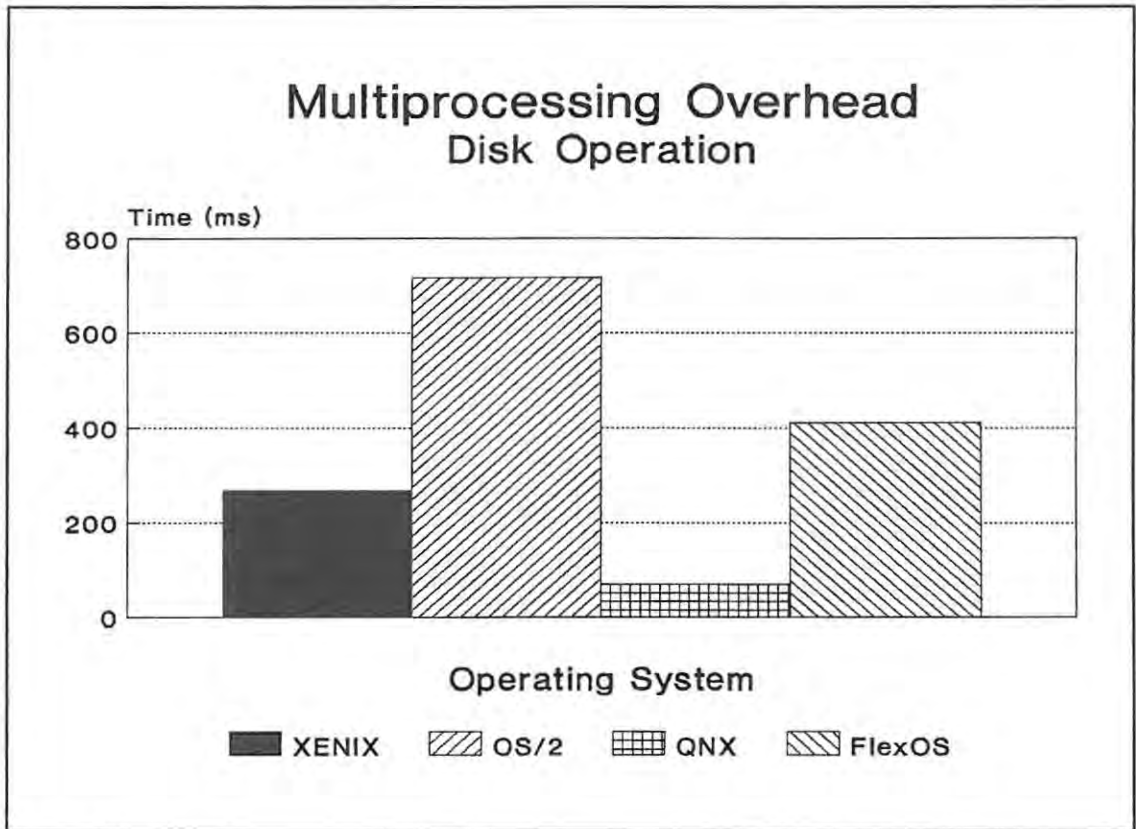

Figure 21

cases. It was also found that the process creation time under OS/2 depended on the number of processes that were created.

XENIX The process creation time for the memory operation (fork) was 19.1 ± 3.9 ms.
The process creation time for the disk operation (fork/exec) was 175 ± 15 ms.

OS/2 The process creation time for the disk operation ranged from 310.5 ± 1.9 ms (for 20 processes) to 315.24 ± 0.53 ms (for 100 processes).

QNX The process creation time for the memory operation (fork) was 12.15 ± 0.38 ms.
The process creation time for the disk operation (create) was 33.85 ± 0.35 ms.

FlexOS The process creation time for the disk operation (e_command) was 289 ± 12 ms.

These results are summarised in Figure 22 and Figure 23. Once again, QNX produced the best results, with the disk operation an order of magnitude better than XENIX or FlexOS. With regard to the memory operations, XENIX came close to the times returned by QNX. The results for FlexOS were obtained using a hard disk with a slightly slower seek time than that used for the other operating systems. This will have a slight effect on the results, but should not be too significant, as the processes used were small and the data transfer rates of the disk drives were identical. It is extremely unlikely that the use of the faster disk drive would have

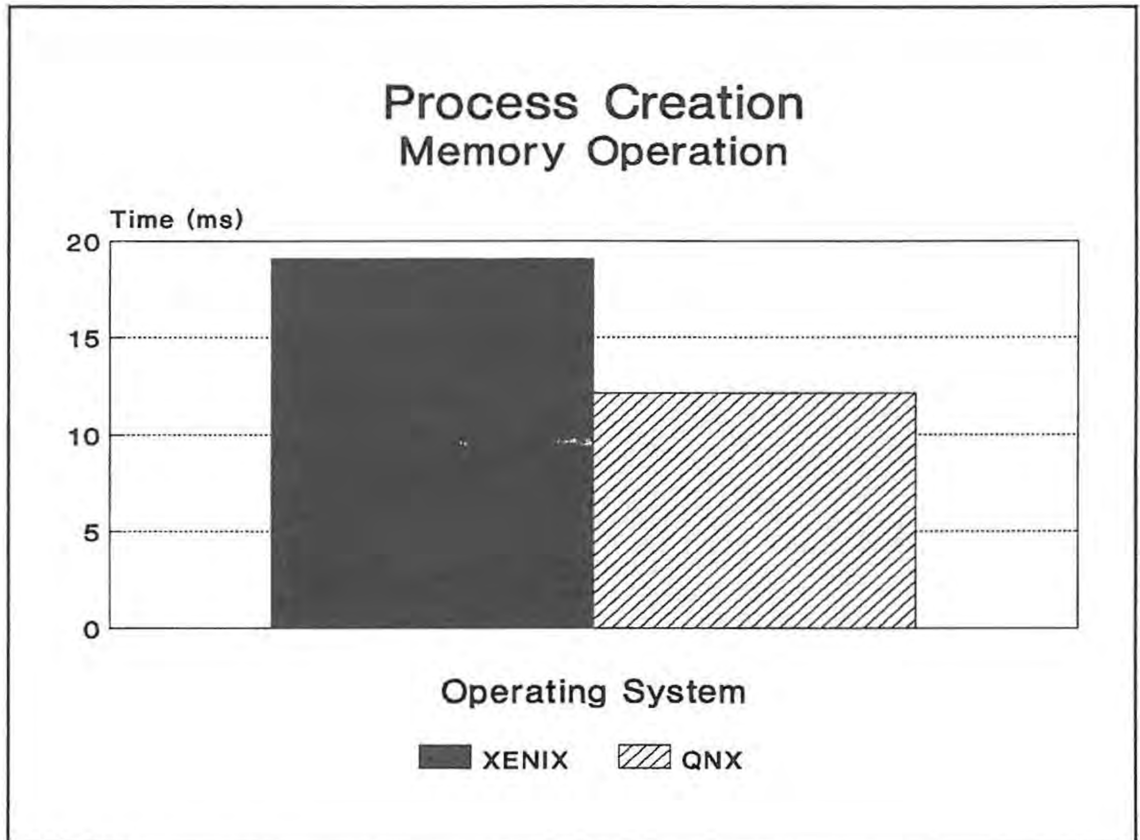

Figure 22

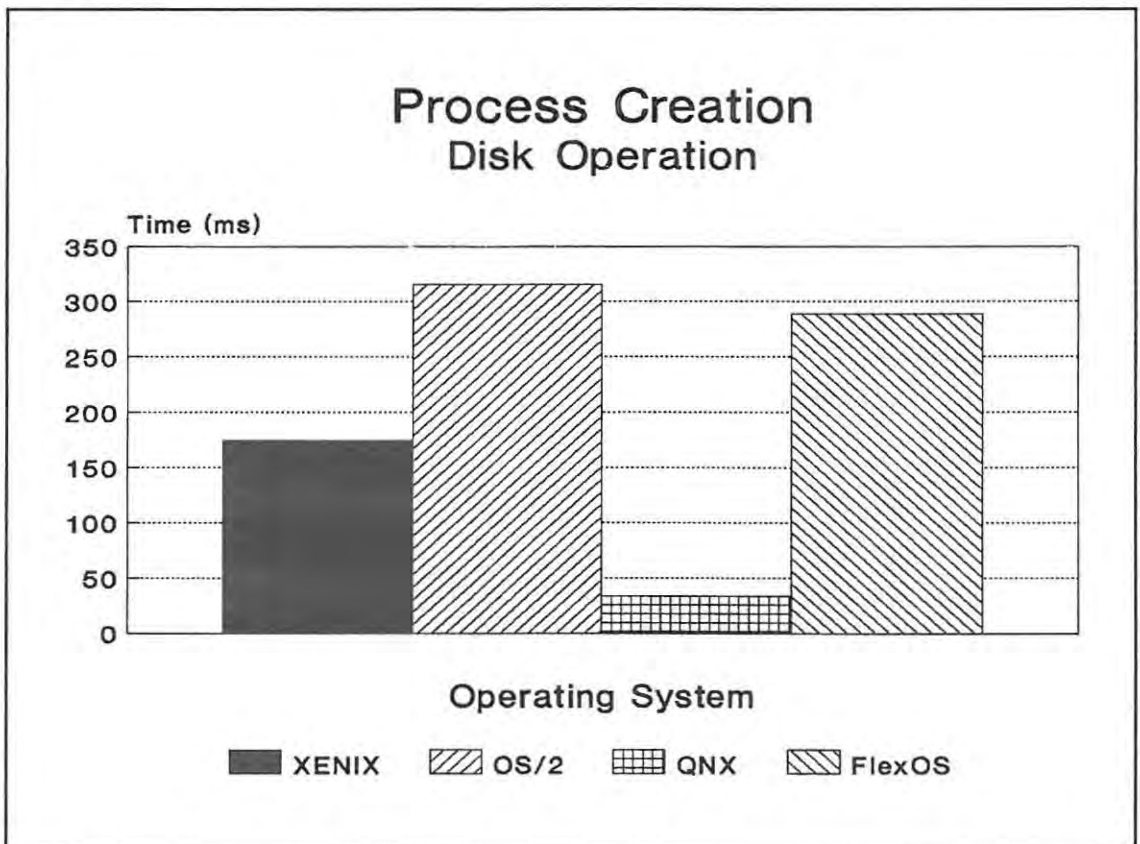

Figure 23

brought the results for FlexOS to those for QNX. A more significant effect arises from the use of the postlinking feature available under FlexOS (discussed in section 6.2.6) which brought the time for the creation of a process from disk down to 144.9 ± 2.6 ms, putting it in the same class as XENIX.

7.1.2. Interprocess Communication

The interprocess communication facilities offered by the operating systems fell into three categories: pipes, messages and shared memory. XENIX and OS/2 both offered all three of these resources, FlexOS offered both pipes and shared memory, and QNX supported only message passing. There were two benchmarks performed in this category. The first test measured the time to send a 100 byte message to a process which simply discarded it. The results are summarised in the following table. A dash (-) signifies that the facility is not available and N/A signifies that the test was not performed for the operating system using that facility. All times are given in milliseconds.

	Pipes	Messages	Shared Memory
XENIX	3.06 ± 0.46	1.89 ± 0.53	8.68 ± 0.74
OS/2	2.232 ± 0.024	N/A	3.858 ± 0.024
QNX	-	0.735 ± 0.037	-
FlexOS	3.484 ± 0.067	-	5.331 ± 0.056

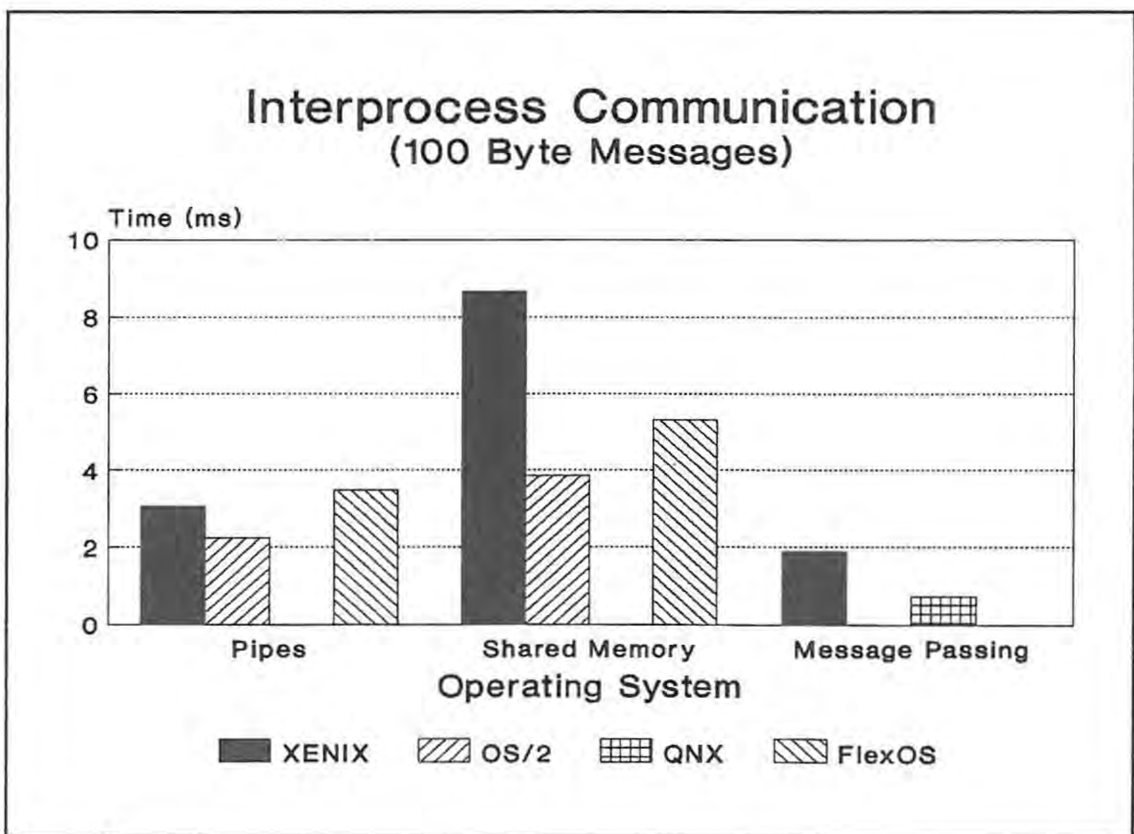

Figure 24

The message passing form of this test was not performed for OS/2 as messages under OS/2 are restricted to being four words (eight bytes) long. The results are illustrated in Figure 24. In general, it can be seen that message passing is the most efficient means of communication, followed by pipes and lastly by shared memory. This last fact is to be expected, since there is the overhead of synchronising the access to the shared memory segment. In general, if there was no need for such synchronisation, one would expect the shared memory facilities to be more efficient than any of the other mechanisms for interprocess communication. Once again, the results for QNX are an order of magnitude better than for any of the other systems. The times for communication using pipes are all fairly close, with OS/2 showing a slight lead over FlexOS and XENIX. In the case of shared memory, OS/2 is considerably more efficient than FlexOS or XENIX.

The second test assessed the time taken to pass a one byte message around a circle of three processes. The results of this test are given in the following table. Again, all values are in milliseconds and the symbols used have the same interpretation as in the previous table.

	Pipes	Messages	Shared Memory
XENIX	3.8518 ± 0.0021	2.959 ± 0.016	N/A
OS/2	2.1610 ± 0.0030	4.3218 ± 0.0029	N/A
QNX	-	0.6795 ± 0.0018	-
FlexOS	2.1222 ± 0.0037	-	N/A

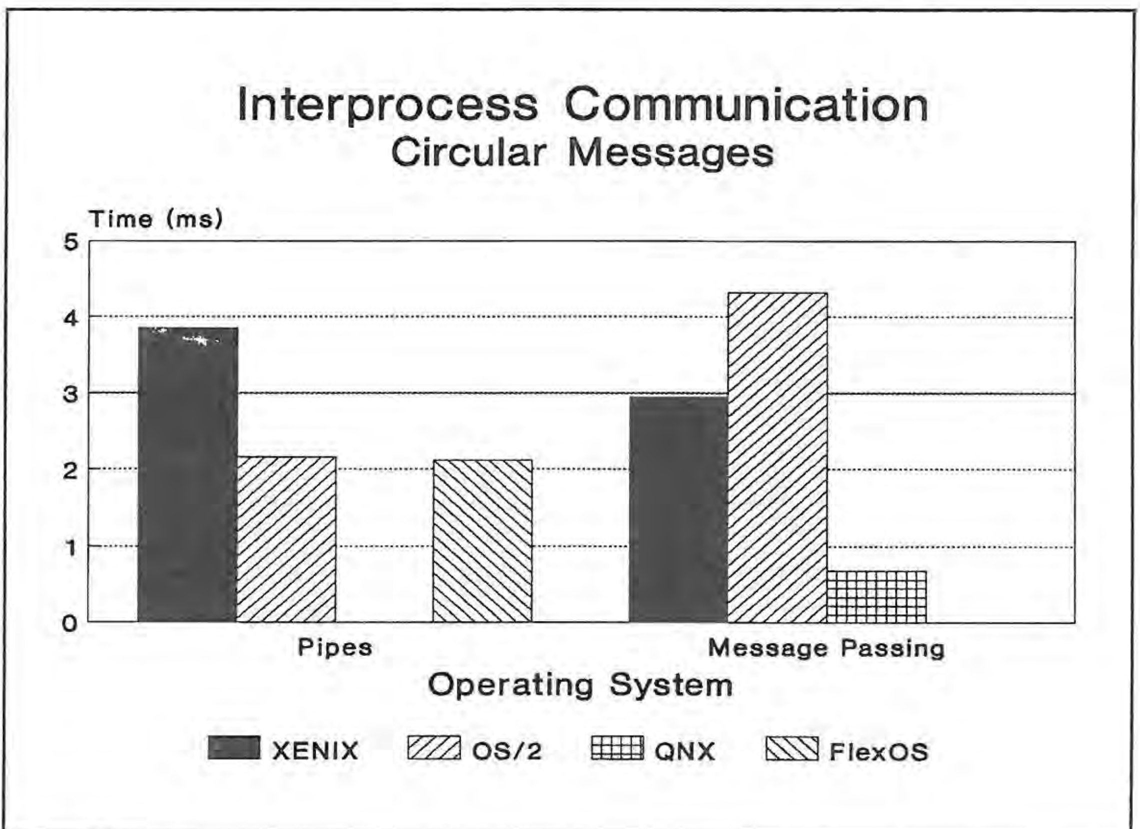

Figure 25

This test was not performed for any of the operating systems using shared memory as the use of such features for passing a single byte of data is highly inefficient. The results are illustrated

graphically in Figure 25, and agree quite closely with those for the first test in this section. Interprocess communication using pipes produced much the same results for OS/2 and FlexOS. However, XENIX did slightly worse in this case. As regards message passing, QNX is an order of magnitude better than the other two systems which offer message passing facilities. The implementation of message passing under OS/2 seems particularly inefficient, but it must be borne in mind that OS/2 messages are eight bytes long and so there was an overhead of seven bytes of unused capacity.

7.1.3. Synchronisation

The facilities offered by the operating systems in this category are difficult to classify into simple classes. The following table provides a summary of the various mechanisms used for the benchmarks.

XENIX	Semaphores are the main synchronisation mechanism, but pipes can also be used for this purpose.
OS/2	RAM semaphores (for use by threads of a single process) and system semaphores (for more general use) are provided. System semaphores were tested using both threads and separate processes.
QNX	Offers a form of semaphore through the attaching and detaching of ports.
FlexOS	Pipes (with no data content) are used.

The results (the amount of time per synchronisation operation, expressed in milliseconds) are given below.

	Operation	Result
XENIX	Semaphores	5.006 ± 0.021
	Pipes	8.01 ± 0.17
OS/2	RAM Semaphores	1.241 ± 0.041
	System Semaphores with Threads	1.532 ± 0.036
	System Semaphores with Processes	1.474 ± 0.041
QNX	Ports	1.000 ± 0.068
FlexOS	Pipes	2.78 ± 0.16

The results are illustrated graphically in Figure 26, using the best case figures for each of the operating systems. Once again, QNX is somewhat more efficient than the rest of the systems, with OS/2 a fairly close second. FlexOS starts to trail rather far behind and the results for XENIX are very poor.

7.1.4. Exception Handling

As was stated in the general discussion of the benchmarks in chapter two, this was probably the least satisfactory of the benchmarks. However, the results do provide a "ballpark" value for the performance of the operating systems, and, in particular, may provide some indication of the *relative* efficiency of each of the systems. The features of the operating systems that were tested in this section fell into two categories: system calls to suspend a process and have the operating system resume it (a sleep operation), and system calls whereby one process can alert another to the fact that some event has occurred. XENIX offered both of these facilities; the former by

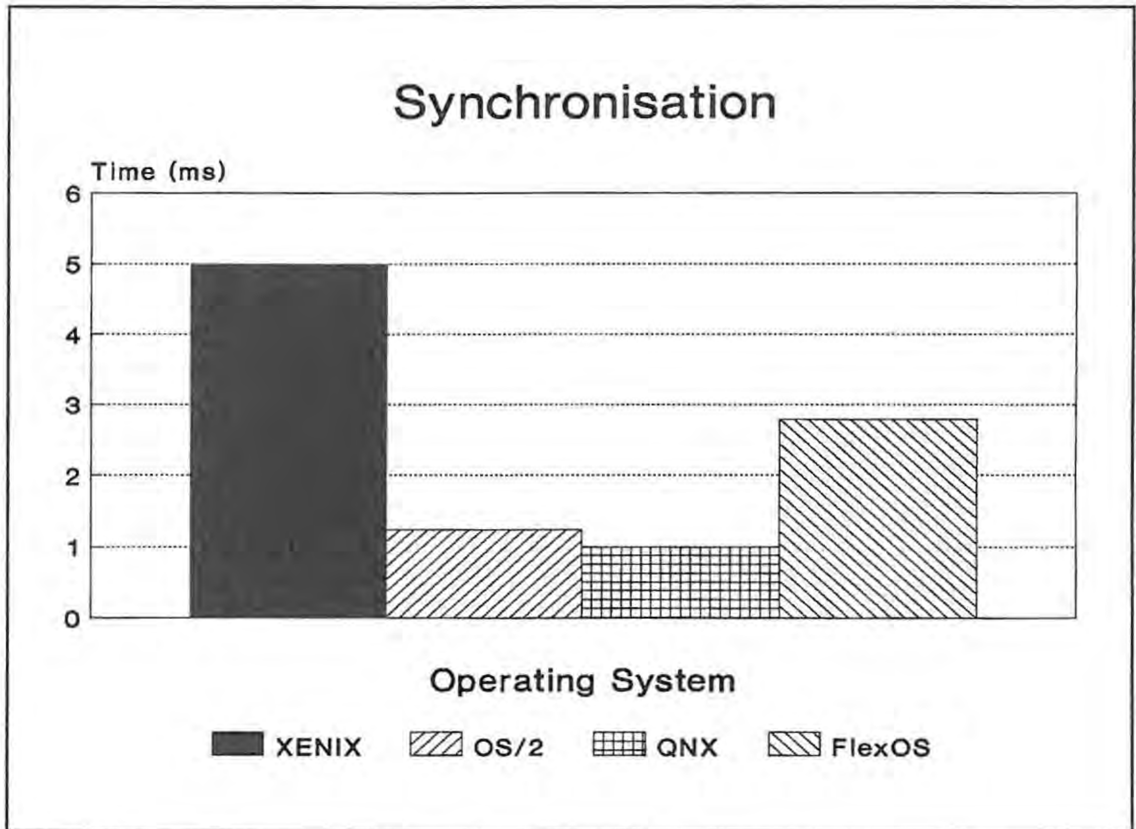

Figure 26

means of the `nap()`[1] system call and the latter by means of the `kill()` and `signal()` system calls. OS/2, as usual, provided a wide range of features in this category, with a sleep system call, periodic and one-off timer services to signal semaphores, and flags for one process to let another know that an event has occurred. QNX provided facilities for a timer to signal a port and also to set an exception on a process. FlexOS provided a synchronous timer facility (sleep operation) and an asynchronous timer facility. The results are summarised in the following table (all values in milliseconds).

	Operation	Result
XENIX	Nap	8.7
	Kill/Signal	10.37
OS/2	Sleep	0.19
	Asynchronous timer	1.3
	Periodic timer	0.46
	Flag (with threads)	0.14
	Flag (with processes)	0.42
QNX	Signal	1.7
	Exception	1.7
FlexOS	Synchronous timer	0.56
	Asynchronous timer	0.92

[1]This system call, which provides millisecond resolution, is a nonstandard feature of XENIX. The timer services defined under UNIX System V provide a resolution of only one second.

Without trying to infer too much from such poor data, it would appear that XENIX is rather inefficient in this area, and that OS/2 and FlexOS are both relatively good. As far as the individual results are concerned, it is apparent that (with the exception of the thread-based flag operation under OS/2) the sleep operation is considerably more efficient than the signal form. This is to be expected, of course, since the sleep operation involves just the suspended process and the operating system. The signal type of operation, on the other hand, involves the operating system in passing the "event" from one process to another.

7.2. The Process Control System and Plant Simulation

All of the operating systems ran the process control simulation quite adequately. However, a few points can be made about their general performance. Subjectively, XENIX appeared to run the least smoothly. The screen displays were rather "jerky" and there were noticeable delays at times. Some of this may have been due to the way in which a process had to be dedicated to handling the screen output (as discussed in section 3.3). QNX ran the simulation very smoothly, but required more active processes in order to implement the communication between the subsystems of the simulation. The other operating systems with their support of shared memory segments were able to do without these communication processes.

Quantitative results were obtained for XENIX and OS/2, where a low priority performance monitoring task was implemented. (This was done right at the end of the study, and, unfortunately, the other two operating systems were no longer available for testing.) This gave a performance index value of 0.91 for OS/2 and 0.64 for XENIX. This clearly highlights the fact that XENIX was not really as capable of handling such a system as OS/2. Presumably, the other systems would be closer to the level reached by OS/2 (or even better), judging from the results found for the benchmarking process.

7.3. Qualitative Comparison

Qualitatively (subjectively, perhaps) the systems fall into two groups. XENIX and QNX both offer a relatively small number of system calls with simple interfaces (with the exception of the newer System V features in XENIX). On the other hand, OS/2 and FlexOS offer a far more complex interface between the operating system and the program. This fact makes the former operating systems much easier to work with - it is possible to *remember* all the system calls and their parameters. When working with OS/2 or FlexOS it is essential to have the reference manuals close at hand to check on the various options that are available. This is, of course, a classic trade-off between power and flexibility on one hand, and ease of use on the other. The provision of a large range of services and many options makes it possible for the operating system facilities to be tailored to suit a particular situation. (Morris and Brooks [1988] give a detailed evaluation of the system call interfaces to UNIX and OS/2 and they also highlight the difference in the complexity of the interface in their discussion.)

The approach of QNX and XENIX has been to provide facilities that are less configurable, but hopefully still suitable for most cases. For XENIX this means it has been made as general purpose as possible; there is no way of setting priorities exactly, for example, since this is usually unnecessary in a general purpose operating system. As should be expected of the operating system which gave birth to the C language, the interface between the operating system and the C programming language is very good. In the case of QNX, it has been tailored for real-time, distributed applications, which is ideal for those real-time systems that are the concern of this study. The flexibility of OS/2 has been provided to allow it to be used for general purpose applications (its main purpose) but to provide the necessary facilities for time-critical applications when necessary. FlexOS is aimed at the real-time end of the spectrum, and presumably its flexibility is simply meant to give the designer of a real-time system as much freedom as possible.

Turning to the subject of the utilities and the programming support environment provided by the operating systems, it is hard for the other systems to match up to XENIX in this regard. By far the oldest of the systems tested, XENIX shows its maturity in the depth and number of available utilities. OS/2 is probably the nearest contender in this area, and appears to be catching up rapidly as it gains increasing acceptance. Both of these systems have found support from third party software houses and there are a large number of packages, such as word processors, that combine to make these systems suitable for use as full system development environments (from the use of CASE tools for the design of applications, through to the production of the documentation). The other two systems (FlexOS and QNX) are aimed at a more specialised market and so do not offer as many general purpose packages. Similarly, the utilities (such as editors) tend to be somewhat less sophisticated.

Another point that should be considered is the range of hardware platforms supported by the operating system. This another area where UNIX has a great advantage, in that it supports a large range of hardware platforms from almost all the modern CISC and RISC microprocessors through to supercomputers such as the Cray series. FlexOS is available for the Intel range of 80x86 processors, and for the Motorola 680x0 range as well. Both OS/2 and QNX restrict themselves to the popular 80x86 processors. OS/2 supports only the 80286 and those processors that are supersets of the 80286 architecture, while QNX supports the older 8088/8086 and 80186 processors as well.

The last area to be considered in this section is that of documentation. Again, this is an area in which the greater maturity of XENIX stands it in good stead. The documentation is consistently laid out, with good indexes and cross referencing, and examples of the use of commands or system calls in many cases. The main problem with the XENIX documentation is its volume and verbosity! In the case of OS/2, the documentation that was used was not really "standard" in any sense (as explained in section 4.4). Having examined the Microsoft Software Developers Kit very briefly, it would appear to be almost as thorough as the XENIX documentation with regard to the available system calls. QNX offers rather terse documentation, with poor layout in certain areas (such as the discussion of the `set_exception()` system call, mentioned in section 5.4). The indexing is another weak point. However, it is adequate and offers examples of the use of the system calls and cross references. The FlexOS documentation was possibly the weakest of any of the operating systems, suffering from a relatively poor layout,

extreme brevity and a lack of examples.

7.4. Discussion

The temptation at the end of a study such as this is to declare one of the operating systems the clear "winner". However, the systems that were studied covered a wide spectrum. They ranged from the general purpose, multi-user XENIX operating system, through OS/2 which is general purpose, single-user and configurable for time-critical applications, to the small, real-time QNX system and the larger, real-time FlexOS. As such, the operating systems assessed in this study meet different criteria. For example, in hard real-time systems with tight time constraints XENIX would not be suitable. However, in a less demanding situation it could possibly be used, and the advantages of its more mature features might even make it desirable. Cramer [1988] shows how the facilities offered by UNIX can easily be enhanced to provide better real-time operation. From another viewpoint, the real-time variations of UNIX (such as H-P/UX from Hewlett-Packard and REAL/IX from MODCOMP) would certainly bear consideration if a complete system development environment was desired. This point is supported by Mizuhashi and Teramoto [1989] who state that evaluations of the use of RX-UX 832 (a real-time version of UNIX developed by NEC) show that it has helped ease the design and implementation of real-time systems and has improved both productivity and the quality of software. Astley [1987] also gives a fairly detailed discussion of the changes that are required to UNIX to produce real-time versions. Furht *et al* [1989] have a good discussion of the motivation behind the REAL/IX operating system, and a comparison of the features of eight different real-time dialects of UNIX.

The other three operating systems all provide the kind of performance that is required for real-time systems, but again the factors that distinguish them need to be weighed up and assessed against the type of application that is being developed. In addition, the availability of other facilities that may be required (such as word processing and CASE tools, if the operating system is to be used to support the complete system development cycle) should also be considered. FlexOS has the anomalies in its scheduling algorithm that were discussed in section 6.2.5, and the lack of robustness characterised by frequent system crashes (discussed in section 6.3). These problems were drawn to the attention of Digital Research, but no reply was received from them.

OS/2 performed quite adequately and has the advantage of increasing acceptance as a general purpose operating system, leading to a large number of third party utilities and development tools becoming available. In some ways it combines the advantages of UNIX with the real-time performance required for certain time-critical process control applications. It also has the advantage of a sophisticated, graphical windowing environment for the generation of operator and managerial displays. As far as serious real-time applications are concerned, OS/2 is not ideal. The designers of OS/2 state that the kernel was designed for simple implementation and reasonable performance for *non-real-time* applications [Kogan and Rawson, 1988]. The main reason for this is that the kernel is nonpreemptible (although it is interruptible).

Finally, QNX offers excellent performance (undoubtedly better than any of the other operating

systems), but with a relatively poor level of support in terms of utilities, system development tools and documentation. One of the strengths of QNX is that the message passing paradigm, on which it is based, is very elegant. Although this approach can lead to the need for greater numbers of processes (compared with a system which supports access to shared memory, for example) the amazing efficiency of its message passing routines make this factor relatively unimportant. It would appear to be the "winner", if such a choice had to be made, on the basis of its performance, which is, after all, the fundamental criterion that needs to be considered for time-critical applications.

The message passing implementation of QNX makes it ideal for distributed processing systems (indeed it was developed with this in mind). Coupled with the fact that it is a small, high performance operating system, this gives cause to consider the desirability of hosting QNX on the transputer architecture [INMOS, 1988]. The transputer is a fast microprocessor which is intended for distributed parallel processing, using a message passing paradigm. Research has shown that transputers make an excellent platform for the implementation of real-time systems [Hull and Zarea-Aliabadi, 1989]. Thus, there is good reason to believe that a version of QNX for the transputer would be successful. Efforts to port operating systems such as UNIX to the Transputer (such as the DISTRIX project [McCullagh and Smit, 1989] and Helios [Grimsdale, 1989]) have found many obstacles, and the closer match between the fundamental approaches of QNX and the transputer may well help to overcome these.

8. Conclusions

This study has investigated the performance of several operating systems for microcomputers, namely XENIX System V, OS/2, QNX and FlexOS. These operating systems range from general purpose operating systems to real-time operating systems. The main emphasis of the assessment was on the suitability of the operating systems for real-time applications such as process control.

The assessment procedure involved designing a suite of benchmarks that could be used to compare different operating systems. This meant identifying those properties that all real-time operating systems have in common and designing a set of tests that was, as far as possible, independent of the actual way in which a particular operating system implemented a given feature. The properties that were examined in this study were **concurrency** (the creation of new processes and the concurrent execution of processes), **interprocess communication**, **synchronisation** and **exception handling** (the voluntary suspension of processes for a specified duration and the reaction of processes to external events). Results for the performance of the features of the operating systems which provided these properties were used as the basis of the quantitative comparison of the systems.

It is important to note that these results do not reflect the actual low level performance of the operating systems. Such results (for example preemption times, often quoted in advertisements) are impossible to determine without special tools or access to the source code. Rather, the results of this study reflect the perceived performance of the operating systems, often in worst case situations. This approach may well be of more value, since, although factors such as preemption time and task switching time are important, they do not necessarily tell the whole story. For instance, an operating system may boast a very impressive preemption time but have other overheads (for example, in synchronisation) that are unaccounted for by simply quoting the preemption time. It is felt that the results found in this study present a more realistic view of the performance of the operating systems, since they were obtained by using benchmark programs that performed much the same sort of operations as production programs would (such as sending messages from one process to another, or synchronising two processes). The one set of results which is highly unreliable is the exception handling times, for the reasons discussed in section 2.1.4. These results should not be used as the basis for drawing any strong conclusions about the performance of the operating systems in this area.

In addition to this quantitative assessment, it was felt that the suitability of an operating system for use as a development environment is often an important factor in the selection of an operating system for a particular application. This aspect of an operating system is harder to measure quantitatively and so a qualitative approach was taken. This involved the implementation of a small real-time simulation of a manufacturing process and its associated control system under each of the operating systems. The design of this simulation was undertaken using the Ward and Mellor approach to the development of real-time systems. For the special case of simulations, it was found that several enhancements to the basic techniques espoused by Ward and Mellor were of great use. This extension of the Ward and Mellor method of real-time systems design formed a large component of the time spent designing the simulation. The

implementation of the simulation allowed some deeper experience of the operating systems to be gained, compared with that gained from implementing short benchmark programs. This experience was the main basis for the qualitative assessment of the operating systems. In addition, the implementation of a performance monitoring task allowed a quantitative value to be assigned to the performance of the simulation. In general, it appeared that the general purpose operating systems tended to have a better supporting environment, evidenced by such factors as the quality and number of utility programs that were available. On the other hand these systems tended to have less performance than the real-time systems.

In terms of the quantitative results, the real-time operating system QNX from Quantum Software Systems gave the best overall performance. QNX is based on a message passing paradigm and so its extremely efficient message passing facilities are greatly in its favour. In general, the message passing paradigm was found to be an elegant approach to the implementation of real-time systems. In addition, it is ideally suited for use in distributed processing systems.

In terms of the qualitative results, it was felt that UNIX offered the best development environment. However, there are clear signs that a great number of development tools for OS/2 are likely to become available in the near future, and this could make a significant difference to the desirability of using OS/2 as a development environment.

As far as the general suitability of the operating systems for real-time applications is concerned, it was found that XENIX was not really suitable (except perhaps for systems with very little requirement for time-critical processing). The other three operating systems all performed quite adequately in this regard, although FlexOS was subject to some anomalies in extreme situations. The existence of real-time versions of the UNIX operating system, coupled with the good development environment found under this operating system, means that such versions would bear serious consideration for real-time applications.

A study such as this is extremely open-ended, as there are many other operating systems worthy of inclusion in the list of those tested. The benchmark programs have already been supplied to a company in Cape Town for the evaluation of iRMX (a highly configurable real-time operating system from Intel Corporation). Another direction that would be of interest for further study, would be to run the benchmarks on different architectures (such as minicomputers and transputers). This would give some indication of the price/performance factors that affect the use of microcomputers for real-time processing.

9. References

Aburto, A.A., "Problems and Pitfalls", *BYTE* **13**(6), June 1988, 217-224.

Astley, B., "Operating in Real-Time", *Systems International* **15**(7), July 1987, 63-64.

Atkinson, H., "Benchmarking Concurrency", *Systems International* **14**(3), March 1986, 50.

Bach, M.J., *The Design of the UNIX Operating System*, Prentice-Hall (Englewood Cliffs), 1986.

Bemmerl, T., and **Schöder**, G., "A Portable Realtime Multitasking Kernel for Embedded Microprocessor Systems", *Microprocessing and Microprogramming*, **21**, August 1987, 181-188.

Bunnel, M., and **Bunnel**, M., "Real-Time Data Acquisition", *Dr. Dobb's Journal*, **14**(6), June 1989, 36-44.

Cheriton, D.R., **Malcolm**, M.A., **Melen**, L.S., and **Sager**, G.R., "Thoth, a Portable Real-Time Operating System", *Communications of the ACM* **22**(2), February 1979, 105-115.

Conway, M.E., "Design of a Separable Transition-Diagram Compiler", *Communications of the ACM* **6**(7), 1963, 396-408.

Cortesi, D.E., *The Programmer's Essential OS/2 Handbook*, M&T Publishing (Redwood City), 1988.

Cramer, B., "Writing Real-Time Programs under UNIX", *Dr. Dobb's Journal* **13**(6), June 1988, 18-33.

Duncan, R., "Interprocess Communications in OS/2", *Dr. Dobb's Journal* **14**(6), June 1989, 14-25.

Elfring, G., "A Message-Passing Executive", *PC Tech Journal* **5**(1), January 1987.

Furht, B., **Parker**, J., **Grostick**, D., **Ohel**, H., **Kapish**, T., **Zuccarelli**, T., and **Perdomo**, O., "Performance of REAL/IX - Fully Preemptive Real Time UNIX", *Operating Systems Review* **23**(4), October 1989, 45-52.

Gentleman, W.M., "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", *Software - Practice and Experience* **11**(5), May 1981, 435-466.

Grimsdale, C.H.R., "Distributed Operating System for Transputers", *Microprocessors and Microsystems* **13**(2), March 1989, 79-87.

Hehner, E.C.R., "Real-Time Programming", *Information Processing Letters* **30**(1), January 1989, 51-56.

Hildebrand, D., "Message-Passing Operating Systems", *Dr. Dobb's Journal* **13**(6), June 1988, 34-48.

Hull, M.E.C., and **Zarea-Aliabadi**, A., "Real-Time System Implementation - The Transputer and Occam Alternative", *Microprocessing and Microprogramming* **26**(2), June 1989, 77-84.

INMOS Ltd., *Transputer Reference Manual*, Prentice-Hall (Englewood Cliffs), 1988.

Johnstone, J., "The QNX Multiuser/Multitasking Operating System", *Micro/Systems*, **4**(11), November 1988, 58-61.

Kar, R.P., and **Porter**, K., "Rhealstone: A Real-Time Benchmarking Proposal", *Dr. Dobb's Journal* **14**(2), February 1989, 14-24.

Knuth, D.E., *Fundamental Algorithms* (Vol 1 of *The Art of Computer Programming*), 2ed, Addison-Wesley (Reading, Massachusetts), 1973.

Kogan, M.S., and **Rawson**, F.L., "The Design of Operating System/2", *IBM Systems Journal* **27**(2), 1988, 90-104.

Laboratory Technologies Corp., "A Realtime Process Control Package for PCs", *SA Measurement and Control* **2**(8), August 1985, 43-44.

Lyon, A.J., *Dealing with Data*, Pergamon Press (Oxford), 1970.

Mackay, S., "The PC in Process Control", *SA Measurement and Control* **3**(8), August 1986, 16-22.

McCullagh, P.J., and **Smit**, G. de V., "Implementing UNIX on the INMOS Transputer", *Proc Vth Southern African Computer Symposium*, Johannesburg, November 1989, 119-128.

McQuaid, T., "Operating Systems for Personal Computers", *SA Measurement and Control* **2**(5), May 1985, 5-7.

Minasi, M., "OS/2's Multitasking Dashboard", *BYTE* **13**(12), November 1988, 147-151.

Mizuhashi, Y., and **Teramoto**, M., "Real-Time Operating System: RX-UX 832", *Microprocessing and Microprogramming* **27**, August 1989, 533-538

Morris, R.R., and **Brooks**, W.E., "At the Core: An API Comparison", *PC Tech Journal* **6**(12), December 1988, 62-77.

Musstopf, G., "The Influence of Microprocessors on Future Real-Time Systems", *Real-Time Programming 1978*, Ed. Cronhjort, B., (*Proc. IFAC/IFIP Workshop on Real-Time*

Programming, Mariehamn/Aland, Finland, 19-21 June 1978), Pergamon Press (Oxford), 1979, 21-29.

Riley, D.D., *Data Abstraction and Structures*, Boyd & Fraser (Boston), 1987.

Ritchie, D.M., and **Thompson**, K., "The UNIX Time-Sharing System", *Communications of the ACM* **17**(7), July 1974, 365-375.

Smith, J.E., "Characterizing Computer Performance with a Single Number", *Communications of the ACM* **31**(10), October 1989, 1202-1206.

Stewart, A.B., "Problems Relating to the Use of Personal Computers in Process Control", *Elektron* **4**(3), March 1987, 9-15.

van Halm, R., "Real-Time in the Real World", *UNIX World* **VI**(9), September 1989, 58-62.

van Zyl, W., "Realtime Multiprogramming Techniques", *SA Measurement and Control* **3**(8), August 1986, 29-30.

Vose, G.M., and **Weil**, D., "A Benchmark Apologia", *Dr. Dobb's Journal* **14**(2), February 1989, 36-41.

Ward, P.T., and **Mellor**, S.J., *Structured Development for Real-Time Systems* (Vols 1, 2 and 3), Yourdon Press (New Jersey), 1985 (Vols 1 and 2) and 1986 (Vol 3).

Weiss, R. "16-Bit Operating System Opens PC ATs to Real-Time Development", *Electronic Design* **33**(12), May 1985, 169.

Wells, G.C., "Real-Time Programming with UNIX", *Technical Document 87/25*, Department of Computer Science, Rhodes University, July 1987.

Wells, G.C., "The Application of Real-Time Design Techniques to Simulation", *Software Engineering Journal* **4**(6), November 1989, 301-308.

Wells, P., "The 80286 Microprocessor", *BYTE* **9**(12), November 1984, 231-242.

Young, S.J., *Real Time Languages: Design and Development*, Ellis Horwood (Chichester), 1982.

A. Algorithms for Benchmark Programs

A.1. Introduction

In the following sections algorithms for the various benchmark programs are presented. These algorithms are based on the XENIX programs that were used for the benchmarking process, but have had the XENIX specific system calls and data structures replaced by generic forms of system call and data structure. These generic entities are shown in bold type in the algorithms, and are defined in section A.2.2. The algorithms are presented in the same order as the discussion of the benchmarks.

In order to simplify the process of porting the benchmarks from one operating system to the next, the timing interface was developed in terms of a header file (`timer.i`) which included the definition of a type `TIME` and two routines with the following C prototypes.

```
void set_time (TIME *);
/* Set time to current clock value */

void report_time (TIME *, TIME *);
/* Print out elapsed time */
```

One algorithm which does not fall into any of the four areas of real-time features that were studied is the basic Sieve of Eratosthenes, which is presented below, as a full C program. This was also the algorithm that was used for background processes when assessing their impact on foreground processing (of course, the timing calls were removed and the program ran indefinitely). Note the use of the `set_time()` and `report_time()` functions.

```
/* Eratosthenes Sieve Prime Number Program.  Executes once.
   George Wells  --  2 January 1990 */

#include <stdio.h>
#include "timer.i"

#define true 1
#define false 0
#define size 8190
#define sizepl 8191

char flags[sizepl];

void main ()
{ int i, prime, k, count, iter;
  TIME t1, t2; /* Timer ticks */

  printf("1 sieve: 10 iterations\n");
  set_time(&t1);
  for (iter = 1; iter <= 10; iter++)
    { count = 0;
      for (i = 0; i <= size; i++)
        flags[i] = true;
      for (i = 0; i <= size; i++)
        { if (flags[i])
            { prime = i + i + 3;
              k = i + prime;
              while (k <= size)
                { flags[k] = false;
                  k += prime;
                }
              count++;
            }
```

```
        }
    }
    set_time(&t2);
    report_time(&t1, &t2);
} /* sieve.c */
```

A.2. Data Types and System Calls Used in Algorithms

The generic data types and system calls used in the algorithms are defined in the following two sections. In the case of the system calls, each one is presented in the form of a C function prototype, followed by a brief explanation of the use and effects of the system call. In both sections the types and calls are presented in alphabetic order.

A.2.1. Generic Data Types

`FILE_HANDLE` This type is used for all input and output files, and also for pipes. See `Read()`, `Write()` and `CreatePipe()` in section A.2.2.

`MESSAGE_HANDLE` This type is used to control access to message queues. A message queue is an ordered list of messages, which may be of any length. The operating system maintains the queue, and suspends any process which attempts to add an item to the queue when the internal buffers used to store messages become full. See `CreateMessageQueue()`, `SendMessage()` and `ReceiveMessage()` in section A.2.2.

`PID` This type is used for the process identifiers by which the operating system recognises processes. See `GetParentsProcessID()` and `SendSignal()` in section A.2.2.

`SEMAPHORE` This type is used for simple counting semaphores. A semaphore has an associated value which is maintained by the system and adjusted by the system calls which make use of semaphores. See `CreateSemaphore()`, `Wait()` and `Signal()` in section A.2.2.

`SHARED_MEM_HANDLE` This type is used by the system to control access to shared memory segments. These are areas of memory which may be accessed by more than one process (normally, a process has access only to its own private memory area). See `CreateSharedMemorySegment()` and `AttachSharedMemorySegment()` in section A.2.2.

A.2.2. Generic System Calls

`char * AttachSharedMemorySegment (SHARED_MEM_HANDLE Shm);`

This system call takes a shared memory segment handle (returned from a call to `CreateSharedMemorySegment()`) and returns a pointer to the area of shared memory. This pointer can be used by the process to access the shared memory segment.

`MESSAGE_HANDLE CreateMessageQueue (void);`

This system call returns a handle which can be used to access a message queue. See also `SendMessage()` and `ReceiveMessage()`.

`void CreatePipe (FILE_HANDLE Pipe[2]);`

This system call is used to create a pipe. Two file handles are placed in the parameter `Pipe`. `Pipe[0]` is used for reading information from the pipe, and `Pipe[1]` is used for writing information to the pipe. Processes reading information from an empty pipe will be suspended until another process writes some information to the pipe. Similarly, processes writing information to the pipe will be suspended if the internal buffers allocated to the pipe are full.

void CreateProcessFromDisk (char * ExeFileName);
This system call will load an executable program from the file named by ExeFileName into memory and start the execution of the program. The program executes as a child of the process that makes this system call. See GetParentsProcessID() and WaitForChildProcess().

void CreateProcessFromMemory (void (* Func)(), ...);
This system call creates a new process by making a duplicate copy of the calling process. The new process executes a child of the process that makes this system call. The execution of the child process commences from the beginning of the function Func. Any other parameters specified are passed to Func, as parameters. Execution of the child process ceases when control returns from the function Func. See WaitForChildProcess().

void CreateSemaphore (SEMAPHORE * Sem, unsigned InitialValue);
This system call creates a new semaphore. The system handle for the new semaphore is placed in the parameter Sem. The value associated with the semaphore is set to InitialValue. See Wait() and Signal().

SHARED_MEM_HANDLE CreateSharedMemorySegment (int Size);
This system call creates a shared memory segment. The size of the segment is Size bytes. The system handle returned by this system call can be used by processes in calls to the AttachSharedMemorySegment() system call to gain access to the memory segment.

PID GetParentsProcessID (void);
This system call returns the process identifier of the parent of the calling process. See SendSignal().

void JoinFileDescriptors (FILE_HANDLE F1, FILE_HANDLE F2);
This system call makes the file handle F2 a duplicate of F1. The effect of this is to make both file handles refer to the same file. If F2 already refers to an open file then it is closed before being joined to F1.

void Read (FILE_HANDLE File, char * Buffer, int Size);
This system call reads Size bytes from the file represented by the file handle File into the area of memory pointed to by Buffer. The calling process is suspended if there is less than Size bytes of information available, until such time as the request for data can be satisfied. See Write().

void ReceiveMessage (MESSAGE_HANDLE MessageQueue, char * Buffer, int Size);
This system call removes a message from the head of the message queue represented by the message handle MessageQueue and places it in the area of memory pointed to by Buffer. Buffer is assumed to be at least Size bytes in extent. The calling process is suspended if there is no message in the queue until such time as another process writes a message to the queue.

void SendMessage (MESSAGE_HANDLE MessageQueue, char * Buffer, int Size);
This system call adds a message to the tail of the message queue represented by the message handle MessageQueue. The message is taken from the area of memory pointed to by Buffer, and is assumed to be Size bytes long. If the internal buffers used by the operating system to store the message queue are full, the calling process is suspended until such time as space becomes available in the buffers, due to another process removing a message or messages from the message queue.

void SendSignal (PID Process);
This system call sends a signal to the process whose process identifier is Process. If the process has set up a signal handler then it will be automatically invoked. Otherwise the receiving process continues execution uninterrupted. See SetSignalHandler() and GetParentsProcessID().

void SetSignalHandler (void (* Func)());
This system call sets the function Func as the signal handler for the calling process. If a signal is received by the process then execution is diverted to the start of the handler function Func. If the handler function does not terminate the process then execution continues from the point of the interruption when the handler function returns. Once a signal has been handled in this way the action to be taken on receipt of a signal is reset to the default (signals are ignored). See SendSignal().

void Signal (SEMAPHORE Sem);
This system call increments the value associated with the semaphore Sem. This may allow

a process that was waiting on the signal to decrement the semaphore and continue its execution. See `Wait()` and `CreateSemaphore()`.

`void Sleep (long Interval);`
This system call suspends the calling process for approximately the length of time specified by `Interval`. The time is specified in milliseconds. The actual interval for which the process is suspended is dependent on the resolution of the system clock.

`void SuspendProcess (void);`
This system call suspends the calling process until a signal is received. If a signal handler has been set for the process, then, when a signal is received, the signal handling function will be invoked, and, if the signal handler does not terminate the process, execution continues from the statement following the `SuspendProcess()` call. If no signal handler has been set for the process then execution simply continues from the statement following the `SuspendProcess()` call. See `SetSignalHandler()` and `SendSignal()`.

`void Wait (SEMAPHORE Sem);`
This system call checks the value associated with the semaphore `Sem`. If this value is greater than zero, then the value is decremented by one and the system call returns immediately. If the value is zero then the calling process is suspended until such time as the semaphore is signalled by another process. See `Signal()` and `CreateSemaphore()`.

`void WaitForChildProcess (void);`
This system call suspends the calling process until such time as one of the child processes of the calling process terminates. If one of the child processes has already terminated than the call returns immediately. If the calling process has n child processes then the call returns immediately. See `CreateProcessFromDisk()` and `CreateProcessFromMemory()`.

`void Write (FILE_HANDLE File, char * Buffer, int Size);`
This system call writes `Size` bytes to the file represented by the file handle `File` from the area of memory pointed to by `Buffer`. See `Read()`.

A.3. Concurrency

A.3.1. Multitasking and Process Switching

This benchmark was performed by executing a number of Sieves of Eratosthenes in parallel and comparing the resulting times with that for the single Sieve of Eratosthenes. There were two main versions of this algorithm. The first made use of the disk-based process creation features of some of the operating systems and the second made use of the memory-based process creation facilities.

A.3.1.1. Disk-Based Process Creation Facility

```
/* Eratosthenes Sieve Prime Number Program.
   Takes the number of sieves to be executed from the command line,
   and executes them by using the disk-based process creation system
   call.
   George Wells  --  2 January 1990 */

#include "timer.i"

void main (int argc, char ** argv)
{ int number /* of sieves */,
      i /* loop counter */;
  TIME t1, t2 /* Timer ticks */;

  if (argc < 2)
    fatal_error("\nuse: nsieve <number of sieves> [ priority ]\n");
```

```
  number = atoi(argv[1]);
  if (number < 1 || number > 100)
    fatal_error("\nInvalid number of sieves (1..100)\n");

  printf("Executing %d sieves in parallel, using the disk-based system call\n",
         number);

  set_time(&t1);
  for (i = 1; i <= number; i++)
    CreateProcessFromDisk("SIEVE");
  for (i = 1; i <= number; i++)
    WaitForChildProcess();
  set_time(&t2);

  printf("\n%d sieves using CreateProcessFromDisk(): ", number);
  report_time(&t1, &t2);

} /* csieve.c */
```

The program SIEVE referred to in the above algorithm is an executable version of the Sieve of Eratosthenes algorithm presented in section A.1, without the timing calls.

A.3.1.2. Memory-Based Process Creation Facility

```
/* Eratosthenes Sieve Prime Number Program.
   Takes the number of sieves to be executed from the command line,
   and executes them by using the memory-based process creation system
   call.
   George Wells  --  2 January 1990 */

#define true 1
#define false 0
#define size 8190
#define sizepl 8191

#include "timer.i"
#include <stdio.h>

char flags[sizepl];  /* The sieve */
void sieve (void);

void main (int argc, char ** argv)
{ int number /* of sieves */,
      i /* loop counter */;
  TIME t1, t2;

  if (argc != 2)
    fatal_error("\nuse: nsieve <number of sieves>");
  number = atoi(argv[1]);
  if (number < 1 || number > 100)
    fatal_error("\nInvalid number of sieves (1..100)");
  printf("Executing %d sieves in parallel, using the memory-based system call\n",
         number);

  set_time(&t1);
  for (i = 1; i <= number; i++)
    CreateProcessFromMemory(sieve);
  for (i = 1; i <= number; i++)
    WaitForChildProcess();
  set_time(&t2);

  printf("\n%d sieves using CreateProcessFromMemory(): ", number);
  report_time(&t1, &t2);

} /* nsieve.c */

void sieve (void)
{ int i, prime, k, count, iter;
```

```
  for (iter = 1; iter <= 10; iter++)
    { count = 0;
      for (i = 0; i <= size; i++)
        flags[i] = true;
      for (i = 0; i <= size; i++)
        { if (flags[i])
            { prime = i + i + 3;
              k = i + prime;
              while (k <= size)
                { flags[k] = false;
                  k += prime;
                }
              count++;
            }
        }
    }
} /* sieve */
```

A.3.2. Process Creation

Again there were two types of algorithm used for this section - disk-based and memory-based. In both cases the algorithm followed was to create a given number of child processes, each of which did nothing but exit immediately. The parent process made no attempt to wait for the termination of the children.

A.3.2.1. Disk-Based Process Creation Facility

```
/* Program to test process creation time by creating empty child processes,
   using the disk-based process creation facility.
   George Wells  --  2 January 1990 */

#include "timer.i"

main (int argc, char ** argv)
{ TIME t1, t2 /* Timer ticks */;
  int num_children,
      i; /* Loop counter */

  if (argc != 2)
    fatal_error("\nuse: procreate num_children\n");
  num_children = atoi(argv[1]);

  set_time(&t1);
  for (i = 0; i < num_children; i++)
    CreateProcessFromDisk("CHILD");
  set_time(&t2);

  printf("\nCreate %d child processes: ", num_children);
  report_time(&t1, &t2);

} /* procreate.c */
```

The program CHILD referred to in the above algorithm is an executable form of the empty C program shown below.

```
/* Empty child process to test process creation.
   George Wells  --  2 January 1990 */

main ()
{} /* child */
```

A.3.2.2. Memory-Based Process Creation Facility

```
/* Program to test process creation time by creating empty child processes,
   using the memory-based process creation facility.
   George Wells  --  2 January 1990 */

#include "timer.i"

void child (void)
/* Child process - does nothing except exit immediately */
{ exit(0);
} /* child */

void main (int argc, char ** argv)
{ TIME t1, t2 /* Timer ticks */;
  int num_children,
      i;  /* Loop counter */

  if (argc != 2)
    fatal_error("\nuse: fcreate num_children\n");
  num_children = atoi(argv[1]);

  set_time(&t1);
  for (i = 0; i < num_children; i++)
    CreateProcessFromMemory(child);
  set_time(&t2);

  printf("\n%d child processes created using CreateProcessFromMemory(): ", num_children);
  report_time(&t1, &t2);

} /* fcreate.c */
```

A.4. Interprocess Communication

There were two tests performed in this section. The first made use of 100 byte messages and a receiving process that did nothing except receive the messages. The second used a circle of three processes where the first process would send a one byte message to the second process, which would pass it on to the third process, which would pass it back to the first process. The first process would then proceed to send the second message, and so on.

A.4.1. Simple Message Passing

This benchmark was performed using several different mechanisms, namely message passing, pipes and shared memory.

A.4.1.1. Simple Message Passing using Messages

```
/* Program to test message passing by sending messages of 100 bytes to a null
   receiver using messages.
   George Wells  --  2 January 1990 */

#include "timer.i"

void receiver (void);

MESSAGE_HANDLE msg_id;
```

```
void main (int argc, char ** argv)
{ TIME t1, t2 /* Timer ticks */;
  int i;      /* Loop counter */
  unsigned int num_messages;
  char message[100];

  if (argc != 2)
    fatal_error("\nuse: mess100 num_messages\n");
  num_messages = atoi(argv[1]);

  msg_id = CreateMessageQueue();

  CreateProcessFromMemory(receiver);

  for (i = 0; i < 100; i++)
    message[i] = '#';

  set_time(&t1);
  for (i = 0; i < num_messages; i++)
    SendMessage(msg_id, message, 100);
  set_time(&t2);

  printf("\n%u messages sent using IPC: ", num_messages);
  report_time(&t1, &t2);

} /* mess100.c */

void receiver (void)
/* Function to receive messages */
{ char message[99];

  for (;;)
    ReceiveMessage(msg_id, message, 100);
} /* receiver */
```

A.4.1.2. Simple Message Passing using Pipes

Typically, pipes interface with the usual file handling facilities and are most easily used by redirecting the standard input and output files of the processes working with the pipes. This is the method which is reflected by the algorithm given below.

```
/* Program to test message passing by sending messages of 100 bytes to a null
   receiver using a pipe.
   George Wells -- 2 January 1990 */

#include "timer.i"

void main (int argc, char ** argv)
{ TIME t1, t2 /* Timer ticks */;
  int i;      /* Loop counter */
  FILE_HANDLE mess_pipe[2];
  unsigned num_messages;
  char message[100];

  if (argc != 2)
    fatal_error("\nuse: pipe100 num_messages\n");
  num_messages = atoi(argv[1]);

  CreatePipe(mess_pipe);

  JoinFileDescriptors(stdinput, mess_pipe[0]);
  CreateProcessFromDisk("NULLREC");

  for (i = 0; i < 100; i++)
    message[i] = '#';

  set_time(&t1);
  for (i = 0; i < num_messages; i++)
    Write(mess_pipe[1], message, 100);
```

```
  set_time(&t2);

  printf("\n%d messages sent using a pipe: ", num_messages);
  report_time(&t1, &t2);

} /* pipe100.c */
```

As can be seen from the above algorithm, the receiving process is a separate program, the algorithm for which follows.

```
/* Null receiver for messages using pipes.
   George Wells  --  2 January 1990 */

void main (int argc, char ** argv)
{ char message[100];

  for (;;)
    Read(stdinput, message, 100);
} /* nullrec.c */
```

A.4.1.3. Simple Message Passing using Shared Memory

This benchmark made use of a shared memory segment common to both the sending and receiving processes. This necessitated the use of semaphores to synchronise access to the shared area of memory.

```
/* Program to test message passing using shared memory and semaphores.
   George Wells  --  2 January 1990 */

#include "timer.i"

#define size 100

void receive (unsigned);

SEMAPHORE sem1, sem2;
SHARED_MEM_HANDLE shm_id;

void main (int argc, char ** argv)
{ unsigned num_messages,
           i; /* Loop counter */
  int status  /* Of semaphore operations */,
      j;      /* Loop counter */
  TIME t1, t2 /* Timer ticks */;
  char message[size],
       * shm_area;

  if (argc != 2)
    fatal_error("\nuse: shm100 <number of messages>\n");
  num_messages = atoi(argv[1]);

  CreateSemaphore(&sem1, 1);
  CreateSemaphore(&sem2, 0);

  shm_id = CreateSharedMemorySegment(size);

  CreateProcessFromMemory(receive, num_messages);

  shm_area = AttachSharedMemorySegment(shm_id);

  for (i = 0; i < size; i++)
    message[i] = '*';

  set_time(&t1);
  for (i = 0; i < num_messages; i++)
    { Wait(sem1);
      for (j = 0; j < size; j++)
```

```
        shm_area[j] = message[j];
      Signal(sem2);
    }
  set_time(&t2);

  printf("\n%u messages sent using shared memory: ", num_messages);
  report_time(&t1, &t2);
} /* shm100.c */

void receive (unsigned number)
/* Function to receive messages */
{ unsigned i;
  int j;
  char message[size],
       * shm_area;

  shm_area = AttachSharedMemorySegment(shm_id);

  for (i = 0; i < number; i++)
    { Wait(sem2);
      for (j = 0; j < size; j++)
        message[j] = shm_area[j];
      Signal(sem1);
    }

} /* receive */
```

A.4.2. Circular Message Passing

This benchmark was performed for message queues and pipes only. There is no reason why shared memory could not have been used, but the extra overhead due to the semaphores required for synchronising the access to the shared memory area would have become excessive in this case where only a byte at a time was being exchanged.

A.4.2.1. Circular Message Passing using Messages

```
/* Program to test message passing by sending messages around a ring of processes,
   using message queues.
   George Wells -- 2 January 1990 */

#include "timer.i"

void circle (MESSAGE_HANDLE, MESSAGE_HANDLE);

main (int argc, char ** argv)
{ TIME t1, t2 /* Timer ticks */;
  int i,       /* Loop counter */
      num_messages;
  MESSAGE_HANDLE msg_id1, msg_id2, msg_id3;
  char message[1],  /* Sent message */
       rmessage[1]; /* Received message */

  if (argc != 2)
    fatal_error("\nuse: circlep num_messages\n");
  num_messages = atoi(argv[1]);

  msg_id1 = CreateMessageQueue();
  msg_id2 = CreateMessageQueue();
  msg_id3 = CreateMessageQueue();

  CreateProcessFromMemory(circle, msg_id1, msg_id2);
  CreateProcessFromMemory(circle, msg_id2, msg_id3);

  set_time(&t1);
```

```
  for (i = 0; i < num_messages; i++)
    for (message[0] = 'A'; message[0] <= 'Z'; message[0]++)
      { SendMessage(msg_id1, message, 1);
        ReceiveMessage(msg_id3, rmessage, 1); 0);
      }
  set_time(&t2);

  printf("\n%d 1 byte messages sent using IPC: ", num_messages*26);
  report_time(&t1, &t2);

} /* circlem.c */

void circle (MESSAGE_HANDLE in_msgq, MESSAGE_HANDLE out_msgq)
{ char message[1];

  for(;;)
    { ReceiveMessage(in_msgq, message, 1);
      SendMessage(out_msgq, message, 1);
    }
} /* circle */
```

A.4.2.2. Circular Message Passing using Pipes

```
/* Program to test message passing by sending messages in a ring of processes,
   using pipes.
   George Wells  --  2 January 1990 */

#include "timer.i"

void circle (FILE_HANDLE, FILE_HANDLE);

main (int argc, char ** argv)
{ TIME t1, t2 /* Timer ticks */;
  int i,
      num_messages;
  FILE_HANDLE pipe1[2],
              pipe2[2],
              pipe3[2] /* Pipe descriptors for circle */;
  char message,        /* Sent message */
       rmessage;       /* Received message */

  if (argc != 2)
    fatal_error("\nuse: circlep num_messages\n");
  num_messages = atoi(argv[1]);

  CreatePipe(pipe1);
  CreatePipe(pipe2);
  CreatePipe(pipe3);

  CreateProcessFromMemory(circle, pipe1[0], pipe2[1]);
  CreateProcessFromMemory(circle, pipe2[0], pipe3[1]);

  set_time(&t1);
  for (i = 0; i < num_messages; i++)
    for (message = 'A'; message <= 'Z'; message++)
      { Write(pipe1[1], &message, sizeof(message));
        Read(pipe3[0], &rmessage, sizeof(rmessage));
      }
  set_time(&t2);

  printf("\n%d 1 byte messages sent: ", num_messages*26);
  report_time(&t1, &t2);

} /* circlep.c */

void circle (FILE_HANDLE in_pipe, FILE_HANDLE out_pipe)
{ char message;

  for(;;)
    { Read(in_pipe, &message, sizeof(message));
      Write(out_pipe, &message, sizeof(message));
```

```
    }
} /* circle */
```

A.5. Synchronisation

This test was performed by first running an unsynchronised program, consisting of two processes. The second step was to run a synchronised version of the same program, and by comparing the execution times the time taken by the synchronisation facilities could be calculated.

A.5.1. Unsynchronised Program

```
/* Program to test synchronisation.
   This version makes no use of synchronisation, but simply sets up two
   processes - one writing "Hi" and one "Ho" on the screen.
   George Wells  --  2 January 1990 */

#include "timer.i"

void hiho (int, char *);

void main (int argc, char ** argv)
{ int number; /* Number of HiHo's */
  TIME t1, t2 /* Timer ticks */;

  if (argc != 2)
    fatal_error("\nuse: hiho <number of HiHo's>\n");
  number = atoi(argv[1]);
  if (number < 1 || number > 1000)
    fatal_error("\nInvalid number of HiHo's (1..1000)\n");

  set_time(&t1);
    CreateProcessFromMemory(hiho, number, "Hi");
    CreateProcessFromMemory(hiho, number, "Ho");
    WaitForChildProcess();
    WaitForChildProcess();
  set_time(&t2);

  printf("\n%d HiHo's without synchronisation: ", number);
  report_time(&t1, &t2);
} /* hiho.c */

void hiho (int number, char * message)
/* Procedure to print number messages */
{ int i;

  for (i = 0; i < number; i++)
    printf(message);

} /* hiho */
```

A.5.2. Synchronised Program

```
/* Program to test synchronisation.
   This version uses semaphores to synchronise two processes - one writing "Hi"
   and the other writing "Ho" on the screen.
   George Wells  --  2 January 1990 */

#include "timer.i"

void hiho (int, char *, SEMAPHORE, SEMAPHORE);
```

```
void main (int argc, char ** argv)
{ int number;  /* Number of HiHo's */
  TIME t1, t2; /* Timer ticks */
  SEMAPHORE sem1, sem2;

  if (argc != 2)
    fatal_error("\nuse: hihoss <number of HiHo's>\n");
  number = atoi(argv[1]);
  if (number < 1 || number > 1000)
    fatal_error("\nInvalid number of HiHo's (1..1000)\n");

  CreateSemaphore(&sem1, 1);
  CreateSemaphore(&sem2, 0);

  set_time(&t1);
    CreateProcessFromMemory(hiho, number, "Hi", sem1, sem2);
    CreateProcessFromMemory(hiho, number, "Ho", sem2, sem1);
    WaitForChildProcess();
    WaitForChildProcess();
  set_time(&t2);

  printf("\n%d HiHo's with synchronisation using semaphores: ", number);
  report_time(&t1, &t2);
} /* hihoss.c */

void hiho (int number, char * message, SEMAPHORE my_sem, SEMAPHORE brothers_sem)
/* Procedure to print number messages, using my_sem and brothers_sem for
   synchronisation */
{ int i;

  for (i = 0; i < number; i++)
    { Wait(my_sem);
      printf(message);
      Signal(brothers_sem);
    }

} /* hiho */
```

A.6. Exception Handling

There were two main algorithms used in this test. The first made use of the process suspension (or sleep) facilities of the operating systems and the second the process signalling facilities. The time taken by the exception handling facilities was measured indirectly by assessing the impact that the execution of the benchmark program had on the performance of a timer process. The algorithm for the timer task was the Sieve of Eratosthenes, shown below. This program is referred to as TIMER in the succeeding algorithms.

```
/* Eratosthenes Sieve Prime Number Program.  Executes continuously,
   timing each iteration.
   George Wells  --  2 January 1990 */

#include "timer.i"
#define true 1
#define false 0
#define size 8190
#define sizep1 8191

char flags[sizep1];

main (int argc, char ** argv)
{ int i, prime, k, count, iter;
  TIME t1, t2; /* Timer ticks */

  for (;;)
    { set_time(&t1);
      for (iter = 1; iter <= 10; iter++)
```

```
    { count = 0;
      for (i = 0; i <= size; i++)
        flags[i] = true;
      for (i = 0; i <= size; i++)
        { if (flags[i])
            { prime = i + i + 3;
              k = i + prime;
              while (k <= size)
                { flags[k] = false;
                  k += prime;
                }
              count++;
            }
        }
    }
  set_time(&t2);
  printf("\n10 sieves: ");
  report_time(&t1, &t2);
 }
} /* timer.c */
```

A.6.1. Process Suspension

```
/* Program to test exception handling.
   The program loops waiting for a timer interrupt at intervals of a given
   number of milliseconds.
   George Wells  -- 2 January 1990 */

#include "timer.i"

void main (int argc, char ** argv)
{ int interval /* Number of milliseconds delay */;
  TIME t1, t2  /* Timer ticks */;

  if (argc != 2)
    fatal_error("\nuse: int1 <interval>\n");
  interval = atoi(argv[1]);
  if (interval < 1 || interval > 100)
    fatal_error("\nInvalid interval (1..100)\n");

  CreateProcessFromDisk("TIMER");

  for (;;)
    Sleep((long)interval);

} /* int1.c */
```

A.6.2. Interprocess Signalling

```
/* Program to test interrupt handling.
   The program loops waiting for a timer interrupt at intervals of a given
   number of milliseconds.
   George Wells  -- 2 January 1990 */

#include "timer.i"

void sig_handler (void);
void interrupter (void);

long interval;

void main (int argc, char ** argv)
{
  if (argc != 2)
    fatal_error("\nuse: int2 <interval>\n");
  interval = atoi(argv[1]);
  if (interval < 1 || interval > 100)
    fatal_error("\nInvalid interval (1..100)\n");
```

```
  CreateProcessFromDisk("TIMER");

  CreateProcessFromMemory(interrupter);

  for (;;)
    { SetSignalHandler(sig_handler);
      SuspendProcess();
    }

} /* int2.c */

void sig_handler (void)
{} /* sig_handler */

void interrupter (void)
/* Function to interrupt the parent at regular intervals */
{ PID ppid /* Parent's process ID */;

  ppid = GetParentsProcessID();
  for (;;)
    { Sleep(interval);
      SendSignal(ppid);
    }
} /* interrupter */
```

B. Results of Benchmarks

This appendix presents the data that was collected for the benchmarks of the operating systems. The results are presented in the same order as in the body of this report. The headings for the columns of data are either the names of the programs (together with any parameters that may have been specified), or an indication of the number and priority of background tasks that were used.

Each test was repeated ten times and the average of the ten results found. In addition, the standard deviation of the result is given. In some cases a ratio has also been calculated. This is generally the ratio of the result in one column with that in the first column. Occasionally, where a table has been split into two to fit across the page, the ratio is of the result in that column with the first column in the first section of the table. In certain cases a further row has been added, giving the result of some calculation on the result, such as the length of time per operation repeated in the test.

B.1. XENIX System V

The background tasks in this case are indicated by headings of the form `+b <number> <nice-value>`, where `<number>` reflects the number of background processes that were used (one or ten) and `<nice-value>` specifies the priority. The priorities are specified in terms of a change to the standard priority, where a negative nice-value implies increasing the priority of the background processes and a positive value decreasing the priority of the background processes. A value of zero means that the background processes ran at the same priority as the benchmark program.

B.1.1. Concurrency

In this section the programs used are as follows.

sieve The sequential form of the Sieve of Eratosthenes.

nsieve The concurrent form of the Sieve of Eratosthenes, using the `fork()` system call.

csieve The concurrent form of the Sieve of Eratosthenes, using the `exec()` system call in conjunction with the `fork()` system call.

procreate The creation of child processes from disk, using the `exec()` system call in conjunction with the `fork()` system call. The child processes were empty programs, which terminated immediately.

fcreate The creation of child processes from memory, using the `fork()` system call. The child processes were empty functions, which terminated immediately.

	sieve	nsieve 1	csieve 1	nsieve 2	csieve 2	nsieve 10	csieve 10
1	3.04	3.04	3.28	6.08	6.46	30.36	32.20
2	3.04	3.04	4.76	7.94	7.56	31.62	33.80
3	3.04	3.06	3.22	6.08	7.56	30.38	32.22

4	3.06	3.04	4.76	6.10	6.44	30.38	33.78
5	3.04	3.04	3.22	7.92	6.44	31.62	33.22
6	3.04	4.96	4.78	6.06	7.54	30.38	33.80
7	3.04	3.04	3.22	6.06	6.44	30.38	32.20
8	3.04	3.04	4.74	7.92	6.44	31.62	33.80
9	3.04	3.04	3.22	6.08	7.56	30.38	32.20
10	3.04	3.04	4.78	6.08	7.54	30.38	33.80
Average	3.04	3.23	4.00	6.63	7.00	30.75	33.10
Std Dev'n	0.01	0.60	0.77	0.87	0.55	0.58	0.73
s/sieve	3.04	3.23	4.00	3.32	3.50	3.08	3.31

	procreate 100	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	19.42	16.12	*	15.50	37.58	*	22.08
2	17.78	19.64	*	15.64	23.82	*	17.02
3	18.18	18.22	*	17.06	40.22	*	17.56
4	19.08	16.20	*	15.66	39.46	*	22.36
5	15.30	16.30	*	18.46	36.32	*	18.22
6	15.30	21.32	*	19.52	27.10	*	17.50
7	16.04	15.24	*	17.02	35.40	*	16.02
8	18.58	16.66	*	16.12	52.50	*	17.16
9	19.08	16.24	*	17.04	64.32	*	17.46
10	16.18	15.32	*	15.58	22.12	*	17.06
Average	17.49	17.13		16.76	37.88		18.24
Std Dev'n	1.48	1.95		1.28	12.87		1.70
Ratio		0.98	*	0.96	2.17	*	1.04
s/child:	0.175						

	procreate 20	procreate 30	procreate 40	procreate 60	procreate 80
1	0.20	2.12	7.04	10.10	10.38
2	0.20	7.54	4.58	8.04	12.32
3	0.20	2.26	4.06	9.08	11.14
4	3.30	6.50	5.78	13.26	12.38
5	0.20	4.82	4.68	7.32	12.58
6	0.20	4.10	7.28	10.10	15.26
7	0.20	4.22	6.68	10.32	15.26
8	0.20	3.12	4.22	15.46	14.40
9	0.18	2.20	4.86	7.62	12.42
10	0.20	3.46	6.30	9.26	12.52
Average:	0.51	4.03	5.55	10.06	12.87
Std Dev'n	0.98	1.71	1.10	2.55	1.38
Ratio	0.03	0.23	0.32	0.57	0.74
s/child	0.025	0.134	0.139	0.168	0.161

	fcreate 100	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	1.62	2.42	*	2.32	37.14	*	6.04
2	1.60	2.76	*	2.40	28.42	*	3.24
3	2.38	2.16	*	1.78	35.06	*	4.84
4	2.32	3.40	*	2.24	27.10	*	3.46
5	1.58	1.58	*	2.32	34.26	*	3.62
6	2.38	3.20	*	2.36	29.38	*	2.38
7	1.58	2.42	*	1.78	36.28	*	2.40
8	1.58	2.66	*	2.26	38.20	*	2.72
9	2.40	2.20	*	2.32	37.08	*	3.46
10	1.62	3.42	*	1.78	9.20	*	3.52
Average:	1.91	2.62		2.16	31.21		3.57
Std Dev'n	0.39	0.59		0.26	8.44		0.71
Ratio		1.38	*	1.13	16.38	*	1.87
s/child:	0.019						

	fcreate 20	fcreate 40	fcreate 60	fcreate 80
1	0.20	1.36	1.06	1.28
2	0.18	1.44	1.06	1.30
3	0.18	0.56	1.04	1.32

4	0.18	0.56	1.06	1.28
5	0.20	0.56	1.04	2.70
6	0.22	0.56	1.06	2.66
7	1.72	0.56	1.04	2.66
8	0.20	1.38	1.12	2.60
9	0.18	0.54	1.06	1.26
10	0.18	0.54	1.18	1.28
Average:	0.34	0.81	1.07	1.83
Std Dev'n	0.48	0.36	0.04	0.68
Ratio	0.18	0.42	0.56	0.96
s/child:	0.017	0.020	0.018	0.023

B.1.2. Interprocess Communication

In this section the programs used are as follows.

`mess100` The 100-byte message passing program, using the UNIX System V message facilities.

`pipe100` The 100-byte message passing program, using pipes.

`shm100` The 100-byte message passing program, using the UNIX System V shared memory facilities.

`circlep` The one-byte, circular message passing program, using pipes.

`circlem` The one-byte, circular message passing program, using the UNIX System V message facilities.

	mess100 10	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	0.00	0.02	0.04	0.02	0.00	*	0.00
2	0.04	0.00	0.00	0.04	0.02	*	0.02
3	0.02	0.04	0.00	0.00	10.22	*	0.00
4	0.00	0.02	0.00	0.00	0.00	*	0.00
5	0.02	0.00	0.04	0.02	0.00	*	1.98
6	0.04	0.04	0.02	0.02	0.00	*	0.02
7	0.00	0.04	0.02	0.02	10.38	*	0.00
8	0.02	0.00	0.04	0.00	0.02	*	0.00
9	0.02	0.00	0.00	0.04	0.00	*	0.04
10	0.04	0.04	0.00	0.02	0.00	*	0.00
Average:	0.02	0.02	0.02	0.02	2.06		0.21
Std Dev'n	0.01	0.02	0.02	0.01	4.28		0.62
Ratio		1.00	0.80	0.90	103.20	*	10.30

	mess100 100	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	0.16	0.12	0.16	0.14	10.16	*	1.84
2	1.84	0.16	0.12	0.16	10.36	*	3.10
3	0.12	0.14	0.14	0.14	19.18	*	3.06
4	1.84	0.16	0.16	0.16	10.16	*	1.86
5	0.16	0.16	1.84	1.84	10.10	*	0.18
6	0.14	0.12	0.14	0.16	8.10	*	0.12
7	0.16	0.16	0.16	0.14	10.16	*	0.18
8	0.14	0.12	0.16	0.16	10.12	*	0.12
9	0.16	1.84	0.16	0.14	8.14	*	0.16
10	0.14	0.16	0.14	1.82	10.12	*	0.12
Average:	0.49	0.31	0.32	0.49	10.66		1.07
Std Dev'n	0.68	0.51	0.51	0.67	2.95		1.20
Ratio		0.65	0.65	1.00	21.93	*	2.21

	mess100 1000	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	1.48	3.56	*	3.54	21.12	*	3.32

2	2.52	2.46	*	2.52	21.16	*	3.38
3	1.46	2.48	*	1.48	21.18	*	2.52
4	1.48	3.56	*	2.44	21.08	*	3.56
5	2.54	3.54	*	2.54	21.10	*	2.46
6	2.52	1.46	*	3.54	21.10	*	2.46
7	1.46	2.48	*	1.48	21.10	*	3.54
8	1.48	3.58	*	1.46	21.12	*	2.52
9	1.46	3.54	*	3.52	21.10	*	2.46
10	2.52	1.46	*	2.52	21.12	*	2.46
Average:	1.89	2.81		2.50	21.12		2.87
Std Dev'n	0.52	0.82		0.80	0.03		0.48
Ratio		1.49	*	1.32	11.16	*	1.52

	mess100 10000	mess100 20000	mess100 30000	mess100 40000
1	15.18	29.64	44.38	59.20
2	15.18	29.62	45.60	59.18
3	15.18	29.60	45.60	59.24
4	14.80	29.62	44.42	59.16
5	14.80	29.60	44.38	60.82
6	15.20	29.56	45.62	59.16
7	15.24	29.60	44.40	59.18
8	15.20	30.40	44.42	60.80
9	15.18	30.40	44.40	59.18
10	15.20	30.40	45.58	59.22
Average:	15.12	29.84	44.88	59.51
Std Dev'n	0.16	0.36	0.59	0.65
Ratio		1.97	2.97	3.94

	pipe100 10	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	0.00	0.00	0.02	0.02	0.02	*	0.02
2	0.02	0.02	0.02	0.00	0.02	*	0.02
3	0.02	0.02	0.02	0.02	0.02	*	0.02
4	0.00	0.00	0.00	0.02	0.02	*	0.02
5	0.02	0.02	0.00	0.00	0.02	*	0.02
6	0.02	0.00	0.02	0.02	0.00	*	0.00
7	0.02	0.02	0.02	0.00	0.02	*	0.02
8	0.00	0.00	0.02	0.02	0.00	*	0.02
9	0.02	0.00	0.02	0.02	0.02	*	0.02
10	0.02	0.02	0.02	0.00	0.02	*	0.00
Average:	0.01	0.01	0.02	0.01	0.02		0.02
Std Dev'n	0.01	0.01	0.01	0.01	0.01		0.01
Ratio		0.71	*	0.86	1.14	*	1.14

	pipe100 100	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	0.12	0.12	0.14	0.14	0.12	*	0.12
2	0.14	1.68	0.12	0.12	0.12	*	0.12
3	0.14	0.12	0.12	0.14	0.14	*	0.12
4	0.12	0.14	0.12	1.68	0.12	*	0.14
5	1.66	0.14	0.14	0.14	0.12	*	0.14
6	0.12	0.12	0.14	0.14	0.14	*	0.12
7	0.12	1.68	2.76	0.12	0.12	*	0.14
8	0.14	0.12	0.12	1.68	0.14	*	0.14
9	0.12	0.54	0.12	0.12	0.14	*	0.12
10	0.14	1.68	0.12	0.14	0.14	*	0.14
Average:	0.28	0.63	0.39	0.44	0.13		0.13
Std Dev'n	0.46	0.70	0.79	0.62	0.01	*	0.01
Ratio		2.25	1.39	1.57	0.46	*	0.46

	pipe100 1000	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	2.50	4.38	4.36	4.50	39.20	*	3.78
2	3.44	5.54	4.34	3.42	28.08	*	6.80
3	3.48	5.50	4.48	3.42	28.08	*	6.16
4	3.46	5.48	4.48	3.42	29.04	*	3.40

5	3.44	3.44	4.48	4.60	39.04	*	4.50
6	3.20	4.50	4.48	3.40	39.46	*	4.36
7	3.48	5.48	4.48	3.42	28.06	*	5.50
8	2.52	5.62	4.48	3.42	28.12	*	6.50
9	2.52	3.42	4.48	4.58	29.08	*	3.44
10	2.52	3.64	4.48	3.42	39.06	*	4.50
Average:	3.06	4.70	4.45	3.76	32.72		4.89
Std Dev'n	0.45	0.89	0.05	0.52	5.29		1.20
Ratio		1.54	1.45	1.23	10.71	*	1.60

	pipe100 10000	pipe100 20000	pipe100 30000	pipe100 40000
1	25.52	48.56	73.00	97.82
2	25.52	48.58	73.02	98.22
3	24.40	48.60	73.04	98.30
4	24.52	48.78	73.02	98.52
5	24.46	48.64	73.30	98.20
6	24.26	48.58	73.12	98.36
7	25.52	49.18	73.10	97.30
8	24.46	48.64	73.12	98.08
9	24.44	48.62	73.02	98.48
10	24.42	48.64	73.18	97.04
Average:	24.75	48.68	73.09	98.03
Std Dev'n	0.51	0.18	0.09	0.47
Ratio		1.97	2.95	3.96

	shm100 10	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	0.08	0.08	0.08	0.08	86.40	*	1.60
2	0.08	0.06	1.16	0.08	86.42	*	1.66
3	0.08	0.08	0.08	0.08	86.42	*	1.50
4	0.08	0.08	0.08	0.08	87.42	*	1.50
5	0.08	0.08	0.08	0.08	86.42	*	1.50
6	0.08	0.08	0.08	0.08	86.42	*	1.48
7	1.92	0.08	0.08	0.08	86.42	*	1.48
8	0.08	0.08	1.18	0.08	86.42	*	1.48
9	0.06	0.08	0.08	0.08	87.42	*	1.52
10	0.08	0.08	0.08	0.08	87.42	*	1.44
Average:	0.26	0.08	0.30	0.08	86.72		1.52
Std Dev'n	0.55	0.01	0.44	0.00	0.46		0.06
Ratio		0.30	1.15	0.31	330.98	*	5.79

	shm100 100	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 1 5
1	1.18	1.34	2.02	1.04	986.40	*	4.50
2	1.18	1.16	2.06	2.90	987.42	*	4.36
3	1.16	2.66	2.12	1.16	986.42	*	4.58
4	1.18	2.58	2.02	1.14	986.42	*	4.54
5	1.18	2.66	2.22	0.82	986.42	*	4.52
6	1.18	1.18	2.10	1.06	987.44	*	4.52
7	0.82	2.60	2.02	1.18	987.42	*	4.46
8	1.18	2.68	2.02	1.06	987.42	*	4.44
9	1.18	1.20	2.00	1.20	986.42	*	4.54
10	0.80	2.54	2.04	1.16	986.42	*	4.56
Average:	1.10	2.06	2.06	1.27	986.82		4.50
Std Dev'n	0.15	0.69	0.06	0.55	0.49		0.06
Ratio		1.87	1.87	1.15	893.86	*	4.08

	shm100 1000	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 1 5
1	8.20	15.66	20.10	13.32	9975.18	*	37.48
2	8.20	16.14	20.22	12.76	9946.18	*	37.58
3	9.76	15.14	20.26	12.60	9986.18	*	37.52
4	8.20	16.14	20.34	12.46	9986.18	*	37.50
5	8.20	15.18	20.42	13.28	9987.18	*	37.48
6	9.80	16.02	20.32	13.26	9955.18	*	37.58
7	8.24	16.86	20.46	13.28	9986.16	*	37.50

8	8.18	16.14	20.28	12.68	9986.16	*	37.58
9	9.82	15.12	20.34	12.52	9987.18	*	37.46
10	8.20	16.10	20.34	12.50	9989.06	*	37.44
Average:	8.68	15.85	20.31	12.87	9978.46		37.51
Std Dev'n	0.73	0.54	0.10	0.35	14.48		0.05
Ratio		1.83	2.34	1.48	1149.59	*	4.32

	shm100 10000	shm100 20000	shm100 30000	shm100 40000
1	82.02	143.16	168.52	192.16
2	82.00	133.50	167.52	192.14
3	82.00	134.48	168.50	192.12
4	82.02	138.26	168.50	192.12
5	82.00	133.48	167.52	192.12
6	82.00	133.50	167.50	192.14
7	82.00	144.64	168.48	192.12
8	82.02	162.14	167.52	191.86
9	82.02	141.16	167.50	192.14
10	82.00	134.18	168.48	192.18
Average:	82.01	139.85	168.00	192.11
Std Dev'n	0.01	8.47	0.49	0.09
Ratio		1.71	2.05	2.34

	circlep 100	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	30.06	37.06	*	32.12	294.60	*	47.04
2	30.04	32.12	*	31.12	321.26	*	48.56
3	30.04	34.10	*	36.04	292.34	*	47.60
4	30.06	37.14	*	36.14	283.70	*	48.26
5	30.04	36.02	*	32.10	314.16	*	48.58
6	30.04	33.12	*	36.22	282.52	*	47.38
7	30.04	37.16	*	33.12	311.14	*	48.22
8	30.02	36.16	*	33.12	313.14	*	48.34
9	30.04	36.12	*	35.06	284.24	*	48.34
10	30.04	38.16	*	32.18	291.12	*	49.26
Average:	30.0420	35.7160		33.7220	298.8220		48.1580
Std Dev'n	0.0108	1.8608		1.8526	13.8625		0.6165
Ratio		1.19	*	1.12	9.95	*	1.60

	circlep 50	circlep 150	circlep 200	circlep 250	circlep 300
1	14.92	45.06	60.06	75.00	90.10
2	15.08	44.90	60.00	75.04	90.02
3	15.08	45.08	60.02	75.16	90.06
4	15.06	45.08	60.00	75.08	90.10
5	15.06	44.82	60.04	75.84	90.08
6	15.06	45.06	60.02	75.10	90.02
7	15.08	45.04	60.06	75.12	90.10
8	15.08	46.92	60.00	75.10	90.96
9	15.08	45.04	60.06	75.12	90.10
10	15.06	45.10	60.06	75.10	90.08
Average:	15.0560	45.2100	60.0320	75.1660	90.1620
Std Dev'n	0.0463	0.5763	0.0256	0.2286	0.2676
Ratio	0.50	1.50	2.00	2.50	3.00

	circlem 100	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	23.14	34.12	*	30.08	202.20	*	33.22
2	23.14	31.14	*	30.12	201.18	*	30.58
3	22.86	35.14	*	30.14	181.14	*	29.42
4	23.14	34.12	*	30.12	231.18	*	29.44
5	23.14	34.14	*	30.12	222.18	*	29.44
6	23.16	33.88	*	29.08	222.18	*	31.42
7	23.14	35.14	*	31.12	201.16	*	29.42
8	23.14	34.14	*	30.12	211.16	*	30.44
9	22.84	33.14	*	30.12	221.18	*	29.42
10	23.14	34.12	*	30.12	201.16	*	31.44

Average:	23.0840	33.9080		30.1140	209.4720		30.4240
Std Dev'n	0.1172	1.0738		0.4564	14.1304		1.2177
Ratio		1.47	*	1.30	9.07	*	1.32

	circlem 50	circlem 150	circlem 200	circlem 250	circlem 300
1	12.50	34.28	45.72	57.12	68.54
2	12.58	35.72	45.70	57.12	69.44
3	11.42	34.28	46.30	57.12	69.44
4	11.42	34.28	46.28	57.12	69.46
5	42.58	34.28	46.30	57.12	69.46
6	11.44	34.28	46.30	57.12	69.46
7	11.42	35.72	46.30	57.12	68.54
8	11.42	34.28	46.30	57.12	68.56
9	12.58	34.28	46.30	58.88	68.54
10	11.42	34.26	46.30	57.12	69.46
Average:	14.8780	34.5660	46.1800	57.2960	69.0900
Std Dev'n	9.2478	0.5770	0.2351	0.5280	0.4451
Ratio	0.64	1.50	2.00	2.48	2.99

B.1.3. Synchronisation

In this section the programs used are as follows.

hiho	The unsynchronised HiHo program
hihoss	The synchronised HiHo program, using the UNIX System V semaphore facilities.
hihosp	The synchronised HiHo program, using pipes.

	hiho 1000	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	1.12	2.06	2.06	2.56	21.32	*	3.84
2	1.12	2.06	3.56	1.12	21.28	*	3.10
3	1.10	2.06	2.02	1.12	21.20	*	3.04
4	1.12	2.06	3.56	2.62	21.34	*	3.36
5	1.10	2.56	2.02	1.10	21.32	*	3.58
6	1.12	2.58	3.56	1.12	21.32	*	2.06
7	1.12	2.12	2.02	2.60	21.32	*	3.58
8	1.12	2.58	3.56	1.12	21.32	*	2.06
9	1.12	2.04	2.02	1.10	21.32	*	3.58
10	1.12	2.58	3.30	2.58	21.32	*	2.06
Average:	1.1160	2.2700	2.7680	1.7040	21.3060		3.0260
Std Dev'n	0.0080	0.2498	0.7437	0.7236	0.0380		0.6708
Ratio		2.03	2.48	1.53	19.09	*	2.71

	hihoss 1000	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	6.14	10.36	14.06	9.16	9995.06	*	28.10
2	6.12	12.12	14.46	9.18	9996.02	*	26.12
3	6.12	11.82	14.26	9.02	9996.02	*	26.30
4	6.12	12.18	14.30	9.08	9997.02	*	26.32
5	6.12	10.60	14.26	9.06	9924.00	*	26.36
6	6.12	11.26	12.62	8.74	9997.02	*	28.30
7	6.12	12.70	13.52	8.64	9996.02	*	28.38
8	6.14	12.28	14.40	8.62	9994.02	*	28.28
9	6.12	11.36	14.34	9.46	10008.02	*	28.28
10	6.10	11.50	14.56	9.18	9996.04	*	28.30
Average:	6.1220	11.6180	14.0780	9.0140	9989.9420		27.4740
Std Dev'n	0.0108	0.7103	0.5571	0.2554	22.2850		0.9829
Ratio		1.90	2.30	1.47	1631.81	*	4.49

	hihosp 1000	+b 1 0	+b 1 -5	+b 1 5	+b 10 0	+b 10 -5	+b 10 5
1	9.22	14.00	*	12.40	75.64	*	17.50
2	8.76	15.30	*	13.24	78.04	*	16.12
3	8.90	14.26	*	13.28	102.24	*	17.10
4	9.24	14.32	*	13.30	112.32	*	17.18
5	9.22	14.26	*	12.14	112.32	*	18.30
6	9.10	14.30	*	13.40	80.58	*	19.28
7	9.24	14.12	*	13.24	99.22	*	17.32
8	9.22	14.42	*	12.32	86.62	*	18.28
9	9.24	15.72	*	13.24	66.62	*	17.32
10	9.10	14.26	*	13.30	99.22	*	17.20
Average:	9.1240	14.4960		12.9860	91.2820		17.5600
Std Dev'n	0.1587	0.5267		0.4638	15.1519		0.8187
Ratio		1.59	*	1.42	10.00	*	1.92

B.1.4. Exception Handling

In this section the programs used are as follows.

`timer` The Sieve of Eratosthenes program, used as a basic timing program. It is also run by each of the other programs in this section and is the source of the results in the other columns.

`int1` The interrupt driven program, using the `nap()` system call to suspend itself for a given number of milliseconds.

`int2` The interrupt driven program, using a child process which uses the `nap()` system call to suspend itself for a given number of milliseconds and then sends a signal to the parent process. The parent process has a signal handler set up which does nothing, except return immediately.

	timer	int1 1	int2 1	int1 20	int2 20
1	3.04	6.04	8.66	4.22	5.32
2	3.04	6.04	6.70	4.14	5.30
3	3.04	6.06	7.04	4.22	6.68
4	3.04	6.04	6.34	4.22	5.30
5	3.04	6.04	7.10	3.78	5.34
6	3.04	6.04	7.16	4.22	6.68
7	3.04	6.04	7.44	4.22	5.32
8	3.04	6.06	7.26	4.22	5.30
9	3.04	6.04	7.06	3.78	6.68
10	3.04	6.04	7.00	4.20	5.32
Average:	3.0400	6.0440	7.1760	4.1220	5.7240
Std Dev'n	0.0000	0.0083	0.3030	0.1787	0.6444
Ratio		1.99	2.36	1.36	1.88

	int1 40	int2 40	int1 60	int2 60
1	3.38	4.12	3.26	4.44
2	4.62	4.12	3.26	3.56
3	3.38	4.12	4.72	4.44
4	3.38	3.90	3.26	3.56
5	4.64	4.12	3.26	4.44
6	3.38	4.12	3.26	4.44
7	4.62	4.12	4.72	3.56
8	3.38	4.12	3.26	4.44
9	3.36	4.12	3.26	3.56
10	4.62	4.12	3.26	4.44
Average:	3.8760	4.0980	3.5520	4.0880
Std Dev'n	0.6207	0.0691	0.6070	0.4373

Ratio	1.28	1.35	1.17	1.34

B.2. OS/2

The background tasks in this case are indicated by headings of the form p <number> <priority> and t <number> <priority>, where <number> indicates the number of background tasks (one or ten) and <priority> indicates the priority of the background tasks as an absolute value. The priorities are all within the TIME CRITICAL scheduling class found under OS/2. A numerically higher value for the priority implies a task has greater priority. The base priority used for the test programs was 15. The p or t in the column headings denotes whether the background tasks were run as processes or as threads.

B.2.1. Concurrency

In this section the programs used are as follows.

sieve The sequential form of the Sieve of Eratosthenes.

psieve The concurrent form of the Sieve of Eratosthenes, using processes (the DosExecPgm() system call).

tsieve The concurrent form of the Sieve of Eratosthenes, using threads (the DosCreateThread() system call).

ssieve The concurrent form of the Sieve of Eratosthenes, using sessions (the DosStartSession() system call).

procreate The creation of child processes from disk, using the DosExecPgm() system call. The child processes were empty programs, which terminated immediately.

	sieve	psieve 1	psieve 2	psieve 10	tsieve 1	tsieve 2	tsieve 10
1	2.19	2.94	5.65	27.56	3.72	7.47	28.50
2	2.19	2.91	5.59	27.56	3.72	7.25	28.53
3	2.19	2.90	5.60	27.56	3.72	7.21	28.53
4	2.19	2.91	5.59	27.60	3.72	7.22	28.50
5	2.19	2.88	5.59	27.56	3.72	7.47	28.50
6	2.19	2.87	5.60	27.60	3.72	7.47	28.53
7	2.16	2.90	5.59	27.56	3.72	7.22	28.50
8	2.19	2.90	5.59	27.59	3.72	7.22	28.50
9	2.18	2.91	5.60	27.56	3.71	7.22	28.53
10	2.19	2.91	5.59	27.56	3.72	7.22	28.50
Average:	2.1860	2.9030	5.5990	27.5710	3.7190	7.2970	28.5120
Std Dev'n	0.0092	0.0179	0.0176	0.0170	0.0030	0.1137	0.0147
s/sieve	2.1860	2.9030	2.7995	2.7571	3.7190	3.6485	2.8512

	ssieve 1	ssieve 2	ssieve 10
1	3.12	5.87	28.82
2	2.94	5.71	28.59
3	2.97	5.71	28.60
4	2.93	5.75	28.59
5	2.94	5.75	28.59
6	2.94	5.75	28.66
7	2.97	5.75	28.62

8	2.94	5.72	28.63
9	2.97	5.75	28.60
10	2.94	5.72	28.66
Average:	2.9660	5.7480	28.6360
Std Dev'n	0.0533	0.0440	0.0665
s/sieve	2.9660	2.8740	2.8636

	procreate 100	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	31.53	31.19	93.16	*	31.59	760.90	*
2	31.56	30.72	92.50	*	31.53	763.78	*
3	31.44	31.15	92.81	*	31.53	765.69	*
4	31.56	31.25	93.43	*	31.53	758.65	*
5	31.53	31.21	93.50	*	31.53	756.53	*
6	31.53	31.22	93.75	*	31.53	773.62	*
7	31.56	31.25	93.25	*	31.53	758.53	*
8	31.44	31.25	92.29	*	31.50	758.81	*
9	31.56	31.21	93.87	*	31.53	746.56	*
10	31.53	31.25	93.22	*	31.54	770.75	*
Average:	31.5240	31.1700	93.1780		31.5340	761.3820	
Std Dev'n	0.0441	0.1532	0.4862		0.0211	7.2630	
Ratio		0.99	2.96	*	1.00	24.15	*
s/child:	0.315						

	t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1	31.59	89.63	*	32.13	790.56	*
2	31.54	90.56	*	31.56	758.22	*
3	31.56	93.37	*	31.56	768.47	*
4	31.53	92.78	*	31.56	765.87	*
5	31.57	91.91	*	31.59	783.16	*
6	31.53	91.03	*	31.56	765.79	*
7	31.53	92.38	*	31.54	768.25	*
8	31.56	92.37	*	31.53	770.82	*
9	31.53	92.37	*	31.56	768.34	*
10	31.54	93.06	*	31.59	780.84	*
Average:	31.5433	91.9460		31.6180	772.0320	
Std Dev'n	0.0199	1.1242		0.1716	9.2439	
Ratio	1.00	2.92	*	1.00	24.49	*

	procreate 20	procreate 40	procreate 60	procreate 80
1	6.12	12.57	18.87	25.21
2	6.21	12.57	18.87	25.22
3	6.22	12.53	18.88	25.18
4	6.22	12.56	18.91	25.19
5	6.22	12.56	18.88	25.22
6	6.22	12.56	18.87	25.12
7	6.22	12.53	18.87	25.22
8	6.22	12.57	18.87	25.21
9	6.22	12.57	18.91	25.19
10	6.22	12.53	18.78	25.18
Average:	6.2090	12.5550	18.8710	25.1940
Std Dev'n	0.0298	0.0169	0.0339	0.0291
Ratio	0.20	0.40	0.60	0.80
s/child:	0.310	0.314	0.315	0.315

B.2.2. Interprocess Communication

In this section the programs used are as follows.

`pipe100` The 100-byte message passing program, using pipes.

`shm100` The 100-byte message passing program, using a shared memory segment.

`circlep` The one-byte, circular message passing program, using pipes.

`circleq` The one-byte, circular message passing program, using message queues.

	pipe100 1	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	0.00	0.00	0.25	*	0.00	2.50	*
2	0.00	0.03	0.25	*	0.00	2.50	*
3	0.00	0.00	0.25	*	0.00	2.50	*
4	0.00	0.00	0.25	*	0.03	2.50	*
5	0.00	0.00	0.25	*	0.00	2.50	*
6	0.00	0.00	0.25	*	0.00	2.50	*
7	0.00	0.00	0.25	*	0.00	2.50	*
8	0.00	0.00	0.25	*	0.00	2.50	*
9	0.00	0.00	0.25	*	0.00	2.50	*
10	0.00	0.00	0.25	*	0.00	2.50	*
Average:	0.00	0.00	0.25		0.00	2.50	
Std Dev'n	0.00	0.01	0.00		0.01	0.00	
Ratio		??	??	*	??	??	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		0.00	0.25	*	0.00	2.50	*
2		0.00	0.25	*	0.00	2.50	*
3		0.00	0.25	*	0.00	2.50	*
4		0.00	0.25	*	0.00	2.50	*
5		0.00	0.25	*	0.00	2.50	*
6		0.00	0.25	*	0.00	2.50	*
7		0.00	0.25	*	0.00	2.50	*
8		0.04	0.25	*	0.00	2.50	*
9		0.00	0.25	*	0.00	2.50	*
10		0.00	0.25	*	0.00	2.50	*
Average:		0.00	0.25		0.00	2.50	
Std Dev'n		0.01	0.00		0.00	0.00	
Ratio		??	??	*	??	??	*

	pipe100 10	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	0.04	0.03	2.50	*	0.00	25.00	*
2	0.03	0.03	2.50	*	0.03	25.00	*
3	0.00	0.03	2.50	*	0.03	25.00	*
4	0.04	0.00	2.50	*	0.00	25.00	*
5	0.03	0.00	2.50	*	0.03	25.00	*
6	0.00	0.04	2.50	*	0.03	25.00	*
7	0.03	0.03	2.50	*	0.03	25.00	*
8	0.03	0.03	2.50	*	0.00	25.00	*
9	0.03	0.03	2.50	*	0.03	25.00	*
10	0.03	0.00	2.50	*	0.04	25.00	*
Average:	0.03	0.02	2.50		0.02	25.00	
Std Dev'n	0.01	0.01	0.00		0.01	0.00	
Ratio		0.85	96.15	*	0.85	961.54	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		0.03	2.50	*	0.00	25.00	*
2		0.03	2.50	*	0.03	25.00	*
3		0.00	2.50	*	0.03	25.00	*
4		0.04	2.50	*	0.03	25.00	*
5		0.03	2.50	*	0.00	25.00	*
6		0.04	2.50	*	0.03	25.00	*
7		0.00	2.50	*	0.03	25.00	*
8		0.03	2.50	*	0.03	25.00	*
9		0.03	2.50	*	0.04	25.00	*
10		0.00	2.50	*	0.03	25.00	*
Average:		0.02	2.50		0.03	25.00	

Std Dev'n		0.02	0.00		0.01	0.00	
Ratio		0.85	96.15	*	0.96	961.54	*

	pipe100 100	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	0.22	0.25	25.00	*	0.25	250.00	*
2	0.22	0.22	25.00	*	0.25	250.00	*
3	0.22	0.22	25.00	*	0.21	250.00	*
4	0.22	0.22	25.00	*	0.22	250.00	*
5	0.25	0.22	25.00	*	0.25	250.00	*
6	0.21	0.25	25.00	*	0.25	250.00	*
7	0.22	0.22	25.00	*	0.25	250.00	*
8	0.22	0.25	25.00	*	0.25	250.00	*
9	0.22	0.22	25.00	*	0.25	250.00	*
10	0.22	0.22	25.00	*	0.25	250.00	*
Average:	0.2220	0.23	25.00		0.24	250.00	
Std Dev'n	0.0098	0.01	0.00		0.01	0.00	
Ratio		1.03	112.61	*	1.09	1126.13	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		0.25	25.00	*	0.25	250.00	*
2		0.25	25.00	*	0.22	250.00	*
3		0.21	25.00	*	0.25	250.00	*
4		0.25	25.00	*	0.25	250.00	*
5		0.22	25.00	*	0.25	250.00	*
6		0.25	25.00	*	0.25	250.00	*
7		0.25	25.00	*	0.28	250.00	*
8		0.25	25.00	*	0.25	250.00	*
9		0.22	25.00	*	0.25	250.00	*
10		0.22	25.00	*	0.25	250.00	*
Average:		0.24	25.00		0.25	250.00	
Std Dev'n		0.02	0.00		0.01	0.00	
Ratio		1.06	112.61	*	1.13	1126.13	*

	pipe100 1000	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	2.22	2.25	250.00	*	2.38	2500.00	*
2	2.25	2.25	250.00	*	2.41	2500.00	*
3	2.22	2.25	250.00	*	2.37	2500.00	*
4	2.25	2.25	250.00	*	2.37	2500.00	*
5	2.22	2.25	250.00	*	2.41	2500.00	*
6	2.22	2.25	250.00	*	2.38	2500.00	*
7	2.25	2.25	250.00	*	2.38	2500.00	*
8	2.22	2.25	250.00	*	2.37	2500.00	*
9	2.25	2.25	250.00	*	2.38	2500.00	*
10	2.22	2.25	250.00	*	2.37	2500.00	*
Average:	2.2320	2.2500	250.0000		2.3820		
Std Dev'n	0.0147	0.0000	0.0000		0.0147	0.0000	
Ratio		1.01	112.01	*	1.07	1120.07	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		2.35	250.00	*	2.43	2500.00	*
2		2.25	250.00	*	2.44	2500.00	*
3		2.25	250.00	*	2.50	2500.00	*
4		2.28	250.00	*	2.53	2500.00	*
5		2.25	250.00	*	2.50	2500.00	*
6		2.28	250.00	*	2.53	2500.00	*
7		2.25	250.00	*	2.53	2500.00	*
8		2.25	250.00	*	2.53	2500.00	*
9		2.25	250.00	*	2.53	2500.00	*
10		2.25	250.00	*	2.53	2500.00	*
Average:		2.2567	250.0000		2.5050	2500.0000	
Std Dev'n		0.0304	0.0000		0.0369	0.0000	
Ratio		1.01	112.01	*	1.12	1120.07	*

	pipe100 10000	pipe100 20000	pipe100 30000	pipe100 40000
1	23.34	46.72	70.06	93.41
2	23.34	46.69	70.06	93.44
3	23.35	46.72	70.06	93.41
4	23.34	46.69	70.06	93.41
5	23.34	46.69	70.06	93.40
6	23.38	46.72	70.06	93.43
7	23.34	46.68	70.06	93.43
8	23.34	46.68	70.06	93.40
9	23.35	46.72	70.06	93.41
10	23.34	46.69	70.06	93.41
Average:	23.3460	46.7000	70.0600	93.4150
Std Dev'n	0.0120	0.0167	0.0000	0.0128
Ratio		2.00	3.00	4.00

	shm100 1	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	0.03	0.00	0.00	*	0.00	0.00	*
2	0.00	0.00	0.00	*	0.00	0.00	*
3	0.00	0.00	0.00	*	0.00	0.00	*
4	0.00	0.00	0.00	*	0.00	0.00	*
5	0.00	0.00	0.00	*	0.00	0.00	*
6	0.00	0.00	0.00	*	0.00	0.00	*
7	0.00	0.00	0.00	*	0.00	0.00	*
8	0.00	0.00	0.00	*	0.00	0.00	*
9	0.00	0.00	0.00	*	0.03	0.00	*
10	0.00	0.00	0.00	*	0.00	0.00	*
Average:	0.00	0.00	0.00		0.00	0.00	
Std Dev'n	0.01	0.00	0.00		0.01	0.00	
Ratio		0.00	0.00	*	1.00	0.00	*

	t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1	0.00	0.00	*	0.00	0.00	*
2	0.00	0.00	*	0.00	0.00	*
3	0.00	0.00	*	0.00	0.00	*
4	0.00	0.00	*	0.00	0.00	*
5	0.00	0.00	*	0.03	0.00	*
6	0.00	0.00	*	0.00	0.00	*
7	0.00	0.00	*	0.00	0.00	*
8	0.00	0.00	*	0.00	0.00	*
9	0.00	0.00	*	0.00	0.00	*
10	0.00	0.00	*	0.00	0.00	*
Average:	0.00	0.00		0.00	0.00	
Std Dev'n	0.00	0.00		0.01	0.00	
Ratio	0.00	0.00	*	1.00	0.00	*

	shm100 10	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	0.22	0.19	2.41	*	0.22	33.85	*
2	0.19	0.22	2.41	*	0.19	29.38	*
3	0.18	0.19	2.41	*	0.18	29.34	*
4	0.22	0.00	2.41	*	0.19	29.38	*
5	0.19	0.18	2.41	*	0.19	29.37	*
6	0.18	0.19	2.41	*	0.18	29.35	*
7	0.22	0.22	2.40	*	0.19	29.34	*
8	0.16	0.18	2.41	*	0.19	29.38	*
9	0.18	0.19	2.41	*	0.19	29.37	*
10	0.22	0.22	2.41	*	0.19	29.38	*
Average:	0.20	0.18	2.41		0.19	29.81	
Std Dev'n	0.02	0.06	0.00		0.01	1.35	
Ratio		0.91	12.29	*	0.97	152.11	*

	t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1	0.19	2.41	*	0.22	29.37	*

2		0.22	2.41	*	0.19	29.37	*
3		0.22	2.41	*	0.19	29.38	*
4		0.18	2.41	*	0.22	29.38	*
5		0.19	2.41	*	0.19	29.37	*
6		0.21	2.40	*	0.19	29.37	*
7		0.22	2.40	*	0.19	29.38	*
8		0.18	2.40	*	0.19	29.38	*
9		0.19	2.41	*	0.22	29.37	*
10		0.22	2.41	*	0.18	29.37	*
Average:		0.20	2.41		0.20	29.37	
Std Dev'n		0.02	0.00		0.01	0.00	
Ratio		1.04	12.28	*	1.01	149.87	*

	shm100 100	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	0.53	0.53	24.91	*	0.53	254.34	*
2	0.53	0.53	24.91	*	0.53	254.38	*
3	0.53	0.53	24.91	*	0.53	254.34	*
4	0.53	0.53	24.91	*	0.57	254.35	*
5	0.53	0.50	24.91	*	0.56	254.37	*
6	0.53	0.53	24.91	*	0.56	254.35	*
7	0.53	0.53	24.91	*	0.53	254.37	*
8	0.53	0.53	24.91	*	0.54	254.38	*
9	0.53	0.50	24.91	*	0.53	254.34	*
10	0.50	0.54	24.91	*	0.56	254.38	*
Average:	0.53	0.53	24.91		0.54	254.36	
Std Dev'n	0.01	0.01	0.00		0.02	0.02	
Ratio		1.00	47.27	*	1.03	482.66	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		0.53	24.91	*	0.53	254.37	*
2		0.53	24.91	*	0.57	254.38	*
3		0.53	24.91	*	0.57	254.37	*
4		0.53	24.91	*	0.57	254.38	*
5		0.54	24.91	*	0.57	254.38	*
6		0.53	24.90	*	0.53	254.37	*
7		0.53	24.90	*	0.56	254.38	*
8		0.53	24.90	*	0.53	254.37	*
9		0.53	24.91	*	0.56	254.37	*
10		0.53	24.91	*	0.53	254.38	*
Average:		0.53	24.91		0.55	254.38	
Std Dev'n		0.00	0.00		0.02	0.00	
Ratio		1.01	47.26	*	1.05	482.69	*

	shm100 1000	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	3.84	3.84	249.91	*	3.97	2504.38	*
2	3.84	3.84	249.91	*	3.97	2504.38	*
3	3.84	3.87	249.91	*	3.97	2504.37	*
4	3.84	3.84	249.90	*	4.00	2504.35	*
5	3.87	3.85	249.90	*	4.00	2504.37	*
6	3.87	3.88	249.90	*	4.00	2504.35	*
7	3.87	3.85	249.91	*	3.97	2504.34	*
8	3.87	3.85	249.91	*	3.96	2504.38	*
9	3.87	3.88	249.91	*	3.97	2504.37	*
10	3.87	3.84	249.91	*	4.00	2508.84	*
Average:	3.8580	3.8540	249.9070		3.9810		
Std Dev'n	0.0147	0.0156	0.0046		0.0158	1.3424	
Ratio		1.00	64.78	*	1.03	649.25	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		3.94	249.91	*	4.15	2504.35	*
2		3.94	249.91	*	4.22	2504.37	*
3		3.94	249.91	*	4.21	2504.37	*
4		3.94	249.91	*	4.25	2504.38	*

5	3.94	249.91	*	4.22	2504.34	*
6	3.94	249.91	*	4.22	2504.37	*
7	3.94	249.91	*	4.22	2504.38	*
8	3.94	249.91	*	4.22	2504.34	*
9	3.94	249.91	*	4.25	2504.35	*
10	3.94	249.91	*	4.22	2760.38	*
Average:	3.9400	249.9100		4.2180	2529.9630	
Std Dev'n	0.0000	0.0000		0.0260	76.8057	
Ratio	1.02	64.78	*	1.09	655.77	*

	shm100 10000	shm100 20000	shm100 30000	shm100 40000
1	39.13	78.03	117.00	155.97
2	39.10	78.03	117.00	155.94
3	39.10	78.06	117.00	155.94
4	39.13	78.07	117.00	155.93
5	39.12	78.06	117.00	155.94
6	39.10	78.06	117.00	155.91
7	39.13	78.06	117.00	155.94
8	39.09	78.03	117.00	155.93
9	39.09	78.03	117.00	155.94
10	39.09	78.03	117.00	155.94
Average:	39.11	78.05	117.00	155.94
Std Dev'n	0.02	0.02	0.00	0.01
Ratio		2.00	2.99	3.99

	circlep 100	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	16.85	17.00	1950.00	*	18.19	19500.00	*
2	16.85	17.03	1950.00	*	18.19	19500.00	*
3	16.87	17.03	1950.00	*	18.19	19500.00	*
4	16.84	17.00	1950.00	*	18.19	19500.00	*
5	16.85	17.03	1950.00	*	18.19	19500.00	*
6	16.87	17.00	1950.00	*	18.19	19500.00	*
7	16.84	17.03	1950.00	*	18.18	19500.00	*
8	16.88	17.03	1950.00	*	18.18	19500.00	*
9	16.84	17.00	1950.00	*	18.18	19500.00	*
10	16.87	17.03	1950.00	*	18.18	19500.00	*
Average:	16.8560	17.0180	1950.0000		18.1860	19500.0000	
Std Dev'n	0.0143	0.0147	0.0000		0.0049	0.0000	
Ratio		1.01	115.69	*	1.08	1156.86	*

	t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1	16.97	1950.00	*	17.81	19500.00	*
2	16.97	1950.00	*	17.81	19500.00	*
3	16.97	1950.00	*	18.28	19500.00	*
4	17.00	1950.00	*	18.28	19500.00	*
5	16.97	1950.00	*	18.28	19500.00	*
6	17.06	1950.00	*	18.28	19500.00	*
7	17.10	1950.00	*	18.28	19500.00	*
8	17.10	1950.00	*	18.25	19500.00	*
9	17.09	1950.00	*	18.25	19500.00	*
10	17.09	1950.00	*	18.25	19500.00	*
Average:	17.0389	1950.0000		18.1770	19500.0000	
Std Dev'n	0.0576	0.0000		0.1840	0.0000	
Ratio	1.01	115.69	*	1.08	1156.86	*

	circlep 50	circlep 150	circlep 200	circlep 250	circlep 300
1	8.40	25.19	33.63	42.03	50.43
2	8.81	25.19	33.59	42.00	50.43
3	8.85	25.22	33.63	42.00	50.40
4	8.81	25.22	33.63	42.00	50.43
5	8.82	25.22	33.59	42.00	50.40
6	8.84	25.22	33.62	42.03	50.44
7	8.81	25.21	33.63	42.03	50.41

8	8.81	25.18	33.59	42.03	50.44
9	8.81	25.22	33.59	42.03	50.41
10	8.85	25.22	33.63	42.03	50.44
Average:	8.78	25.21	33.61	42.02	50.42
Std Dev'n	0.13	0.02	0.02	0.01	0.02
Ratio	0.52	1.50	1.99	2.49	2.99

	circleq 100	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	33.69	34.09	3900.00	*	37.37	39000.00	*
2	33.69	34.12	3900.00	*	37.37	39000.00	*
3	33.72	34.13	3900.00	*	37.37	39000.00	*
4	33.69	34.12	3900.00	*	37.34	39000.00	*
5	33.71	34.10	3900.00	*	37.38	39000.00	*
6	33.72	34.13	3900.00	*	37.35	39000.00	*
7	33.72	34.09	3900.00	*	37.38	39000.00	*
8	33.72	34.12	3900.00	*	37.38	39000.00	*
9	33.72	34.09	3900.00	*	37.37	39000.00	*
10	33.72	13.13	3900.00	*	37.37	39000.00	*
Average:	33.7100	32.0120	3900.0000		37.3680	39000.0000	
Std Dev'n	0.0134	6.2940	0.0000		0.0125	0.0000	
Ratio		0.95	115.69	*	1.11	1156.93	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		34.28	3900.00	*	37.25	39000.00	*
2		34.29	3900.00	*	37.25	39000.00	*
3		34.28	3900.00	*	37.34	39000.00	*
4		34.25	3900.00	*	37.38	39000.00	*
5		34.25	3900.00	*	37.38	39000.00	*
6		34.25	3900.00	*	37.38	39000.00	*
7		34.25	3900.00	*	37.35	39000.00	*
8		34.25	3900.00	*	37.38	39000.00	*
9		34.28	3900.00	*	37.37	39000.00	*
10		34.28	3900.00	*	37.37	39000.00	*
Average:		34.2644	3900.0000		37.3450	39000.0000	
Std Dev'n		0.0162	0.0000		0.0492	0.0000	
Ratio		1.02	115.69	*	1.11	1156.93	*

	circleq 50	circleq 150	circleq 200	circleq 250	circleq 300
1	16.85	50.56	67.56	84.32	101.13
2	16.84	50.56	70.07	84.28	101.13
3	16.84	50.57	70.25	84.29	101.16
4	16.84	50.56	69.10	84.28	101.13
5	16.85	50.56	69.10	84.28	101.16
6	16.88	50.59	69.09	84.28	101.16
7	16.87	50.59	69.09	84.31	101.16
8	16.87	50.57	69.09	84.28	101.15
9	16.88	50.56	69.13	84.28	101.31
10	16.88	50.56	69.10	84.28	101.22
Average:	16.86	50.57	69.16	84.29	101.17
Std Dev'n	0.02	0.01	0.68	0.01	0.05
Ratio	0.50	1.50	2.05	2.50	3.00

B.2.3. Synchronisation

In this section the programs used are as follows.

`hiho` The unsynchronised HiHo program

`hihosr` The synchronised HiHo program, using RAM semaphores (and threads).

`hihoss` The synchronised HiHo program, using system semaphores and threads.

`hihosp` The synchronised HiHo program, using system semaphores and processes.

	hiho 1000	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	4.81	4.81	9.03	*	4.66	49.59	*
2	4.81	4.81	10.72	*	4.66	59.56	*
3	4.81	4.81	10.50	*	4.66	67.09	*
4	4.81	4.82	11.25	*	4.65	49.59	*
5	4.82	4.84	10.00	*	4.65	49.57	*
6	4.81	4.81	10.28	*	4.65	54.60	*
7	4.81	4.81	10.28	*	4.65	54.56	*
8	4.84	4.84	11.50	*	4.65	59.56	*
9	4.81	4.82	11.00	*	4.66	59.56	*
10	4.82	4.81	11.03	*	4.69	59.56	*
Average:	4.8150	4.8180	10.5590		4.6580	56.3240	
Std Dev'n	0.0092	0.0117	0.6793		0.0117	5.4789	
Ratio		1.00	2.19	*	0.97	11.70	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		4.75	10.34	*	4.75	52.19	*
2		4.75	9.94	*	4.75	59.69	*
3		4.75	9.44	*	4.75	57.22	*
4		4.75	9.22	*	4.75	52.19	*
5		4.75	9.68	*	4.75	57.18	*
6		4.75	9.72	*	4.75	59.66	*
7		4.75	9.69	*	4.75	54.72	*
8		4.75	9.47	*	4.75	62.18	*
9		4.75	9.94	*	4.79	64.69	*
10		4.75	9.94	*	4.75	47.19	*
Average:		4.7500	9.7380		4.7540	56.6910	
Std Dev'n		0.0000	0.3033		0.0120	4.9710	
Ratio		0.99	2.02	*	0.99	11.77	*

	hihosr 1000	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	6.07	6.06	250.53	*	6.09	2505.03	*
2	6.03	6.06	250.53	*	6.13	2505.03	*
3	6.06	6.06	250.53	*	6.12	2505.03	*
4	6.06	6.07	250.53	*	6.10	2505.03	*
5	6.07	6.09	250.53	*	6.12	2505.04	*
6	6.06	6.06	250.53	*	6.12	2505.03	*
7	6.06	6.09	250.53	*	6.10	2505.06	*
8	6.06	6.09	250.53	*	6.12	2505.03	*
9	6.03	6.10	250.53	*	6.13	2505.03	*
10	6.06	6.06	250.53	*	6.09	2505.03	*
Average:	6.0560	6.07	250.53		6.11	2505.03	
Std Dev'n	0.0136	0.02	0.00		0.01	0.01	
Ratio		1.00	41.37	*	1.01	413.64	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		6.09	250.53	*	6.25	2505.03	*
2		6.06	250.53	*	6.25	2505.03	*
3		6.06	250.53	*	6.22	2505.03	*
4		6.06	250.53	*	6.25	2505.03	*
5		6.06	250.53	*	6.25	2505.03	*
6		6.07	250.53	*	6.25	2505.03	*
7		6.06	250.53	*	6.25	2505.03	*
8		6.06	250.53	*	6.25	2505.03	*
9		6.06	250.53	*	6.25	2505.03	*
10		6.09	250.53	*	6.25	2505.03	*
Average:		6.06	250.53		6.25	2505.03	
Std Dev'n		0.01	0.00		0.01	0.00	

Ratio		1.00	41.37	*	1.03	413.64	*

	hihoss 1000	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	6.37	6.38	250.53	*	6.41	2505.03	*
2	6.34	6.37	250.53	*	6.44	2505.03	*
3	6.35	6.38	250.53	*	6.40	2505.03	*
4	6.34	6.37	250.53	*	6.44	2505.03	*
5	6.35	6.37	250.56	*	6.41	2505.03	*
6	6.34	6.38	250.57	*	6.44	2505.03	*
7	6.35	6.37	250.53	*	6.40	2505.04	*
8	6.34	6.38	250.53	*	6.44	2505.03	*
9	6.35	6.37	250.56	*	6.41	2505.03	*
10	6.34	6.38	250.53	*	6.44	2505.04	*
Average:	6.3470	6.38	250.54		6.42	2505.03	
Std Dev'n	0.0090	0.01	0.02		0.02	0.00	
Ratio		1.00	39.47	*	1.01	394.68	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		6.35	250.53	*	6.50	2505.03	*
2		6.34	250.53	*	6.56	2505.03	*
3		6.38	250.56	*	6.53	2505.03	*
4		6.34	250.53	*	6.53	2505.03	*
5		6.34	250.53	*	6.54	2505.04	*
6		6.35	250.56	*	6.53	2505.03	*
7		6.37	250.53	*	6.53	2505.03	*
8		6.35	250.53	*	6.56	2505.03	*
9		6.37	250.53	*	6.53	2505.03	*
10		6.38	250.53	*	6.53	2505.03	*
Average:		6.36	250.54		6.53	2505.03	
Std Dev'n		0.02	0.01		0.02	0.00	
Ratio		1.00	39.47	*	1.03	394.68	*

	hihosp 1000	p 1 10	p 1 15	p 1 20	p 10 10	p 10 15	p 10 20
1	6.31	6.40	250.69	*	6.47	2507.37	*
2	6.28	6.40	250.66	*	6.47	2507.40	*
3	6.28	6.41	250.65	*	6.47	2507.41	*
4	6.28	6.41	250.66	*	6.47	2507.37	*
5	6.32	6.41	250.65	*	6.47	2511.84	*
6	6.28	6.37	250.66	*	6.47	2507.35	*
7	6.29	6.40	250.65	*	6.47	2507.37	*
8	6.28	6.37	250.66	*	6.47	2507.41	*
9	6.29	6.41	250.68	*	6.46	2507.40	*
10	6.28	6.38	250.66	*	6.50	2507.34	*
Average:	6.2890	6.40	250.66		6.47	2507.83	
Std Dev'n	0.0137	0.02	0.01		0.01	1.34	
Ratio		1.02	39.86	*	1.03	398.76	*

		t 1 10	t 1 15	t 1 20	t 10 10	t 10 15	t 10 20
1		6.43	250.69	*	6.62	2507.40	*
2		6.43	250.65	*	6.66	2507.37	*
3		6.47	250.66	*	6.66	2507.43	*
4		6.47	250.68	*	6.66	2507.41	*
5		6.44	250.69	*	6.66	2507.43	*
6		6.44	250.69	*	6.66	2507.44	*
7		6.44	250.68	*	6.65	2507.44	*
8		6.47	250.66	*	6.68	2507.41	*
9		6.44	250.65	*	6.65	2507.41	*
10		6.44	250.66	*	6.65	2511.85	*
Average:		6.45	250.67		6.66	2507.86	
Std Dev'n		0.02	0.02		0.01	1.33	
Ratio		1.03	39.86	*	1.06	398.77	*

B.2.4. Exception Handling

In this section the programs used are as follows.

`timer` The Sieve of Eratosthenes program, used as a basic timing program. It is also run by each of the other programs in this section and is the source of the results in the other columns.

`int1` The interrupt driven program, using the `DosSleep()` system call to suspend itself for a given number of milliseconds.

`int2` The interrupt driven program, using the repeated interval timer service (`DosTimerStart()`) to signal a semaphore at intervals of a given number of milliseconds.

`int3` The interrupt driven program, using the one-off timer service (`DosTimerAsync()`) to signal a semaphore at an interval of a given number of milliseconds. The program then restarted the timer.

`intt` The interrupt driven program, using a child thread which uses the `DosSleep()` system call to suspend itself for a given number of milliseconds and then sends a signal to the parent thread. The parent thread has a signal handler set up which does nothing, except return immediately.

`intp` The interrupt driven program, using a child process which uses the `DosSleep()` system call to suspend itself for a given number of milliseconds and then sends a signal to the parent process. The parent process has a signal handler set up which does nothing, except return immediately.

	timer	int1 1	int2 1	int3 1	intt 1	intp 1
1	2.19	2.25	2.28	2.28	2.22	2.19
2	2.22	2.28	2.25	2.32	2.19	2.22
3	2.19	2.25	2.25	2.32	2.22	2.22
4	2.19	2.25	2.28	2.28	2.22	2.19
5	2.22	2.25	2.28	2.32	2.22	2.22
6	2.18	2.28	2.25	2.28	2.22	2.22
7	2.21	2.25	2.25	2.32	2.18	2.19
8	2.18	2.25	2.29	2.32	2.21	2.22
9	2.19	2.28	2.25	2.28	2.21	2.22
10	2.22	2.28	2.28	2.31	2.18	2.22
Average:	2.1990	2.26	2.27	2.30	2.21	2.21
Std Dev'n	0.0158	0.01	0.02	0.02	0.02	0.01
Ratio		1.03	1.03	1.05	1.00	1.01

	int1 20	int2 20	int3 20	intt 20	intp 20
1	2.25	2.25	2.28	2.22	2.21
2	2.25	2.25	2.28	2.22	2.22
3	2.25	2.28	2.32	2.22	2.22
4	2.28	2.25	2.31	2.21	2.22
5	2.25	2.28	2.31	2.22	2.22
6	2.25	2.25	2.31	2.22	2.22
7	2.25	2.25	2.32	2.22	2.19
8	2.25	2.29	2.31	2.19	2.22
9	2.25	2.25	2.31	2.22	2.19
10	2.29	2.28	2.32	2.22	2.22
Average:	2.26	2.26	2.31	2.22	2.21
Std Dev'n	0.01	0.02	0.01	0.01	0.01
Ratio	1.03	1.03	1.05	1.01	1.01

	int1 40	int2 40	int3 40	intt 40	intp 40
1	2.22	2.22	2.25	2.19	2.22
2	2.21	2.21	2.25	2.19	2.22
3	2.21	2.21	2.25	2.19	2.22
4	2.22	2.22	2.25	2.22	2.22
5	2.22	2.25	2.25	2.22	2.21
6	2.22	2.22	2.25	2.22	2.22
7	2.22	2.22	2.25	2.21	2.22
8	2.22	2.22	2.25	2.22	2.19
9	2.22	2.22	2.25	2.19	2.22
10	2.22	2.22	2.22	2.22	2.19
Average:	2.2180	2.2210	2.2470	2.2070	2.2130
Std Dev'n	0.0040	0.0104	0.0090	0.0142	0.0119
Ratio	1.01	1.01	1.02	1.00	1.01

	int1 60	int2 60	int3 60	intt 60	intp 60
1	2.22	2.22	2.25	2.22	2.21
2	2.22	2.22	2.25	2.19	2.22
3	2.22	2.25	2.25	2.18	2.22
4	2.22	2.21	2.25	2.22	2.22
5	2.22	2.21	2.25	2.22	2.22
6	2.22	2.21	2.25	2.19	2.22
7	2.21	2.21	2.25	2.19	2.22
8	2.21	2.21	2.25	2.22	2.22
9	2.21	2.21	2.25	2.19	2.18
10	2.11	2.21	2.21	2.22	2.21
Average:	2.2060	2.2160	2.2460	2.2040	2.2140
Std Dev'n	0.0323	0.0120	0.0120	0.0162	0.0120
Ratio	1.00	1.01	1.02	1.00	1.01

B.2.5. Special Tests Performed Using OS/2

Two series of special tests were performed using OS/2. The first series repeated some of the benchmarks with the print spooler disabled, in order to assess the effect that the spooler had on the execution time of programs. The second series was performed using the dynamic scheduling algorithm, rather than the absolute scheduling algorithm used for all the other tests (the print spooler was enabled during this second series of tests). In the following tables of results there is a row titled "Normal". This is the result that was obtained for the program during the main series of tests detailed above. The ratios given are those of the special test to the normal value. In each case the column headings refer to the same programs as in the main series of test detailed above.

B.2.5.1. Results With the Print Spooler Disabled

	sieve	psieve 10	ssieve 10	tsieve 10	procreate 100
1	2.66	28.56	28.65	36.10	30.78
2	2.66	28.10	28.47	36.12	30.72
3	2.62	28.28	28.44	36.09	30.71
4	2.65	28.10	28.47	36.10	30.75
5	2.66	28.09	28.43	36.09	30.78
6	2.63	28.09	28.44	36.09	30.15
7	2.66	28.25	28.47	36.09	30.69
8	2.65	28.09	28.47	36.10	30.68
9	2.65	28.13	28.43	36.09	30.69

10	2.66	28.12	28.47	36.09	30.69
Average:	2.65	28.18	28.47	36.10	30.66
Std Dev:	0.01	0.14	0.06	0.01	0.17
Normal:	2.19	27.57	28.64	28.51	31.52
Ratio:	1.21	1.02	0.99	1.27	0.97

	pipe100 1000	shm100 1000	circleq 100	circlep 100
1	2.28	4.34	34.84	17.47
2	2.31	4.32	34.81	17.46
3	2.28	4.31	34.85	17.50
4	2.31	4.34	34.84	17.47
5	2.25	4.31	34.85	17.50
6	2.25	4.35	34.84	17.47
7	2.29	4.34	34.84	17.46
8	2.31	4.34	34.82	17.47
9	2.32	4.31	34.84	17.47
10	2.28	4.32	34.84	17.50
Average:	2.29	4.33	34.84	17.48
Std Dev'n	0.02	0.01	0.01	0.02
Normal:	2.23	3.86	33.71	16.86
Ratio:	1.03	1.12	1.03	1.04

	hiho 1000	hihosr 1000	hihosp 1000	hihoss 1000
1	5.15	6.50	6.75	6.82
2	5.15	6.50	6.75	6.81
3	5.16	6.50	6.72	6.81
4	5.16	6.50	6.72	6.82
5	5.16	6.50	6.72	6.81
6	5.16	6.50	6.72	6.81
7	5.16	6.50	6.72	6.81
8	5.16	6.53	6.72	6.81
9	5.16	6.53	6.72	6.82
10	5.19	6.53	6.72	6.81
Average:	5.16	6.51	6.73	6.81
Std Dev:	0.01	0.01	0.01	0.00
Normal:	4.82	6.06	6.29	6.35
Ratio:	1.07	1.07	1.07	1.07

B.2.5.2. Results Using the Dynamic Scheduling Algorithm

	sieve	psieve 10	ssieve 10	tsieve 10	procreate 100
1	2.19	27.63	28.81	31.16	29.81
2	2.16	27.59	28.66	31.12	30.00
3	2.19	27.60	28.60	31.15	31.44
4	2.19	27.59	28.59	31.12	30.60
5	2.19	27.56	28.63	31.13	24.47
6	2.19	27.60	28.63	31.16	30.25
7	2.16	27.56	28.59	31.12	31.16
8	2.19	27.59	28.85	31.12	31.16
9	2.19	27.60	28.65	31.13	31.34
10	2.19	27.59	28.69	31.13	26.09
Average:	2.18	27.59	28.67	31.13	29.63
Std Dev:	0.01	0.02	0.09	0.02	2.27
Normal:	2.19	27.57	28.64	28.51	31.52
Ratio:	1.00	1.00	1.00	1.09	0.94

	pipe100 1000	shm100 1000	circleq 100	circlep 100
1	2.37	4.25	35.63	18.06
2	2.37	4.22	35.69	18.06
3	2.43	4.22	35.63	18.07
4	2.40	4.22	35.59	18.06

5	2.40	4.21	35.53	18.06
6	2.37	4.22	35.60	18.06
7	2.37	4.22	35.60	18.06
8	2.41	4.22	35.56	18.06
9	2.41	4.22	35.62	18.06
10	2.41	4.21	35.65	18.07
Average:	2.39	4.22	35.61	18.06
Std Dev:	0.02	0.01	0.04	0.00
Normal:	2.23	3.86	33.71	16.86
Ratio:	1.07	1.09	1.06	1.07

	hiho 1000	hihosr 1000	hihosp 1000	hihoss 1000
1	4.84	6.09	6.60	6.53
2	4.93	6.10	6.47	6.53
3	4.81	6.09	6.50	6.54
4	4.84	6.13	6.47	6.53
5	4.75	6.09	6.50	6.56
6	4.75	6.22	6.50	6.56
7	4.75	6.22	6.47	6.56
8	4.75	6.22	6.50	6.54
9	4.75	6.25	6.47	6.53
10	4.75	6.25	6.50	6.53
Average:	4.79	6.17	6.50	6.54
Std Dev:	0.06	0.07	0.04	0.01
Normal:	4.82	6.06	6.29	6.35
Ratio:	1.00	1.02	1.03	1.03

B.3. QNX

The background processes in this case are indicated by column headings of the form `+b <number> <priority>`, where `<number>` represents the number of background processes used (either one or ten), and `<priority>` represents the absolute priority for the background processes. The benchmark programs were run with a priority value of eight. Lower numeric values represent higher scheduling priority.

B.3.1. Concurrency

In this section the programs used are as follows.

sieve The sequential form of the Sieve of Eratosthenes.

nsieve The concurrent form of the Sieve of Eratosthenes, using the `fork()` system call.

csieve The concurrent form of the Sieve of Eratosthenes, using the `create()` system call.

procreate The creation of child processes from disk, using the `create()` system call. The child processes were empty programs, which terminated immediately.

fcreate The creation of child processes from memory, using the `fork()` system call. The child processes were empty functions, which terminated immediately.

	sieve	nsieve 1	csieve 1	nsieve 2	csieve 2	nsieve 10	csieve 10
1	48	47	49	94	97	471	485
2	48	47	49	94	97	471	484
3	47	47	49	94	97	471	484

4	48	47	49	94	97	472	485
5	47	47	49	94	98	471	484
6	47	47	49	94	97	472	484
7	48	47	49	94	97	471	485
8	48	47	49	95	98	471	484
9	47	47	49	94	98	471	484
10	48	47	49	94	97	471	484
Average:	47.6	47	49	94.1	97.3	471.2	484.3
Std Dev'n	0.50	0.00	0.00	0.31	0.47	0.42	0.42
Ticks/sieve	47.6	47	49	47.05	48.65	47.12	48.43
Ave(s):	2.38	2.35	2.45	4.71	4.87	23.56	24.22
s/sieve:	2.38	2.35	2.45	2.35	2.43	2.36	2.42

	procreate 100	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	67	201	*	68	2008	*	68
2	68	201	*	68	2015	*	69
3	67	201	*	69	2016	*	69
4	68	228	*	69	2009	*	68
5	68	218	*	69	2010	*	68
6	68	201	*	69	2017	*	69
7	67	201	*	69	2008	*	69
8	68	201	*	69	2008	*	69
9	68	201	*	69	2008	*	69
10	68	257	*	69	2014	*	69
Average:	67.7	211		68.8	2011.3		68.7
Std Dev'n	0.42	18.39		0.31	3.56		0.42
Ratio:		3.12	*	1.02	29.71	*	1.01
Ticks/child:	0.677						
Ave(s):	3.39	10.55	0.00	3.44	100.57	0.00	3.44
s/child:	0.03						

	procreate 50	procreate 150	procreate 200	procreate 250	procreate 300
1	33	100	134	167	201
2	33	101	134	167	201
3	33	100	134	168	201
4	33	100	134	168	201
5	33	101	134	168	201
6	33	101	134	167	201
7	33	100	134	167	200
8	33	100	134	167	201
9	33	100	133	167	201
10	34	100	134	167	201
Average:	33.1	100.3	133.9	167.3	200.9
Std Dev'n	0.30	0.46	0.30	0.46	0.30
Ratio:	0.49	1.48	1.98	2.47	2.97
Ticks/child:	0.662	0.669	0.670	0.669	0.670
Ave(s):	1.66	5.02	6.70	8.37	10.05
s/child:	0.03	0.03	0.03	0.03	0.03

	fcreate 100	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	24	200	*	24	2000	*	24
2	24	200	*	24	2000	*	24
3	25	200	*	24	2000	*	24
4	24	200	*	24	2000	*	24
5	24	200	*	24	2000	*	24
6	25	200	*	24	2000	*	24
7	24	201	*	24	2000	*	24
8	24	201	*	24	2000	*	24
9	25	200	*	24	2000	*	24
10	24	200	*	24	2000	*	24
Average:	24.3	200.2		24	2000		24
Std Dev'n	0.47	0.42		0.00	0.00		0.00
Ratio:		8.24	*	0.99	82.30	*	0.99
Ticks/child:	0.243						

Ave(s):	1.22	10.01	0.00	1.20	100.00	0.00	1.20
s/child:	0.01						

	fcreate 50	fcreate 150	fcreate 200	fcreate 250	fcreate 300
1	12	37	50	62	75
2	12	37	49	62	74
3	12	37	50	62	75
4	12	37	50	62	75
5	12	37	49	62	74
6	12	37	50	62	75
7	12	37	49	62	74
8	12	37	49	62	74
9	12	37	50	62	75
10	12	37	49	62	74
Average:	12	37	49.5	62	74.5
Std Dev'n	0.00	0.00	0.50	0.00	0.50
Ratio:	0.49	1.52	2.04	2.55	3.07
Ticks/child:	0.240	0.247	0.248	0.248	0.248
Ave(s):	0.60	1.85	2.48	3.10	3.73
s/child:	0.01	0.01	0.01	0.01	0.01

B.3.2. Interprocess Communication

In this section the programs used are as follows.

`mess100` The 100-byte message passing program, using messages.

`circle1` The one-byte, circular message passing program, using messages.

	mess100 1	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	0	8	*	0	80	*	0
2	1	8	*	0	80	*	0
3	0	8	*	0	80	*	0
4	0	8	*	0	80	*	0
5	0	8	*	0	80	*	0
6	0	8	*	0	80	*	0
7	0	8	*	0	80	*	0
8	0	8	*	0	80	*	0
9	0	8	*	0	80	*	0
10	0	8	*	0	80	*	0
Average:	0.1	8		0	80		0
Std Dev'n	0.31	0.00		0.00	0.00		0.00
Ratio:		80	*	0	800	*	0
Ave(s):	0.01	0.40	0.00	0.00	4.00	0.00	0.00

	mess100 10	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	0	44	*	0	440	*	0
2	0	44	*	0	440	*	0
3	0	44	*	0	440	*	0
4	1	44	*	0	440	*	0
5	0	44	*	0	440	*	0
6	0	44	*	0	440	*	0
7	1	44	*	0	440	*	0
8	0	44	*	0	440	*	0
9	0	44	*	1	440	*	0
10	1	44	*	0	440	*	1
Average:	0.3	44		0.1	440		0.1
Std Dev'n	0.47	0.00		0.31	0.00		0.31
Ratio:		146.67	*	0.33	1466.67	*	0.33

Ave(s):	0.02	2.20	0.00	0.01	22.00	0.00	0.01

	mess100 100	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	2	404	*	2	4040	*	2
2	2	404	*	2	4040	*	1
3	2	404	*	2	4040	*	2
4	2	404	*	1	4040	*	2
5	1	404	*	2	4040	*	2
6	1	404	*	1	4040	*	1
7	1	404	*	2	4040	*	2
8	2	404	*	1	4040	*	1
9	1	404	*	1	4041	*	2
10	1	404	*	2	4040	*	1
Average:	1.5	404		1.6	4040.1		1.6
Std Dev'n	0.50	0.00		0.49	0.30		0.49
Ratio:		269.33	*	1.07	2693.40	*	1.07
Ave(s):	0.08	20.20	0.00	0.08	202.01	0.00	0.08

	mess100 1000	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	15	4004	*	15	40040	*	14
2	14	4004	*	15	40040	*	15
3	15	4004	*	15	40041	*	15
4	15	4004	*	15	40041	*	15
5	14	4004	*	15	40040	*	14
6	15	4004	*	15	40041	*	14
7	14	4004	*	14	40041	*	14
8	15	4004	*	15	40040	*	15
9	15	4004	*	15	40041	*	15
10	15	4004	*	14	40041	*	15
Average:	14.7	4004		14.8	40040.6		14.6
Std Dev'n	0.46	0.00		0.40	0.49		0.49
Ratio:		272.38	*	1.01	2723.85	*	0.99
Ave(s):	0.74	200.20	0.00	0.74	2002.03	0.00	0.73

	mess100 10000	mess100 20000	mess100 30000	mess100 40000
1	147	294	440	588
2	147	293	440	588
3	147	294	441	587
4	147	294	440	587
5	147	293	441	587
6	147	294	440	588
7	147	293	441	588
8	146	294	441	587
9	147	294	440	587
10	147	293	441	588
Average:	146.9	293.6	440.5	587.5
Std Dev'n	0.30	0.49	0.50	0.50
Ratio:		2.00	3.00	4.00
Ave(s):	7.35	14.68	22.03	29.38

	circle1 100	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	106	15613	*	106	156121	*	106
2	106	15614	*	106	156121	*	106
3	106	15613	*	106	156120	*	106
4	106	15613	*	106	156121	*	106
5	106	15613	*	105	156121	*	106
6	106	15614	*	106	156120	*	106
7	106	15613	*	106	156121	*	106
8	106	15613	*	106	156121	*	105
9	106	15614	*	106	156120	*	105
10	106	15613	*	106	156121	*	106
Average:	106	15613.3		105.9	156120.7		105.8
Std Dev'n	0.00	0.46		0.30	0.46		0.40

Ratio:		147.30	*	1.0	1472.8	*	1.00
Ave(s):	5.30	780.67	0.00	5.30	7806.04	0.00	5.29

	circle1 50	circle1 150	circle1 200	circle1 250	circle1 300
1	54	158	210	262	315
2	54	158	210	262	315
3	54	158	210	262	314
4	54	158	210	262	314
5	54	158	210	262	314
6	54	158	210	262	315
7	54	158	210	262	314
8	53	159	210	263	315
9	54	158	210	262	314
10	54	158	210	262	314
Average:	53.9	158.1	210	262.1	314.4
Std Dev'n	0.30	0.30	0.00	0.30	0.49
Ratio:	0.51	1.49	1.98	2.47	2.97
Ave(s):	2.70	7.91	10.50	13.11	15.72

	circle1 10	+b 1 8	+b 10 8
1	12	1573	15720
2	12	1573	15720
3	12	1573	15720
4	12	1573	15720
5	12	1573	15720
6	12	1573	15721
7	12	1573	15721
8	12	1573	15720
9	12	1573	15720
10	12	1573	15721
Average:	12	1573	15720.3
Std Dev'n	0.00	0.00	0.46
Ratio:		131.08	1310.03
Ave(s):	0.60	78.65	786.02

B.3.3. Synchronisation

In this section the programs used are as follows.

hiho The unsynchronised HiHo program.

hihos The synchronised HiHo program, using ports.

	hiho 1000	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	77	2010	*	78	20100	*	78
2	78	2010	*	78	20101	*	78
3	78	2010	*	78	20100	*	78
4	77	2010	*	78	20101	*	78
5	78	2010	*	78	20100	*	78
6	78	2010	*	78	20100	*	78
7	78	2010	*	78	20101	*	78
8	78	2010	*	78	20100	*	78
9	78	2010	*	78	20100	*	78
10	78	2010	*	78	20100	*	78
Average:	77.8	2010		78	20100.3		78
Std Dev'n	0.40	0.00		0.00	0.46		0.00
Ratio:		25.84	*	1.00	258.36	*	1.00
Ave(s):	3.89	100.50	0.00	3.90	1005.02	0.00	3.90

	hiho 100	+b 1 8	+b 10 8

1	8	210	2100
2	8	210	2100
3	8	210	2100
4	8	210	2100
5	8	210	2100
6	8	210	2100
7	8	210	2100
8	8	210	2100
9	9	210	2100
10	8	210	2100
Average:	8.1	210	2100
Std Dev'n	0.30	0.00	0.00
Ratio:		25.93	259.26
Ave(s):	0.41	10.50	105.00

	hihos 1000	+b 1 8	+b 1 7	+b 1 9	+b 10 8	+b 10 7	+b 10 9
1	98	8012	*	98	80120	*	98
2	97	8012	*	97	80121	*	98
3	98	8012	*	98	80121	*	98
4	98	8013	*	98	80120	*	98
5	98	8012	*	98	80121	*	98
6	98	8012	*	98	80120	*	98
7	98	8012	*	98	80121	*	98
8	98	8012	*	98	80120	*	97
9	98	8012	*	98	80120	*	98
10	97	8012	*	97	80120	*	98
Average:	97.8	8012.1		97.8	80120.4		97.9
Std Dev'n	0.40	0.30		0.40	0.49		0.30
Ratio:		81.92	*	1.00	819.23	*	1.00
Ave(s):	4.89	400.61	0.00	4.89	4006.02	0.00	4.90

	hihos 100	+b 1 8	+b 10 8
1	10	813	8120
2	10	812	8120
3	11	812	8120
4	10	812	8120
5	11	812	8120
6	10	812	8120
7	10	812	8120
8	10	812	8120
9	11	812	8120
10	10	812	8120
Average:	10.3	812.1	8120
Std Dev'n	0.46	0.30	0.00
Ratio:		78.84	788.35
Ave(s):	0.52	40.61	406.00

B.3.4. Exception Handling

In this section the programs used are as follows.

`timer` The Sieve of Eratosthenes program, used as a basic timing program. It is also run by each of the other programs in this section, and is the source of the results in the other columns.

`interrupt` The interrupt driven program, using the `set_timer()` system call to produce a signal every system clock tick.

`int2` The interrupt driven program, using the `set_timer()` system call to generate an exception

every system clock tick.

	timer	interrupt	int2
1	48	49	50
2	48	50	49
3	47	50	49
4	48	49	50
5	48	49	50
6	48	50	50
7	47	49	49
8	48	49	49
9	48	50	49
10	48	50	50
Average:	47.8	49.5	49.5
Std Dev'n	0.42	0.50	0.50
Ratio:		1.04	1.04
Ave(s):	2.39	2.48	2.48

B.4. FlexOS

The background processes in this case are indicated by column headings of the form +b <number> <priority>, where <number> represents the number of background processes used (either one or ten), and <priority> represents the absolute priority for the background processes. The benchmark programs were run with a priority value of 200. Lower numeric values represent higher scheduling priority.

B.4.1. Concurrency

In this section the programs used are as follows.

sieve The sequential form of the Sieve of Eratosthenes.

csieve The concurrent form of the Sieve of Eratosthenes, using the `e_command()` system call to create multiple processes from disk.

procreate The creation of child processes from disk, using the `e_command()` system call. The child processes were empty programs, which terminated immediately.

	sieve	csieve 1	csieve 2	csieve 10
1	1.969	2.625	4.938	23.812
2	1.969	2.500	4.906	23.781
3	1.969	2.531	4.937	23.781
4	1.969	2.563	4.906	23.781
5	1.969	2.500	4.937	23.781
6	1.969	2.563	4.906	23.781
7	1.969	2.500	4.938	23.781
8	1.969	2.563	4.907	23.782
9	1.969	2.500	4.938	23.781
10	1.969	2.563	4.906	23.906
Average:	1.9690	2.5408	4.9219	23.7967
Std Dev'n	0.0000	0.0297	0.0156	0.0392
s/sieve	1.969	2.541	2.461	2.380

	procreate 100	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	29.532	28.344	*	29.000	*	*	1.719
2	29.750	30.687	*	30.593	*	*	2.688
3	26.968	29.656	*	26.531	*	*	2.719
4	26.719	41.406	*	28.906	*	*	2.593
5	28.844	40.125	*	30.000	*	*	2.656
6	30.094	40.907	*	30.094	*	*	16.125
7	29.500	41.688	*	27.063	*	*	3.969
8	30.031	41.750	*	31.000	*	*	PANIC
9	29.375	39.219	*	30.844	*	*	*
10	28.188	41.750	*	29.938	*	*	*
Average:	28.9001	37.5532		29.3969			4.6384
Std Dev'n	1.2004	4.5691		1.5319			4.9425
Ratio:		1.30	*	1.02	*	*	0.16
s/child:	0.289						

	procreate 20	procreate 40	procreate 60	procreate 80
1	4.156	10.094	15.593	21.250
2	4.125	9.469	14.781	21.844
3	3.687	9.125	17.032	21.407
4	3.125	9.782	15.282	22.344
5	4.344	8.656	17.125	20.656
6	4.062	10.687	17.093	19.562
7	3.687	8.938	15.969	22.968
8	3.437	9.875	14.906	20.531
9	3.812	9.687	16.562	22.687
10	3.375	10.063	15.875	20.500
Average:	3.7810	9.6376	16.0218	21.3749
Std Dev'n	0.3724	0.5757	0.8495	1.0352
Ratio:	0.13	0.33	0.55	0.74
s/child:	0.189	0.482	0.801	1.069

B.4.2. Interprocess Communication

In this section the programs used are as follows.

`pipe100` The 100-byte message passing program, using a pipe.

`shm100` The 100-byte message passing program, using a shared memory segment.

`circlep` The one-byte, circular message passing program, using pipes.

	pipe100 1	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	0.094	0.000	*	0.000	0.000	*	0.000
2	0.000	0.000	*	0.000	*	*	0.000
3	0.000	0.031	*	0.000	*	*	0.000
4	0.000	0.000	*	0.000	*	*	0.000
5	0.000	0.000	*	0.000	*	*	0.031
6	0.000	0.000	*	0.000	*	*	0.000
7	0.000	0.000	*	0.000	*	*	0.000
8	0.000	0.000	*	0.000	*	*	0.000
9	0.000	0.000	*	0.000	*	*	0.000
10	0.000	0.000	*	0.000	*	*	0.000
Average:	0.009	0.003		0.000	0.000		0.003
Std Dev'n	0.03	0.01		0.00	0.00		0.01
Ratio:		0.33	*	0.00	*	*	0.33

	pipe100 10	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	0.032	0.031	*	0.031	0.000	*	*
2	0.000	0.000	*	0.000	*	*	*
3	0.000	0.000	*	0.000	*	*	*

4	0.000	0.000	*	0.000	*	*	*
5	0.032	0.000	*	0.000	*	*	*
6	0.000	0.000	*	0.000	*	*	*
7	0.000	0.000	*	0.000	*	*	*
8	0.000	0.000	*	0.000	*	*	*
9	0.031	0.000	*	0.000	*	*	*
10	0.000	0.000	*	0.000	*	*	*
Average:	0.010	0.003		0.003	0.000		
Std Dev'n	0.01	0.01		0.01	0.00		
Ratio:		0.33	*	0.33	*	*	*

	pipe100 100	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	0.125	0.156	*	0.157	0.094	*	0.063
2	0.094	0.062	*	0.063	*	*	0.094
3	0.062	0.062	*	0.094	*	*	0.093
4	0.062	0.094	*	0.062	*	*	0.062
5	0.125	0.125	*	0.063	*	*	0.062
6	0.062	0.094	*	0.094	*	*	0.063
7	0.062	0.094	*	0.062	*	*	0.125
8	0.062	0.094	*	0.063	*	*	0.062
9	0.125	0.125	*	0.094	*	*	0.063
10	0.063	0.093	*	0.062	*	*	0.125
Average:	0.084	0.100		0.081	0.094		0.081
Std Dev'n	0.03	0.03		0.03			0.02
Ratio:		1.19	*	0.97	*	*	0.96

	pipe100 1000	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	3.656	3.719	*	3.687	4.500	*	6.094
2	3.469	3.469	*	3.500	*	*	3.687
3	3.469	29.532	*	3.500	*	*	3.500
4	3.468	29.469	*	3.500	*	*	3.500
5	3.469	29.469	*	3.469	*	*	3.500
6	3.469	29.500	*	3.469	*	*	3.500
7	3.438	29.469	*	3.500	*	*	3.469
8	3.469	29.469	*	3.500	*	*	3.468
9	3.468	29.531	*	3.500	*	*	3.469
10	3.469	29.469	*	3.500	*	*	3.469
Average:	3.4844	24.3096		3.5125	4.5000		3.7656
Std Dev'n	0.0579	10.3580		0.0594			0.7786
Ratio:		6.98	*	1.01	*	*	1.08

	pipe100 10000	pipe100 20000	pipe100 30000	pipe100 40000
1	24.125	46.875	69.594	92.344
2	23.937	46.657	69.406	92.157
3	23.906	46.656	69.406	92.157
4	23.907	46.687	69.406	92.157
5	23.937	46.565	69.407	92.156
6	23.937	46.687	69.438	92.156
7	23.938	46.656	69.406	92.156
8	23.906	46.657	69.437	92.156
9	23.937	46.688	69.406	92.156
10	23.938	46.656	69.406	92.156
Average:	23.947	46.678	69.431	92.175
Std Dev'n	0.06	0.07	0.06	0.06
Ratio:		1.95	2.90	3.85

	shm100 1	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	0.000	0.000	*	0.000	0.000	*	0.000
2	0.000	0.000	*	0.000	*	*	0.000
3	0.000	0.000	*	0.000	*	*	0.000
4	0.000	0.000	*	0.000	*	*	0.000
5	0.000	0.000	*	0.000	*	*	0.000
6	0.000	0.000	*	0.000	*	*	0.000

7	0.000	0.000	*	0.000	*	*	0.000
8	0.000	0.000	*	0.000	*	*	0.000
9	0.000	0.000	*	0.000	*	*	0.000
10	0.031	0.000	*	0.000	*	*	0.000
Average:	0.003	0.000		0.000	0.000		0.000
Std Dev'n	0.01	0.00		0.00			0.00
Ratio:		0.00	*	0.00	*	*	0.00

	shm100 10	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	1.656	1.656	*	1.688	*	*	3.344
2	1.532	1.906	*	1.656	*	*	3.000
3	1.532	1.782	*	1.500	*	*	1.687
4	1.532	1.782	*	1.500	*	*	1.531
5	1.532	1.781	*	1.500	*	*	1.531
6	1.532	1.781	*	1.532	*	*	1.687
7	1.532	1.719	*	1.531	*	*	1.500
8	1.532	1.750	*	1.531	*	*	1.532
9	1.532	1.781	*	1.531	*	*	1.531
10	1.532	1.781	*	1.532	*	*	1.688
Average:	1.544	1.772		1.550			1.903
Std Dev'n	0.04	0.06		0.06			0.64
Ratio:		1.15	*	1.00	*	*	1.23

	shm100 100	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	2.000	2.000	*	2.000	*	*	2.906
2	1.875	4.719	*	2.031	*	*	4.687
3	1.875	4.625	*	1.875	*	*	2.344
4	1.875	4.594	*	1.843	*	*	1.938
5	1.844	4.594	*	1.875	*	*	2.031
6	1.875	4.593	*	1.875	*	*	1.844
7	1.875	4.594	*	1.843	*	*	2.125
8	1.875	4.563	*	1.875	*	*	2.000
9	1.875	4.594	*	1.875	*	*	2.031
10	1.875	4.594	*	1.843	*	*	2.000
Average:	1.884	4.347		1.894			2.391
Std Dev'n	0.04	0.78		0.06			0.82
Ratio:		2.31	*	1.00	*	*	1.27

	shm100 1000	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	5.469	5.469	*	5.500	*	*	7.813
2	5.344	32.875	*	5.500	*	*	7.125
3	5.313	32.687	*	5.344	*	*	5.500
4	5.312	32.719	*	5.312	*	*	5.468
5	5.313	32.719	*	5.343	*	*	5.313
6	5.312	32.750	*	5.343	*	*	5.500
7	5.313	32.719	*	5.343	*	*	5.312
8	5.312	32.719	*	5.312	*	*	5.312
9	5.313	32.687	*	5.344	*	*	5.500
10	5.312	32.718	*	5.344	*	*	5.750
Average:	5.3313	30.0062		5.3685			5.8593
Std Dev'n	0.0469	8.1792		0.0669			0.8288
Ratio:		5.63	*	1.01	*	*	1.10

	shm100 10000	shm100 20000	shm100 30000	shm100 40000
1	40.000	78.406	116.782	155.156
2	39.875	78.250	116.656	155.062
3	39.875	78.250	116.656	155.063
4	39.875	78.281	116.656	155.063
5	39.875	78.281	116.657	155.062
6	39.875	78.250	116.657	155.062
7	39.875	78.250	116.688	155.063
8	39.875	78.281	116.656	155.063
9	39.875	78.281	116.656	155.062

10	39.875	78.250	116.656	155.062
Average:	39.888	78.278	116.672	155.072
Std Dev'n	0.04	0.05	0.04	0.03
Ratio:		1.96	2.93	3.89

	circlep 100	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	16.531	16.625	*	16.625	17.312	*	17.094
2	16.531	84.157	*	16.843	*	*	18.094
3	16.531	83.843	*	16.625	*	*	16.813
4	16.562	83.812	*	16.593	*	*	16.563
5	16.532	83.750	*	16.625	*	*	16.563
6	16.562	83.843	*	16.593	*	*	16.844
7	16.593	83.750	*	16.625	*	*	16.813
8	16.563	83.813	*	16.594	*	*	16.938
9	16.563	83.843	*	16.625	*	*	16.593
10	16.562	83.750	*	16.625	*	*	16.562
Average:	16.5530	77.1186		16.6373	17.3120		16.8877
Std Dev'n	0.0198	20.1648		0.0700	0.0000		0.4378
Ratio:		4.66	*	1.01	*	*	1.02

	circlep 50	circlep 150	circlep 200	circlep 250	circlep 300
1	9.594	23.500	30.406	37.343	44.312
2	9.625	23.500	30.469	37.375	44.313
3	9.594	23.469	30.406	37.375	44.312
4	9.844	23.469	30.469	37.407	44.282
5	9.594	23.469	30.406	37.657	44.281
6	9.594	23.469	30.469	37.343	44.282
7	9.594	23.500	30.406	37.375	44.312
8	9.594	23.500	30.469	37.375	44.313
9	9.844	23.500	30.406	37.344	44.312
10	9.594	23.468	30.407	37.375	44.281
Average:	9.6471	23.4844	30.4313	37.3969	44.3000
Std Dev'n	0.0989	0.0156	0.0308	0.0887	0.0151
Ratio:	0.58	1.42	1.84	2.26	2.68

B.4.3. Synchronisation

In this section the programs used are as follows.

`hiho` The unsynchronised HiHo program.

`hihos` The synchronised HiHo program, using semaphore pipes.

	hiho 1000	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	6.093	6.125	*	6.125	6.375	*	7.094
2	6.063	8.719	*	6.094	*	*	6.468
3	6.094	8.688	*	6.094	*	*	5.906
4	6.093	8.812	*	6.187	*	*	6.094
5	6.188	8.812	*	6.094	*	*	6.187
6	6.094	8.813	*	6.094	*	*	6.094
7	6.187	8.781	*	6.094	*	*	6.094
8	6.094	8.844	*	6.125	*	*	6.125
9	6.187	8.500	*	6.093	*	*	6.156
10	6.094	8.531	*	6.094	*	*	5.968
Average:	6.1187	8.4625		6.109	6.375	*	6.219
Std Dev'n	0.0458	0.7875		0.03			0.32
Ratio:		1.38	*	1.00	*	*	1.02

	hihos 1000	+b 1 200	+b 1 190	+b 1 210	+b 10 200	+b 10 190	+b 10 210
1	8.969	11.406	*	*	*	*	*

2	8.938	11.219	*	*	*	*	*
3	8.906	11.438	*	*	*	*	*
4	8.968	11.281	*	*	*	*	*
5	8.969	11.281	*	*	*	*	*
6	8.938	11.281	*	*	*	*	*
7	8.938	11.500	*	*	*	*	*
8	8.937	11.188	*	*	*	*	*
9	8.687	11.188	*	*	*	*	*
10	8.718	11.063	*	*	*	*	*
Average:	8.8968	11.2845					
Std Dev'n	0.0991	0.1252					
Ratio:		1.27	*	*	*	*	*

B.4.4. Exception Handling

In this section the programs used are as follows.

- `timer` The Sieve of Eratosthenes program, used as a basic timing program. It is also run by each of the other programs in this section and is the source of the results in the other columns.
- `int1` The interrupt driven program, using the `s_timer()` supervisor call to suspend itself for a given number of milliseconds.
- `int2` The interrupt driven program, using the `e_timer()` supervisor call to schedule an event in a given number of milliseconds.

	timer	int1 1	int2 1	int1 20	int2 20
1	2.000	2.032	2.063	2.063	2.063
2	2.000	2.062	2.062	2.031	2.063
3	2.000	2.063	2.094	2.032	2.062
4	2.031	2.031	2.063	2.062	2.063
5	2.000	2.031	2.093	2.031	2.062
6	2.000	2.063	2.094	2.031	2.094
7	2.000	2.031	2.063	2.062	2.063
8	2.000	2.063	2.093	2.032	2.062
9	2.000	2.062	2.063	2.031	2.063
10	2.000	2.031	2.063	2.063	2.062
Average:	2.0031	2.0469	2.0751	2.0438	2.0657
Std Dev'n	0.0097	0.0157	0.0153	0.0146	0.0099
Ratio:		1.02	1.04	1.02	1.03

	int1 40	int2 40	int1 60	int2 60
1	2.031	2.062	2.031	2.032
2	2.031	2.063	2.031	2.031
3	2.031	2.062	2.031	2.063
4	2.032	2.063	2.031	2.062
5	2.031	2.062	2.031	2.063
6	2.031	2.031	2.031	2.062
7	2.032	2.031	2.032	2.063
8	2.031	2.031	2.000	2.062
9	2.031	2.031	2.031	2.031
10	2.032	2.062	2.032	2.031
Average:	2.0313	2.0498	2.0281	2.0500
Std Dev'n	0.0005	0.0154	0.0094	0.0153
Ratio:	1.01	1.02	1.01	1.02

B.4.5. Special Tests Performed Using FlexOS

Two series of special test were performed using FlexOS. The first series assessed the use of the postlinking utility supplied with FlexOS (see section 6.2.6) for those tests that made use of the facilities for creating child processes from disk. The second series was performed to assess the effect of background processes that performed I/O operations (see section 6.2.5). In each case, the names of the programs used in the column headings are the same as in the main series of tests detailed above.

B.4.5.1. Results With Postlinked Programs

	psieve 10	procreate 100
1	21.688	14.875
2	21.688	14.156
3	21.688	14.375
4	21.688	14.656
5	21.657	14.094
6	21.657	14.656
7	21.688	14.875
8	21.688	14.531
9	21.688	14.531
10	21.656	14.188
Average:	21.679	14.494
Std Dev'n	0.01	0.25

B.4.5.2. Results Using Background Processes With I/O Operations

	procreate 100	+b 1 200	+b 10 200
1	29.532	39.875	126.469
2	29.750	38.813	126.313
3	26.968	41.093	129.438
4	26.719	40.032	125.718
5	28.844	37.937	124.812
6	30.094	39.313	128.062
7	29.500	39.594	125.000
8	30.031	41.343	125.625
9	29.375	38.250	126.531
10	28.188	42.563	121.906
Average:	28.900	39.881	125.987
Std Dev'n	1.20	1.45	1.99
Ratio:		1.380	4.359
s/child	0.289	0.399	1.260

	pipe100 1000	+b 1 200	+b 10 200
1	3.656	26.312	269.562
2	3.469	26.313	280.063
3	3.469	26.250	285.938
4	3.468	26.250	284.375
5	3.469	26.375	282.156
6	3.469	26.187	279.531
7	3.438	26.250	274.719
8	3.469	26.281	276.157
9	3.468	26.250	284.656
10	3.469	26.313	277.219
Average:	3.484	26.278	279.438
Std Dev'n	0.06	0.05	3.78
Ratio:		7.542	80.197

	shm100 1000	+b 1 200	+b 10 200
1	5.469	30.063	285.593
2	5.344	29.906	287.750
3	5.313	30.000	289.031
4	5.312	29.969	286.687
5	5.313	29.906	386.125
6	5.312	29.938	289.875
7	5.313	29.968	287.156
8	5.312	29.906	288.062
9	5.313	29.938	288.219
10	5.312	29.937	285.344
Average:	5.331	29.953	297.384
Std Dev'n	0.05	0.03	30.94
Ratio:		5.618	55.781

	circlep 100	+b 1 200	+b 10 200
1	16.531	76.750	741.532
2	16.531	76.532	744.187
3	16.531	76.657	744.688
4	16.562	76.656	740.468
5	16.532	76.437	743.406
6	16.562	76.532	745.781
7	16.593	76.563	741.719
8	16.563	76.594	746.438
9	16.563	76.562	747.937
10	16.562	76.656	742.094
Average:	16.553	76.594	743.825
Std Dev'n	0.02	0.07	2.28
Ratio:		4.627	44.936