DESIGN OF A DISTRIBUTED UNIX KERNEL AND ITS MODELLING IN CSP
DIX, TREVOR IAN
ProQuest Dissertations and Theses; 1985; ProQuest Dissertations & Theses Global

---

INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

The following explanation of techniques is provided to help clarify notations which may appear on this reproduction.

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.

2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.

3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*

4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

*For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.

U·M·I Dissertation Information Service

University Microfilms International
A Bell & Howell Information Company
300 N. Zeeb Road, Ann Arbor, Michigan 48106

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

8608978

Dix, Trevor Ian

DESIGN OF A DISTRIBUTED UNIX KERNEL AND ITS MODELLING IN CSP

University of Melbourne (Australia) PH.D. 1985

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Design of a Distributed UNIX Kernel
and
its Modelling in CSP

Trevor I. Dix

February 1984

Department of Computer Science
University of Melbourne
Parkville, Australia, 3052.

Submitted to the University of Melbourne in fulfilment of the requirements for the degree of Doctor of Philosophy.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgements

To Rao (alias K. Ramamohanarao) I express my appreciation for his encouragement and support over the last two years. As virtual supervisor, he has played a vital role in bringing this work to completion.

In the early steges of work on the model of CSP, Jean-Louis Lassez and Liz Sonenberg were helpful with their critical comments. Rodney Topor deserves a particular mention for his suggested improvements to draft papers on CSP.

Our resident UNIX guru, Robert Elz, was a ready source of information when requested.

For allowing me to work in the Department of Computer Science at the University of Melbourne, I would like to thank Professor Peter Poole. Employment within the department on several occasions during my candidature also is acknowledged.

Two people deserve mention for kindling the motivation for this thesis. Firstly, Prabhaker Mateti for his belief that UNIX was worthy of study, be that as an exercise in programming correctness. Secondly, to Professor Tony Hoare who during his first visit to Australia asked that computer scientists take up the challenge of providing formal semantics for new language constructs.

It is with great pleasure that I acknowledge my wife, Sue Thomson, who has been particularly supportive over the last three years and has been the family bread winner for the last year. Sue also has managed to convince me on two separate occasions that we should visit that most magnificent of mountain ranges, the Himalayas. The four months that I have spent in this region with its peaks and people during the last three years have been sufficient to recharge my tired cells so as to complete this thesis.

For her assistance with proof reading, I thank Mike Incigneri.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

The thesis presented is that the standard UNIX kernel may be partitioned and distributed over a very local network resulting in increased processing power without unacceptable delays due to associated message passing, and that CSP may be used to model the distributed kernel.

That the UNIX kernel was selected for study is largely due to the impact that the facilities supported by and supplied with the operating system have made on teaching institutions and, more recently, industry. However, the approach that is taken need not be limited to UNIX and may be applicable to other operating systems.

The synchronized message passing commands of CSP adequately encapsulate the issuing and servicing of requests in a distributed kernel. Preemptive commands are proposed for CSP reflecting the need for management of peripheral devices and process priority within operating systems.

We define operational semantics for CSP including commands for exceptions and interruptions. The contribution that extended CSP makes to potentially real-time applications is examined and a scheme is given for the implementation of synchronized message passing in the presence of preemption.

For the partitioning of the kernel, we establish a suitable division of labour through the examination of a standard Version 7 UNIX system. The anticipated message passing throughput is considered in relation to commercially available hardware for a very local network providing very fast, reliable, variable length message transfers. An approach to implementation of the distributed kernel is presented which considers the residence of system tables and the use of existing source code.

A more precise notion of the processes involved in the distribution is presented by a model in CSP. This model shows, at a functional level, the interactions of processes in the network. The distributed model relies on the semantics of CSP and in particular uses language extensions for managing priority and preemption.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Chapter 1. Introduction 1
1.1. The Thesis 1
1.2. Distribution 2
1.3. Why Distribute the UNIX Kernel 2
1.3.1. What is the UNIX Kernel 3
1.4. Modelling 4
1.4.1. Why Model in CSP 5
1.5. Aims 5
1.6. Non-Aims 6
1.7. Chronology of Our Research Effort 6
1.8. Overview 8

Chapter 2. Exceptions and Interrupts in CSP 10
2.1. CSP 11
2.1.1. Suitability 13
2.1.2. Which CSP 13
2.2. Priority in Operating Systems 14
2.3. Preemptive Commands 15
2.3.1. About Execution 17
2.3.2. Informal Example 17
2.4. Process Model 18
2.4.1. Definitions 18
2.4.2. Operations 19
2.4.2.1. Traces 19
2.4.2.2. Processes 19
2.4.3. Composition of Processes 20
2.4.4. Processes After First Steps 22
2.5. Comments on the Process Model 23
2.6. Process Execution 24
2.6.1. Events, Histories and Progress 25
2.6.2. Channels 26
2.6.3. Ready and Possible Functions 26
2.6.4. Execution Function 28
2.7. Priority 31
2.8. Restricted Preemption 32
2.9. Some Examples 34
2.9.1. Asynchronous Buffering 35
2.9.2. Scheduling 36
2.10. In Summary 37

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Chapter 3. Implementation of Preemptive Guards 38
3.1. Input/Output Commands in Guards 39
3.2. A Problem of Contention 40
3.3. Solution by Preemption 42
3.4. Another Implementation of I/O Commands 44
3.4.1. Control Messages 44
3.4.2. States 45
3.4.3. Scheme 46
3.5. Implementation of Preemption 48
3.5.1. Handshaking 48
3.5.2. Algorithm 49
3.6. Some Implementation Details 51
3.6.1. Once Only Boolean Evaluation 51
3.6.2. Nested Preemption and Resumption 51
3.6.3. Bounds on Requests 52
3.6.4. Control Message Overheads 52
3.7. Comparing Generalized and Preemptive Guards 53
3.8. Related Work 55
3.8.1. Buffering Output Commands 56
3.8.2. Port Directed Communication 56
3.8.3. Synchronizing Resources 57
3.8.4. Otherwise as an Alternative 58
3.9. Another Else 58
3.10. In Summary 59

Chapter 4. A Distribution of the UNIX Kernel 61
4.1. Outline of Network 62
4.2. Outline of Distribution 63
4.3. Directions in Related Networks 64
4.4. In Support of the Distribution 66
4.4.1. Dual VAX 67
4.4.2. Satellite Processors 68
4.4.3. S/F UNIX 68
4.4.4. Diskless SUN Workstations 70
4.4.5. Remote Procedure Calls 70
4.4.6. Recapping 71
4.5. A V7 UNIX System 71
4.6. Job Mix 72
4.7. Statistics Gathering 74
4.8. Processor Allocation 75
4.9. System Calls 76
4.9.1. Usage 76
4.9.2. Timing 78
4.9.2.1. Process Management 79
4.9.2.2. I/O Management 80
4.9.3. Observations 81
4.10. Weighted System Call Loads 81
4.11. Disk I/O 83
4.12. Network I/O 85

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.13. Distribution Revisited 87
4.14. In Summary 88

Chapter 5. A Model of a Distributed UNIX Kernel 90
5.1. Level of Granularity 91
5.2. Language Orientation 92
5.3. Modelling Process Multiplexing 93
5.4. User Machines 94
5.4.1. User Processes 95
5.4.2. System Calls 97
5.4.3. Scheduler 99
5.5. File Machines 101
5.5.1. File Manager 102
5.6. Terminal Machines 103
5.7. Priority Revisited 104
5.8. In Summary 105

Chapter 6. Towards a Kernel Implementation 108
6.1. Standard Implementation Table Classification 109
6.1.1. Kernel Resident 109
6.1.2. Per Process Resident 110
6.2. Table Residence in Distributed Kernel 111
6.3. Capabilities 111
6.4. Messages 112
6.5. User Machines 114
6.5.1. Kernel Tables 114
6.5.2. Per Process Tables 114
6.5.3. Scheduling and System Call Code 114
6.5.4. Message Handler 115
6.6. File Machines 115
6.6.1. Server Processes and Message Handler 116
6.6.2. Tables and Code 117
6.7. File Manager 117
6.7.1. For Efficiency 119
6.8. Terminal Machines 120
6.9. In Summary 122

Chapter 7. Conclusions 123
7.1. Distribution of UNIX Kernel 123
7.1.1. Towards Implementation 126
7.2. Model of a Distributed UNIX Kernel 126
7.3. Priority in CSP 127
7.3.1. General Application 128
7.3.2. Real-Time Application 128
7.3.3. Control Message Scheme 129
7.4. Model of CSP 130
7.5. Contribution 130
7.6. Further Work 132

Bibliography 133

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 1

Introduction

Distributed operating systems are becoming more common as the computing fraternity recognizes the convenience and potential advantages that access to distributed facilities can provide. Typically, such systems require support in one operating system for access to resources controlled by another. In this thesis, we investigate a different approach. We propose the distribution of parts of an existing operating system kernel over a network whereby the whole system provides the same functionality as the original operating system. With this aim, we seek to design a distributed UNIX[1] operating system kernel.

Although we choose UNIX for the design of a distributed kernel, we emphasize that the approach to distribution that we will take need not be limited to UNIX. The approach may be applicable to other operating systems provided they have been designed in such a way that a similar dissection of source code is possible.

1.1. The Thesis

The thesis has two parts. The first is that the UNIX kernel [Ritchie74, Thompson78] may be partitioned and distributed over a very local network with the expectation of providing more processing power through truly concurrent execution of processes and of supporting distributed file systems. Message passing that is associated with the distribution must not lead to unacceptable delays and must not totally negate the advantages of distribution.

[1] UNIX is a trademark of Bell Laboratories.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The second part of the thesis is that CSP, Hoare's language for communicating sequential processes [Hoare78], may be used to model the distributed kernel. The model must show, at a functional level, the processes that are involved in the distribution and the interactions of such processes within the network.

1.2. Distribution

We shall investigate the feasibility of distributing the UNIX kernel over a very local network. We term the envisaged network "*very* local" in that the processors would be in very close proximity, typically in the same room. We can therefore require that such machines be connected by high bandwidth buses or channels which must support the real-time data traffic that could be expected for file systems and terminal systems.

In such a network, delays will occur due to the availability of the communications medium and due to data transfers across the medium. Given that we wish to distribute functions of the kernel, the choice of granularity of distribution is governed by such delays. For example, it unrealistic to believe that a very large network of very small processors, each dedicated to performing small sections of the kernel code and thus requiring considerable amount of message passing, would be acceptable. For such fine granularity, it is likely that the overheads associated with the message passing would be sufficient to more than negate the benefits of parallel execution. Consequently, we choose coarse granularity for distributed portions of the kernel. It is at this level that we wish to model the distributed kernel.

1.3. Why Distribute the UNIX Kernel

UNIX is a good example of a modern operating system which has made a major impact on the expectations of computer programmers in respect to the facilities that operating systems must provide. Ritchie and Thompson [Ritchie74] give three major factors which influenced their development of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

UNIX:

- The system was designed by programmers for programmers and importantly was interactive.
- Elegance was considered to be essential, originally being enforced by size constraints. When conflict arose between simplicity and efficiency, the line towards simplicity was taken.
- The system was self supporting, so that designers soon found the deficiencies.

The popularity of UNIX is partly due to the suite of system programs that are available and the interface between user programs and the operating system kernel. Since becoming operational in 1971, the number of installations has grown rapidly. In 1978 there were over 600 installations, mainly in computer science educational institutions for whom the licensing cost was minimal. This number has risen to 1200 educational user sites (3600 installations), 600 commercial sites (1000 installations) and 300 US Government sites (600 installations) at May 1983. In addition, there are systems from licensed distributors and UNIX lookalike systems.

Ritchie and Thompson say "the success of UNIX lies not so much in new inventions, but rather in the full exploitation of a carefully selected set of fertile ideas ...".[2] For example, forks were present in GENIE [Deutch65] and much of the system was based on the methodology used and experience gained from MULTICS [Organick72].

1.3.1. What is the UNIX Kernel

One question which naturally comes to mind is, "what is the UNIX kernel?".

From the user's point of view, the answer is that part of the operating system which provides a standard set of system calls which give user and

[2] [Ritchie74] page 374.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

utility programs access to the UNIX file system, including the terminal system, and the capability to execute programs under the supervision of the kernel.

At the other end of the scale, as seen from the designer's point of view, the kernel is a set of processes that are multiplexed using interrupts and traps as typical triggering mechanisms. These processes provide the interface between user processes and the system resources — the processor, memory and input/output devices.

We shall take an intermediate stance since we have not discussed such processes.

Definition:

> The **kernel** is all of the operating system code that would normally execute in supervisor or privileged mode on a single processor implementation and the associated system data structures. This includes code implementing process switching, memory management, supervisor calls and device control. In the distributed operating system kernel, we shall also include message passing code.

We shall refer to the *standard* UNIX kernel and the *standard* UNIX implementation. By this we mean the kernel and implementation which is available as a standard release. We contrast this with the *distributed* kernel or implementation, which refers to our design.

1.4. Modelling

By developing a model of the distributed kernel, we intend to show the processes that are involved in the system and the communication that must occur among them. We require a notation that will suit a distributed system and that will abstract away from the details of implementation. A better understanding of the distribution, and also of the standard kernel, should be possible through such a high level representation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.4.1. Why Model in CSP

Lauer and Needham [LauerH78] have discussed the duality of operating system structures and more recently remote procedure calls have been raised as another alternative [Reid80] [Andrews83]. We shall see whether a message-oriented language like CSP is suited to the modelling task, especially in the light of Hoare's statement that CSP is not intended to be a complete programming language [Hoare78].

The language CSP was designed to be suitable for implementation on a network where individual processes execute on a separate processor (with local memory). In addition, CSP was intended to be implementable by multiplexing process execution on a single processor. The distributed system that we envisage will have several processors on each of which will execute many processes, that is, a blend of both situations. Moreover, in the framework of a model for communicating sequential processes, Hoare has addressed operating system examples [Hoare80a]. We wish to see if the model extends to our application.

1.5. Aims

We are seeking answers to the following questions:

- Can the UNIX kernel be easily partitioned and distributed?
- What is a suitable partitioning?
- What sort of implementation would be suitable?
- Can CSP be used to model the distributed UNIX kernel?
- Are the techniques that are used in the model suited for any other applications?

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.6. Non-Aims

We do not intend to make a general study of distributed computer systems, nor to examine all the UNIX based systems which provide remote file access and process invocation.

No consideration of robustness and fault tolerance, that might be required of local area and long-haul networks, is to be undertaken. The distributed UNIX kernel will be as tolerant of crashes by input/output devices as the standard kernel is.

We do not undertake an implementation.

1.7. Chronology of Our Research Effort

To explain the way in which our work developed, we present a chronology of research effort.

Within the Department of Computer Science at the University of Melbourne, work was beginning on the application of a fast, time division multiplexed switch that had been built in the Department for networking between two UNIX machines. One group took the approach towards providing virtual file systems and processes [Vines82]. Whereas, the author started to look at the kernel source code to determine the feasibility of partitioning and distributing the kernel over a network.

Around this time, Professor Hoare visited Australia presenting his model for communicating sequential processes [Hoare80a, Hoare80b]. Hoare's examples included a batch operating system and a discrete event simulation. We thought it should be possible to model the standard UNIX operating system and the distributed operating system using Hoare's model. Closer examination revealed that the techniques employed in the simulation model were not applicable to the scheduling of user processes. So, we set about developing extensions to CSP which could support the envisaged task.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The adoption of interrupts, and the more restrictive exceptions, followed quite naturally by considering the multiplexing of user processes to be a form of interruption. Guarded commands provided an ideal mechanism with which to express preemption of this form. Interrupt commands appeared to be well suited to capture message passing between distributed parts of the operating system and for actual device interrupts.

The new commands were included in Hoare's trace model of communicating sequential processes. However, it was realized that the temporal qualities of preemption were not expressed by this model. As a consequence we developed operational semantics based on environments. The resulting technical report "Exceptions and Interrupts in CSP" was published about one year later [Dix83b]. The scheduling problem is addressed in that paper.

During the latter part of the above development period, work began on determining the characteristics of a Version 7 UNIX system with respect to input/output requirements and system call usage in the operating system. We had a particular distribution of the kernel in mind and wished to see whether such an approach was feasible on empirical grounds. These early results were presented in "Towards a Distributed UNIX Kernel" [Dix83a] and they led us to continue with the approach but with certain refinements. Subsequently, additional timing figures and data transfer statistics have been gathered.

Having developed preemptive commands for CSP, our research followed two directions. The first was to devise a scheme whereby input/output guards, preemptive guards and input/output commands could be supported. After the start of this work, a scheme for generalized input/output guards was published by Buckley and Silberschatz [Buckley83]. This work was encouraging as we had already determined that preemptive commands could be used as an alternative to generalized guards. It also provided a useful precedent for presentation. The algorithm we developed for synchronization in the presence of preemption is presented in "Preemptive Guards in CSP" [Dix83c]. This paper also forwards

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

preemption in CSP as a programming technique for real-time applications.

The second research direction we took at this time was to develop a model in CSP of the distributed UNIX kernel that we were designing. We saw that with the extensions for preemption we could model the scheduling of user processes and we knew that although message passing was synchronized in CSP, asynchronous models could be produced using interrupt type preemption. Such models could be applied to our distribution of the UNIX kernel to show the processes and their interactions at a functional level. This would filter through to the processes that exist in a standard UNIX implementation. The resulting model of a distributed UNIX kernel is given in [Dix84].

The above chronology explains the directions that will be followed in this thesis. Each of the above four papers contribute towards an individual chapter. The only other chapter, excluding the introduction and conclusions, addresses some of the practical problems associated with dissecting the code in the standard UNIX kernel for distribution.

1.8. Overview

In chapter 2, we review the history of CSP and draw attention to the lack of support for priority. We introduce preemptive commands for exception and interruption in order to provide such facilities for applications like operating systems. Formal operational semantics are presented using a model of process execution which incorporates the necessary temporal properties associated with preemption.

We consider the contribution that preemption in CSP can offer other potentially real-time applications in chapter 3. An implementation scheme for communication in the presence of preemption is presented and comparisons are made with other communication schemes.

Chapter 4 examines the distribution of the kernel from an empirical view. The envisaged type of distribution and its relationship to existing systems is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

presented. The characteristics of an existing Version 7 UNIX implementation are given with respect to usage and time to service system calls and input/output loads. From these figures we estimate the loads that would require servicing by the machines and the communications network in the distributed system.

In chapter 5, we present a model of the distributed kernel in CSP. The model is at a high level and shows processes within distributed components and the interactions that must occur between them.

Chapter 6 returns to the practical problems associated with a distributed implementation. We present an approach which aims to use as much of the existing system source code as possible except where efficiency would be compromised within the system. Residence of code and tables throughout the network is considered.

Finally, chapter 7 presents conclusions and reviews the contribution of this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 2

Exceptions and Interrupts in CSP

This chapter presents additions to Hoare's language for communicating sequential processes, CSP [Hoare78], to cater for preemption of processes for exception and interrupt handling. Operational semantics are given within a model of communicating sequential processes.

Our motivation for the work comes from the desire to present a model of the distributed UNIX kernel which uses these extensions. For this model, a method is required to functionally specify the operating system with separation between typical system processes, such as the user process scheduler, the file system and the supervisor. The description also is to be amenable to decomposition into further processes. CSP seems well suited to these requirements.

Section 2.1 briefly looks at the work of Hoare and other researchers with CSP. We then discuss the need for priority in operating systems in section 2.2. The remainder of the chapter deals with the introduction of exceptions and interrupts into CSP.

An informal introduction to the proposed preemptive commands is given and then Hoare's process model [Hoare80a] is presented with some changes and incorporating preemption (sections 2.3 and 2.4). Section 2.5 addresses the inadequacies of this model with respect to the expression of process priority which is necessary to define the semantics of preemption. A model of execution is given in section 2.6 which defines the operational semantics of preemption within a given environment.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The discussion in sections 2.7 and 2.8 concerns the assignment of a priority ordering to input/output alternatives in CSP and includes a further extension to the preemptive commands. Finally, two examples are presented in section 2.9.

The contents of sections 2.3 to 2.8 appear in [Dix83b].

2.1. CSP

We present a brief history and list of the work relating to semantics, proof and models for CSP.

In 1978, Hoare published what he termed an ambitious attempt to find a single simple solution to the problems asscciated with process communication and synchronization [Hoare78]. The proposed language has come to be known as CSP.

Until that time, shared memory tended to be used for communication and a variety of methods were applied for synchronization. These included semaphores [Dijkstra68], events (for example, PL/I), critical regions [BrinchHansen72, Hoare72], monitors [Hoare74, BrinchHansen75] and path expressions [Campbell74].

CSP supports the input and output of messages as primitives of parallel programming. Hoare chose synchronized unbuffered message passing instructions to provide both mechanisms.

Message passing in itself was not new at this time. In fact message primitives were used as a basis for the RC4000 operating system kernel [BrinchHansen70]. The novel features of CSP included the introduction of nondeterminism through Dikjstra's guarded commands [Dijkstra75] and the use of input commands in guards (input guards). The language was also intended to be supportable on monoprocessors (by multiplexing) and on a network of processors each having its own memory.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Denotational semantics of nondeterminism, concurrency and communication for CSP were given by Francez et al. [Francez79]. Thereafter, Hoare worked towards the objective of providing a basis for the proof of CSP programs and investigated models for communicating processes.

A model for communicating sequential processes was presented in [Hoare80a]. Details will be given in section 2.4 when we build upon this model. Suffice to say that the model is based upon traces of possible process behaviour using operators that look like program constructs. The model is simple and, as given by Hoare, deliberately avoids certain problems of nondeterminism. Later, Hoare et al. presented a theory of communicating sequential processes which addressed nondeterminancy [Hoare81a]. Again the model is set theoretic and employs traces and refusal sets.

A calculus of total correctness for communicating processes is given in [Hoare81b] for proving that a process satisfies an assertion describing its intended behaviour. Hehner and Hoare take another look at processes in [Hehner83] for which the axioms and proof rules of the calculus become theorems about processes which are defined by predicates.

The initial thrust of other researchers into CSP was towards proving correctness. Three proof systems may be found in [Apt80], [Misra81] and [LevinG81]. A comparison of the techniques is given in [Misra81]. The detection of distributed termination in CSP is covered in [Francez80] and [Misra82].

An operational semantics for CSP appears in [Plotkin82]. The approach therein employs labelled transition sequences and the presentation is quite abstract. In section 2.6, we take a different approach to operational semantics for CSP including the extensions for preemption. Ours is less abstract and intentionally captures the nature of the environment showing possible delays associated with communication and possible deadlock.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.1.1. Suitability

Certainly, other languages which have an abstraction for distributed processes and message passing could be used as a suitable tool for the desired application for which we will use CSP. It is with the research effort mentioned above behind us that we choose CSP. The semantics of CSP are well defined and understood, and, although we shall not require the facility, proof methods for parallel CSP programs exist. Another appealing property of Hoare's model, that we shall extend, is simplicity which is due to its set theoretic basis.

CSP is intended for the programming of processes which may be distributed over a network of small processors. Such processes do not have shared memory for communication. Instead, CSP supports input/output message primitives. The separation of processes in CSP for distribution to individual processors in a network suits our functional distribution requirements. Pairs of synchronized communication commands may be used to replace user calls to the supervisor and subsequent returns. Also, messages adequately encapsulate the passing of requests for input and output of data within the operating system.

Other communication structures may be suited to that modelling task. However, CSP can be used to provide equivalent facilities. For example, remote procedure calls are modelled by[1]

Server :: *[?request(id,params)→ service(params); id!reply(result)].

With the extensions for preemption, it will be shown that asynchronous buffering can also be modelled.

2.1.2. Which CSP

With regard to the definition of the language CSP, we use the notation of [Hoare78] in that "[", "]" and "*[" are used for bracketing alternatives and

[1] This is similar to the construction given in [Francez83].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

repetition. However, the following two changes as applied by Levin and Gries [LevinG81] are assumed:

- Input and output commands may appear in guards.[2]
- Distributed termination is not supported. Specific termination signals must be used to terminate repetition involving input/output guards.[3]

2.2. Priority in Operating Systems

An operating system, by necessity, must include facilities to handle peripheral devices. Current hardware usually provides priority interrupts as a means of assigning the processor to device handlers. However, CSP does not support an analogue to this situation. Further, CSP is also designed to be able to execute on a single processor. In such an environment, and for any network on which more processes are required than there are processors, processor multiplexing is necessary. But within operating systems, certain processes are required to have priority over others; for example, system above user processes. Again CSP is unable to model this.

With respect to alternative communication in CSP, which is expressed as a set of guards containing input/output commands, the nondeterministic nature of guard evaluation prevents any assignment of a priority ordering to communications. For purely boolean guards (which only require evaluation of an expression as true or false), the modelling of a priority ordering of evaluation is possible using loops (see section 2.7). However, if an input/output command occurs in the set of guards, such a technique does not fulfil the requirements.

---

[2] This does not mean that the so called generalized i/o guards of [Bernstein80] and [Buckley83] are supported. An input (output) guard must match an output (input) non-guard command. See chapter 3.

[3] Specifically, alternation will not fail if the communication partner of an i/o guard has terminated and similarly repetition will not terminate. See [LevinG81] page 285 for a bounded buffer example with specific quit signals.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In related research, Lauer and Shields [LauerP78] found it necessary to apply the priority operator of Campbell [Campbell76] in systems programming examples using path expressions to model exceptions and interrupts. There is support in languages like PL/I, Mesa, Ada[4] [Ada82] and CLU [Liskov79], for providing program language facilities for handling exceptions. Modula [Wirth77a], Modula-2 [Wirth83] and SR [Andrews81a, Andrews81b] are examples of languages which provide for device interrupt handlers. The Series/1 distributed operating system (SODS/OS) [Sincoskie80] supports exceptional and interrupt messages as an integral part of its message passing primitives.

The issues of concern for exception handling are discussed in Levin's thesis [LevinR77]. Levin stresses that the term "error" should not be used as a synonym for exception and that the latter may include the former. We point out that the occurrence of an exception may be a desirable event playing much the same role as an interrupt (as in fact, Levin's scheme does), but not requiring resumption of the preempted process. That is, an exception may be likened to a preferred alternative, so much so that preemption is required. This view supports Levin's and is diametrically opposed to the notion of an error.

Given that preemptive situations exist in practice, the commands proposed in this chapter allow for priority exception and interrupt handling in CSP. Guarded commands are used to provide the mechanism for preemption.

2.3. Preemptive Commands

The proposed commands are

P **except** Q and P **interrupt** Q.

Intuitively, P is a process which may be preempted by the execution of Q. Should P successfully complete execution so does the preemptive command.

[4] Ada is a trademark of the U.S. Government Ada Joint Program Office.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Whether an exception or interrupt occurs and Q is executed, depends on the environment. Generally, the environment of these processes may best be imagined as a process R executing in parallel. If R realizes a preemptive condition for a preemptive command, it raises that condition for Q. Preemption of P by Q must then occur at the earliest possible moment. The raising of a preemptive condition by R may be achieved by attempting to synchronize with Q.

For the except command, preemption results in the execution of Q, thus preventing any further execution of P.

For the interrupt command, preemption suspends execution of P, executes Q and, on successful completion of Q, resumes P.

Q is restricted to CSP's alternative command. That is,

$Q = [g_1 \rightarrow cl_1 \; \square \; g_2 \rightarrow cl_2 \; \square \; \ldots \; \square \; g_n \rightarrow cl_n]$

Terminology :

- P is called the *preemptible command* and Q the *preemptive command*.
- The guards, g_i, are called *preemptive guards* and each may contain a boolean expression and/or an input/output command.
- A guard is

  (1) *enabled* if the boolean expression evaluates to true, or is not present,

  (2) *ready* if the communication can proceed without causing a delay, or is not present,

  (3) *true* if *enabled* and *ready*.

Restrictions :

- P and Q cannot communicate.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- P cannot make assignments to variables contained in boolean expressions of preemptive guards of Q,[5] whereas I can in "I; P **interrupt** Q", for example.

2.3.1. About Execution

Immediately prior to the execution of P, the preemptive guards of Q are examined. If any guard is true, preemption occurs and Q is executed. For exception, preemption implies the termination of P. Whereas for interruption, P is resumed after successful completion of Q and is then interrupted again if any preemptive guards are true. The execution of a command list, cl_i, occurs if g_i is true. Should more than one guard be ready a nondeterministic choice is made.

Due to the restriction of assignment by P to variables contained in preemptive guards, only those guards which were enabled but not ready can become true during the execution of P. However, after interruption, Q may change such variables and preemptive guards must be reevaluated on resumption of P.

The preemptive commands enable one to establish a priority ordering of Q over P during the execution of P. The preemptive guards of Q serve as the triggering mechanism for the execution of Q. Q may be viewed as a preferred alternative over the whole of the execution of P.

Preemption is particularly well suited to a multiplexed implementation for P and Q. At no time can both P and Q be actively executing (assuming Q does not use some form of busy wait).

2.3.2. Informal Example

The following example illustrates the application of both preemptive commands informally. Consider the vending machine

---

[5] A relaxation of this requirement, to use once only evaluation of boolean expressions, provides for a simple implementation (see chapter 3).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
VM = *[[?5_cents→eject_packet_of_biscuits] interrupt
          [?insert_service_key→restock_biscuits; ?remove_service_key]
     ] except [?power_down→skip]
```

Whether waiting for insertion of 5 cents, ejecting a packet of biscuits or attempting to eject when there are no packets left, the serviceman may open the machine by inserting the service key. After being restocked and the removal of the service key, the vending machine resumes where execution was interrupted. This process is repeated except if the power goes down, in which case it is possible to be cheated of 5 cents.

2.4. Process Model

Having presented an informal introduction to the preemptive commands, a formal definition will now be given within Hoare's model for communicating sequential processes. Most of the definitions in sections 2.4.1 to 2.4.3 may be found in [Hoare80a]. Any major differences are noted in the text. We use the bracketing symbols "(" and ")" in the model to distinguish it from the language notation "[" and "]".

2.4.1. Definitions

A *symbol* represents an atomic event in which a process may participate. The symbol ✓ represents successful termination.

A *trace* is a finite sequence of symbols recording a possible behaviour of a process up to some point in time. <> is the null trace.

The concatenation of two traces s and t, is represented by st. For example, if s=<a,b> and t=<x,y,z>, then st=<a,b,x,y,z>. The symbol ✓ occurs only at the end of a trace. To maintain this condition s<✓>r $\underline{\Delta}$ s<✓>.

A *process* P is defined to be a set of traces such that

$$\langle\rangle \in P,\ st \in P \implies s \in P$$

That is, a process is a prefix closed set of traces.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABORT $\underline{\Delta}$ {<>}, represents a process which never succeeds in doing anything.

SKIP $\underline{\Delta}$ {<>,<✓>}, represents a process which always successfully terminates.

Notice that the set of processes constitutes a complete lattice, under set inclusion, with ABORT at the bottom.

2.4.2. Operations

2.4.2.1. Traces

(a) *Sequence* :

If the trace s does not contain ✓ then s;t $\underline{\Delta}$ s and (s<✓>r);t $\underline{\Delta}$ st (in which case s<✓> is said to successfully terminate).

(b) *Restriction* :

s⌈A (s restricted to A) is the trace formed by omitting all symbols not in the set of symbols A from the trace s.

(c) *Closure* :

A^* is the set of all finite traces of symbols from the set of symbols A.

2.4.2.2. Processes

The *alphabet* of a process, $\bar{P}$, is the set of symbols denoting events in which the process P may participate. For convenience in later definitions, $✓\in\bar{P}$, however, P does not have to contain a trace s<✓>, as is exemplified by ABORT.

The set of *first steps* of P, fs(P) $\underline{\Delta}$ {e | <e>∈P}

The following definition forms the basis for generating prefix closed sets of traces from an event e and a process P.

(e→P) $\underline{\Delta}$ {<>} ∪ {<e>s | s∈P} where e≠✓

The following operations will be required for the definitions of preemptive exception and interruption.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For processes P and Q,

$P.Q \triangleq \{pq \mid p \in P,\ q \in Q\}$ and $P.Q.R = (P.Q).R$

$P:Q \triangleq \{s \mid \exists n>0\ \exists p_1 \ldots p_{n+1} \in P\ \forall i<n\ \exists q_i \in Q.\ s = p_1 q_1 ; p_2 q_2 ; \ldots p_n q_n ; p_{n+1}\}$

That is, $s \in (P:Q)$ if s is a trace of P containing zero or more occurrences of all traces of Q. Should s contain more than one such occurrence, then a trace of Q must exist which successfully terminates.

$P:Q:R = (P:Q):R$

Later application of these operators for preemption will only occur when $P \neq Q$.

2.4.3. Composition of Processes

In the following e and e_i are symbols, P, P_i and Q are processes.

(a) *Alternation* :

$$(e_1 \rightarrow P_1 \ \square \ \ldots \ \square \ e_n \rightarrow P_n) \triangleq (e_1 \rightarrow P_1) \cup \ldots \cup (e_n \rightarrow P_n)$$

This construction prohibits (P□ABORT) and (P□SKIP). In future the term alternation will also be taken to include the single guard construction $(e \rightarrow P)$.

(b) *Sequential Composition* :

$$(P;Q) \triangleq \{s;t \mid s \in P,\ t \in Q\}$$

Notice that ABORT;P = ABORT and SKIP;P = P

(c) *Parallel Composition* :

$$(P \| Q) \triangleq \{s \mid s \in (\bar{P} \cup \bar{Q})^*,\ (s \lceil \bar{P}) \in P,\ (s \lceil \bar{Q}) \in Q\}$$

The model gives the set of all possible interleavings of symbols. Synchronized communication is modelled by requiring that a symbol in $(\bar{P} \cap \bar{Q})$ may only occur in $(P \| Q)$ if it occurs simultaneously in P and Q. The definition is easily extended for more than two processes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(P‖Q) may only terminate successfully if both P and Q could. So, (P‖ABORT)≠P if P≠ABORT.

Due to the inclusion of ✓ in the alphabet of all processes, this definition is more restrictive than that of Hoare's. If P does not contain a trace which successfully terminates, Hoare allows (P‖Q) to terminate successfully when Q terminates successfully. So, (P‖ABORT)=P.

(d) *Repetition* :

For $P = (P_1 \Box \ldots \Box P_n)$,

$$*P \triangleq SKIP \cup P \cup (P;P) \cup (P;P;P) \cup \ldots$$
$$= SKIP \cup (P;*P)$$

As described by Hoare [Hoare80a], the desired solution of the recursive equation is the least fixed point, under set inclusion.

Hoare defined repetition of this kind to be a slave process which could not itself successfully terminate and which relied upon a master process to successfully terminate and hence terminate in the parallel composition. This is analogous to using distributed termination to abort. Such is not the case with the above definition for which termination of repetition is determined by P itself. The removal of system support for distributed termination requires P to be guarded by boolean expressions. Consequently, initialization must accompany repetition, such as (I;*P).

(e) *Exception* :

For $\bar{P} \cap fs(Q) = \{\}$,

$$P \textbf{ except } Q \triangleq (P.Q)$$

Notice that no communication is allowed between P and Q and that Q cannot be SKIP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(f) *Interruption* :

For $\bar{P} \cap fs(Q) = \{\}$,

P **interrupt** Q $\triangleq$ P:Q

All the above definitions preserve prefix closure and hence define processes.

2.4.4. Processes After First Steps

Hoare [Hoare80a] defines the possible future of process P as

P **after** s $\triangleq$ {t | st∈P} where s∈P is a trace of its past

and gives the following theorem which enables one to consider the first steps of a process and the subsequent associated behaviour.

$$P = \bigcup_{e \in fs(P)} (e \rightarrow (P \textbf{ after } \langle e \rangle))$$

The possible behaviour of composed processes after a first step e is as follows.

For e∈fs(P) and e≠✓,

(P;Q) **after** <e> = (P **after** <e>);Q

(*P) **after** <e> = (P **after** <e>);*P

For e∈(fs(PoQ) and e≠✓ where o is □, ||, **except** or **interrupt**,

(P□Q) **after** <e> = **if** e∈fs(P)∩fs(Q) **then** (P **after** <e>)∪(Q **after** <e>)
else if e∈fs(P) **then** P **after** <e>
else Q **after** <e>

(P||Q) **after** <e> = **if** e∈fs(P)∩fs(Q) **then** (P **after** <e>)||(Q **after** <e>)
else if e∈fs(P) **then** (P **after** <e>)||Q
else P||(Q **after** <e>)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(P except Q) after <e> = if e∈fs(P) then (P after <e>) except Q
else Q after <e>

(P interrupt Q) after <e> = if e∈fs(P) then (P after <e>) interrupt Q
else (Q after <e>);(P interrupt Q)

The above theorems will be applied in section 2.5.

2.5. Comments on the Process Model

The process model represents a process as a set of traces (where each trace gives a possible history) and is thereby sequential. For parallel composition, synchronized communication occurs simultaneously and other events are interleaved. The model does not allow for two atomic events to occur concurrently. One may represent duration of an event using two events - start and finish of that event.

The occurrence of synchronized communication may involve the passing of a message which effectively assigns the value of the message to the variable. For simplicity, such communication can be represented by the channel name within the model. Hoare [Hoare78] commented on the equivalence of process naming in input/output pairs and the alternative using channel names, which Misra and Chandy [Misra81] adopted. For example, the communicating processes P and Q could contain the commands Q?channel(variable) and P!channel(value) respectively, or the commands channel?variable and channel!value. More recently, Francez has presented conventions for computed communication targets and unspecified communication targets along with extensions for cooperating proofs [Francez83].

Nondeterminism of alternation and of repetition is nicely expressed in the process model as the nondeterministic choice from a set of alternative traces. Similarly parallelism is represented by nondeterministic choice from a set of traces containing possible interleavings of events. Exception and

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

interruption also are represented by alternative traces. However, the model does not address the requirement given in section 2.3, that preemption must occur at the earliest possible moment.

For example, consider ((P **except** Q)‖R‖S). Suppose the environment of P **except** Q, R‖S, is such that R is ready to communicate with P and similarly S with Q. If in addition, P is ready to communicate with R and Q with S, then it is essential that communication between Q and S has priority; that is, preemption occurs.

This inadequacy can only be overcome by considering the environment of the processes. The willingness of parallel processes to communicate may influence the occurrence of preemption. Similarly the termination of repetition relies on the values of local variables. The process model does not capture such properties. An operational model is given in the next section which does capture these properties.

2.6. Process Execution

The process model given in section 2.4 provides information about the possible execution of a process. However, the actual execution is influenced by the environment and by nondeterminism. For example, the termination of repetition requires all guards to be false. Process preemption is required as soon as an exception or interrupt condition occurs. The ability of a process to proceed may be governed by the willingness of another process to communicate (the global nondeterminism of Francez et al. [Francez79]), or by the values of local variables which may have resulted from communication (local nondeterminism). An operational model of execution is given in this section which addresses these notions.

The model of execution

- introduces the required priority of preemptive events,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- makes explicit the termination of repetition,
- represents the possibility of waiting for channel events.

It will be noticed that the model dictates that an implementation must not wait for an input/output event when another guard is ready to execute.

2.6.1. Events, Histories and Progress

Only symbols representing the following *events* are permitted within the model:

(a) *Local* : unconditional action, for example, assignment.

(b) *Boolean* : evaluation of a boolean expression as true.

(c) *Channel* : input or output of a set of values.

(d) *Mixed Boolean / Channel* : evaluation of a boolean expression as true and input or output of a set of values.

In addition the symbol ✓ has been introduced and may be considered as a local event.

The *history* of a process at any given time is denoted by the trace of symbols representing those events which have been executed by that process up to that time.

Progress is the transition from history t to history t<e> by the execution of the event represented by the symbol e.

In general, the *environment* of a process determines whether such progress is possible. The *internal environment* of a process is defined by the values of variables resulting from the history t of the process. The *external environment* depends on communication with parallel processes and will be addressed in the next section.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.6.2. Channels

Hoare [Hoare81b] introduces channel communication as follows. A process communicates by sending and receiving messages on named channels. A message output by one process along a channel is received instantaneously by all other processes connected by that channel, provided that all these processes are simultaneously prepared to input that message.

Associated with each process P is a *channel variable* c.ready$_P$ which, at a given time, denotes the set of messages which P is prepared to communicate on channel c. So if c.ready$_P \neq \{\}$ then P wishes to communicate on channel c.

Using the above ready sets, for processes $P_1 \| \ldots \| P_n$ which are connected to channel c, we define the set of messages that may be communicated on that channel at a given time, the *channel ready set*,

$$c.READY \triangleq \{x \mid x \in \bigcap_{i=1}^{n} c.ready_{P_i} \text{ and only one } P_i \text{ is prepared to output } x\}$$

Notice that, provided one of the parallel processes is making progress, the value of c.READY may change. At any given time, as given by the history of the parallel composed processes, c.READY captures the capability of communication to occur on channel c.

For any one of the parallel processes P_i above, c.READY defines the *external environment* with respect to channel c. Relative to time as viewed from P_i the external environment may change.

2.6.3. Ready and Possible Functions

Associated with each process P, are two sets of boolean valued functions $r_t(e)$ and $p_t(e)$, where $t\langle e \rangle \in P$.

For local, boolean, channel and mixed boolean/channel events represented by the symbols l, b, c and (b,c) respectively, the *ready* function, r, is defined:

(a) $r_t(l) \triangleq$ true

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(b) $r_t(b) \triangleq b$ (that is, evaluation of b)

(c) $r_t(c) \triangleq$ true if c.READY≠{}

false otherwise

(d) $r_t((b,c)) \triangleq r_t(b) \wedge r_t(c)$

That is, the *ready* function represents the ability to execute the guard successfully *now* having already progressed to history t. A local event is always ready to execute. Execution of a boolean event is dependent on that boolean being true. A channel event is dependent on the channel.

The ready function captures the environment of a process at the time of evaluation. The evaluation of boolean expressions uses the internal environment whereas the readiness of communication relies on the external environment.

The *possible* function, p, is defined:

(a) $p_t(l) \triangleq$ true

(b) $p_t(b) \triangleq b$ (that is, evaluation of b)

(c) $p_t(c) \triangleq$ true

(d) $p_t((b,c)) \triangleq p_t(b)$

That is, the *possible* function represents the ability to execute a guard successfully *now* or in the *future*. Since there is no distributed termination of input/output guards, it is assumed that a channel will always be able to communicate sometime in the future.

The definitions of r and p are extended to processes as follows.

$r(P) \triangleq$ true if $\exists e \in fs(P).r_{<>}(e)$=true

false otherwise

$p(P) \triangleq$ true if $\exists e \in fs(P).p_{<>}(e)$=true

false otherwise

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Notice that r(ABORT) = p(ABORT) = false and r(SKIP) = p(SKIP) = true.

The environment of each sub-process within a composition, depends on the state of its own channels and the values of its variables resulting from its history. Importantly, there is no need for a sub-process to know the global history of the composition nor the global environment.

2.6.4. Execution Function

The execution function, E(P), returns a set of terminating traces of process P executing in a particular environment. E is defined recursively so that each call describes the next steps of P, possibly after a wait. Local and global nondeterminism are resolved by the possible and ready functions.

The following construct will be used to represent a wait for a channel event to occur if no other events are ready.

when boolean **do** S $\underline{\Delta}$ **if** boolean **then** S
else when boolean **do** S

Each application of the execution function results in one of the following three outcomes.

(i) Termination occurs - with success or failure.

(ii) Nonterminating deadlock occurs - because the boolean in "**when** boolean **do** S" never becomes true.

(iii) Progress occurs - a set of next events and consequent application of the execution function.

The set of *ready first steps* of P, rfs(P) $\underline{\Delta}$ {e | <e>$\in$P, r(e)=true}. That is, those events which can occur now.

For a symbol e and a set of traces A, <e>A $\underline{\Delta}$ {<e>t | t$\in$A}.

The execution function is defined as in the following five cases.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(a) ABORT *and* SKIP:

$$E(ABORT) \triangleq \{<>\} \quad \text{and} \quad E(SKIP) \triangleq \{<\surd>\}$$

Termination, with failure for ABORT and success for SKIP, occurs in this case.

(b) *Alternative, Sequential and Parallel Composition* :

For P equal to $(e \rightarrow Q)$, $(Q \Box R)$, $(Q;R)$ or $(Q \| R)$,

$$E(P) \triangleq \textbf{if } p(P) \textbf{ then } \textbf{when } r(P) \textbf{ do } \bigcup_{e \in rfs(P)} <e>E(P \textbf{ after } <e>) \textbf{ else } \{<>\}$$

Three situations are possible (as in cases (d) and (e) below).

(i) p(P) is false, resulting in termination with failure.

(ii) p(P) is true but r(P) never becomes true, resulting in nonterminating deadlock. This can occur if the required channel event never occurs.

(iii) r(P) is or becomes true, resulting in progress of P.

Simplification of compositions with ABORT and SKIP are as follows.

$E(ABORT;P) = E(ABORT)$ and $E(SKIP;P) = E(P)$ by definition of ";",

$E(ABORT \| SKIP) = E(ABORT \| ABORT) = E(ABORT)$ and

$E(SKIP \| SKIP) = E(SKIP)$ by definition of "$\|$".

(c) *Repetitive Composition* :

Termination of repetition is succinctly expressed by

$$E(*P) \triangleq \textbf{if } p(P) \textbf{ then } E(P;*P) \textbf{ else } E(SKIP)$$

So this case reduces to cases (a) and (b).

(d) *Except Composition* :

For $P \neq SKIP$,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

E(P except Q) $\underline{\Delta}$ if (p(P) or p(Q)) then

when (r(P) or r(Q)) do

if r(Q) then E(Q)

else $\bigcup_{e \in rfs(P)}$ <e>E((P after <e>) except Q)

else {<>}

and

E(SKIP except Q) $\underline{\Delta}$ if r(Q) then E(Q)

else E(SKIP)

The effect of an exception causing the preemption of P by Q is now obvious. If a guard of Q is ready to be executed then it does; otherwise a step is taken in P.

For simplification of ABORT,

E(ABORT except Q) = E(Q)

(e) *Interrupt Composition* :

For P≠SKIP,

E(P interrupt Q) $\underline{\Delta}$ if (p(P) or p(Q)) then

when (r(P) or r(Q)) do

if r(Q) then E(Q;(P interrupt Q))

else $\bigcup_{e \in rfs(P)}$ <e>E((P after <e>) interrupt Q)

else {<>}

and

E(SKIP interrupt Q) $\underline{\Delta}$ if r(Q) then E(Q;(SKIP interrupt Q))

else E(SKIP)

If interruption occurs then Q executes, and if Q terminates successfully, P interrupt Q continues.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For simplification of ABORT,

$$E(\text{ABORT } \textbf{interrupt } Q) = E(*Q;\text{ABORT})$$

2.7. Priority

Consider the set of exception handlers, E_1, E_2, ..., E_n, and the set of interrupt handlers, I_1, I_2, ..., I_n, and a process P.

P **except** $[E_1 \,\square\, \ldots \,\square\, E_n]$ and P **interrupt** $[I_1 \,\square\, \ldots \,\square\, I_n]$

define processes which execute P until preempted by any E_i (respectively I_i). The handlers all have the same priority. Only one handler may preempt P at any given moment of time.

P **except** E_1 ... **except** E_n and P **interrupt** I_1 ... **interrupt** I_n

introduces a priority ordering to the handlers increasing from 1 to n. At any instant, P or E_i (I_i) may be preempted by E_j (I_j) where $1 \leq i < j \leq n$. Any parenthesized grouping of the above does not alter the priority ordering.

Priorities for handlers may be determined during execution. Consider for example,

$[b_{11} \wedge g_1 \rightarrow cl_1 \,\square\, b_{12} \wedge g_2 \rightarrow cl_2 \,\square\, \ldots]$
except $b_{21} \wedge g_1 \rightarrow cl_1 \,\square\, b_{22} \wedge g_2 \rightarrow cl_2 \,\square\, \ldots]$
except ...
]

The additional booleans, b_{ij}, may be restricted such that if $\exists i,j$: b_{ij}=true then $\forall k \neq i$ b_{kj}=false, by initialization prior to execution of the command, to provide dynamic handler priority.

It was stated in section 2.2 that the nondeterministic nature of guard evaluation prevents any assignment of a priority ordering to alternative communication in CSP. Justification of that statement is now given.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consider firstly a solution for ordering boolean guard evaluation in, for example, an alternative command. The guards, $boolean_i$, are attempted in order 1 to n by iteration. Should attempt=n and done=false, then the process aborts.

```
done:=false; attempt:=0;
*[¬done→
   [attempt<n→ attempt:=attempt+1];
   *[attempt=1∧boolean1→ P1; done:=true
    [] ...
    []attempt=n∧booleann→ Pn; done:=true
 ] ]
```

However, if an input/output command occurs in the set of guards, a similar solution does not satisfy the requirements, as is illustrated by

```
done:=false; attempt:=0;
*[¬done→
   [attempt<n→ attempt:=attempt+1];
   [attempt=1; ?channel1→ P1; done:=true
   [] ...
   []attempt=n; ?channeln→ Pn; done:=true
 ] ]
```

Iterative selection reaches the first input guard, $?channel_1$, and the operational semantics of CSP stipulate that a wait for communication must occur should the partner not be ready. Hence, no other alternatives are attempted. We know of no method to solve the problem in CSP as defined in [Hoare78]. Attempts seem always to be foiled by the nondeterminism of guard evaluation.

2.8. Restricted Preemption

In section 2.3, the initialization of variables in preemptive guards was mentioned with reference to "I; P **interrupt** Q". Implicit to such initialization is scope of variables. That is, such variables must be in the scope of I, P and Q. This further explains the suitability of the preemptive

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

commands to a multiplexed implementation in that should I, P and Q not be resident in the same local memory as such variables, then access to the variables would not be possible without further communication. The problems of mutual exclusion in such cases are well known.

Critical sections may be modelled using restricted preemption. Notice that Hoare's CSP does not permit shared variables. A process which inputs a new value and outputs the current value may be defined to provide a shared variable facility.

The scope of restricted preemption is given by the bracketing symbols "⟦" and "⟧" and is defined by

⟦P⟧ **except** Q $\underline{\Delta}$ P U Q

```
E(⟦P⟧ except Q) Δ if (p(P) or p(Q)) then
                    when (r(P) or r(Q)) do
                       if r(Q) then E(Q)
                       else E(P)
                 else {<>}
```

and

⟦P⟧ **interrupt** Q $\underline{\Delta}$ P U (Q;P) U (Q;Q;P) U ...

```
E(⟦P⟧ interrupt Q) Δ if (p(P) or p(Q)) then
                       when (r(P) or r(Q)) do
                          if r(Q) then E(Q;(⟦P⟧ interrupt Q))
                          else E(P)
                    else {<>}
```

In both cases Q may only preempt the set of first events of P. In general, restricted preemption is used to reverse the priority of P and Q after P has commenced execution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The application of restricted preemption is well suited to modelling hardware when a process performs several sequential events, none of which can be interrupted.

For software processes, the reversal of priority may for example be used to enable P to meet critical time constraints once it has started to execute. Also, restricted preemptive commands may model an operating system which frequently disables processor interrupts while modifying system tables.

Critical sections may be modelled for sequential compositions containing restricted preemption by defining

(P;[[Q]];R) **except** S $\underline{\Delta}$ (P **except** S) U (P;([[Q]] **except** S) U (P;Q;(R **except** S))

and

(P;[[Q]];R) **interrupt** S $\underline{\Delta}$ (P **interrupt** S);([[Q]] **interrupt** S);(R **interrupt** S)

Another application is exemplified by the following nested restricted exception commands.

$[[g_1 \rightarrow P_1]]$ **except** $[[g_2 \rightarrow P_2]]$ ... **except** $[[g_n \rightarrow P_n]]$

If more than one guard is ready to execute then $(g_i \rightarrow P_i)$ will proceed where i is the maximum subscript of those ready guards. Hence $(g_1 \rightarrow P_1)$ becomes the default action.

2.9. Some Examples

The application of preemption to catch typical programming errors such as floating point overflow requires system support to raise predefined exceptions, like NUMERICAL_ERROR in Ada [Ada82], and could be handled by a mechanism similar to

P **except** [System?NumericalError→ (recover or propagate)].

Leaving the issue of error handling aside, we illustrate the use of the preemptive commands in the following examples. Not all the processes are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

coded; omitted code is self-evident.

2.9.1. Asynchronous Buffering

With communication in CSP being synchronous, typically asynchronous buffering between a producer and a consumer process is achieved by the introduction of another process which implements a bounded buffer. An alternative scheme is now presented.

Consider the following simplified producer/consumer situation where Consumer will consume any items in the buffer but otherwise does not wish to wait. C, takes items from a circular buffer until it is empty:

```
C = *[c<b→ consume(buf(c mod Nbuf)); c:=c+1].
```

B inputs into the buffer:

```
B = [b-c<Nbuf; Producer?(buf(b mod Nbuf))→ b:=b+1].
```

We can now define a consumer process:

```
[Consumer :: buf: (0..Nbuf-1)item; c, b: integer; b:=1; c:=1;
             C interrupt B
]
```

The reader will notice that problems of mutual exclusion to shared variables have been carefully avoided.

Assuming that Producer is ready and waiting to send an item when the interrupt construct is reached in Consumer (for which it must wait), then B will perform asynchronous buffering for C. However, communication between Producer and Consumer is synchronized and can only occur when both are ready. The example can of course be extended to perform the more usual producer/consumer function of repetition until mutual agreement for termination occurs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.9.2. Scheduling

This example illustrates a small operating system for which S schedules processes P_1, ..., P_n. There is no need for all P_i to be identical and more than one may be scheduled to execute concurrently. To facilitate scheduling, each P_i runs under the control of a virtual process, V(i). The system looks like,

```
[(||i:1..n) V(i) || S || Server || Clock],*
```

where Server supplies a remote service for P_i's and Clock is a hardware process. V(i) and S are defined as follows:

```
V(i) = [S?go; P_i interrupt [S?pause→ S?go]; S!terminate
       ] except [S?abort→ skip]

S = [table:(1..n)state; more:boolean; more:=true;
     *[more→
        [] (□i:1..n) V(i)?terminate→ table(i):=DONE;
        [] Server?abort(j)→ V(j)!abort; table(j):=DONE;
        ] except [Clock?tick→ skip];
        reschedule
      ]; Server!done
    ]
```

where reschedule executes V(i)!pause and V(j)!go for some i and j or assigns false to more if there are no P_i left unterminated. Server is like the remote procedure call model of section 2.1.1. Server notifies S if a P_i issues an illegal request and receives an exception message from S to terminate when all have terminated. Notice that the clock exception is inhibited if any other scheduler action is in progress when the hardware signal (tick) occurs.

---

* We use the notation (||i:m..n) and (□i:m..n) for n-m+1 parallel and alternative like processes, respectively.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.10. In Summary

In this chapter, we have proposed preemption for CSP. An informal introduction was given showing the syntax of the **except** and **interrupt** commands and outlining requirements for the execution of preemption. Modifications to Hoare's purely set theoretic process model were then presented with definitions for the extensions. We also indicated the changes that were necessary because distributed termination was no longer supported.

A model of process execution which captured the environment of a process, both internal and external, was given. Importantly, no notion of a global environment was necessary for consideration of a sub-process in a composition. The channel ready set was defined to capture the external environment of a process in such a way that the channels may be regarded as arbiters.

An operational semantics of CSP, including the extensions for preemption, were defined by the execution function. The progress of a process through execution steps was represented as were possible delays for communication and possible deadlock.

A restricted form of preemption was then proposed. Finally, examples were given of the modelling of critical sections, asynchronous buffering and process scheduling.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 3

Implementation of Preemptive Guards

Having proposed preemptive commands for CSP in the previous chapter, we now take a closer look at preemptive guards from a practical viewpoint, with particular reference to implementation. Although we intend to use the preemptive commands in a functional model of a UNIX kernel, we feel that no proposal should be forwarded for a language which is impractical to implement.

We maintain that there exists at least one class of applications for which the preemptive commands are well suited. Obviously, such applications benefit from the ability to preempt one process in favour of another and are real-time in nature.

Current processor hardware typically supports traps and device interrupts. Handlers for such preemptive mechanisms are implicitly multiplexed and often must satisfy real-time constraints required of the operating system that they are used to implement. The operational semantics of the preemptive commands also stipulate real-time servicing within the granularity of events. Similarly, a multiplexed implementation is suitable and will be assumed hereafter.

In this chapter, we give an implementation scheme for preemptive guards where both the preemptible command and the preemptive command are multiplexed. We follow the same approach as that taken in [Dix83c].

Section 3.1 gives the alternatives that have been proposed by others regarding the implementation of input/output guards in CSP. By way of comparison, a potentially real-time problem that has appeared in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

literature which involves a re-writable single buffer is examined and a solution is given using preemptive guards (sections 3.2 and 3.3).

We then present implementation details — firstly, a scheme for synchronization without preemption in section 3.4, then, the complete scheme in section 3.5. The remainder of the chapter examines the control message overheads and makes comparisons with related work.

3.1. Input/Output Commands in Guards

In the original proposal for CSP, Hoare stipulated that an input command could appear as a non-guard or as an *input guard*, with a matching output command occurring as a non-guard, as is illustrated by

[A:: [B?x→ ... □ ...]] ‖ [B:: C?y; ...; A!x; ...] ‖ [C:: ...; B!y; ...].

An implementation of this scheme is simple. The sending process may be regarded as the *master*. The receiving, *slave*, process waits for a request from a matching process. The sender waits for acknowledgement of the request, after which the communication proceeds.

Hoare gave reasons for the possible inclusion of *output guards* with matching input non-guards in CSP, not the least being symmetry. Hence the following example of i/o guards would be supported:

[A:: ...; [B?x→ ... □ C!y→ ...]] ‖ [B:: ...; A!x; ...] ‖ [C:: A?y; ...].

Silberschatz gave an implementation of i/o guards which required the declaration of a master-slave relationship between processes, so that a strict ordering among processes at the i/o command level is obtained [Silberschatz79]. Interestingly, Levin and Gries [LevinG81] have commented that the inclusion of output guards made easier the development of their proof scheme for CSP. We accept that i/o commands in guards are desirable and (perhaps with the introduction of some restrictions) are easily implemented.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Generalized i/o guards, where an input guard matches an output guard, were proposed by Bernstein [Bernstein80], as in

[A:: ...; [B?x→ ... □ ...]] ‖ [B:: ...; [A!x→ ... □ ...]].

Bernstein gave an implementation based on i/o guards probing for the state of their communication partners. Buckley and Silberschatz [Buckley83] have recently presented a more comprehensive implementation scheme for generalized i/o guards.

In sections 3.3 and 3.7, we will examine the use of the proposed preemptive commands as an alternative to generalized i/o guards.

3.2. A Problem of Contention

The following example is a simplified version of that in [Kieburtz79] and forms part of Bernstein's justification for the support of generalized i/o guards in CSP. We comment on the proposed solution using generalized i/o guards and, in section 3.3, present an alternative based on preemption.

The problem involves two processes which control the position of a point on a screen. Update computes a new position and communicates it to Display. Three desirable conditions are placed on the processes.

(1) Update should be able to compute a new value, rather than waiting to communicate an old one.

(2) Display should only refresh the screen with the most recent value computed.

(3) Communications are to be kept to a minimum.

The solution which Bernstein proposes involves the introduction of a buffering process.

```
Update::  *[true→ compute(x,y) □ Buffer!(x,y)→ skip]
Buffer::  *[Update?(x,y)→ B:=true □ B;Display!(x,y)→ B:=false]
Display:: *[true→ refresh(x,y) □ Buffer?(x,y)→ skip]
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The boolean latch, B, in Buffer is introduced to prevent unnecessary communication of the old point value to Display.

The implementation of generalized i/o guards given by Bernstein is based on the requirement that each i/o command sends a *probe* asking the communication partner if synchronization can occur now. A reply of YES, NO or BUSY will be received and must be waited for. For P_i probing to establish communication with P_j,

- P_j replies YES, if it is waiting at a matching i/o command.
- P_j replies NO, if it is not at a matching i/o command.
- If P_j is at an alternative construct with an enabled and matching i/o command but P_j is itself probing, then P_j sends BUSY if $j<i$ or delays replying if $j>i$ until its reply is received. This protocol is necessary to avoid a perverse type of busy deadlock.

No information about the state of a neighbour is stored except in the latter case. Probes simply determine whether communication is possible at a particular instant of time.

Buckley and Silberschatz criticize Bernstein's imprecise semantics of process behaviour after receiving a reply, and give a complete algorithm of their own. However, the important point in both algorithms is:

> If a NO message is returned, the communication partner is deemed to be responsible for making a similar probe to synchronize communication.

Excluding coincidental rendezvous where communication partners are simultaneously probing each other, Buffer will receive a NO reply for probes to both Update and Display, and will wait to be probed subsequently.

Notice that if Buffer was omitted, communication would rely on coincidental rendezvous between Update and Display which would rarely occur since the true guards of Update and Display imply that neither should wait for communication if i/o guards are not ready.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The most serious problem with the above solution and for which Bernstein criticizes CSP, relates to the non-determinism of guard evaluation and the absence of a priority function associated with i/o guards. In Display, only fairness considerations ensure that input will be received from Buffer sometime after the latch becomes true. However, this may not occur immediately. That is, the old point may be displayed for a series of repetitions when a new value is available.

While not wishing to quote in their entirety the rejected alternative solutions given by Bernstein (for which the reader should refer to [Bernstein80]), it is interesting to note that the following solution, which does not use generalized i/o guards and satisfies the desirable conditions (1) and (2) above, is not considered.

```
Update:: *[compute(x,y); Buffer!(x,y)]
Buffer:: *[Update?(x,y)→ skip □ Display!(x,y)→ skip]
Display:: *[Buffer?(x,y); refresh(x,y)]
```

A problem of contention for the communications facility exists in this solution if either computing the new point or refreshing the screen is fast. The network, in a distributed implementation, could become saturated by the requests to Buffer from the faster process. Similarly, the control messages associated with probes that are required to support generalized i/o guards in Bernstein's solution, are an additional burden on the network. Even if the latch in Buffer is false, two control messages are still required to establish that Buffer does not want to communicate with Display. The scheme to be presented in section 3.5, implements the above solution without probes and with only 2 control messages for any synchronization.

3.3. Solution by Preemption

We apply the **interrupt** command to Bernstein's problem yielding the following solution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Update::  *[compute(x,y); Display!(x,y)]
Display:: *[refresh(x,y)] interrupt [Update?(x,y)→ skip]
```

The **interrupt** command enables the establishment of a relative priority between commands. The receiving of a new point is given priority over refreshing the screen. Notice that the inclusion of an alternative with a boolean guard which is always true is no longer necessary. Generalized i/o guards are unnecessary. The use of an i/o guard and a matching non-guard i/o command is sufficient.

The above solution assumes a faster relative speed of refreshing compared with a computation of a new point. In the implementation to be presented later, all i/o guards, including preemptive guards, are passive and do not instigate synchronization. Consequently, Display will refresh with the old point until Update communicates a new point. If the relative speeds were reversed, then interruption would be applied to Update using a preemptive output guard.

```
Update::  *[compute(x,y)] interrupt [Display!(x,y)→ skip]
Display:: *[refresh(x,y); Update?(x,y)]
```

The above assumption may in fact be drawn from Bernstein's solution in which we notice the following. The latch in Buffer is introduced to prevent unnecessary communication by Display. However, if Update was faster per repetition than Display, there would be no need for such a latch. Most of the time a new point would be awaiting Display in Buffer.

Under certain conditions, it may be desirable to apply the **except** command. Firstly, if the refresh time is known to be quite small compared with the time to perform synchronized communication, and secondly, a slight delay in Update is tolerable, possibly for the duration of the refresh, then the following Display may be suitable:

```
Display:: *[skip except [Update?(x,y)→ skip]; refresh(x,y)]
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

There is nondeterminism as regards the availability of a new point from Update. However, if a point is available, the except command ensures that input will occur. There is no nondeterminism associated with the guard selection as was the case for Bernstein's Display. In fact, this expresses precisely the required relationship between the alternatives of Bernstein's Display.

3.4. Another Implementation of I/O Commands

We give an implementation of i/o commands which is simple and requires a small number of control messages. The scheme will be expanded later to allow for preemption. We point out that the algorithm for synchronization omits consideration of purely boolean guards.

The basis of the scheme is that *non-guard* i/o commands are responsible for *requesting* communication. I/o guards do not actively seek (by sending requests to partners) to communicate. In this way non-guard i/o commands play the role of masters of matching i/o guards. A general relationship between computation states of processes, as expressed by i/o guards, may be used rather than that which Silberschatz suggested. In the case of matching non-guard i/o commands, only one of the processes can become the master.

3.4.1. Control Messages

Two processes, P_i and P_j, have matching i/o commands for channel c if, for example, they contain P_j!c(x) and P_i?c(y) respectively. The protocol to establish synchronized communication between matching i/o commands uses two control messages.

NOTIFY(i,j,c): Process P_i sends a NOTIFY to process P_j indicating that P_i is about to wait at a non-guard i/o command for channel c. The direction of transfer could also be included in c if checking is required.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

COMMIT(i,j,c): P_i has received a NOTIFY from P_j for channel c; P_i has now reached a matching i/o command (guard or non-guard) and now commits itself to the transfer.

We assume control messages

(1) cannot overtake one another,

(2) arrive reliably and without error, and

(3) can only be received when the addressed process is willing to accept them.

The last assumption precludes two processes which contain matching non-guard i/o commands from simultaneously sending a NOTIFY to each other. For a locally distributed system which employs a single path, bus-type communication, for example ethernet, the medium itself may assist in enforcing this assumption. Simultaneous mutual notifications are due to delays in the transport mechanism in a multiple path network or configurations where message hops are necessary. Such an occurrence can be detected by the arrival of a notify control message when an acknowledge control message is expected and can be resolved, for example, by always allowing the sending process (containing the output command) to become the master, thus ignoring the receiver's notification. In general however, the lower level protocol supporting control messages will be required to enforce the assumptions on such a network.

3.4.2. States

We define the following states for individual processes:

E: Normal execution, not attempting to perform an i/o command.

N: Reached a non-guard i/o command and must become slave if a suitable NOTIFY already has been received from a matching i/o partner, or become the master.

W: Reached a non-guard i/o command, NOTIFY sent and has not received COMMIT.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

G: Reached i/o guard and either has not received a suitable NOTIFY or has received one and has not yet sent COMMIT.

C: Committed to transfer.

3.4.3. Scheme

Table 3.1 presents the algorithm for process P_i synchronizing with process P_j for channel c. The table gives the current state, including conditions on variables, the sent or received message (annotated with ! or ?), and the subsequent action that must occur to result in the next state. The action is sending a message or recording a received message, or is null (represented by ø). The braces represent an ordered set of alternative current states and corresponding next states. The set *enabled* contains the (process,channel) pairs of enabled i/o guards which is established during execution.[1] The pair (j,c) is used for the awaited notification and (m,n) for an unawaited notification.

Table 3.1. Synchronization for i/o commands, with no preemption.

Current State	Message Sent/Received	Subsequent Action	Next State
Receipt of unawaited NOTIFY:			
E **W** **G** ∧ (m,n)∉enabled	? NOTIFY(m,i,n)	req[m]:=n	{**E** **W** **G**
Reach non-guard i/o command:			
N ∧ req[j]≠c	! NOTIFY(i,j,c)	ø	**W**
N ∧ req[j]=c	! COMMIT(i,j,c)	req[j]:=ø	**C**
W	? COMMIT(j,i,c)	ø	**C**
Reach i/o guard:			
G ∧ ¬∃(j,c)∈enabled:req[j]=c	? NOTIFY(j,i,c)	COMMIT(i,j,c)	**C**
G ∧ ∃(j,c)∈enabled:req[j]=c	! COMMIT(i,j,c)	req[j]:=ø	**C**

[1] Other issues concerning the implementation are ignored here, such as, nondeterministic ordering of guard evaluation and presence of true guards which do not involve i/o commands. Also, an efficient implementation need not firstly calculate the enabled set.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

With regard to the use of the request set, consider for example, the state E, where a NOTIFY can be received from any communication partner m for channel n. This causes "req[m]:=n" to be performed, recording the notification, before continuing in state E. Assignment of ∅ to an element of *req* cancels this information.

Consider communication between partners P_i and P_j. With respect to elapsed time, four interleavings of states of P_i and P_j are possible —

(1) and (2) P_i reaches non-guard i/o command before/after P_j reaches non-guard i/o command,

(3) P_i reaches non-guard i/o command before P_j reaches i/o guard, and

(4) P_i reaches non-guard i/o command after P_j reaches i/o guard.

Cases 1 and 2 are effectively the same. Figure 3.1(a) illustrates 1 and 3 (with **N** and **G** respectively), showing the interleavings of states and the control messages associated with the synchronization protocol, and figure 3.1(b) illustrates 4. The NOTIFY of figure 3.1(a) is unawaited by P_j. In the case of pure synchronization, no message transfer is necessary. The control message protocol performs the required task.

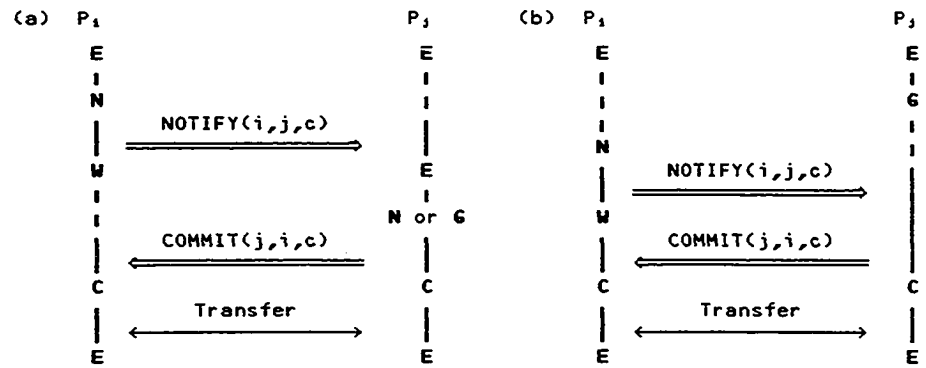


Figure 3.1. Scheme for CSP synchronized i/o commands, with no preemption.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Deadlock occurs if

(1) a process in state W never receives a COMMIT, or

(2) a process in state G never receives a NOTIFY.

One particular situation can be detected where state N is entered and req[j]=h such that h≠∅ and h≠c. Deadlock will occur between P_i and P_j as they will wait for each other to commit on different channels if NOTIFY(i,j,c) is sent. When a NOTIFY is received, the control message mechanism can check that the request is valid and hence detect a possible deadlock situation.

Notice that an interrupt mechanism is assumed to exist whereby P_j can receive notification from P_i. The schemes of Bernstein and of Buckley and Silberschatz require a similar mechanism to query the state of a process.

3.5. Implementation of Preemption

We now extend the previous scheme for preemption.

3.5.1. Handshaking

A handshaking protocol is necessary for preemptible processes and the following additional control signals are introduced for that purpose.

REQUEST(i,j,c): P_i sends REQUEST to P_j indicating that P_i is about to wait at a non-guard i/o command for channel c and that P_i *may be preempted.*

SHAKE(i,j,c): P_i has received REQUEST from P_j for channel c and has reached a matching i/o command (guard or non-guard). P_i sends SHAKE asking if P_j can commit to the transfer or if P_j has been preempted and wishes to cancel the request.

CANCEL(i,j,c): P_i has received SHAKE from P_j for channel c but has been preempted. P_i sends CANCEL indicating that the request is no longer current.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We also introduce one more state associated with handshaking for preemption.

S: SHAKE has been sent to a preemptible process and has not received COMMIT or CANCEL.

The handshake protocol for P_i synchronizing with P_j for channel c is:

send REQUEST(i,j,c), receive SHAKE(j,i,c) and then send COMMIT(i,j,c).

3.5.2. Algorithm

The algorithm for synchronization is given in Table 3.2, again for P_i synchronizing with P_j for channel c, but with a possible preemption by P_p on channel q. The term *request* is taken to include both NOTIFY and REQUEST control messages. We extend the information stored in *req* to reflect whether handshaking is required or not. Each process has a set *preempt* which contains the enabled preemptive guard (process,channel) pairs and is empty when the process is not preemptible. The pair (p,q) is used to represent a preemptive request.

In state S_x only COMMIT or CANCEL will be received to complete the handshaking protocol. We wish to synchronize quickly and so we assume that the response will be received in the minimum time that the transport mechanism permits. If P_i is not completing a preemptive handshake, (y,z)=(j,c), otherwise (y,z)=(p,q). For efficiency, the receipt of a CANCEL causes the process to resume the previous state X.

An *outstanding request* occurs when, for example, P_i has sent REQUEST(i,j,c) and is then preempted. Only if the requested process, P_j, sends SHAKE(j,i,c) does P_i send CANCEL(i,j,c). In this case, 3 control messages have been wasted but a preemptive transfer has occurred. Notice that a subsequent request by P_i sent to P_j cancels a previous outstanding request by overwriting the element in *req*. A process can only have as many outstanding requests as it has communication partners.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 3.2. Synchronization of i/o commands, with preemption.

Current State	Message Sent/Received	Subsequent Action	Next State
Receipt of unawaited non-preemptive request:			
{E, W, G} ∧ (m,n)∉preempt ∧ (m,n)∈enabled	? NOTIFY(m,i,n)	req[m]:=(n,C)	{E, W, G}
{E, W, G} ∧ (m,n)∉preempt ∧ (m,n)∈enabled	? REQUEST(m,i,n)	req[m]:=(n,S)	{E, W, G}
Reach non-guard i/o command, no request received:			
N ∧ preempt=∅ ∧ ¬∃x:req[j]=(c,x)	! NOTIFY(i,j,c)	∅	W
N ∧ preempt≠∅ ∧ ¬∃x:req[j]=(c,x)	! REQUEST(i,j,c)	∅	W
W	? COMMIT(j,i,c)	∅	C
W	? SHAKE(j,i,c)	COMMIT(i,j,c)	C
Reach non-guard i/o command, request received:			
N ∧ req[j]=(c,C)	! COMMIT(i,j,c)	req[j]:=∅	C
N ∧ req[j]=(c,S)	! SHAKE(i,j,c)	req[j]:=∅	S_W
Complete handshake with (j,c) or (p,q):			
S_X where X∈{E,N,W,G}	? COMMIT(y,i,z)	∅	C
S_X where X∈{E,N,W,G}	? CANCEL(y,i,z)	∅	X
Reach i/o guard, no request received, await request:			
G ∧ ¬∃(j,c)∈enabled:req[j]=(c,C)	? NOTIFY(j,i,c)	COMMIT(i,j,c)	C
G ∧ ¬∃(j,c)∈enabled:req[j]=(c,S)	? REQUEST(j,i,c)	SHAKE(i,j,c)	C
Reach i/o guard, request received:			
G ∧ ∃(j,c)∈enabled:req[j]=(c,C)	! COMMIT(i,j,c)	req[j]:=∅	C
G ∧ ∃(j,c)∈enabled:req[j]=(c,S)	! SHAKE(i,j,c)	req[j]:=∅	S_G
Reach preemptive command, preemptive request received:			
E ∧ ∃(p,q)∈preempt:req[p]=(q,C)	! COMMIT(i,p,q)	req[p]:=∅	C
E ∧ ∃(p,q)∈preempt:req[p]=(q,S)	! SHAKE(i,p,q)	req[p]:=∅	S_E
Within preemptive command, receipt of preemptive request:			
{E, W, G} ∧ (p,q)∈preempt	? NOTIFY(p,i,q)	COMMIT(i,p,q)	C
{E, W, G} ∧ (p,q)∈preempt	? REQUEST(p,i,q)	SHAKE(i,p,q)	{S_E, S_W, S_G}
Receipt of handshake for outstanding request:			
{E, W ∧ (m,n)≠(i,j), G}	? SHAKE(m,i,n)	CANCEL(i,m,n)	{E, W, G}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.6. Some Implementation Details

3.6.1. Once Only Boolean Evaluation

In the informal introduction to the preemptive commands in chapter 2, we specified that, for P **except** Q or P **interrupt** Q,

> P cannot make assignments to variables contained in boolean preemptive guards of Q.

It is obvious that this restriction frees the implementation of the need to constantly re-evaluate any boolean part of a preemptive guard (preemptive boolean). As such the restriction is introduced for pragmatic reasons. We also stated that

> immediately prior to the execution of P, the preemptive guards of Q are examined and Q is executed if any are true.

We now bring to notice the fact that an implementation need only evaluate any preemptive booleans once on entering a preemptive command. A compiler could allow a process to make assignments to such variables but we consider that would be a bad practice. If restricted preemption is supported, thus providing a mechanism to disable preemption, then after the execution of a critical section, the preemptive booleans may be re-evaluated routinely.

3.6.2. Nested Preemption and Resumption

Preemptive commands may be nested. Hence, when a preemption occurs, the current and inner level preemptive pairs must be removed from the preempt set before continuing with execution. In addition, the type of preemption must be noted, making resumption of the preempted command possible after successful completion of preemption. Such resumption requires the return to state E and again routine re-evaluation of preemptive guards is necessary.

One further complication may also occur in that it may be necessary for another notification to be sent if resumption is to a non-guard i/o command.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Consider the following:

P:: ...; [Q!X(x); ...] **interrupt** [Q!Y(y)→ Q!Z(z)]

Q:: ...; P!Y(y); ...; P?Z(z); [P?X(x)→ ...]

The request sent by P for channel X may well have been overwritten by a request for channel Z (here it is timing dependent, but need not be). Resumption to state E must check if another request must be sent. Notice that if P had not sent a request to Q for channel Z, the request would still be outstanding and current on resumption of P. It is possible to avoid such a complication but is at the cost of efficiency.

3.6.3. Bounds on Requests

With regard to the maximum number of requests that need to be saved in *req* at any one time, N-1 serves as a bound, where there are N processes in the system. However, if the compiler ensures that an addressed process contains a matching i/o command, then the size of *req* for P_i is bounded by the number of communication partners, N_i, which may well be small.

3.6.4. Control Message Overheads

We now seek bounds on the number of control messages that must passed for synchronization to occur.

The minimum number of control messages that are necessary for synchronization between P_i with P_j, where neither is preemptible, is 2 — NOTIFY and COMMIT. The minimum number where either may be preempted is 3 — REQUEST, SHAKE and COMMIT.

Now consider the worst case of P_i, P_j and P_k, all of which may be preempted, where P_i is trying to synchronize with P_j for channel c and P_j decides to synchronize with P_i, but P_k sends a preemptive request to P_i for channel d. This situation requires 6 control messages. Firstly REQUEST(i,j,c), REQUEST(k,i,d), SHAKE(i,k,d) and COMMIT(k,i,d) are sent. Then

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the transfer on channel d occurs. Finally SHAKE(j,i,c) and CANCEL(i,j,c) are sent. For longer chains of preemptive requests, where P_k gets preempted and so on, the maximum number of control messages exchanged among any three processes is still 6 per synchronization.

3.7. Comparing Generalized and Preemptive Guards

Generalized i/o guards have been strongly supported in the literature and we shall see that they provide a natural solution to some problems. In this section, we shall make some comparisons between applications of generalized i/o guards and preemptive guards.

The implementation given in [Bernstein80], was based on all processes probing communication partners for state information. A statically derived ordering of processes was applied to resolve the potentially cyclic problem that the probed process may itself be probing. Buckley and Silberschatz elaborated on this scheme and introduced additional control messages for retries in this case [Buckley83]. Firstly, we shall look at their proposed implementation.

The underlying principle is that all i/o commands seek to initiate communication. A process P_i reaching a non-guard sends *COMMIT*(i,j)[2] which is held until P_j can reply with *YES*(j,i). In this respect, their scheme is similar to that given in section 3.4.

For i/o guards however, three replies are possible for P_i sending *QUERY*(i,j).

(1) *YES*(j,i) — P_j is waiting at a matching i/o command (having previously failed in its attempt to synchronize with P_i or other processes in its enabled guards).

---

[2] We use italics to represent their control signals. Notice that their presentation does not explicitly show channel names.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(2) *NO*(j,i) — P_j is not prepared to synchronize.

(3) *BUSY*(j,i) — a non-committal response where j>i.[3]

In case (3), P_j will later send *RES*(i,j) to which P_i will send *RETRY*(i,j) and be assured of a *YES* or *NO* reply. Should P_j itself be probing, it can delay responding for j<i.

In the worst case for an alternative command in P_i, $3M_i+6_i$ control messages must be transferred to establish synchronization, where M_i is the number of processes named in the enabled guards of an alternative command and G_i is the number of processes with a static priority order less than P_i.

We shall compare the above scheme with that presented in this chapter in two situations which represent extremes for either scheme. The first has

- a non-guard i/o command matching an i/o guard, and
- no preemption.

The second requires different solutions each making full use of

- generalized i/o guards, or
- preemption.

For the first situation, consider

[A::...; B!x] ‖ [B:: ...; [A?x→ ...]].

Both schemes require the same number of control messages if the non-guard is reached first; that is 2. For the generalized scheme, if the alternative is reached first, then at best 4 control messages will be required — *QUERY*, *NO*, *COMMIT* and *YES*. Notice that normally there would be other alternative i/o guards which would also be tried.

In the second situation, we consider an example that illustrates perhaps the most convenient use for generalized i/o guards. The situation involves a

[3] This uses the same static ordering as in [Bernstein81].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

set of N_s server processes that are available at the request of a set of N_v client processes:

```
Client = ...; [(□i:1..Ns) Server[i]?Req→ s:=i]; ...
Server = *[[(□j:1..Nv) Client[j]!Req→ u:=j]; ...]
```

After synchronization, Client[u] has the services of Server[s]. At most $3N_v+G_i$, control messages will be required to synchronize a client and a server process, where $G_i<N_s$. Notice that this generalized solution is very concise (which is the basis for its choice).

Application of preemptive commands requires a different approach:

```
Client = ...; *[¬granted→ choose next s; Server[s]?Req(granted)]; ...
Server = *[[(□j:1..Nv) Client[j]!Req(true)→ u:=j];
          [...] interrupt [(□j:1..Nv) Client[j]!Req(false)→ skip]]
```

In order to make a comparison we will assume that $N_v>N_s$ and that one server will always be found to be free if all servers are polled through requesting service. For each synchronized communication between a client and a server, 2 control messages are required plus the data transfer, which we shall include as a message. A maximum of $3N_s$ messages are required for a client to acquire a server if all servers are polled under the same assumptions as above.

Hence, the number of messages required for both solutions are of the same order. The application of interruption makes explicit the probes that are used in the generalized i/o guard solution. However, from the programmer's point of view, the generalized i/o guard solution may be considered to be preferable.

3.8. Related Work

With regard to implementation of CSP, the work of Silberschatz in [Silberschatz79] is the most closely related to the scheme for synchronization presented in this chapter. As mentioned earlier, the master-slave relationship had to be established which governed which process was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

responsible for requesting communication. We take the view that i/o guards, including preemptive guards, are slaves. The only difference between such guards is that a preemptive guard is enabled over the whole of execution of a preemptible command until preemption occurs, whereas an i/o guard is enabled for the duration of guard execution.

Nicholson and Tsang [Nicholson83] describe a multiprocessor implementation of CSP, which is imbedded in Modula-2, using small Motorola 6809 processors connected by an Ethernet. We mention this implementation because it addresses the original scheme that Hoare proposed where a single process resides in each processor. No buffering of messages occurs and the Ethernet is used as an arbiter, so no priority is given to i/o guards (of which only input guards are supported).

We now review other related work that has taken or provides an alternative approach to preemption in CSP.

3.8.1. Buffering Output Commands

In their comments on CSP, Kieburtz and Silberschatz proposed that output messages be buffered [Kieburtz79], thus assisting in the avoidance of a cyclic form of deadlock. To assist in synchronization, additional primitives were also given. Roper and Barter [Roper81] report an implementation of a CSP derived language on a single processor which uses buffered message output. The consequence of such buffering (given buffer availability) is that output commands always succeed and are therefore not supported in guards.

In the model of the operating system, we want to show such buffering. Therefore, we do not consider buffered output commands as suitable.

3.8.2. Port Directed Communication

The reason usually cited for supporting port directed communication is to provide a synchronized service facility, via a port, without knowing user

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

process names. Silberschatz has given a scheme for CSP [Silberschatz81] which is based on his earlier scheme [Silberschatz79]. The owner of a port can have input or output guards using the port name, whereas users can not. The implementation required the non-guard i/o commands to signal a request to synchronize. CSP/80 [Jazayeri80], another single processor implementation of a CSP based language (which has C as a host language), uses such a scheme. Here, ports must be declared as guarded for them to be used in guards.

Extended naming conventions have been proposed in [Francez83] and could be applied as an alternative for port directed communication. Again referring to our intended application, we shall have process names for direct process referencing.

3.8.3. Synchronizing Resources

Andrews' language for distributed programming, SR [Andrews81a], supports some facilities that CSP with preemption can. A resource — processes and data — must reside on a single machine (which it can share with other resources). Hence, the implementation implicitly supports multiplexing. Buffered output commands and remote procedure calls are supported. SR has input commands as guards within an input construct of the form "in ... ni", and also can have priority as a function of message content. This latter facility may be used to advantage to schedule buffered output. To provide a similar facility in CSP with preemption, we must have an interrupt handler which inserts the message in the appropriate position in an ordered queue.

In [Andrews81c], an example of a job manager, or dispatcher, is presented. The example has some similarity to our scheduler example of chapter 2 which was developed independently at around the same time. Moreover, the suspend and activate procedures that are defined in [Andrews81b] could be used for scheduling. However, these facilities were not designed for general exception handling. A process that may be suspended in SR has no control over such suspension, unlike a preemptible process in CSP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.8.4. Otherwise as an Alternative

Schneider defines a guard **otherwise** which is always enabled [Schneider82]. For example, should g_1 not be enabled and ready in

$[g_1 \rightarrow cl_1 \ \square$ **otherwise** $\rightarrow cl_2]$,

cl_2 is executed as default. This is not as general a construct as preemption and has the effect of

$[\![true \rightarrow cl_2]\!]$ **except** $[g_1 \rightarrow cl_1]$.

Notice that with the less restrictive constraints given for the application of **except** at the end of section 3.3, the following would serve as a suitable solution for Display in Bernstein's problem:

*[Update?(x,y)→ skip □ **otherwise**→ refresh(x,y)].

For this example the construct does solve the problem introduced by nondeterminate ordering of guard evaluation. However, where more than one i/o guard is involved, we are forced to use a form of the construction

$[A?a \rightarrow cl_1 \ \square$ **otherwise** $\rightarrow$

$[B?b \rightarrow cl_2 \ \square$ **otherwise** $\rightarrow [A?a \rightarrow cl_1 \ \square \ B?b \rightarrow cl_2]]]$,

and must assume that the implementation will perform the first communication to become ready should the last alternative be reached.

3.9. Another Else

We take this opportunity to suggest that the following construct would introduce a more suitable form of priority for guards where no preemption is required:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[g_{11}→ cl_{11} ▯ g_{12}→ cl_{12} ▯ ...

else g_{21}→ cl_{21} ▯ g_{22}→ cl_{22} ▯ ...

⋮

]

where guards g_{1i} have priority over guards g_{2j} and so on. Should guard g_{1k} be executed, we can state that

$$\forall h<j\ \forall i:\ g_{hi} \text{ is } \begin{cases}\text{not enabled, or}\\ \text{enabled and not ready.}\end{cases}$$

Recently, Parnas presented a deterministic guarded alternative construct [Parnas83]

it g_1→ ... **else** g_2→ ... **ti.**

This construct does not include nondeterministic alternatives or provide for i/o commands in guards.

3.10. In Summary

There is at least one class of problems for which Hoare's original CSP is not suited. Such a class would include what we termed Bernstein's problem. More generally, we would include situations concerned with real-time, where avoidance of unnecessary delays between computations and minimizing contention on the communications network are of importance. The introduction of generalized guards does not provide a suitable solution. Typically, a priority ordering is required for the solution of such problems. The introduction of preemption for CSP serves this purpose. We observe that for hardware, support for interrupts and traps in processors has long provided this facility.

Notice that if guaranteed response times are required, we must establish worst case synchronization and subsequent communication times and use techniques such as those forwarded by Wirth [Wirth77b] to determine whether a program can meet its real-time requirements.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An implementation has been given which supports input and output commands in guards, including preemptive guards. In this implementation, multiplexing of the preemptive command seems to be quite a natural approach and may be executed efficiently. The uniform control message protocol for synchronized i/o places no additional burden on the underlying mechanism to support preemption. In fact, only where a process may be preempted and is itself requesting to synchronize, is any additional overhead incurred.

We have made some comparisons between generalized i/o guards and preemptive guards. In the worst case, the number of control messages associated with the application of preemption to provide a facility equivalent to generalized i/o guards is similar. However, if generalized i/o guards are not necessary in a particular application, a saving in control messages is gained since i/o guards do not actively seek a rendezvous by probing communication partners. The price one pays is in the restricted manner that matching i/o commands must be used.

We have also reviewed associated work with particular emphasis on implementations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 4

A Distribution of the UNIX Kernel

A local network provides potential for the exploitation of functionally distributed computing where different machines perform dedicated tasks like terminal handling and file management [Tanenbaum81a]. In this chapter, we are interested in what may be termed a very local network of loosely coupled machines, which might, for example, be in the same room. With such close proximity of machines, we can assume the existence of high bandwidth, directly connected paths between machines which need to communicate. No message re-routing is necessary.

We investigate a particular distribution of the UNIX kernel. The approach we wish to take is to distribute functions of the kernel over different machines to gain power through parallelism. We emphasize that this approach is quite different from that taken in the now quite common distributed operating systems which usually enable remote file access and sometimes remote procedure calls or process invocations between fully self-contained machines over a network. In our case, the kernel itself is partitioned and distributed over the network which as a whole constitutes the operating system.

Notice that the distribution of kernel functions is not new either; for example the CDC Cyber series have multiple strongly coupled CPU's and multiple Peripheral Processors. What is new however, is that we seek to take an existing operating system and distribute the kernel over a network such that no one machine contains a fully-fledged UNIX kernel. We also wish to use as much of the original code as is possible without compromising the efficiency

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of the distributed implementation.

Section 4.1 outlines the type of network under consideration and section 4.2 introduces the type of distribution that we consider will be suitable. We summarize the directions taken in related distributed systems in section 4.3 and then section 4.4 examines the support from existing systems for our distribution.

The behaviour of an extant version 7 (V7) UNIX system is investigated in section 4.5. In particular, the performance characteristics of that system under a day-to-day job mix are used to derive the typical loads that are generated by user processes (sections 4.6-4.9). Sections 4.10-4.13 present an account of the load that a distribution could be expected to support in the light of the previous analysis and draws on previous work regarding the availability of hardware suitable for the network.

4.1. Outline of Network

Figure 4.1 illustrates the type of network that is under consideration. Although the diagram shows two shared buses, only the following direct communication paths are necessary:

terminal machine ↔ all user machines

user machine ↔ all file machines

Top-end microcomputers or minicomputers are envisaged for the network. Up to 64 terminals might be supported by the terminal machine. The user machines must be homogeneous processors and must support memory management. The file machines are likely to be homogeneous and their architecture must support fast bursty i/o transfers to/from mass storage and the network.

For the communications network, we shall examine the suitability of a broadcast channel such as Ethernet [Metcalfe76, Ethernet80], which can support 10 Mbit/sec data transfers with a very short contention interval over a short cable, in preference to high bandwidth directly coupled buses.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

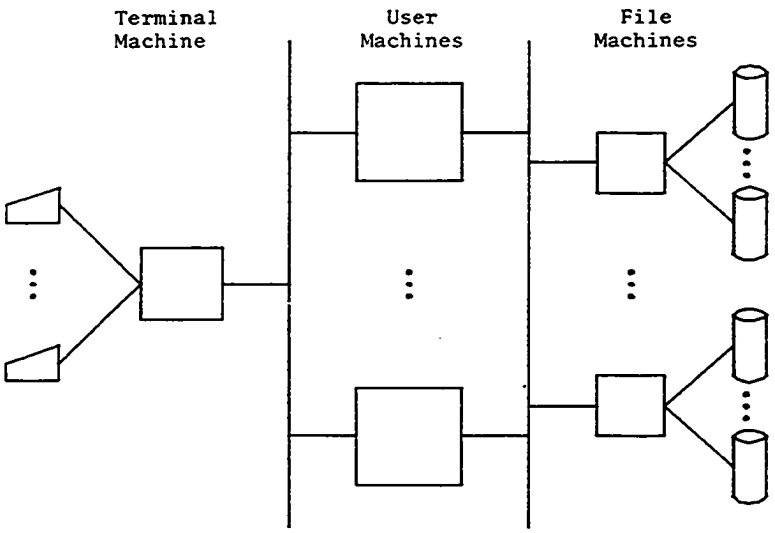


Figure 4.1. Network under consideration.

In some ways the thesis is also about cost effectiveness. We are not looking at providing power equivalent to a large mainframe which may employ intelligent channels or peripheral processors to handle a large number of interactive users and background processes. However, we are seeking to provide significantly more processing power than a single processor UNIX implementation and to support access to distributed file systems in a uniform and fast manner.

The network need only have one file machine in the minimal configuration. Part of this chapter is directed towards finding a balance for the user machine to file machine ratio, with the object being to have a sufficient number of user machines to almost saturate the file machines with file related requests.

4.2. Outline of Distribution

The approach to distribution is to off-load as much system code from machines which execute user processes as is possible while maintaining

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

efficiency in the implementation. A natural boundary for distribution exists in the UNIX kernel in the form of system calls. However, on the grounds of efficiency, not all these calls warrant distribution.

Referring to figure 4.1, the user machines execute user processes. Several users may be executing programs interactively on any one user machine to which their terminal has been logically connected. Process management is performed locally as are the process related system calls.

The terminal machine provides terminal read/write system calls, terminal device i/o control (ioctl) system calls and asynchronous character buffering. The file machines provide all file system related system calls and associated device control system calls, and asynchronous mass storage device handling.

To the user, the system appears as a normal single processor UNIX implementation. The fact that users are sharing the frontend terminal machine and the backend file machines is transparent. The intention is to provide a service in these machines which is comparable in speed to having the system call performed on the user machine but which will involve less processing on the user machine. The user machine is thereby free to execute another ready user process. The limiting factors in the servicing of requests are the message transport mechanism and mass storage access.

4.3. Directions in Related Networks

Early research with local networks took a practical approach by making extant operating systems co-operate to satisfy remote information requests. More recently, there has been a diversification in such research which includes:

1. *Remote file access and process invocation*:

Particular UNIX related systems include the Network UNIX System [Chesson75], Resource Sharing UNIX System [Holmgren78], SDS/NET [Antonelli80], COCANET [Rowe82] and the Newcastle Connection [Brownbridge82].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The usual approach taken in these systems is to add to the kernel a mechanism for addressing a remote file, either directly by extending the information given in the file name, or by catching the reference at the level of a directory entry which contains remote access particulars.

2. *Pools of processors and service facilities*:

Such systems range from the provision system call servers for UNIX based satellite processors [Lycklama78] [Barak80], and file servers like those for the diskless SUN workstations [Cheriton83a, Cheriton83b] and the S/F-UNIX Virtual Circuit Switch based system [Luderer81], through the more distributed Cambridge system model [Wilkes80] and the utilities of Medusa [Ousterhout80a, Ousterhout80b], to the very general facilities proposed for the Amoeba distributed operating system [Tanenbaum81b]. These last two references base their file systems on UNIX.

3. *Remote procedure calls*:

This class encompasses many of the issues in 1. but seeks to integrate mechanisms into a programming language for a distributed system [Nelson81] [Shrivastrava82].

4. *Reliability*:

Of particular interest is LOCUS [Popek81, Walker83] which supports UNIX and those items of class 1 above but is designed for reliability.

5. *Parallel task solution*:

Both STAROS [Jones79] and Medusa provide for sets of cooperating parallel processes, called task forces, on Cm* [Swan77a, Swan77b]. The results of the parallel solution of certain problems are presented in [Gehringer82].

We will not attempt to mention the plethora of other literature concerned with distributed operating systems. Brownbridge et al. give a good review of systems that may be considered as forerunners of their system. The provision of general file servers is discussed in [Birrell80]. Sturgis et al.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Sturgis80] present the issues of remote file systems.

Of these directions, the distribution presented in this thesis is most related to the second class. Functionally, a specific set of servers are involved for the UNIX terminal and file systems. We have already commented on the difference between distributed operating systems and the approach we wish to take. Our approach lies between provision of the remote access facilities of the first class and the general services facilities of the second. Message passing without language support is envisaged. Such messages generally will be buffered for a waiting process rather than causing the instantiation of a remote procedure.

Regarding the parallel solution of tasks, the duplication of kernel code in user machines and the distributed kernel code in other machines addresses this problem only for the kernel. We do not seek to provide additional parallelism for user task groups.

No additional consideration for reliability than exists in the standard UNIX kernel will be given. The system would be serviceable provided one terminal machine, one user machine and one file machine was available. Although we will not consider bootstrapping, we note that unserviceable machines could be detected and logically disconnected from the distributed system during bootstrap.

4.4. In Support of the Distribution

Some of the existing work with distributed and multiprocessor systems provides direct support for the proposed distribution and also provides some lessons that are applicable. We shall examine five different systems. Each provides performance bounds and/or lessons based on empirical experience which are applicable in our distribution and encouraging for success.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.1. Dual VAX

Gobal and Marsh [Gobal82] present their experiences with a dual VAX-11/780 system running a modified Berkeley 4.1BSD UNIX. The second processor was used in place of the synchronous backplane interconnect terminator and consequently memory was shared (on two memory controllers) with one copy of the kernel. One processor was deemed the master with respect to kernel execution.

The closely coupled dual VAX system provides an extreme throughput bound for our distribution. With a benchmark of parallel compilations and text formatting, which in our experience is slightly compute bound, the system showed a 90% speedup[1] (where 100% means doubled throughput).

For this system, contention for the master processor, and consequently the kernel, is the limiting factor in process execution. It gives us an upper performance bound in that we may consider the second processor to be dedicated to user process execution and the master processor to be dedicated to kernel tasks but which may execute user processes when otherwise idle. We may view the implementation as a form of synchronized message passing using shared memory. We will introduce message passing across the network which must contribute more overhead. Hence, we could never expect such an increase in throughput.

Of more importance is the fact that in order not to reach saturation with disk transfers, the disk buffer cache was increased from 100 blocks to 1000 blocks of 1024 bytes. McKusick et al. [McKusick83] have noted that the increase of the basic block size from 512 bytes in the previous versions to 1024 bytes in 4.1BSD UNIX *more than doubled* the file system performance. The buffer cache increase smooths out peak output demand and can lead to increased availability of frequently used blocks.

---

[1] We use percentage speedup = $100*(t_0-t_1)/t_1$, where t_0 is the time taken by a single processor and t_1 is the time taken by dual processors.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.2. Satellite Processors

Barak and Shapir report on a system using a PDP-11/45 UNIX host with one PDP-11/10 satellite processor [Barak80]. Their implementation introduces considerable overheads in that extra layers of software are involved at the host on which a proxy process executes to perform requests for the satellite processor.

A compute bound benchmark using both processors to execute user processes executed in 60% of the time that it would have taken on the host processor. For comparison with the dual VAX, this is a speedup of 67%. The authors state that tasks requiring many system calls are much slower but unfortunately they give no figures.

We can see that the overhead of messages over the network to the proxy user process has produced a considerable reduction in speedup. The approach gives us a lower bound for throughput increase. We are sure that, for maximum efficiency, the proxy user process approach should be avoided and that the host should be dedicated to servicing requests.

4.4.3. S/F UNIX

Luderer et al. describe a distributed UNIX system with a network based on a virtual circuit switch [Luderer81]. The system has user machines running an S-UNIX kernel which supports local terminals and a local file system (that is, normal UNIX) and gives access to remote file systems. The file server machines run an F-UNIX kernel with a server process (executing in kernel mode) per user/file machine circuit.

Benchmarks were performed to detect the time difference between local file and remote file usage. Throughput decreased by about 35%. The increase from 70 to 99 sec/script in the benchmark was entirely due to an increase from 42 to 69 sec/script in system time on the user machine. To explain this phenomenon, they state that data transmission can be overlapped with local

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

processing, however, the offloaded file service is compensated for by the communication load.

Closer examination of their figures reveals that the benchmark became network bound with i/o wait time increasing by 5%. Although their time division multiplexed switch has a 7 Mbit/sec aggregate data rate, we suspect that the across circuit rate is too slow. The other problem that they point out is data copying.

The authors stated that their system constitutes an advance in distributed file systems. They have measured the kernel level transfer rates at 125 kbyte/sec, and a small packet send and acknowledge time of 0.9 msec.

Luderer et al. chose not to offload all the file system from the user machines. In support of this decision they give the following reasons:

(a) Access to more than one file system is via a singularly rooted tree on each user machine.

(b) Bootstrapping is easier.

(c) There are potential efficiency gains for local load modules.

(d) Personal UNIX systems (down market versions of S-UNIX) should have the option of local files.

On these points we make the following comments with regard to our proposed distribution. Our system is designed to provide a single root; moreover, the root is physically the same for all user machines. Bootstrapping depends on particular processor hardware. Auto-load instructions or rom bootstraps are becoming commonplace, but certainly there are advantages in having local devices for bootstrapping during development. With their implementation and hardware, local load modules are more efficient. The last point is not relevant to our situation.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.4.4. Diskless SUN Workstations

The V-kernel of Cheriton and Zwaenepoel [Cheriton83a] is message oriented, being based on Thoth [Cheriton79] and Verex, and runs on SUN MC68000 workstations. VAX UNIX systems are used to provide file servers over an Ethernet. The file system is supported by general purpose inter-process communication facilities.

Cheriton and Zwaenepoel found that the 3 messages required by their protocol to transfer a 512 byte page over a 3 Mbit/sec Ethernet with 8MHz processors involved an elapsed time of 5.6 msec. The user processor time was 2.5 msec and the server processor time was 3.3 msec. For comparison, a similar message took 3.3 msec locally. Without considering disk transfers and latency, the system could handle about 175 of such transfers per second.

The important point that Cheriton and Zwaenepoel make concerns the intelligence of their Ethernet interfaces for which data must be copied in and out. They say that even with DMA the system would still have to copy headers and/or data buffers for their interfaces.

The lesson from their work re-iterates that of Luderer et al.. Transfers over the network are not the time consuming entity in the system, rather it is the processing in assembling and disassembling the message.

4.4.5. Remote Procedure Calls

Nelson undertook performance evaluations of remote procedure calls over Ethernet [Nelson81]. Nelson gives an interesting summary of performance lessons which include:

- Special purpose protocols are good.
- Use microcode for better performance (up to a factor of 30 improvement).
- Avoid copying wherever possible.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The tables of remote procedure timings presented in Nelson's thesis are not directly transferable to our work. However, as an extreme of his performance figures, we note that a microcoded remote procedure call between Dorado machines with no parameters took 150 μsec. This involved 2 packets of 5 words (10 bytes). Such figures certainly give cause for hope with regard to processing times for message assembly and disassembly.

4.4.6. Recapping

We can expect increased throughput by distributing user processes, and hence the load, over user machines in a network with a file server machine. The network must be fast, typically transferring at 10 Mbit/sec.

We have seen the importance of intelligent DMA interfaces. Thus, disassembly of a message can be performed by first deciphering a special purpose protocol and then appropriately redirecting data to end-point buffers without the need for copying. A similar scheme can be used for the assembly of a message. Without such interfaces, the overhead of copying is at least as much as performing the request locally. Existing systems indicate that 175 kbyte/sec transfer rates are sustainable without such interfaces.

4.5. A V7 UNIX System

The V7 UNIX system that we shall examine to determine the feasibility of the proposed distribution is shown in figure 4.2. The processor is a Perkin Elmer PE3240 which has 4 Mbytes of memory. Three quite slow 40 Mbyte Ampex disks are attached to one controller and can perform concurrent seeks, and one 80 Mbyte MSM80 (Digital Equipment) disk drive with its own controller provide the mass storage for the system. Up to 30 user terminals operating at 9600 baud may be connected to the system with the usual user load being 12-20 during the day.

The version of UNIX that is run is Version 7. Originally the system came from the University of Wollongong, but it has been extensively modified at the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

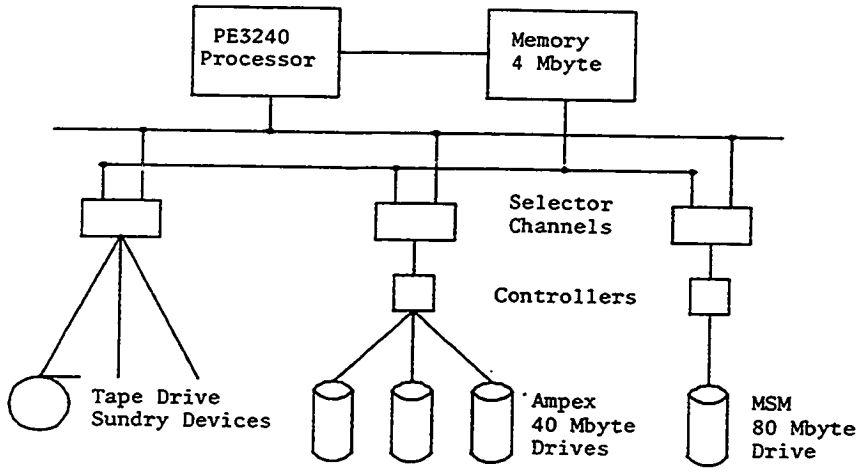


Figure 4.2. PE3240 hardware configuration.

University of Melbourne. Many of the available utilities are from Berkeley's 4.1BSD.

A preliminary presentation of some of the following work appears in [Dix83a]. At that time, the system had 1.25 Mbytes of memory and hence some of the performance characteristics of the system have changed.

4.6. Job Mix

No special job mixes have been used in the derivation of the figures in the the following sections. Mainly, the machine is used for text processing, operating system development and application programming in C, APL and Prolog. Tables 4.1 and 4.2 are derived from a summary of accounting figures for the system which were collected by the system accounting program, sa[2], over a period of about 5 months.

Table 4.1 shows those programs which use a significant amount (>1%) of the total cpu processing time. The table is ordered by percentage of total cpu

[2] The programs **sa** and **iostat**, that are referred to in this chapter, are documented in sections 8 and 1, respectively, of UNIX Programmer's Guide.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

usage. It includes the average ratio of user to system time, and average i/o and percentage i/o of all transferred disk blocks that the system attributes directly to each program. The last two entries are for all processes and for the sum of the C compiler passes and including linking.

Table 4.1. Major CPU usage programs.

Program	% Total CPU Usage	User:System Time	Ave i/o	% i/o	Description
vi	13	2.1 : 1	285	12.7	screen editor
nroff	10	4.4 : 1	1247	55.1	text formatter
mpass3	7.5	20.2 : 1	1708	1.4	music synthesis
newcsh	4.2	1.2 : 1	207	2.8	command processor
as	3.8	7.6 : 1	182	2.8	assembler
cpp	3.6	6.5 : 1	182	3.6	C preprocessor
more	3.3	3.8 : 1	78	3.0	scroller
ndk	2.9	1 : 3.8	142	.77	printer filter
sh	2.9	3.7 : 1	65	5.6	command processor
make	2.8	8.4 : 1	105	.88	program maintenance
c1	2.5	5.5 : 1	177	2.7	C compiler pass 2
c0	2.3	4.1 : 1	180	3.1	C compiler pass 1
prolog	2.1	7.4 : 1	282	.34	prolog interpreter
apl	1.4	8.9 : 1	430	.20	apl interpreter
w	1.2	1 : 16.9	58	.35	user process info
ld	1.1	1 : 1.1	925	4.6	linker
ls	1.0	1.2 : 1	47	2.1	list directory
total	100	2.0 : 1	97	-	of all accounting
cc	9.7	3.9 : 1	170	17.6	total C passes, etc. & link

We see that about 10% of cpu processing time is used each for C compilations and linking, file editing and text formatting; a music synthesis program also uses a similar amount of processing time. Importantly, almost 68% of disk i/o is attributable to two programs — vi and nroff.

Table 4.2 gives the most frequently used programs (>1% frequency). This table is ordered by frequency of program usage. In this group are many of those programs in the first group. We have excluded the system daemons and the system house-keeping programs from these tables, thus concentrating on user programs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.2. Most frequently used programs.

Program	% Frequency	User:System Time	Ave i/o	% i/o	Description
sh	8.4	3.7 : 1	65	5.6	command processor
ls	4.4	1.2 : 1	47	2.1	list directory
vi	4.3	2.1 : 1	285	12.7	screen editor
more	3.8	3.8 : 1	78	3.0	scroller
rm	2.4	1 : 9.3	27	.66	remove file
cc	2.0	1 : 25.4	44	.88	C compiler driver
cpp	1.9	6.5 : 1	182	3.6	C preprocessor
echo	1.9	1 : 4.0	9	.18	echo args - quick ls
c0	1.7	4.1 : 1	180	3.1	C compiler pass 1
u	1.6	1 : 4.3	24	.40	system users
c1	1.5	5.5 : 1	177	2.7	C compiler pass 2
as	1.3	7.6 : 1	182	2.8	assembler
newcsh	1.3	1.2 : 1	207	2.8	command processor
cat	1.1	1 : 1.9	41	.45	print file

4.7. Statistics Gathering

The figures collected from the PE3240 system to be presented in section 4.9 used two programs, **stats** and ss, both being extensively modified versions of the i/o statistics program **iostat**, which collects statistics from tables planted inside the kernel. Either absolute statistics from system bootstrap or incremental statistics after sleeping for a period of time, including overnight and over the weekend, may be collected. **iostat** can provide the mutually exclusive percentage times that the system is idle, executing in user mode, executing in system mode and waiting for disk i/o, terminal i/o and disk i/o statistics including buffer cache hits.

The extra tables and code to increment elements were added to the kernel. A table recording the use of individual system calls was maintained. For the **read** and **write** system calls,[3] a breakdown for each type — disk, terminal and pipe — was also kept. Also, the system kept a table of the current active processes and those that were waiting for terminal and disk i/o.

[3] We use bold face for system calls. The reader is referred to [Ritchie74] and section 2 of the UNIX Programmer's Manual for complete descriptions of UNIX system calls.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.8. Processor Allocation

From table 4.1, we can see that the allocation of the processor has a ratio of about 2:1 for user to system time. System time includes process scheduling, interrupt processing and system call processing. Figure 4.3 shows general agreement with this ratio over a range of different processor loads. The graph plots percentage user and system times against percentage idle times which are grouped in ranges of 10% (0 to <10, 10 to <20, etc.). The sampling was done every minute.

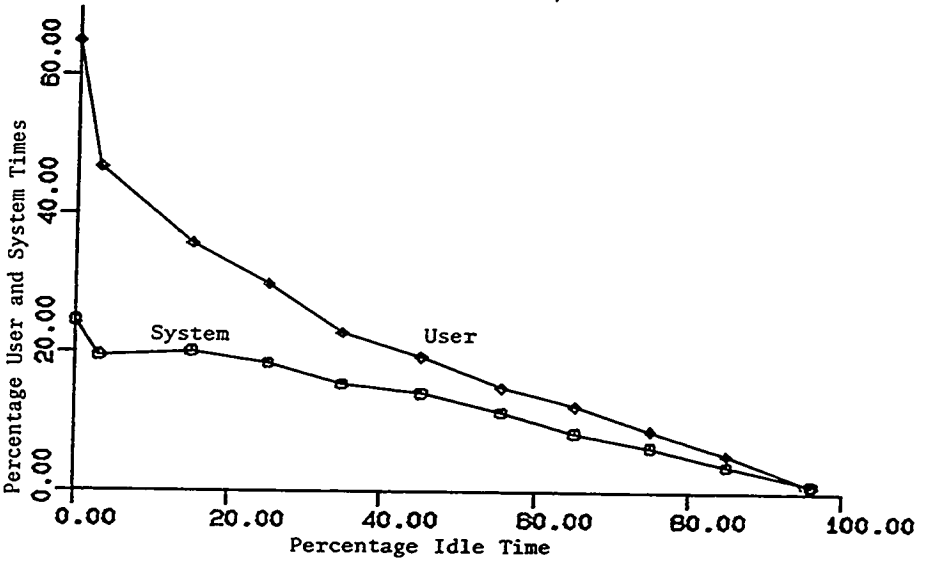


Figure 4.3. Percentage user and system time versus idle time.

When the system is greater than 40% idle, the system time is quite significant, but there is processing power to spare. The disk i/o wait time can be considerable on this system due to the slow disks and goes up to 25% at 5% idle.

Figure 4.4 plots percentage system time against percentage user time (grouped by user time). We see a roughly linear relationship up to about 60% user time where the percentage system time must drop. We have extrapolated linearly back to the axis and obtained a figure of 7% system time. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

provides an estimate of the time required for scheduling the processor in a moderately busy system (from the graph, <70% idle time). This averages out at about half of the system time.

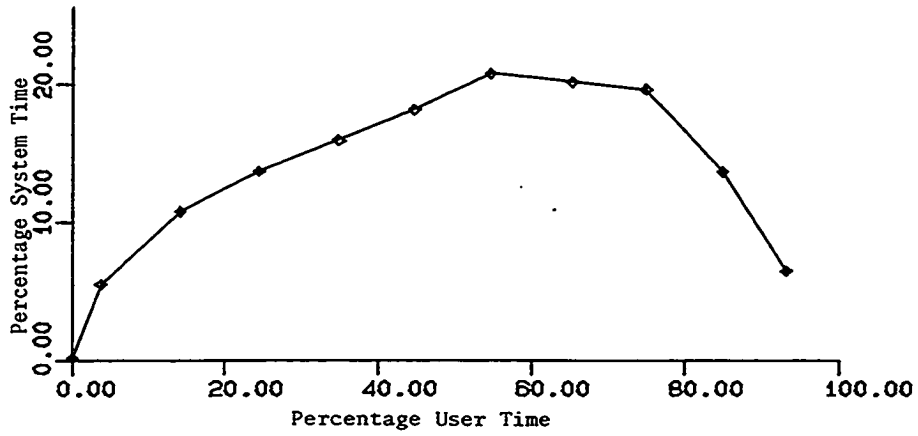


Figure 4.4. Percentage system time versus user time.

4.9. System Calls

UNIX may be characterized by the supported set of system calls that provide for the creation and management of user processes and for access to the file system. We will now examine the use of and times for the V7 UNIX system calls.

4.9.1. Usage

Table 4.3 shows the relative percentages of system calls. The figures were collected over a 5 month period. Only system calls of greater than 0.1% significance (which is of the order of one per two process forks) are included. Some 88% of the total system call usage involves the file and terminal systems. In particular, **read** and **write** contribute some 63% of all system calls.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 4.3. Percentage of significant system calls.

(i) I/o Related		(ii) Process Management	
System Call	%	System Call	%
read	37.84	signal	5.74
write	25.38	alarm	1.42
seek	9.61	time	0.80
close	7.11	break	0.73
ioctl	1.97	exece†	0.53
stat	1.81	getpid	0.45
dup	1.52	ftime	0.37
open	1.21	fork	0.31
fstat	0.30	exit	0.31
chdir	0.30	wait3	0.29
unlink	0.25	times	0.27
creat	0.24	pause	0.22
sync	0.12	wait	0.19
		getuid	0.15

† exece uses open, read and close system calls.

Table 4.4 gives a breakdown of the read/write system calls for disk, terminal and pipe i/o. In addition, a sub-category is given for terminal reads for which only one character was available and similarly for writes of one character. The swapping figures, which are presented relative to read/write system calls for comparison, show that with a large memory, swapping is negligible. It should be noted that V7 UNIX imposes a maximum size for programs.

Table 4.4. Distribution of read/write system calls.

Device Type	Read	Write
All	59.86	40.14
Disk	23.14	7.82
Terminal	20.64	30.13
Pipe	16.06	1.82
1 Character Terminal	15.77	23.27
Swap	0.011	0.004

In [Dix83a], it was stated that one character terminal writes were excessive due to a library bug and that a 30% reduction in such writes could be expected. This table shows that such a reduction did occur when compared

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with a similar table in that paper.[4] Another difference is a 10% increase in pipe usage and a subsequent decrease in disk writes. We believe this is due to the almost threefold increase in available memory and subsequent pipelining by users even during heavy cpu usage periods. Certainly swapping has dropped from 0.5% to 0.015%.

Without taking system call execution times and message assembly/disassembly times into consideration, we can offload 88% of system call processing. We shall see in chapter 6 that **seek**, **dup** and most of **close** can be performed locally, as well as pipe reads and writes. Such a distribution of processing would perform

39% locally on a user machine,
32% on a terminal machine, and
29% on a file machine.

4.9.2. Timing

Generally, the technique used for timing system calls was to perform the call at least 1000 times within a program which was run when there was no other load on the system. An average is then obtained. This overcomes the problem of only having a 50 Hz clock.

However, other problems are encountered. For example, it is impossible to measure an **open** without a **close**, which an **exit** will do if it is omitted anyway. The alternative, to do 1000 **opens**, also is not possible, due to the limit on open files. Instead, we will use some representative system call times that we can measure.

We shall look at timings relating to process management and then the i/o system related calls.

---

[4] We should point out that much of the 1 character terminal system calls are due to the particular hardware that is used for across system logins and file transfers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.9.2.1. Process Management

Firstly, we will use some system call timings to provide an estimate of kernel entry/exit overheads and of process switching.

1. Kernel entry/exit overhead - 0.34 msec.

The system calls which involve almost no processing, **getuid**, **setuid**, **getgid**, **setgid** and **getpid**, were used to provide this estimate.

2. Process switching - 2.8 msec.

To obtain this figure, two processes communicating via alternating pipes were used to force context switches and this was adjusted by similar pipe usage in a single process.

3. Process creation.

The time for a process fork depends on the program size. We tried two extremes of the **fork** system call:

8 kbyte - 17.5 msec.

408 kbyte - 258 msec.

4. Process exit - 4.7 msec.

The process creation and exit times were measured by forking and waiting 1000 times and obtaining the times used by the children, which merely exit. The total child system time approximates the exit time and the time for the wait (1.6 msec) was subtracted to obtain the fork time.

5. Program load and execute.

Again two extremes were measured for the **exece** system call:

6 kbyte - 24 msec.

406 kbyte - 148 msec.

Interestingly, the real-time for both tests was only marginally larger than the total of system time and user time. This indicates that read ahead was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

effective, even with the slow disks and that load time with large *bss* arrays (requiring zero initialization) is compute bound.

Typically, the other process management system calls which require an entry to be made in the process structure take a similar time to **signal** - 0.54 msec.

4.9.2.2. I/O Management

1. Read/write file - 2.1 msec.

This figure was obtained for 512 bytes accessed sequentially on block boundaries.

2. Read/write pipe - 1.4 msec for 512 bytes.

3. Read/write terminal.

This is a difficult time to obtain. We only tested terminal writes and expect reads to require slightly more processing time. A range of buffer sizes were used.

1 character - 2.8 msec.
20 characters - 3.2 msec.
80 characters - 4.5 msec.
512 characters - 13.8 msec.

4. Create file.

To measure this the fork timing scheme was used where the child did a **creat** call. Three tests were done creating a file in the home directory:

relative path - 6.4 msec.
absolute path, 9 directories deep - 17.8 msec.
absolute path, 19 directories deep - 28.1 msec.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. File statistics (**stat**) - 2.6 msec.

6. Seek file - 0.46 msec.

lseek does no file i/o, hence the call is relatively fast.

The **close** system call, which is the other well used call in this category, does not often cause i/o. The command processor, **sh**, is responsible for the majority of closes on files, most of which are not open. Also, it is only when a reference count for an open file goes to zero that the file is really closed. Times for **open** are assumed to be similar to **creat** and times for **dup** similar to **seek**.

4.9.3. Observations

Firstly, we notice that there are several cheap system calls. These should be performed locally in the user processor. In addition, it would be preferable if **seek**, **dup** and **close** were performed locally.

For file **read** and **write**, the time of 2.1 msec is about that reported for the message passing overheads by Cheriton and Zwaenepoel with copying required by the Ethernet interface. This means that no gain on the user machine with regard to offloading processing would be achieved with similar interfaces. For **creat**, **open** and **stat** a gain would be made.

4.10. Weighted System Call Loads

We shall now derive estimates of weighted system call loads for the three types of machines. By this we mean the expected relative execution time load that each machine must support to provide a particular set of distributed system calls. This may be determined by summing individual system call percentages weighted by the time taken to perform each system call, and allowing for message assembly/disassembly times. We shall use the percentage system call distributions and sub-distributions from tables 4.3 and 4.4 and weight each percentage system call by the time estimates for each from the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

previous two sections. We shall also adopt the suggested distribution of **seek**, **dup** and **close**.

In order to make this calculation, we need to assume a time for the message processing overhead associated with a distributed system call. We have a lower bound of 0.34 msec for kernel entry/exit on user machines and an upper bound of 2 msec from the work of Cheriton and Zwaenepoel.

Assumption:

The average time to assemble and disassemble messages associated with any system call is 1 msec.

We believe this is a realistic, probably conservative, estimate of message processing overheads with better interfaces. Notice that with such an overhead we must gain from kernel distribution.

We illustrate our derivation of weighted system call execution times with two examples. For **signal**, which is implemented on the user machine, the weighted load is 5.7 * 0.54 (5.7% of system calls being signals and each taking 0.54 msec). Whereas read/write disk has 20 * 1 on the user machine (20% of all system calls being disk read/write calls with 1 msec message overhead) and 20 * (1 + 2.1) on the file machine (20% with 1 msec overhead and 2.1 msec system call processing).

For **close**, we assume that 6% are performed locally in the user machine and 1% require a message to the file machine. Notice that we have assumed that the time to process 512 bytes is representative of file reads and writes (which will be the case for programs that use the standard i/o library).

For terminal read/writes, we shall assume that the average number of bytes is 20 and that **ioctl** calls take a similar time to writes, that is, 3.2 msec.

The weighted system call loads are:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

User machine	99
File machine	96
Terminal machine	130

For comparison, the weighted system call loads for no kernel distribution, that is, all system calls performed locally with no message overhead, is 199. Hence, the system call processing has been reduced by 50% on the user machines, including message processing. The total weighted system call load is 325 for the distribution, an increase of 60% over all machines.

Scaling the loads to 100% we get:

User machine	30%
File machine	30%
Terminal machine	40%

This gives a quite well balanced distribution of system call processing.

4.11. Disk I/O

So far we have assumed in all our calculations that the disks on the file machines can keep up with the requests from user processes. Now, we shall see if this is a reasonable assumption.

UNIX smooths disk i/o transfers by using a buffer cache. Delayed buffer writing, write behind, and anticipatory read ahead, when file access is detected as being sequential, are applied. Disk queues are ordered by minimal seek per sweep, so a large cache can reduce disk latency overheads when disk activity is high.

Figure 4.5 summarizes disk block transfer and buffer cache statistics for the V7 PE3240 system. The figures were collected each minute during weekdays between the hours of 10:00 and 22:00 over a period of months. The graph plots total disk transfers per minute, with a breakdown of reads, writes, read aheads and buffer cache hits, against percentage idle time. The averages are grouped in ranges of 10% idle time.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

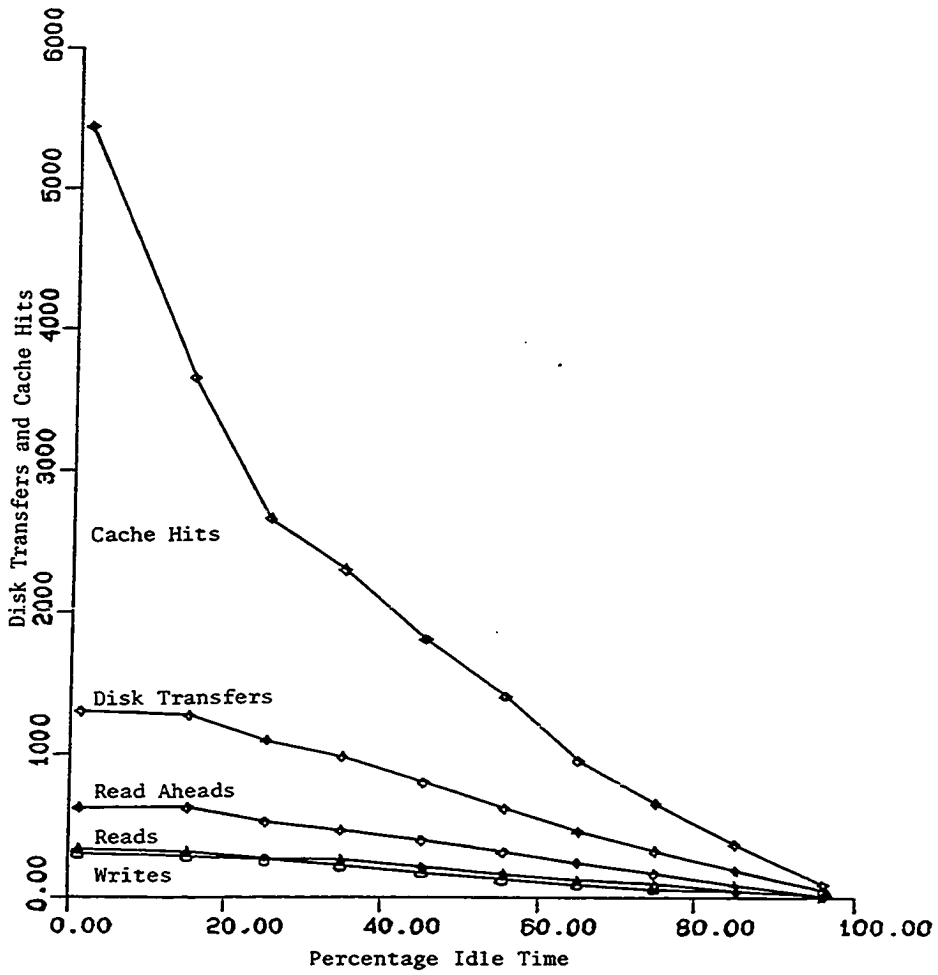


Figure 4.5. Disk transfers and cache hits versus percentage idle time.

For a busy system, less than 20% idle time, the average number of transfers is greatest at about 1300 transfers/minute, of 512 byte blocks. For comparison, the maximum transfers detected per minute were:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Total	3031
Reads	1743
Read aheads	2132
Writes	1358

We estimate the theoretical maximum disk transfer rate, with allowances for latency, is about 6000 blocks/min.

We have already commented on the slowness of our disks. For Berkeley's 4.1BSD UNIX, 175 kbytes/sec can be expected [McKusick83] with 1024 byte transfers being requested of intelligent disks (with decay in performance occurring as the free list becomes randomized). The 4.2BSD file system has been revamped to increase throughput by optimal placement of blocks and using up to 8 kbyte block requests. Up to 39% of the disk bandwidth has been observed and with maximum rates of 466 kbytes/sec over the MASSBUS of a VAX-11/750 being recorded.

It is reasonable to expect that, with the adoption of 1 kbyte buffers, a rate of 100 kbyte/sec would be maintainable, provided the load was spread over multiple disks which supported good transfer rates. This corresponds to a rate of 12000 blocks of 512 bytes per minute. That is, 4 times the maximum on the PE3240 system and 9 times the average.[5]

We could expect to support disk transfers for up to 9 user machines on one file machine but we would require fast disks that can handle requests for blocks larger than the de facto physical standard of 512 bytes.

4.12. Network I/O

The network must match the disk i/o transfer rates and accommodate the additional overhead of request and acknowledgement for the protocol. Although

[5] We emphasize the slowness of the Ampex disks on the PE3240 system which tends to be disk bound. Experience with a VAX 4.2BSD UNIX system shows that faster disks make a considerable difference to attainable transfer rates.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

not all disk transfers have a 1:1 relationship with file i/o system calls, we believe it provides a good estimate of network load which we shall use.

We shall allow 10% for the protocol overheads. Hence, a file machine handling 100 read/writes per second of 1 kbyte blocks would participate in producing a load of 120 kbytes/sec on the network.

With regard to the network architecture that we have mentioned so far, the virtual circuit used by Luderer et al. was found to support kernel to kernel transfers of 125 kbyte/sec and could support the file i/o load. We will now look at the ability of an Ethernet to satisfy the above network load.

Since the paper of Metcalfe and Boggs [Metcalfe76] detailing the characteristics of Ethernet, further theoretical and simulation analyses of Ethernet have been performed [Almes79] [Blair82].

We notice that 120 kbytes/sec is about 10% of the carrying capacity of a 10 Mbit/sec Ethernet. The work of Almes and Lazowska with mixed packet sizes showed no appreciable difference in response times with such a load. From their work we estimate the perceived efficiency, that is the ratio of minimum possible and average transfer times, to be at least 90% and an average response time of around 200 μsec can be expected. The average transfer time for a request, an acknowledgement and data, would be about 1.3 msec.

For the terminal side of the network, Blair and Shepherd suggest that for short packets, which are often associated with terminal i/o, a ring, such as the Cambridge Ring [Wilkes79], provides a better medium than an Ethernet. No conclusive studies have been performed comparing the software overheads associated with either. However, Collinson reports of successful use of a Cambridge Ring with a VAX-11/780 running UNIX and PDP-11's for frontend terminal handling [Collinson82]. This approach is different from that which we are proposing. A device driver for the ring was implemented as a character

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

special file[4] on the VAX system which used the character lists in much the same way as a normal terminal driver.

4.13. Distribution Revisited

Originally we noted a 2:1 user to system time ratio. Closer examination of programs which account for most of the processor usage revealed a ratio of 4:1 was more applicable. Since the system call processing time is reduced by half due to distribution, this ratio becomes 8:1. We could expect 1 file machine to maintain 9 user machines before it ran out of processor time to service requests.

Consideration of disk transfers led us to a similar figure when allowance was made for increased disk throughput by doubling the transfer block size. We also could expect typical high bandwidth networks to support the generated communications traffic.

In the V7 PE3240 system we examined, the terminal machine processing, as measured by the weighted system call load, was slightly more than for the file machine. We will see (in chapter 6) that a slightly more efficient implementation of the terminal system code is possible. However, it would appear that the terminal system could prove to be the bottle-neck in the distributed system.

Of course the terminal handling could be done on the user machines, effectively adding more processing to those machines. The weighted system call loads would become:

User machine 176 ie. 63%

File machine 96 ie. 37%

The problem with such a scheme is that the flexibility of user to user, that is, terminal to terminal, communication is lost. Also, the reduction in user

[4] See [Ritchie74] for an explanation of special files.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

time on the user machine is small. Consequently, a file machine could handle 10 such user machines under the same conditions. Interestingly, this ratio of 10:1 is the same as the estimate by Cheriton and Zwaenepoel for diskless SUN workstations with attached terminals and a file server.

We have noted the need for network interfaces which reduce the amount of copying that the kernel must do. A time division multiplexed switch is installed on the PE3240 machine. The switch can support 4 concurrent 2 Mbyte/sec synchronous transfers using DMA. In [Dix83a], we presented a preliminary analysis of the disk and network loads that are now given in this chapter. It was determined that the switch could easily support the traffic expected on the network. No work has been done concerning contention for port connection over the switch, which will perform arbitration but not retries on failure to connect.

Now, recall that swapping is not supported in the distribution. Consequently, when a message containing data for a user process arrives, that process must be resident (assuming it has not been killed) and the data can be transferred directly into its buffer area. In standard UNIX, where swapping is supported, such a scheme cannot be used because the user process may have been swapped out after it issued the read request. The decision not to support swapping has provided for a more efficient implementation. Similar reasoning can be used for the file machine so that time spent copying can at least be reduced and perhaps eliminated.

4.14. In Summary

In this chapter, we have proposed a distribution of the standard V7 UNIX kernel over a very local network. Related research indicates that current processor and network hardware will support the distribution. More intelligent interfaces than have commonly been used for network connection must be used to prevent copying.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The distribution requires user machines to be connected to file machines and to the terminal machine. The terminal machine handles all i/o for the character oriented devices. Each user machine supports multiple users. Each file machine supports a UNIX file system. The division of labour in processing system calls across these machines seems to be quite well balanced.

Consideration of disk transfer rates that a UNIX file system can sustain and of file requests that are generated by user processes, enabled us to estimate that 1 file machine could support 8-9 user machines without being overloaded.

It proved to be difficult to determine terminal processing overheads in the network. However, we estimated that 8 user machines could be supported by 1 terminal machine. More user machines could be added. This would simply mean that at times the terminal machine would be overloaded. Terminal output tends to occur in bursts and short delays can be (and are) often tolerated.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 5

A Model of a Distributed UNIX Kernel

In this chapter we present a model of the distributed UNIX kernel that we proposed in chapter 4. The model uses CSP including the extensions for preemption that were given in chapter 2. Most of the material in this chapter appears in [Dix84].

The main purpose in presenting this model is to contribute to an increased understanding of the distributed UNIX kernel. Also, a large amount of C code in the original V7 implementation may be used for our distributed kernel.[1] As a consequence, the material presented here makes an additional contribution in that it also models most of the original implementation.

The adequacy of CSP for modelling operating systems also is being put to the test in this chapter. In particular, we wish to apply the preemptive commands to determine whether they provide sufficient expressive power. We have given the solutions for typical operating system problems, such as scheduling, asynchronous buffering and providing service facilities, in chapters 2 and 3. Now we will adopt similar schemas in a more complex situation. Notice that although we are interested in presenting a model, the ability to produce code for a kernel written in CSP also adds credence to the application of preemption in CSP as a general programming language construct for real-time situations.

---

[1] We shall comment on the amount of the original code that is usable for a distributed implementation in chapter 6.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

One problem encountered in the preparation of this chapter was in deciding what degree of detail is suitable for inclusion in the model. We believe that there is nothing to be gained in recoding the UNIX kernel in its entirety in CSP. Rather, we present only those details that we consider are essential to the understanding of the kernel design. The chosen level of granularity that the model portrays is given in section 5.1 and corresponds to the major distributed functions. We review the relationships between languages classified by their design orientation in section 5.2. Section 5.3 explains the representation within the model of process multiplexing, which is a basic requirement of the implementation.

Sections 5.4 to 5.6 give models for user, system and terminal machines respectively. With regard to the degree of detail, the user machine model is the most comprehensive. This corresponds with the fact that scheduling user processes is quite difficult to model. Less detail is shown for file machines and terminal machines which are more easily modelled. For these machines models are presented which show the multiplexing and demultiplexing of messages for requests and results, and the role of the device handlers. The static priority ordering of parallel processes is revisited in section 5.7.

5.1. Level of Granularity

The network proposed in chapter 4 has

(1) user machines, on which user processes would execute with the support of a supervisor and scheduler,

(2) file machines, which support the UNIX file (mass storage) system, and

(3) terminal machines, which support the UNIX terminal (character) systems.

We shall differentiate between such machines by presenting individual models. However, as it is not the intention to fully emulate all the details of the kernel, we shall model at the functional level of major system processes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The essential functional features of the UNIX kernel are the user process interface, the scheduler, the file system, the terminal system and the device drivers. We choose these as the level of granularity for the model and include consideration for user processes and a supervisor that provides the user process interface.

With the chosen granularity, details of how the kernel implementation is programmed in C are avoided. Rather, we will distinguish between processes, both user and kernel, and show the messages that must be passed among them. We do this to make explicit the message passing between processes that are distributed on different machines. In an implementation however, message primitives need not be used to provide all process interfaces. Also, we emphasize the real-time nature of some message passing and synchronization.

5.2. Language Orientation

Lauer and Needham [LauerH78] have commented on the duality of message passing and monitor calls for operating system structures. Their opinions were confined to single processor implementations. More recently, Reid further examined the conjecture with the inclusion of remote procedures [Reid80]. Generally in agreement with the above researchers, Andrews and Schneider [Andrews83] make the following observations about procedure-oriented languages (with monitors), message-oriented languages (with send and receive primitives) and operation-oriented languages (with remote procedure calls):

> At an abstract level, the three types of languages are interchangeable. One can transform any program using one mechanism into another using a different mechanism. However, each class provides flexibility which is not present in the others.

In the case of the distributed kernel, the design relied on the ability to support the expected message throughput over the communications network. The style imposed by CSP emphasizes such messages and in this regard supports the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

above observation.

We make the point that not all languages of the above three classes are designed with the same applications in mind. For example, not all might support mechanisms for associating priorities with events (be they calls or messages) or for device handling. Hence, it may not be possible to produce an equivalent program, even in a similarly orientated language.

Another consideration in the choice of language is performance, which for an operating system is crucial. In turn, this relies on the underlying hardware. The efficiency of the primitives provided by a language can only be as good as the hardware capabilities permit. Lauer and Needham cited two microcoded implementations as empirical support for their statement that monitors and message passing are comparable in terms of performance.

The distributed UNIX kernel epitomizes the above dependence on hardware at the implementation level. Some messages will need to pass over the network, others will be implemented by shared memory, and the rest as parameters to traps and reentrant procedure calls. The model abstracts away from the implementation.

From the above discussion we can draw the conclusion that any language which has an abstraction for message passing and process priority would serve the purpose for which we intend to use CSP. We have chosen CSP mainly because it has well defined semantics and that its synchronization and message passing primitives nicely encapsulate the features we wish to highlight. However, another major reason is that we believe the style which it will force upon us leads to a clear presentation. The latter point it should be noted is somewhat of a value judgement.

5.3. Modelling Process Multiplexing

Multiplexing of the processor among the processes resident on the individual machines is implicitly necessary. We can model the relinquishing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of the processor using CSP's synchronized i/o commands. A process which reaches an i/o command for which the communication partner is not ready to synchronize no longer requires the processor.[2]

Two schemas will be used. The first uses the user/server schema:

```
User(i):: ...; Server!request(data); Server?results(variable); ...
Server:: *[(□i:1..NUsers) User(i)?request(variable)→
                                        ...; User(i)!result(data)].
```

The server relinquishes the processor when there is no request to service (in general, when there is no true guard). On the other hand, the user, on making a request, then waits for a reply; thereby relinquishing the processor to the server.

The second schema uses the preemptive commands. Implicitly, P and Q in P **except** Q and P **interrupt** Q are multiplexed. The preemptive guards of Q are passive and act as alternatives that are distributed over the execution of P. However, should a preemptive guard become true, even if P is waiting to synchronize, then transfer of control is given to Q.

Obviously, if all processes on a given processor are waiting at an i/o command, then the processor is idle. Presumably a device interrupt or a message from another machine will alter the situation.

5.4. User Machines

A model of user processes, the scheduler and the supervisor is required for the user machines.

We restate the two assumptions that were introduced in chapter 4 and which simplify the situation.

---

[2] This approach has been used for at least one multiplexed implementation of Ada tasks where an attempted rendezvous provides a suitable point at which to context switch.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(1) Memory is so cheap that swapping of user processes is unnecessary.

(2) Forking is done locally in the user machine and signal processing is not distributed.

The first problem that we tackle is that user processes are created dynamically by forking in UNIX. CSP is suited to statically declared processes.[3] However, the UNIX kernel has an abstraction for both existent and non-existent processes. Such information is kept in the so called process structure or *proc* table, which is defined by "struct proc proc[NProc]", where NProc is the configured maximum number of processes the system can support. We extend this abstraction by introducing NProc virtual processes, VProc, which will run user processes.

The implementation of system calls within the UNIX kernel is as extra code for a user process that is executed in kernel (privileged) mode via a trap or SVC type call. The kernel is reentrant in this regard. We choose to abstract reentrancy by a set of identical processes, SysCall, which provides the system call interface. The set of VProc and SysCall processes form the supervisor. To provide a static name for user processes, we shall bind VProc and SysCall declarations:

```
UserMachine =
    [(||i:1..NProc) (VProc(i) || (u(i):UserInfo; SysCall(i))) || Scheduler]
```

A per user process structure, u, is also defined for each SysCall.

5.4.1. User Processes

Virtual processes run user processes under control of the scheduler. Herein, lies the next problem. Given that a user process is executing, the scheduler must be able to stop and start the user process, thus providing a

[3] Hoare does use dynamic processes in some examples for his model of communicating sequential processes where the mathematical notation is well suited [Hoare80a]. We wish to avoid the issues associated with naming or addressing of such processes.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

time slice.

We define a virtual process:

```
VProc =
    [UserProc:process;
     *[Scheduler?run(UserProc) →
        [UserProc interrupt [Scheduler?stop → Scheduler?start]
        ] except [Scheduler?die → skip]
    ] ]
```

So a virtual process waits for the scheduler to send information about a user process which is to be run. We then execute the user process, effectively using an **exec**, until an interruption or exception occurs due to the scheduler. Hence, we model a **fork** by assigning a VProc and using an **exec** to run the core image.

The preemptive guards, Scheduler?stop and Scheduler?die, may be considered to distribute over UserProc. Scheduler?die has priority over Scheduler?stop which has priority over UserProc, in accordance with their nesting. However, only when such a message is available (synchronization in this case) will preemption occur and the guarded command subsequently be executed.

In the case of interruption by the scheduler after the stop message, VProc will wait for a start message, thus modelling the release of the processor. On receipt of a start message, resumption of UserProc occurs. For an exception, UserProc is terminated, presumably because of a fatal signal or error.

In the UNIX kernel implementation, user processes wait for the trap return from system calls which includes status values. To model this, user processes apply the user schema:

```
SysCall!s(parameter_s); SysCall?s(status_s)
```

where s denotes a particular system call.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Notice that the declaration of the supervisor binds pairs of VProc and SysCall. This enables a user process to reference the bound SysCall process directly, and conversely a particular SysCall to reference VProc and in fact be addressing an exec'ed user process.

Most calls require at least one parameter to be an address to/from which data is transferred. To illustrate, consider:

```
SysCall!read(file,address(buffer),length); SysCall?read(lengthread)
SysCall!fork; SysCall?fork(processid)
SysCall!exec(filename,address(Image)); SysCall?exec(status);
            [status ok → Image]
```

We pass addresses of buffers using the function *address*. The input messages from SysCall are the status information and indicate the degree of success of the system call.

5.4.2. System Calls

We will illustrate only those system calls that were used as examples above. The function *to* is the inverse of *address* and both are used for buffer pointer manipulation.

```
SysCall =
   *[ VProc?read(file,length,buffer) →
       [file.type=block →
            FileSystem(file.sys)!read(file,length); Scheduler!sleep;
            FileSystem(file.sys)?read(status,to(buffer));
            Scheduler!wakeup; VProc!read(status)
       [] file.type=terminal →
            TerminalSystem(file.sys)!read(file,length); Scheduler!sleep;
            ...
       ]
    [] Scheduler?run(u) → skip
    [] VProc?fork →
        Scheduler!fork(u); Scheduler?fork(processid); VProc!fork(processid)
    [] VProc?exec(filename,image) →
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Scheduler!sleep;
        FileManager!open(filename);
        [(□i:1..NFileMachines) FileSystem(i)?open(status,file) → skip];
        [file is executable and status ok →
            FileSystem(file.sys)!read(file,file.length);
            FileSystem(file.sys)?read(status,to(image));
            Scheduler!wakeup; VProc!exec(status)
        □ file is not executable or status not ok →
            Scheduler!wakeup; VProc!exec(ERROR)
        ]
    □ ...
    ]
```

FileManager mentioned above contains the mount table that indicates directories upon which file systems are mounted and could be resident in the file machine that manages the root file system, (although several alternative strategies are feasible). Consideration for current working directory is ignored here. The file manager redirects calls for file references using an absolute path name to the appropriate file machine.

We have given a model for a selection of system calls. Deliberately, we have concentrated on those which communicate with distributed parts of the kernel. We have omitted many of the finer details.[4] Other system calls either operate in much the same way (i/o system calls, for example), or request the supervisor to alter protected information such as that in the per user process structure. Again we emphasize the difficulty in selecting a suitable division in the presentation of details within the model.

With regard to file information, we have shown above the passing of a file structure by the user process. We could have modelled a more protected scheme by only allowing the user process access to an index, f, into a file table, file say. Then, the first reference above would become "file(f).type".

---

[4] For example, one such point which comes to mind is the examination of the file information on an exec request to see if the object file is a so called "set user id" program, in which case the "effective user id" can be changed in the user structure.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.4.3. Scheduler

The scheduler exists in UNIX as the procedure *swtch* (not to be confused with procedure *sched* which is in fact the swapper and is not under consideration due to a previous assumption). Apart from swapping, there are two sources of calls to *swtch* that are instigated by

(1) System calls: via procedure *sleep* which enables waiting for the availability of a resource or completion of a transfer, or exiting and tracing.

(2) Clock: where priority adjustment or time-out completion results in the existence of a higher priority user process which is ready to run.

We define the scheduler:

```
Scheduler =
    [proc:(1..NProc)ScheduleInfo; Schedule interrupt Clock]
```

Clock is of higher priority than Schedule and the process structure is accessible to both.

Clock is simplified a little by deleting the callout structure which is used for very short timeouts, typically by the terminal system, and is not necessary in a user machine.

```
Clock =
    [ClockDevice?tick →
        update system clock;
        ∀p∈proc: timeout finishes now, set p.status to RUN;
        reschedule := ∃p∈proc: p.priority > proc(current).priority
    ]
```

Schedule uses the flag reschedule to context switch if necessary after a clock interrupt.

```
Schedule =
   *[ reschedule → VProc(current)!stop; [[reschedule:=false]]; Swtch
    [] SysCall(current)?sleep → set proc(current).status to SLEEP; Swtch
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
[] ([]i:1..NProc) SysCall(i)?wakeup →
       set proc(i).status to RUN; VProc(i)!stop
[] SysCall(current)?fork(u) →
       find p: proc(p).status is NULL;
       for proc(p) set status to RUN, parent to proc(current).id, id, etc;
       SysCall(p)!run(u); VProc(p)!run(image(proc(current)));
       VProc(p)!stop;
       SysCall(current)!fork(proc(p).id)
[] SysCall(current)?exit →
       [∃p: proc(p).id is proc(current).parent →
           [proc(p).status is WAIT →
               set proc(p).status to RUN;
               SysCall(p)!wait(proc(current).id);
               set proc(current).status to NULL, etc; Swtch
           [] proc(p).status is not WAIT →
               set proc(current).status to ZOMBIE; Swtch
           ]
       [] ¬∃p: proc(p).id is proc(current).parent →
           set proc(current).status to NULL, etc; Swtch
       ]
[] SysCall(current)?wait →
       [∃p: proc(p).parent is proc(current).id →
           [proc(p).status is ZOMBIE →
               SysCall(current)!wait(proc(p).id)
               set proc(p).status and proc(p).parent to NULL;
           [] proc(p).status is not ZOMBIE →
               set proc(current).status to WAIT; Swtch
           ]
       [] ¬∃p: proc(p).id is processid → SysCall(current)!wait(ERROR)
       ]
[] SysCall(current)?kill →
       VProc(current)!die; as for exit above
[]
...
]
```

It is tempting to omit the details of wait, and hence exit, but so much of UNIX relies on this system call, in particular the shell (command language processor), that they have been included. The function *image* duplicates the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

current process text and data.

Swtch is much the same as the kernel procedure:

```
Swtch =
    [∃p: proc(p).status is RUN →
        current := such p with highest proc(p).priority;
        VProc(current)!start
    ]
```

5.5. File Machines

File machines service requests from the file manager for file open and file stats, and from system call processes for reads, writes, seeks and the like.

```
FileMachine =
    [FileSystem interrupt [([]i:1..NDevices) DeviceHandler(i)]]
```

The file system accepts requests and establishes the appropriate disk queues; starting the device if it is idle. We give an outline of the model where s represents one of the possible service requests and the returned result may be, for example, file information or data that has been read:

```
FileSystem =
   *[ FileManager?s(i,j,parameter*) →
        action* for UserMachine(i).SysCall(j)
    [] ([]i:1..NUserMachines ([]j:1..NProc))
      UserMachine(i).SysCall(j)?s(parameter*) →
        action* for UserMachine(i).SysCall(j);
    [] ([]d:1..NDevices) flag(d) → flag(d) := false;
        [completed action* for queue(d).done →
            UserMachine(i).SysCall(j)!s(status*,result*)
        [] not completed action* for queue(d).done →
            continue action*
    ]   ]
```

The device interrupt handlers interpret the reason for the interrupt and, on success of the transfer, set a flag for the file system. In this way we model sleeping on buffer addresses and wakeups on completion of transfer in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the kernel implementation. The device is assumed to be smart so only one command is necessary to initiate the transfer and only one interrupt will occur.

```
DeviceHandler(i) =
    [Device(i)?(why) →
        [why = done → flag(i) := true;
            put info in queue(i).done;
            start next transfer, if any, in queue(i).do
        □ why = error → retry
    ]   ]
```

Although we present the device interrupt handlers to be at the same priority level, they may in fact be nested, with the highest priority handler at the outer level.

5.5.1. File Manager

The file manager maintains a table of directory names upon which blocked devices have been mounted. Notification of mounts and unmounts are received by the file manager and requests to open files and for file status using file names are redirected.

```
FileManager =
   *[ (□i:1..NUserMachines (□j:1..NProc))
      UserMachine(i).SysCall(j)?mount(directoryname,inode) →
          put (directoryname,inode) in mount table
    □ (□i:1..NUserMachines (□j:1..NProc))
      UserMachine(i).SysCall(j)?open(fname,how,whom) →
          find f: head of fname is largest match with mount(f).name;
          fname := fname - mount(f).name;
          FileMachine(mount(f).inode.machine)!open(i,j,fname,inode,how,whom)
    □ ...
    ]
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.6. Terminal Machines

The terminal system is similar to the file system, but it is character orientated and uses input and output character queues for each terminal. We present two different models for terminal machines. The first takes a similar approach to that of the standard implementation, with additional consideration for messages.

```
TerminalMachine1 =
    [TerminalSystem1 interrupt [(□i:1..NTerminals) TerminalHandler(i)]]
where

TerminalSystem1 =
   *[ (□i:1..NUserMachines (□j:1..NProc))
      UserMachine(i).SysCall(j)?s(t,parameters) →
         perform actions for terminal t and send reply,
         or queue actions for (i,j) on terminal t
    □ (□t:1..NTerminals) flag(t).in → flag(t).in := false;
        [read completed for queued request on terminal t →
            UserMachine(i).SysCall(j)!read(count,queue(t).in(1..count))
        ]
    □ (□t:1..NTerminals) flag(t).out → flag(t).out := false; ...
    □ ...
    ]
and

TerminalHandler(i) =
    [ Terminal(i)?read(char) →
        put char on queue(i).in; flag(i).in := true
    □ Terminal(i)?write →
        start output with next char, if any, in queue(i).out;
        flag(i).out := true
    ]
```

The second model applies a polling technique to terminal devices rather than interrupts (an approach we shall discuss in chapter 6). Message requests

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are handled asynchronously using interrupts.

```
TerminalMachine₂ =
    [TerminalSystem₂ interrupt MessageHandler]

where

TerminalSystem₂ =
   *[ (□t:1..NTerminals) Terminal(t)?read(char) →
         put char on queue(t).in;
         [read completed for queued request on terminal t →
             UserMachine(i).SysCall(j)!read(count,queue(t).in(1..count))
         ]
    □ (□t:1..NTerminals) Terminal(t)?write →
         start output with next char, if any, in queue(t).out
    □ (□t:1..NTerminals) flag(t) →
         if possible perform action, for terminal t, send reply
         and set flag(t) to false
    □ ...
    ]

MessageHandler =
    [ (□i:1..NUserMachines (□j:1..NProc))
      UserMachine(i).SysCall(j)?s(t,parameter,) →
         flag(t) := true;
         queue action, for (i,j) on terminal t
    ]
```

5.7. Priority Revisited

We introduced preemptive commands for CSP specifically for handling exception and interrupt conditions which would generally be associated with a synchronized signal or message. The envisaged use in the model was for device interrupts, scheduler interrupts, and abortion of processes due to error conditions. In this regard, we feel that the style imposed by the preemptive commands is sufficiently expressive — that is, by nesting preemptive commands to define a hierarchy of handlers. We now comment on the adequacy of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

model with regard to the expression of process priority.

Specifically, we shall consider the required static priority of the scheduler over the supervisor. Clock has priority over Schedule in section 5.4 due to the semantics of the interrupt command. Thus, if Schedule is executing, a hardware clock device tick is serviced immediately and the guards of Schedule re-evaluated and rescheduling can occur if necessary. But, what if Schedule is not executing when the tick occurs?

Such problems did not exist in Hoare's original definition of CSP. Recall that CSP was intended for implementation on a network with a single process assigned to a single processor for which multiplexing does not exist.

Facilities to define static priorities have been included in languages such as Modula [Wirth77a], Modula-2 [Wirth83], SR [Andrews81b] and Ada [Ada82]. The adoption of a similar scheme using a **priority** clause for separate processes provides one method to define the required different relative static priorities for the scheduler and the supervisor.

Another solution lies in the interpretation given to a clock tick. If we assume the clock tick causes the required priority interrupt, then Clock will be assigned the processor. Now, by adopting the interpretation that this transfer of control to the clock handler has a stronger consequence and in fact resumes the scheduler, we may imply the required static ordering.

Within the model, we think that the latter interpretation is sufficient for the required task. However, using CSP as a general programming language with preemption may require additional language extensions for defining process priority. Another practical problem which we have not tackled concerns the association of hardware signals with CSP i/o commands.

5.8. In Summary

In this chapter, we have presented a model of a distributed UNIX kernel in CSP. A separation of machine usage was obtained by giving individual models

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

for user, file and terminal machines. The processes that were defined within each of these show the distribution of kernel functions at the chosen level of granularity. The individual models are not the only models that can be derived. This is exemplified by the two models for the terminal system.

Although in some examples, Hoare used dynamic processes in his model of communicating sequential processes [Hoare80a], we have not taken such an approach for user process creation. Instead, we have declared an array of virtual processes and system call processes. Consequently, the model of the supervisor shows the boundedness of user process creation within the UNIX kernel. The use of virtual user and system call processes makes explicit the reentrancy of the system call interface of the UNIX kernel implementation.

For file and terminal machines, multiplexing of actions necessary to perform requests occurs. Some of the required queue manipulation of the file and terminal systems has been briefly presented. The role of device handlers has been shown with the setting of flags to model wakeups on completion of data transfers. Such flags are then noticed by the appropriate system thus modelling the waiting that may be involved. A similar scheme could be used for sleeps and wakeups associated with obtaining system resources, such as i/o buffers, in a more detailed model.

One weakness of the model is concerned with relative process priority. Using preemptive commands, one may build up a hierarchy of handlers. However, only by taking a certain interpretation for transfer of control on the occurrence of a clock interrupt does the model fulfil its role in this respect.

By presenting the model of a distributed UNIX kernel, we have shown that CSP, with the extension for preemption, is suited to the task. A variety of applications of the preemptive commands have been used. Interestingly, none of the UNIX kernel structure as expressed in the programming language C, apart from the process and user data structures, appears in the model. This

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

phenomenon is due to the imposition of the particular style of the language that was used. In particular, a procedure-oriented approach is found in the standard UNIX kernel. It would appear that the use of CSP for the model corroborates Lauer and Needham's observations that a transformation from a procedure-oriented language to a message based language is possible (and vice versa).

In our application, the desire to model the higher level aspects of the distributed kernel, thus showing the interactions of processes in the system, led us to the adoption of a message-oriented language. The clarity of the model may depend upon the reader's familiarity with CSP. However, we venture to say it is at least more understandable than a listing of the kernel source code.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 6

Towards a Kernel Implementation

In this chapter we will present an approach to the implementation of the distributed UNIX kernel. Specific items of concern are

- residence of system tables and additional information contained therein,
- code requirements in the different machines.

The model given in chapter 5 was constrained by the constructs of CSP. Thus we modelled the reentrancy of the kernel with respect to system calls by a set of processes. Also, for the file system and terminal system machines, the model employed polling for requests. We commented that the implementation need not use these approaches. In particular, it would be convenient to use as much of the original C code [Kernighan78] provided the efficiency of the implementation is not compromised.

Licences, copyrights and non-disclosure agreements restrict the amount of information that can be disclosed regarding the UNIX kernel.[1] We shall draw on readily available articles, namely [Ritchie74] and [Thompson78], for much of the UNIX related information. Occasionally however, we shall refer to kernel procedures and tables for which the source code is the ultimate reference.

As it has not been the intention of this thesis to undertake an implementation, we shall delve into details only as far as is necessary to present our approach.

---

[1] A commentary by Lions on the standard version 6 UNIX implementation for PDP-11's [Lions77] has a limited distribution partly for this reason.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sections 6.1 to 6.3 address type, residence and capabilities for system tables concerning both the standard and distributed kernels. We discuss messages in section 6.4. Finally, we take a closer look at the table and code requirements for all machines in the distributed implementation (sections 6.4 to 6.8).

6.1. Standard Implementation Table Classification

Tables in the standard UNIX implementation fall into two classifications — kernel resident and per process resident. This division was introduced to facilitate swapping of user processes. Consequently, the per process resident tables contain information which is not required if the process is not active and swapped out of memory. Most tables are of the kernel resident type.

Although we do not intend to provide for swapping, we shall maintain the above division since the current design is quite suitable and in no way restricts a distributed implementation. No re-design of the process control structures is necessary. However, the distribution of the file system tables and other i/o associated tables and buffers will be necessary. Figure 6.1 shows the division between the classifications for the process control data structures and the file structures in the standard implementation.[2]

6.1.1. Kernel Resident

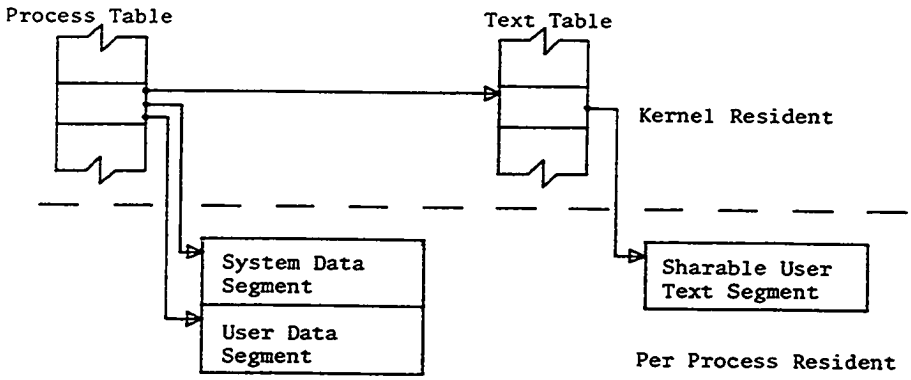
Such tables are compiled into the kernel and cannot be swapped. Included in these tables are the *proc* table and the *text* table containing process control information.[3] The *file* table and the *inode* table contain information regarding open files. Many other tables are associated with data buffering for both block devices (for example, the buffer pool) and character devices (for example, the *tty* table and *clist* buffers).

---

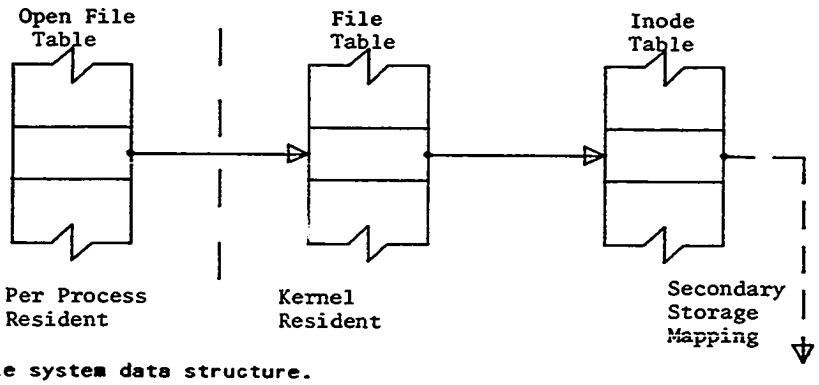
[2] These diagrams are similar to those in [Thompson78] pages 1934 and 1944.

[3] We shall use italics to signify tables and procedures in the kernel as for example, in the *file* table and the procedure *clock*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



(a) Process control data structure.



(b) File system data structure.

Figure 6.1. Standard residence of control and file system data structures.

6.1.2. Per Process Resident

Such tables are called "per user" tables in the literature and the memory allocated for them is sometimes referred to as the *u area* (after the pointer used in the kernel code). They can be swapped. Each resides in the user process data space but are not accessible directly by the user process, only by the kernel and as a consequence of a system call. Space is allocated on demand, that is at process creation (by forking), and includes a stack for the kernel when it is executing on behalf of the user. The size of the per process kernel stack need not be as large as at present (in the V7 PE3240 system it is 4 kbytes) due to the offloading of the file system, which can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

require considerable stack depth. The maximum stack size can be statically calculated by considering the worst case procedure call nesting.

6.2. Table Residence in Distributed Kernel

The adoption of the standard data structures for user process control means that each user machine will contain a data structure like that of figure 6.1(a), which will be used for local process control. Figure 6.2 shows the file system data structure corresponding to figure 6.1(b) and the table residence according to machine type. Mention will be made of the residence of other major data structures in later sections.

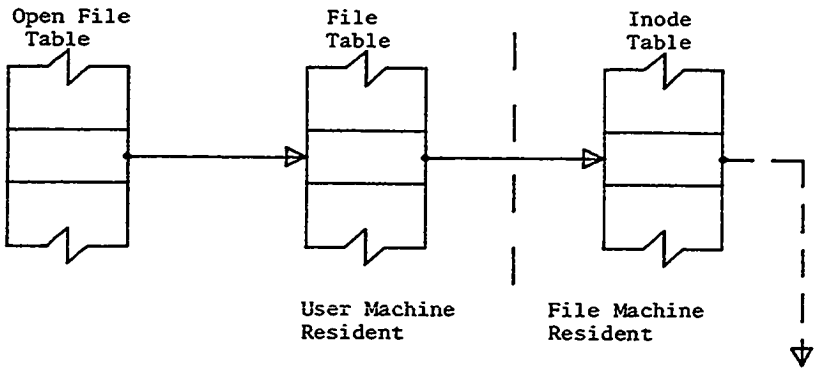


Figure 6.2. Residence of file system structures.

6.3. Capabilities

The standard system uses addresses, more specifically pointers to table elements, as capabilities. So for example, a pointer in the *file* table to an *inode* provides an address to follow for file information and also is deemed to be sufficient security-wise. A distributed implementation must also provide a machine identifier as part of the *inode* pointer. This does not necessarily provide the same degree of security since masquerading may be possible on the message transport medium. However, it does support the same functionality.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In the rest of this chapter, we will assume that any capability that is passed across machines contains the associated machine identifier. Reference to a pointer in the distributed implementation will be taken to mean the address part of the capability as for the standard implementation.

Notice that because of the consistent and disciplined use of header files defining system data structures in the system source code, additional capability information can be inserted into these "#include" type files.

6.4. Messages

We shall make the following assumptions for messages passing between machines:

- Messages shall have a fixed length header/data block of the order of 64 bytes in length.
- Messages may also contain a variable length additional data block up to 1 kbytes in length.

We base our assumptions both on existing remote procedure call mechanisms and on consideration of the information required in this particular case [Vines82]. We will not consider reliability and the like. Notice that an Ethernet would satisfy the above requirements.

All headers contain information for the transport mechanism (network header) and any protocol information including a length for the possibly non-existent additional data block. The data block contains information for the given request or reply. For requests, the type, a reply (wakeup) capability and any other required capabilities would be included in the data block. For replies, the status which is always returned from system calls would be contained in the data block. Such information is easily held in 64 bytes. That is, the header/data block deliberately has been given a length sufficient to specify most types of messages without the need for an additional data block.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

For file and terminal transfers, and other messages requiring more information, the variable length additional data block is used. The 1 kbyte maximum block size has been selected to correspond to the logical disk block size given in chapter 4 for increased throughput.

We will illustrate with some examples.

(1) File Open Request (relative to the current working directory):

The data block would contain OPEN, Read/Write/Update, process user and group identifiers, and *inode* capability of current working directory. The *inode* capability would have been obtained from a previous change directory reply. The additional data block would contain a zero terminated byte string of the form "tom/dick/.../and/harry". Notice that the limitation of the additional data block size, restricts the length of path names to 1 kbytes also.

(2) Terminal I/O Control Write:

The data block would contain IOCTL, Write, terminal capability, control type and the parameter (which is typically 6 bytes), thus including all data for the request.

(3) File Read Request:

The data block would contain READ, *inode* capability, byte position and byte count.

(4) File Read Reply:

The data block would contain the status (actual number of bytes read). The additional data block would contain that many bytes.

Message handlers must reside in all machines. They will be different for each machine type and will be required to use knowledge of processes and some times of request types, thereby making them different from the standard device handlers. The following sections will give requirements for each depending on the machine type.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.5. User Machines

The user process interface to the kernel remains the same as for the standard implementation. The typical "low core" assembly language code to catch traps (SVC's) by user processes, switch to kernel mode, change stacks and effect a procedure call, will be used. The data structures associated with user process control also remain the same.

6.5.1. Kernel Tables

The *proc* and *user* tables are as for the standard implementation.

An entry in the *file* table contains an *inode* capability, a count of processes with this open file and a limited amount of information about the file, including the type of file and type of open. Depending on the file type, the information could include the current position of file (hence, a seek is a local operation), the terminal capability, or the pointer to a pipe and the head and tail pointers.

Message buffers are a critical resource and the number that need be allocated is a tunable configuration parameter. Whether they are used for message headers and data blocks depends on the network hardware. They will be required for pipes, thus making pipe operations local.

6.5.2. Per Process Tables

The *user* table contains the *open file* table which points to a *file* table entry for this user machine. Forked processes inherit open files; when a count in the *file* table goes to zero the file is closed, via a message to the appropriate file machine to release the *inode*.

6.5.3. Scheduling and System Call Code

For scheduling, the context switch procedure *swtch*, the process control procedures *sleep*, *wakeup* and *signal*, and the *clock* interrupt procedure are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

required.

All process related system calls are implemented locally in the user machine in addition to *seek*, *dup* and *pipe*. The other system calls require sufficient code to prepare a message for the appropriate machine, however, some may not require a message to be sent depending on the context.

6.5.4. Message Handler

Both synchronous and asynchronous messages, with respect to user processes can be expected. In the first case, a reply to a request (as a result of a system call) always will occur. The second case occurs when the terminal handler receives an interrupt or quit signal by the user at the keyboard. Here, all processes in the process group associated with that user on that terminal must be signalled. For this purpose, the open read call can be used to inform the terminal machine of the appropriate user machine to send the message to.

So the message handler must send requests and demultiplex the replies using information in the fixed length header/data block to decide whether to do a *wakeup* or a *signal* procedure call.

6.6. File Machines

As has been mentioned, the model in the previous chapter polled for message requests and for completion of requests, whereas the standard implementation uses kernel code reentrancy to advantage with a separate instance for each system call in progress.

Often, system call requests of the standard file system cause a sequence of sleeps for events, such as, waiting for an *inode* or a buffer resource, or for i/o completion. The kernel stack, in the *u area*, contains the context of this kernel process. A call to *sleep* suspends the process and causes a context switch. A subsequent *wakeup* call (by another process) prepares the sleeping

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

process for rescheduling.

There are a considerable number of events which may be waited upon. There are also many contexts in which they may be awaited, like in the different system calls or at different parts of the code for the same system call. In the standard implementation, the kernel stack for a process contains this context information in the form of procedure parameters and local variables.

If polling were used in the distributed file system kernel, all the different contexts would have to be cases in the polling loop and pertinent variable values would need to be explicitly saved. We consider this to require too much re-coding for no additional return.

Instead, we opt for the approach used in the standard implementation which effectively has separate instantiations for each current file system request. In support of this approach, we note the following:

- Most of the system call code remains the same.
- Device handler code remains the same.
- Context switching is easy and has minimal overheads as the file machine always executes in kernel mode.
- Variables (local and parameter) are saved in the instantiation's stack.

6.6.1. Server Processes and Message Handler

We propose that a set of server processes be allocated to perform the system call requests and to establish the reply message. Each server has its own stack, message header/data space, and executes *reentrant code* to perform the requested system call. As such, the set of servers is another critical resource and is tunable at configuration time.

Basically, a server sleeps until awoken by the message handler which will have put a request in its data space. The server is then multiplexed with other servers, however it only relinquishes the processor when it sleeps (for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a resource or a data transfer) or when it is idle again. A server always will assemble a reply message and call the message handler to send it.

The file machine message handler receives requests, finds a server and, some time later, sends a reply. The message handler can access the sleep queue of idle servers for this purpose. To avoid race inefficiencies associated with the semantics of wakeups, the message handler can call *unsleep* for the selected process.

6.6.2. Tables and Code

The remainder of the tables and code in the file machines is very much like that in the standard implementation. We use the same *inode* table and buffer pools.

A simplified process table, *sproc* say, for server process state information is necessary. Context switching requires a minimum of effort; saving or restoring registers in the *sproc* table. Simple first-in-first-out scheduling on the run-queue can be used whereby wakeups add to the end of the queue.

The same code implementing the distributed part of the system calls can be used with little change. The main changes are to the procedures to move data between user and kernel space, which become simpler, and references to the *u area*. Examination of the source code for the file system, about 2500 lines (excluding "#include" files), shows that only 150, that is 6%, refer to the *u area*.

The other device handlers are standard. A local clock handler is required to maintain the current time (which need only be updated each second) so that the time of access can be updated in *inodes*.

6.7. File Manager

The file manager is required for the following system calls: **creat, open, unlink, stat** and **access**. Several alternatives exist for the distributed

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implementation.

Two types of file name paths exist — absolute and relative. An absolute path name starts with a "/", referring to the root directory of the distributed file system. Other path names are relative to the current working directory.[4]

The Newcastle Connection network [Brownbridge82] uses paths such as "/../OtherMachine/And/So/On/..." to access file systems on other machines by referencing above the root directory, as it were. If we were to use this approach, each file machine could pass the system call on to a different machine when a directory was reached which referred to that machine. Initial requests would be passed to the file machine holding the absolute system root directory or the current working directory.

At the other extreme, we could make the simplification that a relative path name is not permitted to cross machines, thus considerably reducing the work required of the file manager. Absolute path names could then be matched against a table of mounted file systems and the longest matching initial substring could be stripped off and the associated *inode* capability substituted as the relative directory. Since relative path names cannot cross file machines, a relative reference can be passed directly to the machine indicated by the current working directory *inode* capability.

The advantages of such a restricted scheme lie in efficiency. However, a major disadvantage is that logically equivalent path names need not refer to the same file due to file systems being mounted across machines.

A compromise can be reached whereby relative path names which are found to reference an *inode* which is on another machine can pass the remainder of the request back to the requesting process which can in turn pass it on to the new file machine. If a direct communications path existed between all file

[4] See [Richie74] for a description of the conventional use of "/", "." and "..", in file names.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

machines, then the request could be passed on directly.

This compromise solution gives uniform semantics to file names. Absolute path names can be considered as relative to the system root directory. The system root directory is assigned at the time of bootstrap and is the root directory on a particular file machine. So absolute path names are passed to the machine holding the system root and relative path names are passed to the machine holding the current working directory. Should a directory be reached which is on another machine as the path is being followed, the request containing the remaining part of the path is passed to that machine. This can be implemented by augmenting the *namei* procedure which follows path names.

The **mount** system call must be altered to allow the mounting of a file system on a directory which exists on a different file machine. Both machines need to make additional entries in their *mount* tables which contain *inode* capabilities for redirecting requests to another file system (and possibly machine). It would be sensible, but not necessary, to restrict across machine mounts only to the root directories of file machines.

6.7.1. For Efficiency

One further point concerning efficiency is worthy of mention here. The above scheme could introduce a bottleneck at the file machine containing the system root directory. In [LauerH78], Lauer and Needham report a similar situation in the CAP system, for which multiple file managers, that shared code, were introduced to overcome the problem.

We can use a similar approach by having a file manager in each user machine. The **mount** system call must also notify the user machines of the absolute path of the mounted file system, which the user machine records in a system mount table, *smount* say. File requests using path names can use this table to direct the request to the appropriate machine. The initial matching substring can be stripped off or an offset included in the fixed length data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

block. In our experience, relative path names rarely cross file system boundaries. So, although we suggest the support of redirection of requests, we believe it will rarely be used and the *smount* table scheme would prove to be quite efficient.

6.8. Terminal Machines

Terminal handling tends to involve a considerable amount of processing per character. Contrasted against this is the relatively slow nature of terminal devices. However, with several terminals doing say 9600 baud output concurrently, the system soon would run out of processing power to service all interrupts on demand. Also, the intelligence of the hardware interface can make marked difference to the number of interrupts that need be handled.

The standard terminal interrupt handler does as much work as possible, such as starting more output if there is any, and putting an input character on the input queue, arranging for it to be echoed and checking if a read request has been satisfied.

Basically two approaches have been used in versions of UNIX terminal handlers — processing interrupts caused by each terminal (used in standard UNIX) and polling terminals after a clock interrupt (used in Berkeley 4.2BSD). The first suits systems where there is relatively little terminal i/o and the second where the load is heavy. Both approaches require interrupts to trigger terminal processing.

However, we are proposing a distributed system where the terminal machine is dedicated to terminal i/o. Unlike the file system, there are only a few cases that need be considered in a polling loop providing terminal servicing. For any terminal, or perhaps group of terminals depending on the hardware, the cases are:

1. character ready for input,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2. character successfully output,

3. i/o control to device successful, and

4. i/o control from device (for example, off-line) available.

After any of these events, the polling loop can determine what action to take in a similar way to the standard implementation.

The advantages for polling terminals are:

- No context switching is required; pertinent data is in the *tty* table (containing terminal state information) and *clist* buffers (containing character lists).
- The system cannot be overloaded with terminal interrupts.

To support polling in the terminal machine, some of the standard implementation code must be re-written. We examined the source code for the terminal support procedures and found that of 1500 lines (excluding "#include" files) only some 200, that is 13%, require changing. This is mainly due to degree of parameterization that occurs in these procedures. In addition, the polling process and the message handler must be written. We feel that this effort would be worthwhile and opt for polling the terminals.

The message handler must:

1. Place the data for write requests on the appropriate output queue.

2. Place input requests on the appropriate input request queue.

3. Place IOCTL requests on the appropriate ioctl request queue.

By appropriate, we mean for the associated terminal. The last two queues mentioned in the requirements are not supported in the standard kernel. The polling process must assemble a reply message and call the message handler to have it sent.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.9. In Summary

In this chapter, we have presented our preferred approach for a distributed kernel implementation. A major objective was to use as much of the original C code as possible without compromising efficiency. In so doing we have shown that the standard kernel is suited to dissection for distribution. This is particularly due to the high degree of parameterization throughout the file and terminal system source code.

The main changes to existing code in the file and terminal systems are for references to the *u area*, involving less than 5% of that code. The new code that is required includes message handlers on all machines and for

User machines - file manager, dissected system calls.

File machines - file manager, server processes, simplified scheduler.

Terminal machines - polling process.

Certainly we have not considered all the details required for a distributed implementation. We have omitted some details completely, for example, the special file "/dev/kmem" which provides access to kernel memory. We believe that sufficient detail has been given to provide an understanding of the approach and to convince the reader that the task does not involve an excessive amount of work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER 7

Conclusions

We have presented a design for a distributed UNIX kernel and a model of the distribution in CSP. In this regard, we support the two thrusts of the thesis — that the UNIX kernel may be partitioned and distributed with the expectation of providing more processing power, and that CSP, albeit with extensions, may be used to model the distribution at a functional level.

Now, we give a summary of the work leading to this conclusion and review the contribution that we make to the field of computer science.

7.1. Distribution of UNIX Kernel

A distribution of the UNIX kernel over a local network has been presented. The kernel includes all code which would normally execute in supervisor or privileged mode, that is, code implementing process switching, memory management, supervisor calls and device control.

The basic aim for distribution was to unload kernel processing from machines that execute user processes onto machines dedicated to kernel functions. Thus system machines containing distributed kernel code can support multiple user machines.

The next problem undertaken was to decide where the division of labour should occur. User processes enter the kernel via system calls. These provide a convenient point at which to capture user requests and distribute the kernel. To assist in making the decision a particular Version 7 UNIX implementation was examined. With some 88% of system calls being related to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

input/output, the use of front and backend machines seems an obvious approach to take. This is further supported by 32% terminal read/write and 20% disk read/write system call distributions.

The other system calls are process management related. With the comparative high cost of message passing that would be necessary to support process management in a machine remote from those executing user processes, local process management provides the solution. Hence, user machines implement all process management related system calls and may call upon the remote backend to assist in the program lookup and subsequent reading of binary code for loading in exec system calls. In fact, the majority of these system calls exist for security reasons and are quite cheap in their implementation.

For the i/o related system calls, a little initial processing must be done on the user machine. Reads/writes must be diverted to frontend and backend machines or processed locally. Different capabilities are required for different system calls. For example, inode pointers for read/writes, effective-user-id for opens and so on. Hence, code must be executed on the user machine to gather these capabilities into a message for remote execution.

The frontend or terminal machine handles the read/write and device control system calls associated with character special files for terminals. The backend consists of at least one file machine supporting block (structured) i/o. Both terminal and file machines require capabilities to be passed in addition to information supplied for the system call by the user process, and both perform the remainder of the system call processing. The file machines may also return capabilities.

The user machines are constrained such that forking is local and no swapping is supported. Local forking means that all process management is local. The only external consideration is the controlling terminal interrupt signal. The system initialization procedure can be used to establish

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

associations between specific terminals and specific user machines (probably via an additional ioctl call). The removal of support for swapping further simplifies user machine kernels. The assumption being that memory is cheap and therefore large.

A minimal system could contain one terminal machine connected to a set of about 8-9 user machines which are in turn connected to one file machine. Suitable hardware for the network would be a ring (or Ethernet) for terminal machine to user machine connectivity and an Ethernet for user machines to file machine. This mix of ring and Ethernet is based on the expected traffic load and message size between machines.

Of course the number of user machines depends on the application environment. The figure of 8-9 was obtained for a job mix at one installation. For installations that are typically computation bound, this could be increased, and decreased for i/o bound installations. It is this flexibility that makes the study of a specific system's characteristics valid for application to other installation job mixes. In addition, other high bandwidth bus or channel structures could also be used in the network. The criterion for selection is governed by speed requirements.

An alternative exists in the case of i/o bound installations to increase the number of file machines. It cannot be expected that an incremental gain in i/o throughput will result from an incremental increase in file machines. This of course is due to additional load on the network. However, judicious choice of mounted file systems for particular application programs such that file referencing is confined to the machines associated with the current working directory may assist greatly in this endeavour.

Perhaps the major advantage of a distribution such as that which is proposed is its versatility with regard to tailoring to specific general loads. The sharing of resources, that is i/o devices and processors, is the other advantage. The approach usually taken, of having one operating system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

request another for file activity, is unnecessary in the kernel distribution. File machines are an integral part of the distributed kernel. Similarly, output directed to terminals logically associated with a "login" on another user machine may be considered as being handled by another part of the distributed kernel.

7.1.1. Towards Implementation

We have indicated how an implementation may be undertaken. Consideration of the distribution of system tables and the necessary additional information that is required has been given. The use of existing source code has been discussed. The source code procedures in the standard kernel for the file system and terminal system are well parameterized. In this regard, the suggested distribution is quite amenable to the use of the majority (about 90%) of the existing C code.

Message handlers, which differ among user, terminal and file machines, must be written, and system call code must be dissected. We proposed that a set of server processes, using reentrant code, be used on each file machine, and that polling be used to service terminals on the terminal machine.

7.2. Model of a Distributed UNIX Kernel

The author, and one may assume many people before and to follow, gained knowledge of the kernel processes and the interactions among them by poring over the source code. From this, the intricacies and implications associated with transfers of control, like calls to procedures which never return or do not return to where one might think, can be worked out.

The model is intended to show clearly and precisely the distribution of the kernel. In addition it presents a view of what the implementation is doing at a logical level. A model could similarly be developed for standard UNIX.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The division and distribution of processes and some system tables is presented in the model. The functionality of distribution is represented and as such, the intended purpose of the model is satisfied. Much of the discussion regarding the duality or interchangeability of operating system structures, like coroutines, remote procedure calls and message processing, is epitomized in the relationship between the model and the implementation.

On the one hand, we have a message based model. On the other hand is an implementation that involves privileged instruction traps for system call entry, coroutines for user process execution and scheduling, messages between processes, remote procedure calls by process instantiation in file machines and queues that are manipulated as a consequence of interrupts in all machines.

The model does not show all these properties. The efficiency requirements of an operating system prohibit the general message constructs such as those applied in the model from directly being used in an implementation.

7.3. Priority in CSP

In CSP, processes may combine to perform a task, like the task forces of STAROS and Medusa. For a network of small processors, each dedicated to one process, the notion of process priority is not required. Moreover, delays for synchronized message transfer may be acceptable since concurrency is provided for as much processing as the program structure allows. Even the debate of whether to permit both input and output guards may not be applicable to this situation. The additional transfers involved in the use of an input guard and a subsequent output command to achieve the effect of an output guard, may not increase the execution time significantly compared with waiting for synchronization. However, in a network where processes are multiplexed, efficiency must be considered. The preemptive commands provide mechanisms to assist in this.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Typical processors support hardware trap and device interrupt mechanisms. The **except** and **interrupt** commands provide natural constructs for dealing with such mechanisms. Although these commands were introduced in this thesis primarily for the operating system application, they can also be applicable in a general situation.

7.3.1. General Application

Situations can be envisaged where solutions are improved by associating a priority ordering with alternatives which involve i/o guards. The use of the **else** construct presented in this thesis is suitable for programming such applications.

Languages exist (for example, CLU, Mesa, PL/1 and Ada) which allow programs to catch exceptions such as arithmetic errors. The **except** command could be similarly applied. In fact, the **interrupt** command could be used to correct the problem and to resume. Other applications, such as the parallel iterative solution of a set of equations could benefit from the ability of one process to preempt another.

7.3.2. Real-Time Application

The existence of facilities for preemption can be vital in real-time applications such as control systems, unless one can guarantee service within known time constraints [Wirth77b].

A classic real-time process is the interrupt handler. To take Modula-2 as an example, modules can be used to provide mutually exclusive queue manipulation, waiting for completion and interrupt process definition. Multiplexing of processes is assumed in such implementations.

We have introduced the interrupt construct for such applications. The binding of two such processes is suited to a multiplexed implementation. An ideal example for multiplexing is P **interrupt** Q, where Q provides asynchronous

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

buffering for P, but Q uses synchronized message passing with its communication partner.

The novel facility that the preemptive commands introduce is the ability to schedule processes. Altering the priority of processes that are multiplexed may be done using interrupt processes that wait for a message from the scheduler after an interrupt message has been received. Other processes in the multiplexed implementation may be executed when the wait occurs. Another example of dynamic process priority is where identical processes are defined for each priority level, but where initialization of mutually exclusive boolean guards determines the priority for a given execution.

7.3.3. Control Message Scheme

A scheme for the implementation of i/o commands in the presence of preemption has been given. The implementation does not support generalized i/o guards. A general master/slave relationship is available through the use of non-guard i/o commands. Preemptive guards can be applied to provide a facility similar to generalized i/o guards.

In the simplest case, where there is no preemption, the implementation is efficient and requires no additional control signal for confirmation by the notifying (non-guard) i/o command. In the presence of preemption at most three additional control signals may form the overhead for any one synchronized transfer. Hence, the scheme is efficient in respect to control signals.

The investigation of Nelson into remote procedure calls using microcode on Dorado machines suggest that for a 10 Mbit/sec ethernet, a remote call with no parameters could be accomplished in about 150 μsec and control messages sent and received in about 50 μsec. This is an encouraging use of message passing mechanisms and adds weight to preemptive constructs being suitable in a real-time network.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

7.4. Model of CSP

The definition of semantics for any programming language is of paramount importance. The approach taken in this thesis was to build upon an existing model using traces which is due to Hoare. Some definitions in Hoare's trace model were altered such that distributed termination was no longer supported. Preemptive commands were also defined.

The model was then extended to provide operational semantics for CSP processes executing in parallel within an environment that depends upon the values of variables resulting from individual histories and channels controlling communication. Importantly, no notion of the global environment (of the composition of all processes) was required. The channel ready sets for a process wishing to communicate was all that one process required to define its external environment at a given time.

The execution function was used to represent delays, progress towards successful or unsuccessful termination, and deadlock. Thus the temporal nature of execution was presented operationally. Apart from making explicit the priority associated with preemption, the execution function also provided operational semantics for termination of repetition.

7.5. Contribution

This thesis makes contributions in two areas of computer science — firstly, as a study for a distributed UNIX kernel, and secondly, in the definition of operational semantics for CSP and in its use for real-time applications.

With regard to kernel distribution, this thesis contributes to the understanding of why the kernel should be distributed in the proposed manner and how such a distribution can be implemented. For this purpose, typical statistics relating to user and system times, i/o loads put on the system and calls to the system by user processes were presented. Such information also contributes to the understanding of resource utilization by the standard UNIX

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

kernel.

The resulting network consists of user, file and terminal machines. The ability of commercially available hardware to satisfy the required message transport mechanism is also considered. As such, the distribution scheme is also applicable to other systems for which a similar division of labour is possible.

The distribution is further augmented by a model of the kernel over the network which presents the functional requirements of the implementation. The functional model contributes not only to the appreciation of the processes and their interactions within a distributed implementation, but also to the standard UNIX kernel in which those processes exist whether as coroutines or as bona fide processes.

With regard to CSP, this thesis contributes to its use as a general purpose message based language in real-time applications. The introduction of priority for alternatives and handlers for exceptions and interrupts provides the necessary facilities.

The operational semantics of CSP including the extensions for preemption contribute beyond the existing model of processes as sets of traces. The definition of execution within an environment takes a novel approach to provide for the temporal qualities of waiting for synchronization, deadlock and progress. The external environment as a function of the channel ready sets narrows the need for global environment considerations down to the channels which act as arbiters.

The final contribution to CSP as a general purpose language is in presenting a scheme for the implementation of message passing in the presence of preemption. The scheme also is suited to situations without preemption. It is shown that with preemptive commands, one does not need to support generalized i/o guards. Further, the master/slave relationship of processes is adequately expressed by the use of non-guard and guard i/o commands

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

respectively. Implementation is greatly simplified by the requirement that non-guard i/o commands must wait for synchronization and therefore can initiate the synchronization.

7.6. Further Work

Based on the ideas presented in this thesis an implementation of a distributed UNIX kernel may be undertaken. A preliminary implementation on a minimal configuration with one terminal machine, one file machine and a set of user machines is suggested. Measurements can then be taken to determine the limits of both the file machine and the terminal machine in servicing requests.

The next step is to add another file machine. The increase of throughput efficiency may then be examined. One would never expect an incremental increase. However, if this can be almost reached, then the distributed kernel scheme could be extended to provide direct access to file systems in a much larger, but still very local, network. This is an alternative worth investigating as opposed to the approach which is now becoming more popular in providing remote file access between remote operating systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bibliography

[Ada83] The Programming Language Ada Reference Manual. ANSI/Mil-Std-1815A. *Lecture Notes in Computer Science, Volume 155*, Springer-Verlag, 1983.

[Almes79] G.T. Almes and E.D. Lazowska. The Behaviour of Ethernet-Like Computer Communications Networks. In *Proceedings 7th Symposium on Operating Systems Principles*, pages 66-81, December 1979.

[Andrews81a] G.R. Andrews. Synchronizing Resources. *ACM Transactions on Programming Languages and Systems*, 3(4), 405-430, October 1981.

[Andrews81b] G.R. Andrews. SR: A Language for Distributed Programming. Technical Report 81-14, Department of Computer Science, University of Arizona, October 1981.

[Andrews81c] G.R. Andrews. The Design and Implementation of SR. Technical Report 81-15, Department of Computer Science, University of Arizona, October 1981.

[Andrews83] G.R. Andrews and F.B. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1), 3-43, March 1983.

[Antonelli80] C.J. Antonelli, L.S. Hamilton, P.M. Lu, J.J. Wallace and K. Yueh. SDS/NET - An Interactive Distributed Operating System. In *Proceedings IEEE Fall COMPCON80*, pages 487-493, September 1980.

[Apt80] K.R. Apt, N. Francez and W.P. de Roever. A Proof System for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3), 359-385, July 1980.

[Barak80] A.B. Barak and A. Shapir. UNIX with Satellite Processors. *Software Practice and Experience*, 10(5), 383-392, May 1980.

[Bernstein80] A.J. Bernstein. Output Guards and Nondeterminism in Communicating Sequential Processes. *ACM Transactions on Programmming Languages and Systems*, 2(2), 234-238, April 1980.

[Birrell80] A.D. Birrell and R.M. Needham. A Universal File Server. *IEEE Transactions on Software Engineering*, SE-6(5), 450-453, September 1980.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Blair82] G.S. Blair and D. Shepherd.
A Performance Comparison of Ethernet and the Cambridge Digital Communication Ring.
Computer Networks, 6(2), 105-113, 1982.

[BrinchHansen70] P. Brinch Hansen.
The Nucleus of a Multiprogramming System.
Communications of the ACM, 13(4), 238-241+250, April 1970.

[BrinchHansen72] P. Brinch Hansen.
Structured Multiprogramming.
Communications of the ACM, 15(7), 574-577, July 1972.

[BrinchHansen75] P. Brinch Hansen.
The Programming Language Concurrent Pascal.
IEEE Transactions on Software Engineering, SE-1(2), 199-207, June 1975.

[Brownbridge82] D.R. Brownbridge, L.F. Marshall and B. Randell.
The Newcastle Connection or UNIXes of the World Unite.
Software Practice and Experience, 12(12), 1147-1162, December 1982.

[Buckley83] G.N. Buckley and A. Silberschatz.
An Effective Implementation for the Generalized Input-Output Construct of CSP.
ACM Transactions on Programming Languages and Systems, 5(2), 223-235, April 1983.

[Campbell74] R.H. Campbell and A.N. Habermann.
The Specification of Process Synchronization by Path Expressions.
In *Lecture Notes in Computer Science, Volume 16*, pages 89-102, Springer-Verlag, 1974.

[Campbell76] R.H. Campbell.
Path Expressions: A Technique for Specifying Process Synchronization.
Ph.D. Thesis, University of Newcastle Upon Tyne, 1976.

[Cheriton79] D.R Cheriton, M.A. Malcolm, I.S. Melen and G.R. Sager.
Thoth, a Portable Real-Time Operating System.
Communications of the ACM, 22(2), 105-115, February 1979.

[Cheriton83a] D.R Cheriton and W. Zwaenepoel.
The Distributed V Kernel and its Performance for Diskless Workstations.
In *Proceedings 9th Symposium on Operating Systems Principles, ACM Operating Systems Review*, 17(5), 129-140, October 1983.

[Cheriton83b] D.R Cheriton.
Local Networking and Internetworking in the V-System.
In *Proceedings 8th Data Communications Symposium, ACM Computer Communications Review*, 13(4), 9-16, October 1983.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Chesson75] G.L. Chesson.
The Network UNIX System.
In *Proceedings 5th Symposium on Operating Systems Principles, ACM Operating Systems Review*, 9(5), 60-66, 1975.

[Collinson82] R.P.A. Collinson.
The Cambridge Ring and UNIX.
Software Practice and Experience, 12(6), 583-594, June 1982.

[Dijkstra68] E.W. Dijkstra.
Co-operating Sequential Processes.
In *Programming Languages*, pages 43-112, F. Genuys (Ed.), Academic Press, 1968.

[Dijkstra75] E.W. Dijkstra.
Guarded Commands, Nondeterminancy and Formal Derivation of Programs.
Communications of the ACM, 18(8), 453-457, August 1975.

[Dix83a] T.I. Dix.
Towards a Distributed UNIX Kernel.
In *Proceedings 6th Australian Computer Science Conference*, pages 11-20, February 1983.

[Dix83b] T.I. Dix.
Exceptions and Interrupts in CSP.
Science of Computer Programming, 3(2), 189-204, August 1983.

[Dix83c] T.I. Dix.
Preemptive Guards in CSP.
Technical Report 83/5, Department of Computer Science, University of Melbourne, August 1983.

[Dix84] T.I. Dix.
A Model of a Distributed UNIX Kernel.
In *Proceedings 7th Australian Computer Science Conference*, pages 23/1-9, February 1984.

[Deutch65] L.P. Deutch and B.W. Lampson.
SDS 930 Time-Sharing System Preliminary Reference Manual.
Project GENIE, Document 30.10.10, University of California at Berkeley, April 1965.

[Ethernet80] Ethernet joint report Digital, Intel and Xerox.
The Ethernet: A Local Area Network, Data Link Layer and Physical Layer Specifications.
September 1980. Reprinted in *ACM Computer Communication Review*, 11(3), 20-66, July 1981.

[Franta81] W.R. Franta, E.D. Jensen, R.Y. Kain and G.D. Marshall.
Real-Time Distributed Computer Systems.
In *Advances in Computer Systems, Volume 20*, pages 39-82, Academic Press, 1981.

[Francez79] N. Francez, C.A.R. Hoare, D.J. Lehman and W.P. de Roever.
Semantics of Nondeterminism Concurrency and Communication.
Computer and Systems Sciences, 19, 290-308, 1979.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Francez80] N. Francez.
Distributed Termination.
ACM Transactions on Programming Languages and Systems, 2(1), 42-55, January 1980.

[Francez83] N. Francez.
Extended Naming Conventions for Communicating Processes.
Science of Computer Programming, 3(1), 101-114, April 1983.

[Gehringer82] E.F. Gehringer, A.K. Jones and Z.Z. Segall.
The Cm* Testbed.
Computer, 15(10), 40-53, October 1982.

[Gobal82] G.H. Gobal and M.H. Marsh.
A Dual Processor VAX 11/780.
In *Proceedings 9th Annual Symposium on Computer Architecture*, *ACM Computer Architecture News*, 10(3), 291-298, April 1982.

[Hehner83] E.C.R. Hehner and C.A.R. Hoare.
A More Complete Model of Communicating Processes.
Theoretical Computer Science, 26(1,2), 105-120, September 1983.

[Hoare72] C.A.R. Hoare.
Towards a Theory of Parallel Programming.
In *Operating Systems Techniques*, pages 61-71, Academic Press, 1972.

[Hoare74] C.A.R. Hoare.
Monitors: An Operating System Structuring Concept.
Communications of the ACM, 17(10), 549-557, October 1974.

[Hoare78] C.A.R. Hoare.
Communicating Sequential Processes.
Communications of the ACM, 21(8), 666-677, August 1978.

[Hoare80a] C.A.R. Hoare.
A Model for Communicating Sequential Processes.
Proceedings of Symposium on Communicating Sequential Processes, University of Woolongong, Department of Computing Science Preprint 80-1, March 1980. Also mainly contained in Technical Monograph PRG-22, Programming Research Group, Oxford University Computing Laboratory, June 1981.

[Hoare80b] C.A.R. Hoare.
Additional Notes on A Model for Communicating Sequential Processes.
Proceedings of Symposium on Communicating Sequential Processes, University of Woolongong, Department of Computing Science Preprint 80-3, March 1980.

[Hoare81a] C.A.R. Hoare, S.D. Brookes and A.W. Roscoe.
A Theory of Communicating Sequential Processes.
Technical Monograph PRG-16, Programming Research Group, Oxford University Computing Laboratory, May 1981.

[Hoare81b] C.A.R. Hoare.
A Calculus of Total Correctness for Communicating Processes.
Science of Computer Programming, 1(2), 49-72, October 1981.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Holmgren78] S.F. Holmgren.
Resource Sharing UNIX.
In *Proceedings IEEE Fall COMPCON78*, pages 302-305, September 1978.

[Jazayeri80] M. Jazayeri, C. Ghezzi, D. Hoffman, D. Middleton and M. Smotherman.
CSP/80: A Language for Communicating Sequential Processes.
In *Proceedings IEEE Fall COMPCON80*, pages 736-740, September 1980.

[Jensen78] E.D. Jensen.
The Honeywell Experimental Distributed Processor - An Overview.
Computer, 11(1), 28-39, January 1978.

[Jones79] A.K. Jones, R.J.Chansler, I. Durham, K. Schwans and S.R. Vegdahl.
STAROS, a Multiprocessor Operating System for the Support of Task Forces.
In *Proceedings 7th Symposium on Operating Systems Principles*, pages 117-127, December 1979.

[Kernighan78] B.W. Kernighan and D.M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[Kieburtz79] R.B. Kieburtz and A. Silberschatz.
Comments on Communicating Sequential Processes.
ACM Transactions on Programmming Languages and Systems, 1(2), 218-225, October 1979.

[LauerH78] H.C. Lauer and R.M. Needham.
On the Duality of Operating System Structures.
In *Proceedings 2nd International Symposium on Operating Systems*, IRIA, October 1978. Reprinted in *ACM Operating Systems Review*, 13(2), 3-19, April 1979.

[LauerP78] P.E. Lauer and M.W. Shields.
Abstract Specification of Resource Accessing Disciplines: Adequacy, Starvation, Priority and Interrupts.
ACM SIGPLAN Notices, 13(12), 41-59, December 1978.

[LevinG81] G.M. Levin and D. Gries.
A Proof Technique for Communicating Sequential Processes.
Acta Informatica, 15(3), 281-302, 1981.

[LevinR77] R. Levin.
Program Structures for Exceptional Condition Handling.
Ph.D. Thesis, Carnegie-Mellon University, 1977.

[Lions77] J. Lions.
A Commentary on the UNIX Operating System.
Course booklet, Department of Computer Science, University of New South Wales, 1977.

[Liskov79] P.H. Liskov and A. Snyder.
Exception Handling in CLU.
IEEE Transactions on Software Engineering, SE-5(6), 546-558, November 1979.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Luderer81] G.W.R. Luderer, H. Che, J.P. Haggerty, P.A. Kirslis and W.T. Marshall.
A Distributed UNIX System Based on a Virtual Circuit Switch.
In *Proceedings 8th Symposium on Operating Systems Principles, ACM Operating Systems Review*, 15(5), 160-168, December 1981.

[Lycklama78] H. Lycklama and C. Christensen.
A Minicomputer Satellite Processor System.
Bell System Technical Journal, 57(6), 2103-2113, July 1978.

[McKusick83] M.K. McKusick, W.N. Joy, S.J. Leffler and R.S. Fabry.
A Fast File System for UNIX.
Technical Report 147, Computer Science Division, University of California at Berkeley, revised July 1983.

[Metcalfe76] R.M. Metcalfe and D.R. Boggs.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM, 19(7), 395-376, July 1976.

[Misra81] J. Misra and K.M. Chandy.
Proofs of Networks of Processes.
IEEE Transactions on Software Engineering, SE-7(4), 417-426, July 1981.

[Misra82] J. Misra and K.M. Chandy.
Termination Detection of Diffusing Computations in Communicating Sequential Processes.
ACM Transactions on Programming Languages and Systems, 4(1), 37-43, January 1982.

[Nelson81] B.J. Nelson.
Remote Procedure Call.
Ph.D. Thesis, Carnegie-Mellon University, 1981.

[Nicholson83] P.D. Nicholson and C.P. Tsang.
A Multi-processor Implementation of CSP.
In *Proceedings 6th Australian Computer Science Conference*, pages 57-64, February 1983.

[Organick72] E.I. Organick.
The MULTICS System: An Examination of its Structure.
M.I.T. Press, 1972.

[Ousterhout80a]
J.K. Ousterhout, D.A. Scelza and P.S. Sindhu.
Medusa: An Experiment in Distributed Operating System Structure.
Communications of the ACM, 23(2), 92-105, February 1980.

[Ousterhout80b]
J.K. Ousterhout.
Partitioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa.
Ph.D. Thesis, Carnegie-Mellon University, 1980.

[Parnas83] D.L. Parnas.
A Generalized Control Structure and its Formal Definition.
Communications of the ACM, 26(8), 572-581, August 1983.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Plotkin82] G.D. Plotkin.
An Operational Semantics for CSP.
In *Proceedings of 2nd Conference on the Formal Description of Programming Concepts*, pages 185-208, June 1982.

[Popek81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin and G. Thiel.
LOCUS: A Network Transparent, High Reliability Distributed System.
In *Proceedings 8th Symposium on Operating Systems Principles, ACM Operating Systems Review*, 15(5), 169-177, December 1981.

[Reid80] L. Reid.
Control and Communication in Programmed Systems.
Ph.D. Thesis, Carnegie-Mellon University, 1980.

[Ritchie74] D.M. Ritchie and K. Thompson.
The UNIX Time-sharing System.
Communications of the ACM, 17(7), 365-375, July 1974. Updated version appears in Bell System Technical Journal, 57(6), 1905-1929, July 1978.

[Roper81] T.J. Roper and C.J. Barter.
A Communicating Sequential Process Language and Implementation.
Software Practice and Experience, 11(11), 1215-1234, November 1981.

[Rowe82] L.A. Rowe and K.P. Birman.
A Local Network Based on the UNIX Operating System.
IEEE Transactions on Software Engineering, SE-8(2), 137-146, March 1982.

[Schneider82] F.B. Schneider.
Synchronization in Distributed Programs.
ACM Transactions on Programming Languages and Systems, 4(2), 179-195, April 1982.

[Shrivastava82] S.K. Shrivastava and F. Panzieri.
The Design of a Reliable Remote Procedure Call Mechanism.
IEEE Transactions on Computers, C-31(7), 692-697, July 1982.

[Silberschatz79] A. Silberschatz.
Communication and Synchronization in Distributed Systems.
IEEE Transactions on Software Engineering, SE-5(6), 542-546, November 1979.

[Silberschatz81] A. Silberschatz.
Port Directed Communication.
The Computer Journal, 24(1), 78-36, February 1981.

[Sincoski80] W.D. Sincoskie and D.J. Farber.
The Series/1 Distributed Operating System: Description and Comments.
In *Proceedings IEEE Fall COMPCON80*, pages 579-584, September 1980.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Sturgis80] H. Sturgis, J. Mitchell and J. Israel.
Design and Use of a Distributed File System.
ACM Operating Systems Review, 14(3), 55-69, July 1980.

[Swan77a] R.J. Swan, S. Fuller and P. Siewiorek.
Cm* a Modular, Multi-microprocessor.
In *Proceedings AFIPS National Computer Conference*, pages 637-644, June 1977.

[Swan77b] R. Swan, A. Bechtolsheim, K-W. Lai and J.K. Ousterhout.
The Implementation of the Cm* Multi-microprocessor.
In *Proceedings AFIPS National Computer Conference*, pages 645-655, June 1977.

[Tanenbaum81a]
A.S. Tanenbaum.
Computer Networks.
Prentice-Hall, 1981.

[Tanenbaum81b]
A.S. Tanenbaum and S.J. Mullender.
An Overview of the Amoeba Distributed Operating System.
ACM Operating Systems Review, 15(3), 51-64, July 1981.

[Thompson78] K. Thompson.
UNIX Implementation.
Bell System Technical Journal, 57(6), 1931-1946, July 1978.

[Vines82] P. Vines, K. Ramamohanarao and M. Briffa.
A Network File System for UNIX.
In *Proceedings 5th Australian Computer Science Conference*, pages 222-232, February 1982.

[Walker83] B. Walker, G. Popek, R. English, C. Kline and G. Thiel.
The LOCUS Distributed Operating System.
In *Proceedings 9th Symposium on Operating Systems Principles*, *ACM Operating Systems Review*, 17(5), 49-70, October 1983.

[Wilkes79] M.V. Wilkes and D.J. Wheeler.
The Cambridge Digital Communication Ring.
Local Area Communications Network Symposium, Mitre Corporation and National Bureau of Standards, May 1979.

[Wilkes80] M.V. Wilkes and R.M. Needham.
The Cambridge Model Distributed System.
ACM Operating Systems Review, 14(1), 21-29, January 1980.

[Wirth77a] N. Wirth.
Modula: A Language for Modular Multiprogramming.
Software Practice and Experience, 7(1), 3-35, January-February 1977.

[Wirth77b] N. Wirth.
Toward a Discipline of Real-Time Programming.
Communications of the ACM, 20(8), 577-583, August 1977.

[Wirth83] N. Wirth.
Programming in Modula 2.
Springer-Verlag, 1983.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.