University of Cape Town
Department of Computer Science

DISTRIX: an Implementation of UNIX on Transputers

by Paul J McCullagh

A Thesis
Prepared under the Supervision of
Associate Professor G de V Smit
In fulfilment of the Requirements for the
Degree of Master of Science in
Computer Science.

University of Cape Town
September 1989.

The University of Cape Town has been given the right to reproduce this thesis in whole or in part. Copyright is held by the author.

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

University of Cape Town

Abstract

Two technologies, ***distributed operating systems*** and UNIX are very relevant in computing today. Many distributed systems have been produced and many are under development. To a large extent, distributed systems are considered to be the only way to solve the computing needs of the future. UNIX, on the other hand, is becoming widely recognized as the industry standard for operating systems.

The transputer, unlike UNIX and distributed systems is a relatively new innovation. The transputer is a concurrent processing machine based on mathematical principles[1]. Increasingly, the transputer is being used to solve a wide range of problems of a parallel nature.

This thesis combines these three aspects in creating a distributed implementation of UNIX on a network of transputers. The design is based on the ***satellite model***. In this model a central controlling processor is surrounded by worker processors, called satellites, in a master/slave relationship.

CR Categories: C.1.2, C.2.4, D.4.0, D.4.1, D.4.7.

Keywords: Multiprocessor, UNIX, operating systems, distributed processing, transputer, satellite model, MINIX.

[1]Communicating Sequential Processes (CSP) by Hoare [Hoare78].

Acknowledgments

It is with great pleasure that I acknowledge the hard work and fortitude of my co-worker and friend Kuyper Hoffman, who was responsible for the development of the DISTRIX file server.

Immeasurable thanks to Associate Professor Riël Smit for his sober-minded assistance and guidance. On more than one occasion, Riël helped me out of a 'tight spot' by his objective and scientific approach to problems.

I would also like to thank those who made the project possible: Professor Kritzinger, head of the computer science department at UCT, Hennie Le Roux of Westerford Technology and ICL South Africa for proposing the project and their financial support. Thanks also to Gavin Gray for his highly professional directorship of the project.

Thanks to the members of the Institute for Electronics at the University of Stellenbosch, particularly, Pieter Bakkes, Prof. Jan du Plessis, Ralph Pina and Elise van der Walt for their work on the hardware aspects of the project, technical assistance and hospitality during our stay at Stellenbosch.

Contents

1 Introduction **1**

1.1 Classification of the DISTRIX Hardware 1

1.2 Distributed Operating Systems 2

1.3 Motivation 4

1.4 The Satellite Model 5

1.5 Project History 6

1.6 Thesis Outline 8

1.7 Related Work 9

2 UNIX on the INMOS transputer **11**

2.1 The INMOS Transputer 11

2.2 Hardware Support for Operating System Functions 12

2.2.1 Memory Protection 13

2.2.2 Peripheral Protection 14

2.2.3 Virtual Memory 14

2.2.4 Interrupts 15

2.2.5 Process Scheduling 15

2.3 Particular Problems in Supporting UNIX System calls 16

2.3.1 The KILL System call 16

2.3.2 The FORK System call 18

2.3.3 The NICE System call 19

2.3.4 The PTRACE System call 19

2.4 Conclusion 20

3 Coupling Additional Processors to UNIX **22**

3.1 Increasing Processing Power 22
3.2 The Satellite Method 23
3.3 Transparency Requirements 25
3.4 Services Provided by the Satellite Processor 26
3.4.1 Allocate a slot 26
3.4.2 Allocate memory 26
3.4.3 Start a process 26
3.4.4 Create a child process 27
3.4.5 Terminate a process 27
3.4.6 Signal a process 27
3.4.7 Read and write memory 27
3.5 The Agent Programs 27
3.5.1 The EXEC System call 28
3.5.2 The FORK System call 29
3.5.3 The SIGNAL System call 29
3.5.4 The KILL System call 31
3.6 The Processor Driver 33
3.6.1 Terminal Lines 33
3.6.2 Kernel Driver 33
3.6.3 Kernel Driver Process 34
3.7 Limitations of the Satellite Method 34
3.7.1 The SIGKILL Problem 34
3.7.2 Problem with Pipes 35
3.7.3 The EXIT Value 36
3.7.4 Signal Timing 36
3.8 System Performance 36

4 The Design of DISTRIX 38
4.1 The Communication Network 39
4.2 The Design of the Mini-kernel 40
4.2.1 The User Layer 40
4.2.2 The Kernel Layer 42
4.2.3 The Communication Layer 44

4.3 The Design of the Central Kernel 44
4.3.1 The Communication Layer 45
4.3.2 The Kernel Layer 45
4.3.3 The Signal Passing Layer 47
4.3.4 The File Server Interface 47
4.4 The System call Protocol 48
4.4.1 The EXEC System call 48
4.4.2 The FORK System call 50
4.4.3 The EXIT System call 51
4.4.4 The WAIT System call 52
4.4.5 The ALARM System call 52
4.4.6 The PAUSE System call 53
4.4.7 The SIGNAL System call 53
4.4.8 The KILL System call 54
4.4.9 The BRK System call 56
4.5 The Problem of Relocating Processes 56
4.6 The Boot Procedure 58

5 Conclusion **60**
5.1 The Current Status of DISTRIX 60
5.1.1 DISTRIX Hardware 60
5.1.2 Booting the DISTRIX Operating System 61
5.1.3 Using DISTRIX 62
5.2 The Future of DISTRIX 63
5.2.1 Data Protection 63
5.2.2 Improved Data Routing 65
5.2.3 Satellite Recovery 65
5.2.4 Process Distribution 66
5.2.5 Process Migration 67
5.2.6 A Parallel-based File Server 68
5.2.7 Distributed File Server 68
5.2.8 Distributed Programming and Debugging 69

A Implementation 70

A.1 The Transputer Microprocessor . . . 70

A.1.1 The Transputer CPU . . . 70

A.1.2 The Instruction Set . . . 71

A.1.3 Transputer Processes . . . 71

A.1.4 The Process Scheduler . . . 72

A.1.5 Communication Channels . . . 72

A.1.6 Hardware Links . . . 73

A.1.7 The Transputer Clock . . . 74

A.2 DISTRIX and MINIX . . . 74

A.2.1 An Overview of MINIX . . . 74

A.2.2 Why MINIX? . . . 76

A.3 Implementation Languages . . . 76

A.3.1 The OCCAM Language . . . 77

A.3.2 C Compilers for the Transputer . . . 77

A.3.3 Logical Systems C . . . 78

A.4 The Implementation of DISTRIX Processes . . . 81

A.4.1 The mini-kernel user process . . . 82

A.4.2 The user program image . . . 83

A.4.3 Running the user program . . . 84

A.4.4 The system call interface . . . 84

A.5 DISTRIX Inter-process Communications . . . 85

A.5.1 The **linkin** and **linkout** processes . . . 86

A.5.2 The DISTRIX exchange processor . . . 86

A.6 Signals on the Central Kernel . . . 87

A.7 The Central Kernel Clock . . . 89

B The Compiler 91

B.1 The Memory Allocation Problem . . . 92

B.2 The Framework for a Solution . . . 92

B.3 Virtual Memory . . . 93

B.3.1 Paging . . . 94

B.3.2 Segmentation . . . 94

List of Figures

1.1 The satellite model for distributed UNIX. 6

1.2 The design of DISTRIX. 7

3.1 The satellite method for distributed UNIX. 24

3.2 Message exchange for a FORK call. 30

3.3 Signal received during system call. 31

3.4 Signal received at the same time as a system call is done. 32

3.5 System call times. 37

4.1 The design of the mini-kernel. 41

4.2 The design of the central kernel. 45

5.1 Current configuration of DISTRIX. 61

A.1 The structure of MINIX. 75

A.2 The DISTRIX process memory map. 83

B.1 Conversion of arrays. 101

B.2 Conversion of a virtual address to a physical address. 101

B.3 Conversion of a physical address to virtual address. 102

Chapter 1

Introduction

This thesis is based on a joint project undertaken by Kuyper Hoffman and myself in the Laboratory for Advanced Computing at the University of Cape Town. The project entailed the creation of DISTRIX [Smit89], a distributed version of the UNIX[1] operating system[2] [Ritch78] running on a network of transputers. The project was initially conducted in conjunction with the Institute for Electronics (IE) at the University of Stellenbosch. The IE was responsible for the hardware aspects of the project while the Laboratory for Advanced Computing was responsible for the system software.

1.1 Classification of the DISTRIX Hardware

One encounters a great deal of diversity in the creation of multi-processor systems today. By way of introduction I will attempt to outline, in broad terms, these different approaches, and to place DISTRIX in the context of this classification.

Multi-processor systems can be classified on a scale which basically involves two factors: the proximity of the processors and the means of connection. On the *loosely coupled* end of the scale we have fairly autonomous systems connected via a local area (or even wider area) network while on the other end of the scale, *tightly coupled* systems involve a number of processors connected to the same bus, sharing memory. A good example of an operating system for a very loosely coupled system is the LOCUS[3] distributed operating system [Popek85]. Amoeba [Mulle86], the V kernel [Cheri84] and the Eden

[1] UNIX is a registered trademark of AT&T Bell Laboratories.
[2] To be more specific, a UNIX version 7 look-alike.
[3] LOCUS is a trademark of Locus Computing Corporation.

project [Almes85] also run on hardware that may be classified as loosely coupled, being based on local area networks. Moving towards the tightly coupled end of the scale we have the ring and star formation distributed systems like the Cambridge Digital Communication Ring [Wilke79], the New Mexico State University Ring-Star System [Karsh83] and the University of California Irvine Distributed System, DCS [Farbe75].

Somewhere between the ring and star formation networks and tightly coupled systems are what may be termed *closely coupled* systems which are those consisting of a number of semi-independent processors in close proximity. In these systems processors do not share memory as in tightly coupled systems. Instead, each processor has its own memory and processors communicate via a common bus or other mechanism. Examples of such systems are the HXDP system [Jense78], AT & T Bell Laboratories' S/Net multicomputer [Ahuja83] and the Butterfly machine [Brown84]. Examples of operating systems for such configurations are LINDA [Carri86], HELIOS [Garne87] and Mach [Coulo88]. DISTRIX also falls in this category.

A fairly popular approach, in terms of UNIX-like systems, is the shared memory (tightly coupled) approach. An example of such a machine is the CM* multi-processor [Swan77]; also Auregen's system 4000 [Glaze83]. Operating systems for these machines are Medusa [Ouste80] and Auros [Glaze83] respectively.

1.2 Distributed Operating Systems

Although hardware is possibly the most important means of distinguishing between multi-processor systems, the operating system ultimately determines the external appearance of the system. One generally views systems whose multi-processor nature is apparent as more loosely coupled than those whose parallel nature is hidden by the operating system. Operating systems whose parallel nature is apparent are more common in hardware configurations that are more loosely coupled than closely coupled systems. For example, the two UNIX network systems LOCUS and the Newcastle Connection [Brown82] illustrate the difference between a transparent network and an explicit network configuration. LOCUS hides the underlying network by unifying the UNIX file system in a transparent manner, whereas in the Newcastle connection the file systems on various machines are unified by adding a new *root* directory which has a directory entry for each machine on the network.

One may go so far as to say that in this case we do not have a distributed operating system but a *network* operating system. Tanenbaum and van Renesse [Tanen85] define a distributed operating system as one that appears to its users like an ordinary centralized operating system but runs on multiple, independent CPUs[4]. The distinction is perhaps an unnecessary one as, in general, operating systems vary in how well they abstract and disguise the underlying hardware. In the case of the Newcastle connection, for example, this is done *less effectively* than in LOCUS.

Nevertheless, the general trend is towards systems that are not necessarily more closely coupled in terms of hardware but rather towards operating systems that make the system *appear* more closely connected by making the multi-processor nature of the system transparent.

DISTRIX achieves a fair degree of transparency. That is, DISTRIX may be used without knowing and without having to know that the system consists of multiple processors. This is an important fact, especially considering that DISTRIX is a version of the UNIX operating system which is and was originally designed as a 'unified' operating system. In this case the word *unified* means that the users do not expect to be restricted (besides normal access control) as to which peripherals, files of other resources they access or with which processes their processes communicate etc.

However, under close examination, the distributed nature of the DISTRIX hardware does become apparent. This is manifest in the introduction of a 'new' system call to DISTRIX called MOVE. The function of MOVE is to move the calling process to another processor[5]. The main reason for adding the MOVE system call was to avoid the problem of having to place process distribution logic in the DISTRIX kernel. However, none of the users need ever use the MOVE system call because the initial process (*init*) uses MOVE to distribute the *login* processes amongst the available processors. Since the FORK system call does not change the distribution of processes in the system this procedure effectively maps terminals onto processors. More details on the MOVE system call are given in chapter 4 section 4.5.

In DISTRIX the problems associated with creating a unified file system in a distributed

[4]Central Processing Units.

[5]That is, any running process may perform the MOVE system call supplying one parameter, the destination processor number, and if the call is successful the process will continue executing on the destination processor.

hardware environment were avoided, rather than solved, by the creation of a single file server processor to which all the peripherals are connected. The work on the file server is described in a separate thesis [Hoffm89]. That which remained, the creation of the UNIX model of processes on the transputer, process control (FORK, EXEC, WAIT, etc.) and the general problem of distributing the processing load is the work of this thesis and further references to the file server will only be made where it is necessary to place the work of this project in the context of the greater project as a whole.

1.3 Motivation

There are a number of reasons for the development of DISTRIX:

1. To create a distributed UNIX operating system.
2. To assess the feasibility of running UNIX on the transputer.
3. To start work that may lead to ongoing research into distributed operating systems.

The reason for the desire to create a distributed version of UNIX lies in both the popularity of the UNIX operating system and the need for a cheap, yet powerful version of the system. Both of these desires are strictly commercial in nature and stem from the fact that the original project was initiated and funded by industry. With this goal in mind the funders of the project chose the transputer for a number of reasons:

1. The transputer micro-processor is both powerful and relatively cheap.
2. It was specifically designed for distributed and parallel processing.
3. Networks of transputers can easily be constructed without support hardware.
4. The IE at the University of Stellenbosch were already familiar with the transputer hardware.

From the outset a system with a more or less one-to-one correspondence between users and processors was envisaged. The transputer was found to be ideal for this purpose. In fact, one user per transputer is even more desirable considering the protection problems outlined in chapter 2.

Commercial gains and objectives aside, DISTRIX has provided a great opportunity for research and proven to be a valuable learning experience. This was certainly the most important objective in mind at the start of the project from the point of view of the Computer Science Department at UCT[6]. As it happened the financial backing from industry and the support from the IE were inadequate for the creation of a commercial UNIX system. Thus, even though commercial objectives have not been met, the academic motivations have been satisfied by the creation of a prototype distributed UNIX operating system. We have proven that the transputer can support UNIX and we have learned a lot about the transputer, UNIX, and distributed operating systems in the process.

1.4 The Satellite Model

Given the objective of the creation of a UNIX operating system on transputers, an investigation was conducted to find an appropriate approach for the design of the new operating system. It was decided to base the design of DISTRIX on the satellite model [Bach86]. This was done for a number of reasons:

- Transputers are stand-alone processors that do not share memory.
- Transputers can easily be connected to form the network required for the satellite model.
- It was decided that the satellite model would be the least complex method, given that limited man-power was available.

The details of the investigation, including further information on the satellite model and other means of creating distributed UNIX systems are outlined in [Smit88]. The work of Janssens *et al.* [Janss86], Bach and Buroff [Bach84] and Russell and Waterman [Russe87] was particularly useful in making the assessment.

The satellite model consists of a central controlling processor connected to an arbitrary number of surrounding satellite processors (figure 1.1). The UNIX kernel runs on the central machine while the user processes run on the satellite machines. For each active user process there is a corresponding stub process or user agent running on the central machine.

[6]The University of Cape Town.

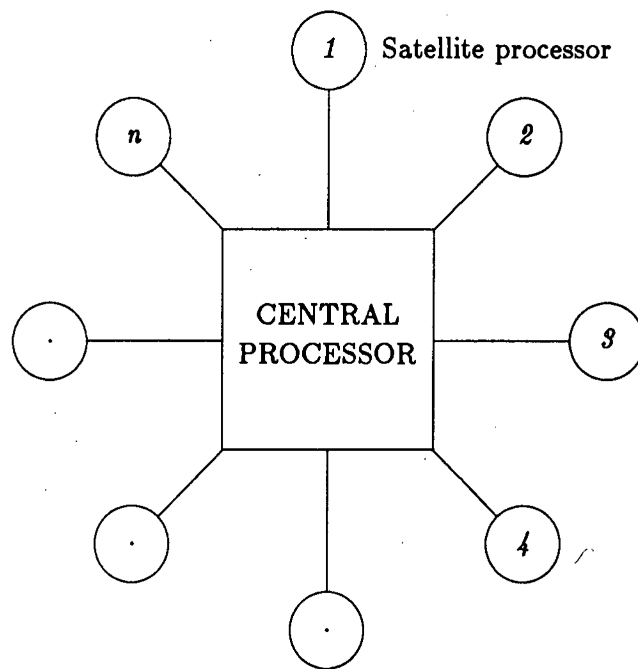

Figure 1.1: The satellite model for distributed UNIX.

A user process runs normally until it issues a system call. At this point the system call is packaged into a message and sent to the process's agent on the central machine. The agent then issues a local system call and the system call function is performed, as usual, by the UNIX kernel on the central machine. The agent packages the results and sends them back to the waiting user process on the satellite machine.

In this way the load of running the user processes can be distributed over a number of processes and a linear increase in processing power at the user level is possible as additional satellite processes are added. Unfortunately the central machine, having to service all system calls, is the limiting factor. As a result DISTRIX was taken a step further with the addition of a remote file server, thus relieving the central machine from servicing all file system calls. In figure 1.2 the overall structure of DISTRIX is illustrated, showing how the file server was removed from the central machine.

1.5 Project History

DISTRIX was derived from the MINIX [Tanen87] operating system. The structure of MINIX lends itself well to a distributed environment as the operating system consists of a number of independent processes which communicate via a message passing system.

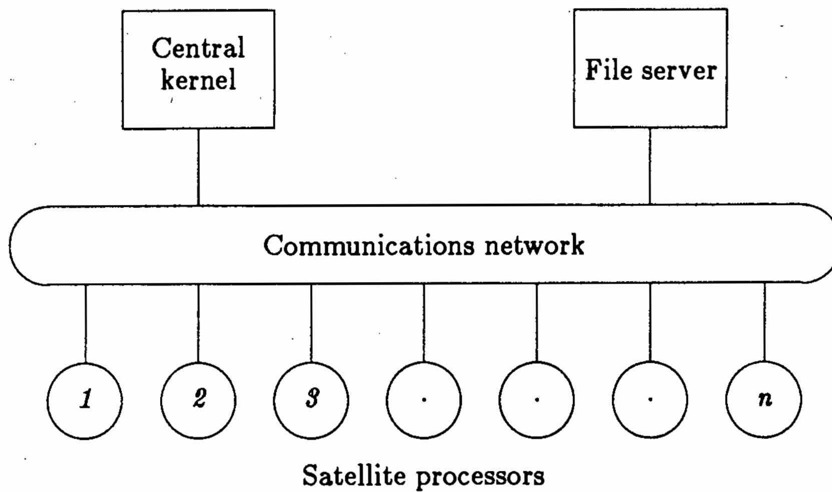

Figure 1.2: The design of DISTRIX.

In his book Andrew Tanenbaum [Tanen87], the creator of MINIX, states that it would be a relatively simple task to turn the file server (one of the operating system processes) into a remote file server. As a result the DISTRIX file server is the MINIX file server with some additions and modifications to enable it to run in a distributed transputer environment. More details on the design of MINIX and the motivation for its use may by found in appendix A section A.2.

In distributing the processing load, most of the MINIX code had to be rewritten for the new environment. As a result, although it behaves as MINIX (outwardly speaking), there is little correspondence between this part of DISTRIX and MINIX. Further references to DISTRIX in this thesis refer to that part of DISTRIX excluding the file server (the file server will be referred to as: *the file server* or *the* DISTRIX *file server*).

As mentioned before, DISTRIX conforms to the satellite model for distributed UNIX. The development therefore consisted of two distinct parts:

- The development of the part of the operating system that is to run on a satellite processor (the mini-kernel).
- The development of the central processor kernel.

The mini-kernel was developed first, using MINIX running on PC/AT as the central processor and a number of transputers as satellites. The result was a *standalone* system of standard MINIX with fairly transparent 'external' processing capabilities. In future, I will

refer to this system as the MINIX-*based system*. This method of adding external processing power to an existing UNIX system has been used before, namely by Barak and Shapir in 1980 [Barak80] and in the New Mexico State University Ring-Star system [Karsh83]. In chapter 3 the MINIX-based system is dealt with in detail and the similarities and differences with the above systems outlined.

The most important result in creating the MINIX-based system, in terms of the project as a whole, was the creation of the mini-kernel which runs on the satellite processors. Once the mini-kernel had been developed and tested, the central kernel was written, duplicating the behavior of the user agent processes and MINIX system functions[7]. Briefly, this was done by having the agents themselves actually perform the system call functions rather than issue a system call to a MINIX kernel. As a result the transputer-based central kernel does not have general operating system capabilities as does MINIX, but was designed specifically for the task that it performs. At this point the file server was integrated into the central/satellite system, completing the DISTRIX operating system. Since then most of the MINIX commands have been ported to DISTRIX. Some of the commands not yet ported are the compiler, assembler and linker. The compiler currently runs under DOS[8] on a PC/AT and the major resistance to its integration into DISTRIX is the fact that it is too large to compile itself. Further details on the compiler may be found in appendix A section A.3 and appendix B.

1.6 Thesis Outline

The next chapter discusses the suitability of the transputer for running UNIX. The chapter describes a number of the problems encountered in the implementation of DISTRIX on transputers as well as the solutions found.

Chapter 3 describes the MINIX-based system in which a number of transputer satellite processors where coupled to a PC/AT running MINIX.

Chapter 4 provides a detailed description of the design of the mini-kernel and central kernel. The chapter also includes details on the agent to user process communication protocols and system call procedures.

Appendix A provides implementation details of various aspects of DISTRIX. Appendix

[7]Excluding the file system calls, of course.

[8]The operating system MS-DOS was written for the IBM PC by Microsoft Incorporated.

B describes an investigation conducted to solve the problem with memory management on the transputer by changing the compiler.

1.7 Related Work

I believe that the DISTRIX project, as a demonstration of UNIX on transputers, is unique. Existing UNIX-like systems on transputers have compromised the structure or functionality of UNIX in some way. The traditional UNIX FORK system call is usually the first to go due to the difficulty of implementing it on the transputer.

The HELIOS operating system [Garne87] for transputers is a good example of this. HELIOS is not first and foremost a UNIX operating system although in his paper Garnet states that an alternative library is supplied that is compatible with the UNIX system calls. However, this is only partially true due to the absence of a conventional FORK, and other problems with UNIX software interrupts on the transputer. In addition to this the HELIOS process model does not correspond to that of UNIX. Instead, HELIOS processes, called tasks, may consist of a number of concurrent processes. No mechanism is provided to interrupt a task in a similar manner to UNIX. Doing things the HELIOS way makes sense because the transputer just does not conform to the UNIX ideal (as will be seen in chapter 2). Nevertheless, having standard UNIX on the transputer has an advantage (over HELIOS) from a programming point of view, namely existing UNIX applications can be ported without changes. In addition, DISTRIX has great potential for research and even greater potential in general if transputer hardware is enhanced.

Another resistance to running UNIX on the transputer is the basic unsuitability of the UNIX kernel to a distributed environment [Blair85]. This stems from the fact that the UNIX kernel relies on a large amount of global state information. The transputer does, in fact, demand a completely different approach in operating system design. Although this is a topic that may require a great deal of research.

Besides HELIOS a number of other operating systems have been announced for multi-transputer machines. Among them are Trollius [Corne88] by the Cornell Theory Center, Meikos by Parsys and Idris by Perihelion. According to Peter Welch [Welch89] Meikos, Idris (and HELIOS) incorporate a strong level of UNIX compatibility. At the application level 'most' UNIX system calls are available and the parallel hardware can be hidden

or made explicit to the application developer or user. In July 1989, Erik Verhulst of Intelligent Systems International, Belgium, announced a product called TROS. According to Verhulst [Verhu89], TROS is an operating system kernel with built-in fault tolerance for transputers. The question of fault tolerance is an important one in the transputer environment (which provides no support in the form of hardware protection capabilities), but was not addressed at all in the development of DISTRIX.

In conclusion, it seems that the general trend with regard to UNIX and the transputer is not to provide 'pure' UNIX (as Welch puts it in his report on UNIX SIG [Welch88]) but rather a system that supports 'most' UNIX system calls, some UNIX-like shell and the common UNIX tools. This may be largely due to the fact that UNIX has set a standard of convenience and usefulness that will be difficult to match if we depart from it completely, in search of a new methodology that is functionally equivalent, yet more suited to a distributed environment.

Chapter 2

UNIX on the INMOS transputer

In this chapter the feasibility of implementing UNIX on a transputer (or a number of transputers) is examined. The next section presents a brief description of the transputer processor. Section 2.2 discusses the general issues involved in getting UNIX to run on one or more transputers. Section 2.3 describes some particular problems encountered when implementing UNIX system calls on the transputer and the possible solutions to these problems.

2.1 The INMOS Transputer

In 1984 INMOS Limited introduced a new processor, the transputer, geared specifically towards the execution of 'psuedo' concurrent processes [Inmos84a, Pount84, Inmos87a]. It was designed essentially as a 'stand-alone' processor requiring no support chips to function correctly and effectively [Walke85].

The transputer has a micro-coded scheduler which controls the running of an arbitrary number of concurrent processes. Machine code level instructions are provided to enable the creation and termination of processes. Processes are typically given a 1 millisecond timeslice and context switches require only 1 microsecond. This is considerably faster than any other micro-processors currently running UNIX.

Information passing between and synchronization of processes is done by means of a simple form of message passing implemented in the transputer micro-code. The concept of a *message* (consisting of a length and a string of bytes) and a *channel* along which a message is passed from one process to another is supported by the transputer hardware.

On a higher level, each transputer communicates with the outside world via four hardware *links*, which transfer data at a rate of 20 Mbits per second. Conceptually there is no difference between communication across a hardware link and internal message passing along a channel; the machine code instruction is the same. In this way networks of transputers can be built by connecting the hardware links as required.

The transputer does not have a bus as do conventional micro-processors, but *link adapters* are available to connect the transputer to the popular busses, thus giving the transputer access to other peripherals. In addition to this, more and more peripherals are being produced which connect directly to the transputer link, allowing faster and more efficient access.

The transputer CPU is also somewhat unconventional in its design. It consists of an *instruction pointer*, a *workspace pointer*, an *operand register* and three general purpose registers, A, B and C, which operate as a 3 word stack. The operand register is used to construct constant values and complex instructions. The register stack is used to hold the arguments and result of an instruction and to pass parameters and return values of a function call. Besides absolute addressing, memory can be addressed relative to the workspace pointer.

The workspace pointer is also of special use to the transputer scheduler. The process queues of ready-to-run processes are maintained as linked lists of workspace pointers. The scheduler has registers which point to the front and back of these queues, thus enabling it to add a process to a queue as the process becomes ready to run and to determine the next process to be run. More details on the transputer are provided in appendix A section A.1.

2.2 Hardware Support for Operating System Functions

Let it be said at the outset that the transputer was *not* designed with an operating system like UNIX in mind. In fact, it may be argued that the transputer was not designed to run an operating system *at all*. Nevertheless, the question may be asked: how suitable is the transputer for running UNIX? In the following sections some general issues in this regard are discussed.

2.2.1 Memory Protection

Since UNIX is a multi-user operating system, users should be able to rely on the operating system for protection from the other users of the system. This includes both protection of data in secondary storage and that of executing programs in main memory. The problem of protection of data in secondary storage is discussed in the following section. With regard to protection of user data in main memory, some of the work can be done by the operating system software, but to ensure complete safety, some hardware support is needed.

The transputer provides no such support for an operating system. In fact, the transputer has one address space (as opposed to multiple virtual address spaces supported by most processors running UNIX). This permits any process to read or write any part of memory. As a result, malicious users cannot be prevented from disrupting the execution of other processes in the system. Perhaps even worse, a program that is under development (or incorrect for some other reason) can quite easily and unintentionally bring the entire system and all its processes to a standstill.

While it is not possible to protect every process from all others, it is possible to protect a group of processes from other groups by executing each group on a different processor. In this case the fact that each transputer has its own memory serves as protection. In the UNIX environment one might, for example, partition processes according to ownership: all processes created by/for a particular user execute on one processor.

This does not solve the more severe problem of protecting the operating system. Every processor, no matter how small its function, must contain some kernel code to control its operation and integrate it into the rest of the system. This code is imminently vulnerable, and corruption would cause that processor to become useless and possibly disrupt other parts of the system. Corrupted processors must be detected and a method found to terminate former activity on that processor gracefully. The processor should then be rebooted without affecting the running of the system as a whole. An example of a similar recovery system may be found in Amoeba [Mulle86]. Amoeba provides a *boot service* which performs the recovery function. Any server in the system may register with the boot server. The boot server periodically polls registered servers and if there is no reply within a specific time the boot server assumes the server has failed and initiates the activation of a new server.

2.2.2 Peripheral Protection

On the transputer all I/O is done via the links. Since the links are easily accessible to all processes, all processes can directly access the peripherals connected to those links. On the transputer there are no privileged instructions or execution levels which allows restricted access to peripherals by the operating system.

If it is known how the data is stored and how to activate a particular peripheral, there is nothing to prevent a user from, for example, accessing any part of the file system. The only way to prevent this is to connect peripherals only to 'privileged' transputers which do not run user programs. To ensure complete safety, links to such transputers (from other less privileged parts of the system) must be carefully monitored.

2.2.3 Virtual Memory

UNIX usually relies on hardware to be able to give each process in the system an unlimited address space in the form of virtual memory. Only part of the program's address space will occupy physical memory at any given time, while the rest resides on disk and is swapped in when required.

To support this, the hardware must provide an address translation mechanism and inform the operating system when a page fault occurs. The process concerned is then suspended while the operating system fetches the page from disk.

The transputer has no virtual memory capabilities. Instead all processes must be located somewhere in the physical address space at all times. In addition, very real limits are placed on the size of the code, stack and data segments of a program. Though off-chip solutions to these problems are being sought ([Bakke88]), they will never be solved adequately for a multi-process system due to the fact that transputer instructions are not restartable. If instructions were restartable, a process that causes a page fault could be descheduled and another process run while the page is fetched. Since the instruction that caused the fault must be stalled[1], the process that is executing the instruction cannot be descheduled[2], thus preventing other ready-to-run processes from becoming active. Therefore, a page fault on a processor causes all processes to be suspended while the page fetch

[1] A stalled instruction is one that has been halted at some point in the execution of that instruction's micro-code.

[2] The transputer is not able to deschedule a process in the middle of executing an instruction, nor is it able to restart the instruction.

is done.

Until INMOS adds some virtual memory support hardware to the transputer, UNIX users will have to be content with solving their memory requirement problems by the addition of physical memory. The transputer has a 4 gigabyte physical address space which should be adequate for applications in the foreseeable future.

2.2.4 Interrupts

The transputer does not support interrupts as do conventional processors. Nevertheless, the transputer is very competent at real-time processing because, instead of an interrupt handling routine which is activated on the occurrence of an event, a dedicated process can be provided to wait for the occurrence of the event. Waiting is always done on a channel, so all events in the transputer are simulated by a channel input.

The solution of having dedicated event handling processes is not acceptable for UNIX environments, since this changes the interrupt model for application programs. Therefore some method of implementing interrupts must be found in order to run UNIX on the transputer. One possible solution is discussed in Section 2.3.1.

2.2.5 Process Scheduling

Most processors to which UNIX has been ported have some instructions to help in the scheduling and switching of processes. These instructions rarely go further than to help the operating system quickly set up the context for the next process it has selected to run. In this area the transputer takes a step further by maintaining process queues and scheduling and descheduling processes in micro-code.

The transputer micro-code scheduler runs processes at one of two priority levels. Each priority level has its own process queue. Processes on the queues are scheduled on a round-robin basis. A high priority process will always be selected to run before a low priority process. In addition, high priority processes can never be pre-empted and once running, will only be descheduled on input from or output to a channel (or link). On the other hand, low priority processes can be pre-empted when their timeslice expires or by a high priority process becoming ready to run.

This implies that if implementors plan to use the built-in scheduler of the transputer to control processes, kernel code must run at high priority level, and user processes will

have to be set at low priority. As a result all user processes run at the same priority level, which is not 'true' UNIX. At least two problems arise should one wish to circumvent the transputer scheduler. Firstly, this would negate one of the greatest advantages of the transputer, namely, the provision of fast, efficient process scheduling. Secondly, implementing such a solution is extremely difficult, if not impossible, considering the lack of interrupts and, more importantly, the relationship between the process queues and some complex instructions such as input, output and block moves.

On the other hand, having only one priority level for user processes on the transputer is not as bad as it would be on many other processors due to the fast switching time and small timeslice. For example, the problem of CPU bound jobs degrading the response of interactive programs (e.g. an editor) is minimal because, even if there were 50 CPU bound jobs scheduled before the CPU bound job, the delay would still only be 50 milliseconds. Considering the speed of the transputer, the interactive program would then be able to execute approximately 10000 instructions before it is descheduled and thus have time to respond to a user's request.

In addition, if a separate processor is given to each user as in the solution described in Section 2.2.1, users can only blame themselves for slow response times.

2.3 Particular Problems in Supporting UNIX System calls

Implementing UNIX, to a large extent, reduces to implementing the system calls. A number of UNIX system calls give particular problems on the transputer due to the transputer's unusual architecture. In the following sections these problems are described and possible solutions examined. Only version 7 system calls [Bell79] have been considered.

2.3.1 The KILL System call

The KILL system call causes what is known as a software interrupt to occur in a selected process. As mentioned before, the transputer does not support the concept of an interrupt so some rather inelegant fiddling has to be done in order to implement this system call on the transputer.

The first problem is to identify and isolate the process that is being signaled. Processes in UNIX are uniquely identified by their process identifiers whereas transputer processes

are uniquely identified by their workspace pointer and priority level.

The workspace pointer is usually used by a program as a stack pointer and therefore the workspace pointer can take on a range of values. However, the operating system can derive the range of values from the process identifier provided by the KILL system call since a table of processes is maintained by the kernel containing, amongst other information, a memory map of each process in the system.

Armed with this range of values and knowing that the process is low priority, the kernel must then search or the particular workspace pointer of the process. This can be found in one of three places depending on the state of the particular process.

1. If the process is descheduled but ready to run, it will be found on the transputer low priority queue.

2. If the process is doing I/O, the workspace pointer will be found at a memory location associated with the channel.

3. If the process was running when the kernel began executing, it will be found stored as the interrupted low priority process at a fixed place in memory, ready to resume execution as soon as no high priority processes are ready to run.

Unfortunately, if the workspace pointer is found in place 3, the process is not 'suitable' for signaling as it is possible that the process is in the middle of a complex instruction such as a block move. So in this case, the process must be allowed to continue executing until its workspace pointer is found in either place 1 or 2. If the process to be signaled is compute bound and the only low priority process ready to run, a dummy transputer process must be introduced to force the process to be descheduled and placed on the process queue. Fortunately this is simple to do on the transputer.

Once the kernel has the workspace pointer, this value can be used to reference memory where the current instruction pointer of the process can be found (stored by the transputer scheduler). The value of the instruction pointer can then be changed to the appropriate interrupt handling routine.

The workspace pointer must also be altered to reflect a procedure call. This can be done either by the kernel or by the process itself just before it executes the interrupt handler. The latter option is preferable from an implementation point of view as the

former may require altering the transputer process queue. However, a danger with this solution is that if signals arrive in quick succession (i.e. a subsequent software interrupt occurs before the stack pointer is adjusted) previous interrupts will be lost.

2.3.2 The FORK System call

By far the greatest problem is presented by the FORK system call. The problem with FORK stems from the fact that the transputer memory is organized into one linear address space whereas some form of segmentation is required.

When a process executes the FORK system call the kernel makes an identical copy of the process (known as the child) and causes it to begin execution as a new process, with its own code and data areas, at the point of exit from the FORK call. The only difference between the execution of the *parent* and the child is that the FORK call returns a different value to each of them.

When the FORK is executed on the transputer, the child process begins with every absolute address stored by the program (naturally) identical to that of the parent. Since the transputer has only one address space, these values all refer to data and code belonging to the parent—an *unacceptable* situation.

To solve this problem it helps to examine the solution used in conventional UNIX systems. Almost every one of these systems uses a hardware supported solution in which every address used by a program is not absolute, but is offset by a base or segment register. The segment registers are then changed when the processor switches context, thus ensuring that all addresses refer to the memory of the process being run.

In brief, the solution is to add another level of indirection to every absolute memory reference. On the transputer this has to be done by software and should be done as efficiently as possible.

One method is to designate some position in a program (e.g. the start of the code section) as the base. Before an address is stored (be it a pointer or a procedure return address) the value of the base is subtracted. Just before the address is used (to access memory or return from a procedure) the base is added. On the transputer this address may be found at run time using a *load pointer to instruction* instruction (appendix B provides a detailed explanation).

Another method is to designate an available register as the base pointer. The operating

system then ensures that this register is set before a process begins its time quantum. The value in this register is either subtracted or added to every address depending on whether it is being stored or used to access memory. Although the second method suggested is more efficient, it is impossible on the transputer as the only available register is the workspace pointer and memory can only be referenced at constant offsets from the workspace pointer.

The additional code to add and subtract the base value must be placed in the program by the compiler. It cannot be done by an assembler because it is a matter of semantics as to whether a value is an address or not. As a result, programmers working in assembler will have to be made aware of the potential problems and the solutions if they want their program to perform the FORK system call correctly. A complete description of the investigation done to solve the FORK system call problem is given in appendix B.

2.3.3 The NICE System call

It is not surprising that the NICE system call, which adjusts the execution priority of a process, has little or no meaning in a system that has only one priority level for all the user processes.

On the other hand, in a complex system of a number of transputers it may be possible to give some meaning to the priority of processes by the number of processes that may run concurrently with a particular process on the same processor. For example, a top priority process may require exclusive use of a processor while a low priority process may share a processor with any number of other processes.

2.3.4 The PTRACE System call

This system call enables a parent process to control the execution of a child process and to examine and change the child's core image. Its primary use is the implementation of breakpoint debugging which unfortunately cannot be done well on a transputer.

There are two aspects of the PTRACE system call that cause problems on the transputer. Firstly, a process that is being traced must be suspended indefinitely on receipt of a signal. In order to do this the same procedure as for a signal must be followed to the point where the workspace pointer is found in either place 1 or 2 as described in Section 2.3.1. The process must then either be removed from the transputer's internal scheduling queue or, if it is waiting on input, the reply message to the process must be stored.

The second problem is to suspend the process as soon as possible after the execution of at least one instruction. There are two main difficulties involved with this:

1. How is it determined that a process has executed an instruction?

2. How soon after that instruction can the process be suspended again?

These problems arise from the fact that the processes are scheduled and descheduled by a micro-coded scheduler and there is no way of circumventing this[3]. Therefore, to determine that a process has executed an instruction (question 1) is a complex task involving a combination of monitoring the process queue and the process currently under execution. Although an interrupting high priority processes can determine which low priority process is currently running, there is, unfortunately, no guarantee that a particular process will be 'caught' during its execution. The smaller the intervals between interrupts, the greater the chance, but doing this greatly increases the overhead on the transputer as a whole.

In answer to question 2, a process can only be suspended again after the micro-code scheduler itself has descheduled the process. In the worst case the process may have executed about 10000 instructions by the time it is descheduled.

Perhaps a far better solution would be to use a simulator that interprets transputer machine code. When the child process issues the system call to indicate it is ready to be traced by its parent, the operating system stops the process and starts an interpreter.

Except for speed it should not be possible to tell the difference between a program that is interpreted and one that is not. In addition, the operating system would have full control over the execution of the program and be able to execute instructions one at a time.

2.4 Conclusion

By introducing some inefficiencies the transputer can be persuaded to run UNIX. It was found that the extra code added by the compiler to implement the fork call, caused programs to be approximately 40% bigger and execute about 30% slower. The overhead is not prohibitive and does not negate the original advantage of the transputer, namely its speed.

[3]Processes will not run on the transputer unless they are scheduled by the on-chip scheduler.

Certain problems such as virtual memory support and kernel protection cannot be solved adequately, although many problems are solved by allocating one processor to each user.

These disadvantages must be seen as a trade-off against high power for low cost. The intended purpose of the system will further influence the trade-off. If, for example, the system is intended mainly for software development, the lack of adequate protection could be a serious deficiency. If the system is to be used for applications such as the correlation of statistical data, the lack of virtual memory may be prohibitive. However, should the system be used mainly for running well-debugged application software without excessive memory requirements, the above mentioned problems may be of little significance.

Nevertheless, the transputer has potential as a basis for large distributed operating systems due to its high speed and ease with which a distributed environment is created. The implementation of the DISTRIX operating system has shown that UNIX can run on a network of transputers. Although there were difficulties, the work has been successful and if INMOS add more hardware support for operating systems to the transputer, DISTRIX may be updated to take advantage of the new features.

Chapter 3

Coupling Additional Processors to UNIX

There are two approaches to using the satellite model (described in chapter 1) to create a distributed version of UNIX.

1. The structure and underlying methodology can be used to create a distributed version of UNIX.
2. It can be used as a method to increase the power of an existing UNIX system.

The second approach differs from the first in that it involves the expanding of an existing uniprocessor system into a distributed system. DISTRIX takes the first approach mainly due to the fact that there is no existing UNIX system running on a transputer. This approach to creating a distributed UNIX is suggested by Bach [Bach86]. As an intermediate step in the creation of DISTRIX the second approach was used. The result of this was the MINIX-*based system*, which is the subject of this chapter.

3.1 Increasing Processing Power

The second approach was first used in 1978 by Lycklama and Christensen [Lyckl78] to create a distributed system. In 1980, Barak and Shapir [Barak80] created a system that consisted of a DEC PDP-11/45 central processor, running UNIX, and PDP-11/10 satellite processors were used to add processing power. Karshmer *et al.* [Karsh83] used it to create the New Mexico State University ring-star system, which was implemented on a PDP-11/34 with LSI-11 satellite processors. This chapter describes a similar system consisting

of a PC/AT running MINIX as the central processor coupled to a number of transputer satellite processors. As mentioned before, this has been called the MINIX-based system.

Barak motivates the use of the satellite approach by explaining how it may provides an economical solution to an increasing demand in computing power. Karshmer *et al.* describe their approach as a relatively easy way of creating a distributed operating system. Both these reasons are relevant, however the MINIX-based system was created for the following reasons:

1. Most importantly, to develop and test the mini-kernel (the kernel code running on the satellite), an integral part of the DISTRIX operating system.
2. To assess the degree of transparency that can be achieved when running user processes on a satellite.
3. To obtain an idea of how much the central processor operating system had to be changed to achieve 2.

The assessment of 2 and 3 above are of interest to those who plan to add satellites processors to an existing UNIX installation. These questions, with particular reference to UNIX are not addressed by Barak and Karshmer.

The exact requirements for transparency are discussed in section 3.3. Section 3.2 gives a brief description of the satellite method and in the sections 3.4, 3.5 and 3.6 details of the implementation are provided. Section 3.7 discusses the achievement (or lack of achievement) of the transparency requirements and finally, in section 3.8, the system performance results are discussed.

3.2 The Satellite Method

This section provides an outline of the satellite method and the terminology used in this chapter. The basic idea is a simple one: a UNIX process is downloaded from the central machine (host) to a closely coupled satellite processor. The user process running on the satellite is referred to as the remote process. A number of remote processes may be running on a particular satellite processor at any one time. In this way the MINIX-based system is an extension to the systems described by Barak and Karshmer, whose satellite processors

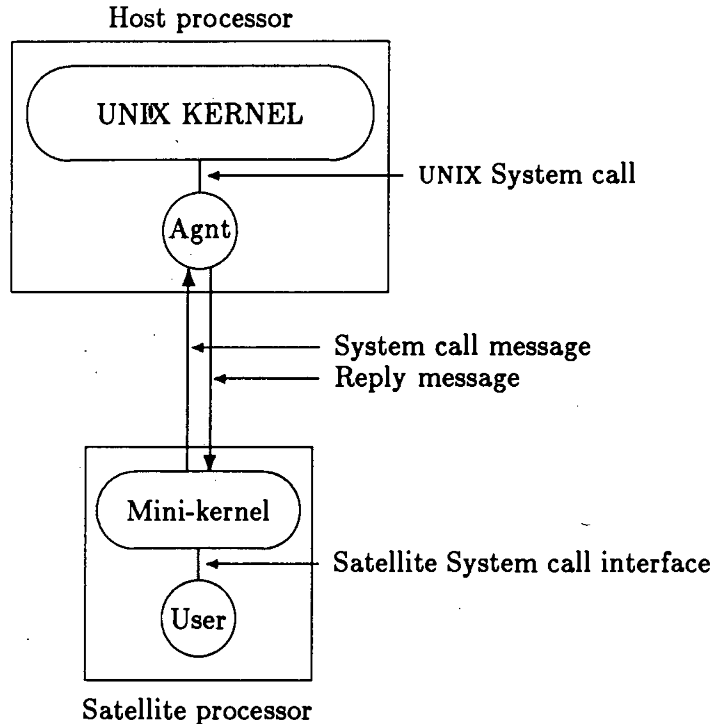

Figure 3.1: The satellite method for distributed UNIX.

could only run *one* remote process at a time[1].

Once running on the satellite the remote process may issue system calls, and interact with processes on the host and on other satellites as if all processes were running on the host. This is done by creating an agent process on the host for every remote process currently running. The parameters of a system call issued by a remote process are sent to its corresponding agent. The agent then issues the system call to the operating system on the central machine. Figure 3.1 indicates a remote process on a satellite processor and its relationship with the corresponding agent process on the central processor. A system call is therefore performed by a form of remote procedure call [Nelso81, Birre84] to the host.

Although most of the system call work can be done by the host, there are some system calls which involve a fair amount of work on the satellite processor itself. These calls are those which involve the creation, signalling and termination of processes in the usual UNIX manner. More details concerning this are given in section 3.4.

In most cases a number of satellite processors are employed depending on the amount

[1]This simplifies the problem somewhat (very little kernel code is needed), although the FORK system call becomes more complicated and less efficient. If a remote process *forks*, the child must be created on a different satellite processor.

of additional processing power required. Although, the increase in power, in terms of user-level processing capacity, is linear, the power of the system as a whole does not increase as a linear function since the host (which performs all the system calls) or the host to satellite communications medium becomes the bottleneck.

In this chapter references made to the satellite processor do not distinguish between the actual processor and the kernel code (mini-kernel) running on the processor. Instead, it is intended that such references refer to both the processor and the kernel code it runs.

3.3 Transparency Requirements

The most important objective in using the satellite method is to increase the processing power of the system without having to alter the host UNIX operating system itself. However, an increase in power would be of little use if processes using it are not, in a sense, part of the original system. The most important aspects of transparency are:

- Interaction with a remote process (process to process and process to peripheral) should be no different to that of a process running on the host.
- Remote processes should be initiated as easily as local processes are under UNIX.

Unfortunately it is not possible to achieve the first of these ideals, and this is discussed further in section 3.7. The second objective is difficult due to the fact that when starting a remote process two pieces of information must be provided:

- The fact that a *remote* process must be started, and
- which satellite processor is to run the process.

The simplest solution is for the user to provide the information at the command line level. In order to make the operation transparent, however, the command line processor (or shell) must be capable of deducing the information. For example, the header of the executable file may contain information as to where the process should run. Another possibility is that the command line processor maintains information on the computing loads of the various satellite processors and sends processes to the processor with the lightest load. In the system created by Barak and Shapir a special process called the *scheduler* is used to allocate satellite processors to the agent (Barak and Shapir use the word *monitor*)

processes. The scheduler communicates with the monitors by means of a small area of shared memory.

It will always be necessary to indicate that a process is remote for two reasons. Firstly, a satellite processor may not be identical to the host processor, therefore programs may have to be specifically compiled to run on the satellite processor. Secondly, certain programs run slower on the satellite processors and a user may wish to run such processes on the host (see section 3.8 on performance).

3.4 Services Provided by the Satellite Processor

The satellite processor provides a number of services which enable the host system to download and run processes.

3.4.1 Allocate a slot

The satellite processor maintains a process table which has a fixed number of process slots for the storage of the process details. Before a process can be created on the processor a slot must be allocated from those available. The host system can then uniquely identify a remote process by its satellite processor number and remote process slot number.

3.4.2 Allocate memory

Having read the executable file header, the UNIX host uses this service to allocate the required amount of memory for the new program. In the case of an EXEC system call the satellite processor will also deallocate memory used by the process before the call. If the satellite processor does not support a virtual memory system, the allocation attempt may fail. In this case the EXEC call will also fail.

3.4.3 Start a process

This service includes the remote starting of a process from the UNIX host and the continuation of a process that has performed an EXEC system call. These two cases are different in that in the case of the *execing* process the satellite processor knows the initial value of the stack pointer. If the process has been created remotely then the initial stack pointer must be supplied by the host.

3.4.4 Create a child process

Creating a child process following a FORK system call is entirely different to the procedure for starting a process. The child process must begin running at a point in the program identical to that of the parent process. Unlike the process that has just performed an EXEC, the child process must be ready to receive a reply to a system call as soon as it begins running.

3.4.5 Terminate a process

This service is provided to stop the execution of a process on the satellite processor. The satellite processor deallocates the process slot and memory occupied by the process.

3.4.6 Signal a process

This service provides a means of sending a software interrupt to a remote process. The satellite processor forces the process to begin the execution of an interrupt handling routine. The address of this routine is stored by the satellite processor, but the signal number must be provided by the host.

3.4.7 Read and write memory

When there is a need to transfer large blocks of data to and from a satellite processor the host uses this service. For example, when a remote process does a WRITE system call it only sends the normal three parameters (file descriptor, buffer address and number of bytes) to the host. The host then reads the data directly from the satellite processor memory using the *read* memory facility.

3.5 The Agent Programs

When a system call is issued the parameters are packaged into a message by the satellite processor and sent to the appropriate agent process. During this time the remote process simply waits for the reply. The agent process running on the host performs the following procedure:

1. Wait for a system call message to arrive.

2. Decode the message determining the system call and its parameters.
3. Make the system call and/or perform any other task associated with that system call.
4. Place the result into a reply message.
5. Send the reply to the remote process.
6. Return to step 1.

Most system calls simply require the agent to unpack the parameters, make the system call, package the reply and send it back to the waiting remote process. However, certain calls require a bit more work and these are discussed in more detail in the following sections.

In all cases the message exchange between a satellite processor and an agent is carefully coordinated to avoid deadlock. This is ensured by the fact that if a message is sent it is clear that the destination process *will* eventually be ready to receive the message. In the case of the agent, each of the steps 2, 3, 4, 5 and 6 are of finite length. Step 5, in particular, is of finite length because the remote process that issued the system call *is waiting* for a reply. Therefore, if a message is sent to an agent it will eventually receive the message. However, UNIX software interrupts make things a little more complicated than this. The problem is described in section 3.5.4.

3.5.1 The EXEC System call

To perform the EXEC system call on behalf of the remote process, the agent does not actually make an EXEC call. Instead, the agent downloads the executable file to the satellite processor.

Before downloading the program, the agent checks the status of the file. If the file is executable and the process has the correct permissions, the agent reads the file header. The file header contains the code, stack and heap sizes required by the process. This enables the agent to send an appropriate memory allocation request to the satellite processor. If any of these actions fail, the agent will send an error reply to the remote process.

The program is downloaded using the *write* memory facility on the satellite processor. Following the downloading of the program, the agent sends a request to the satellite processor to continue the execution of the new program. On successful completion of the

EXEC call, no reply is sent to the process (as is always the case in UNIX), and the agent program returns to the main loop to wait for the next system call.

3.5.2 The FORK System call

On receiving a FORK call message from the remote process, the agent issues a FORK system call itself, thereby creating another agent process. If the FORK fails the agent sends a message to the satellite processor to cancel the FORK call. This is necessary because when the remote process issues the FORK system call the satellite processor allocates a new slot and memory for the child before the FORK is sent on to the agent process. When the satellite receives a FORK cancel message it deallocates the slot and memory already reserved.

Figure 3.2 illustrates the procedure followed and the exchange of messages during a successful FORK. After making the FORK system call (3) the *parent* agent sends a message to the satellite to create a remote child process (6). Meanwhile, the child agent must wait for the remote child process to begin running before sending a reply (10). This synchronization is performed by means of a host IPC[2] (8). At the end of the procedure both agents reply to their respective remote processes (9,10).

3.5.3 The SIGNAL System call

When an agent process starts running it traps all the possible UNIX signals[3]. When a SIGNAL system call is received from the remote process the agent makes a note as to whether the process wishes to trap, ignore or be terminated on receipt of the signal.

Signals ignored by the remote process are ignored by the agent as well. All other signals are trapped by the agent process. When a signal is received by the agent there are two possibilities:

1. If the remote process has trapped the signal, the agent must send a request to the satellite processor to interrupt the process.

2. If the signal was not trapped, the agent must request that the process be terminated.

[2]Inter-process communications.
[3]The SIGKILL signal cannot be trapped, this problem is discussed in section 3.7.1.

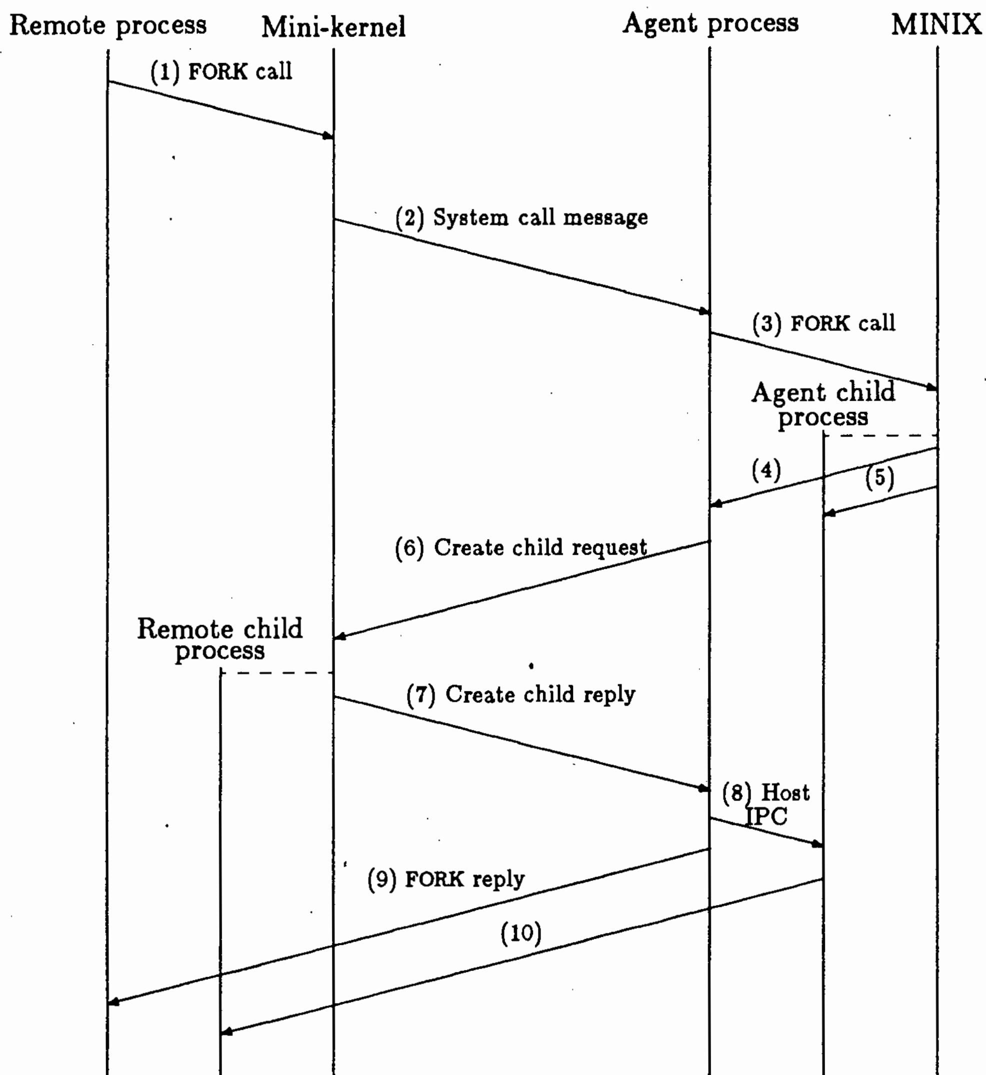

Figure 3.2: Message exchange for a FORK call.

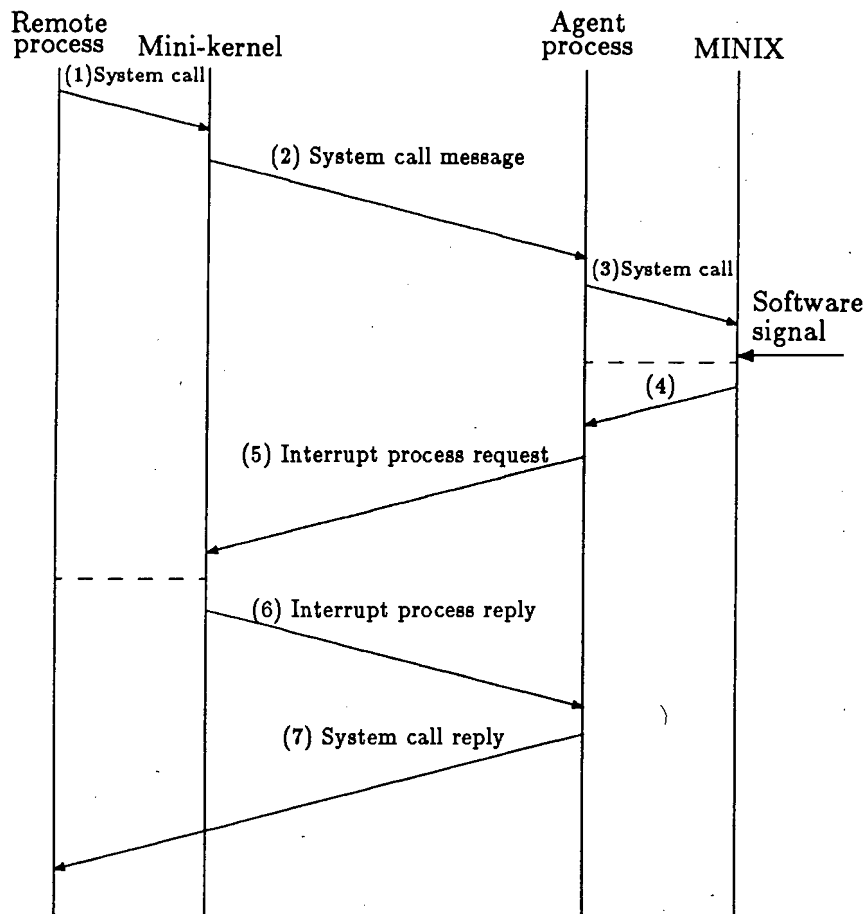

Figure 3.3: Signal received during system call.

If the remote process is terminated the agent itself will exit. There are significant problems with this procedure that limit the effectiveness of the satellite method. These problems are discussed in section 3.7.

3.5.4 The KILL System call

The main difficulty with the KILL system call is with the receipt of signals. These problems are generally timing related, requiring very careful coding to ensure that all possibilities have been accounted for and deadlock cannot occur. The timing problems that can be solved are mentioned here and those that have not been solved are described in section 3.7.4.

When an agent is interrupted there are two possibilities: either the agent was performing a system call on behalf of the remote process or not. Figure 3.3 illustrates the message exchange that occurs if a signal occurs during a system call. The remote process must be

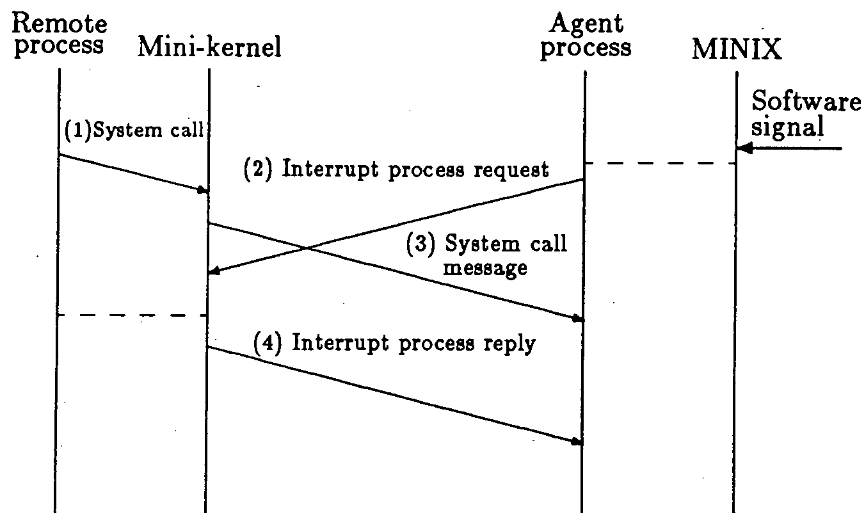

Figure 3.4: Signal received at the same time as a system call is done.

interrupted at the moment the system call ends (i.e. begin executing the interrupt routine immediately after receiving the system call reply). The agent must therefore ensure that the satellite processor receives and completes the interrupt request (5) before the reply to the system call (7) is sent. In order to do this the satellite processor must send a reply to the agent (6) when it has completed the interrupt process request.

However, the reply message results in a further problem (figure 3.4). There is a possibility that the remote process issues a system call (1) and the agent receives a software interrupt at more or less the same time. The system call message (3) may then arrive at the agent when the agent is expecting a reply message (4) from the satellite processor. There are two ways of dealing with this problem:

1. The system call can be buffered until the reply from the satellite processor is received.
2. A reply is only sent by the satellite processor when the remote process has done a system call.

The first solution makes the agent program quite a bit more complicated while the second requires an additional field in the message to the satellite processor to indicate if a reply is required. In the MINIX-based system the second solution was used.

Another problem with signals occurs when the agent receives a signal, requests the termination of the remote process and then *exits* itself. If the remote process had issued a system call just before it was terminated there may be no agent left to receive that

request. This possibility must be taken into account by the driver (as described in the following section), which should simply discard a message bound for a destination that no longer exists.

3.6 The Processor Driver

In order to couple the satellite processors to the host machine the additional processors may be regarded as peripherals. The following sections describe three possible ways of accessing these peripherals.

3.6.1 Terminal Lines

The first possibility is that each processor may be connected to a serial port. This system has the advantage of not having to change the host operating system at all. This method is used in the NMSU ring-star system [Karsh83]. Since a number of agents may be communicating over each terminal line a multiplexer process is required for each line. The multiplexer would be a user level process which would use a form of IPC (messages, shared memory, etc.) to communicate with the agents.

Although a serial port is a simple solution, in most cases it would be considered too slow. For example, transputers can transfer data at a rate of up to 20 Mbps while, in comparison, serial ports do not run much faster than 19200 bps, approximately 1000 times slower!

3.6.2 Kernel Driver

To achieve better performance the driver should take the form of a standard UNIX peripheral driver and be linked directly into the kernel. This will give it access to hardware interrupts and a direct connection to the satellite processor through the system bus, possibly even DMA[4] transfers. The driver would then provide the usual *read* and *write* UNIX peripheral interface. This scheme was used by Barak and Shapir [Barak80].

Since data (a message) may arrive at any time from an satellite processor, the kernel driver cannot be expected to buffer the information indefinitely. It is therefore necessary to provide a permanent reader at the user level. The agents cannot call the driver directly

[4]Direct Memory Access.

as this is inherently ambiguous. After receiving a message from the driver the reader unpacks the message to determine the process identity of the destination agent. It then stores the message until an agent requests a message from the reader. Since there will only ever be one message pending per agent process the reader only requires one buffer per agent.

3.6.3 Kernel Driver Process

In the MINIX-based system a third method was used, due to the unique structure of the MINIX operating system. Device drivers in MINIX are processes. These processes are called tasks and are created when the system boots. Tasks have priority over normal user level processes in MINIX.

The MINIX satellite processor driver was written as a kernel task. The agents communicate with the driver via a special message passing system provided by MINIX. As a result, the conventional *read/write* peripheral protocol was not used.

3.7 Limitations of the Satellite Method

Complete transparency in running remote processes is not possible. The specific reasons for this are discussed in the following sections.

3.7.1 The SIGKILL Problem

The biggest problem with the satellite method is the SIGKILL signal. As mentioned before, the agent must trap all signals in order to pass them on to the remote process. Unfortunately, under UNIX SIGKILL cannot be trapped by any program, not even a **superuser** program.

An agent, receiving this signal on behalf of its remote process is terminated before it can tell the satellite processor to terminate the remote process. The result is that the remote process is not actually terminated by the signal as it should have been. If the remote process now performs a system call there is no agent to receive the call. Even worse is the fact that if the remote process is in an infinite loop it will continue to consume processing time indefinitely and, at the very least, continue to occupy a process slot on the satellite processor.

The first and most obvious solution is to alter UNIX to allow the SIGKILL signal to be trapped. Since most UNIX users do not have access to the operating system source code, this may not be possible. In addition it is also dangerous, as the signal is often the last resort to remove a 'run-away' process. Nevertheless this method was used in the MINIX-based system since all the MINIX source code is available and easily changed.

Another possibility is to have the parent of the agent request the termination of the remote process when the agent has been terminated by a SIGKILL signal. Unfortunately, this may require the UNIX shell and initial processes to be altered as both these processes may be parents of agents. Alternatively, an *agent parent* process may be created whose only task is to run agent processes. When the agent parent has some spare time, it can collect exited agents and check to see if their corresponding remote process needs to be terminated.

The nature of the SIGKILL problem is such that it cannot be ignored and, unfortunately, a lot of work must be done to solve it satisfactorily.

3.7.2 Problem with Pipes

The reading and writing of large blocks of data must be done in stages for two reasons:

- There is a limit to the size of the buffer provided by the agent for data transfers done on behalf of the remote process.
- There is a maximum package size for data transmission from the satellite processor to the host.

This introduces a problem when writing and reading to and from pipes: a number of small writes is not equivalent to one large write even though the total bytes transferred may be the same. For example, if a user has two processes exchanging data by means of a pipe, the user can assume that if blocks of a fixed size are written to a pipe then blocks of the same size can be read from the pipe. However, if such a write is broken into more than one write by an agent, the reader may get less than a complete block if the read happens to occur before an entire write completes.

Increasing the agent buffer does not solve this problem but does make it less likely to occur. This problem was not solved in the MINIX-based system and the users of the system should be made aware of it.

3.7.3 The EXIT Value

When a process is terminated by a signal, the signal number is returned as part of the exit code. Since an agent must trap all signals that can potentially terminate the remote process, the agent itself is not terminated by the signals but instead exits voluntarily. In doing so it is not possible for the agent to return the exit code with the signal number in the correct place as this is filled in by the operating system. Since the agent was not terminated by the signal the exit code is incorrect.

As a result, the parent of the process will not be able to determine the actual cause of termination. There is no way of solving this problem without altering the operating system.

3.7.4 Signal Timing

The main problem with signals on the satellite processors is that they are about an order of magnitude slower then those on the host. This is due to the fact that the signal must travel as a message to its destination as opposed to being a hardware operation. Therefore it is far more likely to happen that a process receives a certain signal and before it can retrap that signal receives another and is terminated as a result. A typical example of this is a user pressing the *DEL key* on the keyboard (causing a SIGINT signal) twice in quick succession, unintentionally causing the user shell to terminate.

It is often the case that the SIGNAL system call has been made by the remote process but has not yet been received by the agent by the time the second signal arrives. There is little that can be done about the slowness of signals and users should be made aware of the problem.

3.8 System Performance

The satellite method only increases the processing power of a system. It was found that I/O bound jobs in fact run slower on the satellite processors than on the host system. The remote procedure calls done by the remote processes were found to be approximately 14 times slower than local calls. By way of comparison, this is much slower than accesses to a remote file server observed by Cheriton and Zwaenepoel in the V system [Cheri83]. They observed an approximate four times slow down. The only explanation for this is the

System call	Time in μseconds		Multiple
	MINIX	Satellite	
OPEN/CLOSE	2290	33604	14.67
READ (1 byte)	3160	66746	21.12
WRITE (1 byte)	4210	50143	11.91
LSEEK	2282	18954	8.30
FSTAT	2510	33342	13.28
GETPID	1140	16670	14.62

Figure 3.5: System call times.

slow data transfer rate achieved over the transputer link to PC bus connection [Hoffm89].

The table in figure 3.5 shows the average times for a number of system calls. The communications overhead, indicated by the figures, is over 80%. The read and write times are for 1 byte transfers. Reads and writes done by a remote process require additional data transfers depending on the amount of data read or written. Each additional data transfer takes approximately 33.3 ms, making the communications overhead much greater in this case.

The transputers used in the MINIX-based system are about 3.5 times faster than the host PC/AT. It was found that CPU bound jobs running on the transputers do not significantly degrade the performance of the host and were, in fact, hardly noticeable if run in the background. When the MINIX-based system is configured with a number of satellite processors a number of processes can be run in parallel. Such a configuration can be used to provide additional computing power for a number of users. For example, 5 users running predominantly CPU bound jobs would bring most systems in the mini-computer range to a near stand-still. However, if a number of satellite processors are connected to the system the degradation is far less noticeable.

The satellite method is very much limited in its application. It will solve the problem of systems that simply require some additional processing power to perform complex calculations (e.g. fourier transforms, mechanical integration, queueing network solution etc.). Institutions that require processing power to aid in the storage and retrieval of data (e.g. typical business applications) will definitely not solve their problems by coupling additional processors to their system using the satellite method.

Chapter 4

The Design of DISTRIX

The previous chapter described the MINIX-based system in which a number of satellite processors were connected to a host machine running MINIX. An important result of the MINIX-based system was the creation of the mini-kernel (the kernel code running on the satellite processor). In this chapter the design of DISTRIX as a whole is described. This includes the mini-kernel, the central kernel and the intervening communications network. Figure 1.2 in chapter 1 illustrates the overall design of DISTRIX, consisting of a central kernel and a file server connected to a number of satellite processors by means of a communications network.

The design of the various components of DISTRIX was motivated by two factors:

- The exclusion of deadlock.
- Conformity to the underlying transputer hardware.

In conforming to the transputer hardware, DISTRIX was designed as a number of independent processes communicating via transputer channels. Since the transputer only supports blocked unbuffered message passing it is necessary to ensure that if a message is sent by one part of the system it will eventually be received by another. The process structure and the rigid interconnection of processes in each of the system kernels ensures that this is the case.

The DISTRIX file server is not totally independent of the central kernel and must, for example, be kept up to date with the creation, termination and location of the user-level processes in the system. Details on the design of the file server are not within the scope of this thesis, although communications between the central kernel and file server are an

integral part of the design of DISTRIX and have been indicated at the appropriate points in section 4.4. In the section on the design of the central kernel, section 4.3.4 provides details of the file server interface.

In the following section the communications network is discussed. Section 4.2 discusses the design of the mini-kernel and the design of the central kernel is described in section 4.3. The central/satellite system call protocols are listed in section 4.4. A new system call, MOVE, was added to DISTRIX and is described in section 4.5. In the final section (4.6) the DISTRIX boot procedure is outlined.

4.1 The Communication Network

The communications network connects processes on the satellite processors with those on the central processors. In order to do this the satellite processors and central processors are each given a system-wide unique *node* number. All messages passed by the communications network have a destination node number, which allows the communications network to forward the message as required.

Each satellite and central processor has an interface to the communications network known as the communications layer (see sections 4.2.3 and 4.3.1). Processes which communicate over processor boundaries send to and receive messages from the local communications layer. Each such process is given a unique process number for that processor. The communications layer uses the process number to identify the specific process for which a message is bound when it is received from the communications network. Messages sent via the communications layer therefore contain both a destination node number and a destination process number.

The communications network itself consists of a number of interconnected processors called *exchanges*. The exchange processors are connected to other processors via the physical links of the transputers. The communications layer of the satellite and central processors send and receive messages to and from the exchanges of the communications network along the transputer links.

Exchanges receive a network message along any link, and forward it along some other link depending on the ultimate destination of the message. In order to do this, each exchange is given a routing table. On receipt of a message, the body of the message is

examined and the destination node number extracted. This address is then used to select (from the routing table) the link on which the message is to be forwarded. The routing tables are set up in such a way that a message is moved closer to its destination each time the message is forwarded. Details on the implementation of the DISTRIX communications may be found in appendix A section A.5.

At this stage the DISTRIX communication layer does not include any data protection facilities. That is, messages may be forged and/or corrupted without detection. The problem of data corruption is not a serious one since the transputer links are 100% reliable under normal conditions. The possibility of forged messages makes it impossible to ensure security in DISTRIX. A form of encryption is required to enable the receiver to verify the source of a message, for example, the use of a digital signature [Needh78]. Data encryption is one of a number of important aspects to be addressed in the future (see section 5.2.1 in the next chapter).

4.2 The Design of the Mini-kernel

The mini-kernel consists of a fixed number of concurrent transputer processes connected by a number of channels. The mini-kernel has a three level, layered design, as illustrated in figure 4.1. At the highest level is the user layer containing the user processes. Below this is the kernel layer which controls the operation of the processor. At the lowest level is the communication layer which provides an interface to the communications network. There is a set protocol for communication between layers. This protocol determines how messages are formatted and passed. Each layer and its functions is discussed in the sections that follow.

4.2.1 The User Layer

The user layer consists of a fixed number of transputer processes which have the task of running the user programs. There is, therefore, an upper limit to the number of user programs that can be run by each satellite processor at any one time. The user processes are numbered *1*, *2*, ..., *n* in figure 4.1.

Processes that are idle (not executing user code) wait on a message from the kernel layer and thus consume no processing power. After a user program has been downloaded

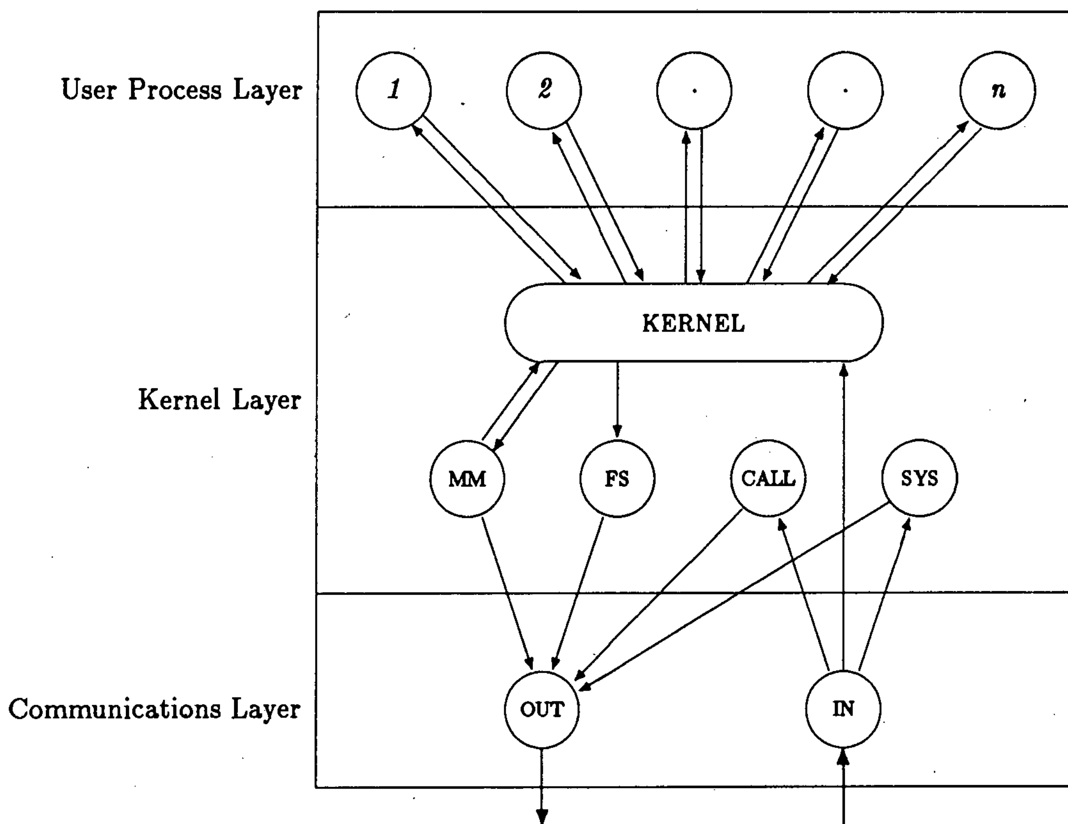

Figure 4.1: The design of the mini-kernel.

into memory by the kernel layer, a user process is selected and the initial instruction pointer and stack pointer values are sent to it which tells the process to begin executing.

Each user process has two channels connecting it to the kernel layer. One channel is used to make system call requests and the other is used to receive system call replies. At a programming level, the library handles the system call to channel communication conversion.

The user to kernel level communication protocol consists of a fixed length message. The field structure of the message varies according to the message type (i.e. the type of the system call). In this way messages containing integers, pointers and strings of bytes are possible. Details on the running of user programs (the implementation of user-level or DISTRIX processes) are given in appendix A section A.4.

4.2.2 The Kernel Layer

This layer controls the execution, signaling and termination of user programs executed by processes in the user layer. It consists of five transputer processes. The label used for each process in figure 4.1 is placed in brackets following each section heading.

The kernel process (KERNEL)

The **kernel** process controls all communication with the user layer. The process waits to receive a message along any of the user process request lines. On receipt of a system call message the information is interpreted and some validity checks are done. If an error is found the **kernel** process may reply immediately along the processes reply channel.

Correct system calls are passed on to either the **fspass** process, if the call is bound for the file server (eg. OPEN, CLOSE, READ, WRITE, etc.), or the **mmpass** process if not a file system call (eg. FORK, EXEC, WAIT, etc.). The **kernel** process also receives messages from some other processes in the kernel and the communication level and passes them to the appropriate user process along a reply channel.

A kernel to communication level message consists of 4 fields: a length, a network node number, a process number and a block of data. The length refers to the number of bytes in the block of data. The node number refers to a particular processor in the system and the process number uniquely identifies a process on that node. In messages received from the communication layer, the network node number parameter is omitted.

The kernel process converts the message from the fixed length format used by the user layer into the above format, and vice versa, before passing the message on.

The mmpass process (MM)

This process checks system calls bound for the central kernel. These include all system calls which involve process management (eg. FORK, EXEC, WAIT, KILL, BRK, etc.).

In some cases work is done before the message is forwarded to the communications layer. An example of this is the FORK system call. On receiving a FORK system call the **mmpass** process will attempt to allocate a process slot and the required memory for the fork, as this is a prerequisite to the call working. If the attempt fails the **mmpass** process replies immediately to the user process without bothering the central kernel with a call that would fail.

In one case, namely BRK, all the work is done by the **mmpass** process. This is due to the fact that the allocation of additional memory to a process is an entirely local problem.

The fspass process (FS)

The **fspass** process checks calls on the way to the file system. Since the file system is removed from the central kernel, messages may follow a very different route to those going to the central kernel. In this case it is up to the **fspass** process to pass the message on to the appropriate process in the communication layer.

The mmcall process (CALL)

This process interacts with the central kernel in order to complete a system call. A number of services (already mentioned in chapter 3 but included here for completeness) are provided to perform various functions, namely:

1. Allocate a slot (process in the user layer) on the mini-kernel.
2. Allocate a fixed amount of memory to a particular slot.
3. Restart a process that has just done an EXEC.
4. Start a child process.
5. Terminate a process.

6. Force a process to execute a signal handling routine.

These services are evoked by the central kernel by means of a message to the **mmcall** process. The actions performed are completed in a finite length of time and a reply is sent to the central kernel.

The sys process (SYS)

The **sys** process reads and writes the processor memory on request. In the case of a write a message is received containing the absolute address, the number of bytes, and the data to be written.

A read request contains the address and the number of bytes to be read. The **sys** process sends the data that was read with the reply. Since read and write requests may be done by any process on any node in the system, all requests contain the network address of the caller. The **sys** process uses the network address to reply to the caller.

4.2.3 The Communication Layer

The processes in this layer communicate with the rest of the system along the physical links of the transputer. Every active link has two processes, one for receiving data and the other for sending data on the link. In figure 4.1 only one active link is shown. Messages bound for remote nodes must be passed to the correct communication process in order that the message is most efficiently routed to its destination.

Messages received from the kernel layer are packaged in the form of a length (consisting of 2 bytes), and a message body before sending along the system network. The message body consists of 3 fields: A node number (2 bytes), a process number (2 bytes) and a variable length data field. On receipt of a message from the communication network the body of the message is examined, the process number and other fields are extracted and the message passed on to the appropriate process in the kernel layer. Messages bound for the user processes are sent to the **kernel** process.

4.3 The Design of the Central Kernel

The central kernel consists of three layers, as illustrated in figure 4.2. The lowest layer is

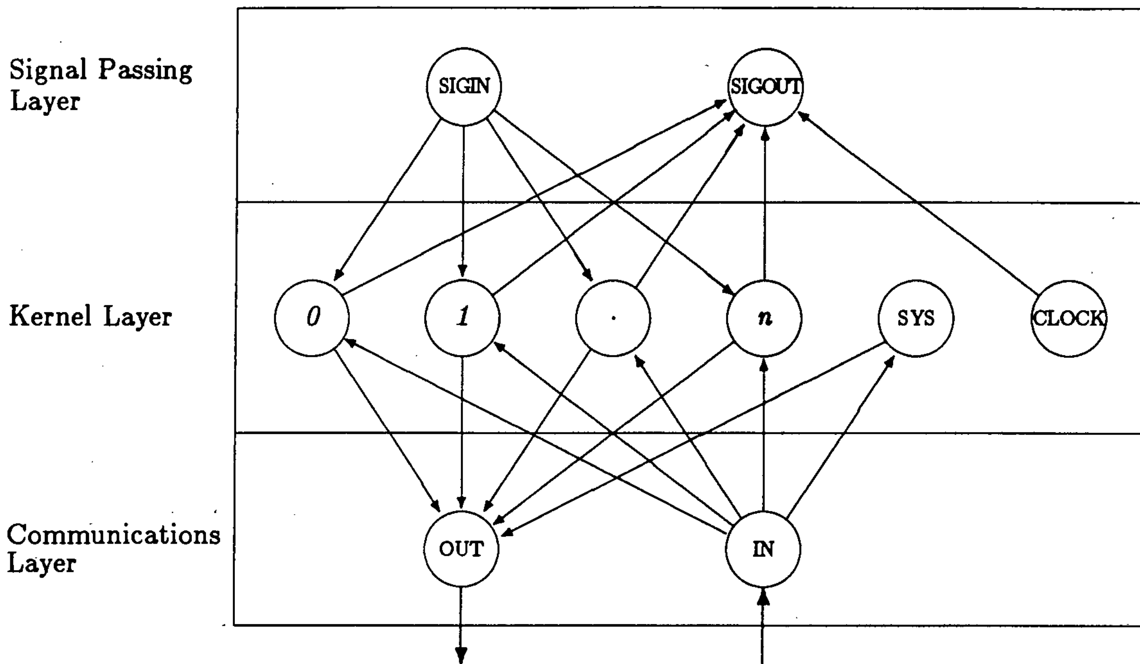

Figure 4.2: The design of the central kernel.

the communication layer, above this is the kernel layer and the highest layer is a specialized signal passing layer.

4.3.1 The Communication Layer

The communications layer of the central kernel is identical to that of the mini-kernel. The format of messages passed from the central kernel layer to the communication layer and vice versa is also the same as in the mini-kernel.

4.3.2 The Kernel Layer

The kernel layer is responsible for coordinating the creation and termination of all user-level processes in DISTRIX as a whole. This layer consists of one process, known as the user agent, for every possible user-level process in the system. The agents are numbered *0*, *1*, ..., n in figure 4.2. As in UNIX, there is a fixed upper limit to the total number of user-level processes in the system at any one time. In DISTRIX this limit is determined by the number of agents available. There is also a limit to the number of user-level processes per satellite. In general, the number of slots on a satellite is less than the number of agents, and the total of the slots on all the satellites is greater than the number of agents.

On the central kernel a global process table is maintained containing information such as: system address, process identity, group identity, parent/child relationships and signal information. This information is accessible to all agents.

An agent may be either 'active' or 'idle'. Each active agent corresponds to a user process running a user program on a satellite processor. An active agent waits for a message (a packaged system call) to arrive from its user process. The following system calls are executed by an agent on behalf of its user process: FORK, EXEC, MOVE (see section 4.5), EXIT, WAIT, SETUID, GETUID, ALARM, PAUSE, KILL, SETGID, GETGID, GETPID and SIGNAL. During the execution of these system calls the agent may have to examine and update information in the global tables, inform the file system of a change in the user-level process structure and send requests to the mini-kernel for work to be done (details in section 4.4). On completion of the system call the results are packaged and sent back to the waiting user process.

When a signal is to be sent from one process to another, the agent of the sending process sends a message to the signal passing layer which buffers and forwards the signal message as soon as the agent of the signalee is ready. The kernel to signal layer protocol consists of an agent number and a signal number. In messages arriving from the signal passing layer the agent number is omitted.

In addition to the user agents, the kernel layer has a clock process. The clock process is responsible for sending alarm signals to agents. A time counter keeps track of the number of seconds since boot time. A list of agents and the time they should be signaled is maintained and is accessible to all processes in the kernel layer. The list is kept in order of earliest time first. To set a timer an agent updates the list with the appropriate agent identifier and time. The clock process awakes once every second and updates the time counter. It then checks the list to see if any pending timers have become current and, if so, it sends an alarm signal to the appropriate agent via the signal passing layer. The details on the implementation of the clock process are given in appendix A section A.7.

The kernel layer also has a **sys** process as does the mini-kernel. This process is used to transfer data from the file server, when the central kernel reads executable files. The **sys** process is identical to the corresponding process in the mini-kernel (section 4.2.2) and therefore an agent reading or writing a file presents an identical interface to the file server as would a user process.

4.3.3 The Signal Passing Layer

The only function of this layer is to pass signal messages from one agent to another. Signals are queued as the target agent may not be ready to receive a signal at all times. The signal passing layer serves to make the passing of messages an asynchronous event. It consists of two processes:

The sigin process

This process waits for a signal message from an agent or the clock process. On receipt of a message it places the signal number in a queue for the destination agent.

The sigout process

The **sigout** process periodically scans all the queues. If it finds an outstanding signal for a particular agent it checks to see if the agent is ready, if so, the signal message is sent.

For further details on the implementation of the signal passing layer see appendix A section A.6.

4.3.4 The File Server Interface

The file server interface is complex due to the fact that in accessing the file system a central kernel agent reverses its basic roll as a server and becomes a client. There are two aspects to the file server interface. The central kernel to file server requests (in which the agents act as clients) and the file server to central kernel requests.

Central kernel to file server requests

The central kernel agents need to access the file server for a number of reasons:

1. To read executable files[1].
2. To inform the file server of the current process configuration, and user process identities.
3. To tell the file server to unblock a process when the process has been signaled.

[1]The agents would also have to access the file server to write core dumps but this is not done by DISTRIX at present.

In order to simplify the file server code the central kernel is considered as *one* client by the file server. As a result, only one agent may access the file server at any one time. Semaphores [Dijks65] are used to ensure mutual exclusion from the file system functions, such as *open*, *close*, *read* and *write*. However, this does not prevent agents from reading executable files concurrently from the file server by using different file descriptors.

File server to central kernel requests

One agent on the central kernel is dedicated to servicing requests from the file server. In performing system call services, an agent only relies on processes that always act as servers, namely the **sigin** process in the signal passing layer and the **sys** and **mmcall** processes on the mini-kernel. As a result, deadlock cannot occur between the file server and the central kernel.

The only service required by the file server is that of signaling user-level processes. This is either in the form of a SIGQUIT or SIGINT interrupt (caused by the user pressing certain keys on a keyboard) or a SIGPIPE signal (caused by a process writing to a broken pipe). The file server agent processes these requests in exactly the same manner as signals are normally processed on the central kernel.

4.4 The System call Protocol

This section describes the procedure that is followed by both the mini-kernel and the central kernel when a process issues a system call. The basic method of handling a system call has already been described, but in the case of a number of system calls (in particular those processed by the central kernel) the procedure is far more complicated and therefore had to be carefully designed.

4.4.1 The EXEC System call

The EXEC call as received by the mini-kernel contains 3 parameters, namely: the length and address of the initial stack of the new program and the name of the file to be executed.

The initial stack is constructed by the EXEC library routine from argument and environment pointers supplied by the user program. The EXEC procedure is as follows:

1. The mini-kernel (**mmpass** process) receives the EXEC call message.

(a) The length and address of the initial stack of the new program is stored.

(b) A message containing only the executable file name is forwarded to the appropriate central kernel agent.

2. The central kernel (user agent) receives the EXEC system call message.

   (a) The agent makes a call to the file server to open the executable file.

   (b) Executability status is checked and then the header of the executable file is read.

   (c) A message is sent to the mini-kernel requesting the required amount of memory.

3. The request for memory is received by the mini-kernel (**mmcall** process).

   (a) If the memory is available it is allocated to the slot.

   (b) The initial stack is checked and copied to the stack area of the new memory.

   (c) If the above operations were successful, the process doing the EXEC stops its execution of the current program and enters an idle state.

   (d) The memory map of the process is updated.

   (e) A reply is sent to the central kernel informing of either success or failure, including memory map details on success.

4. The reply is received by the agent. If the call is successful up to this point the EXEC call is committed to working completely.

   (a) The program image is read from the file system and copied to the correct place in satellite memory using the resident **sys** process.

   (b) Effective user and group identities are adjusted according to *set uid* and *set gid* bits of the executable.

   (c) The file server is informed of any new effective group and user identities.

   (d) A message is sent to the mini-kernel telling it to restart the user process.

5. A message containing the initial instruction pointer of the process to be started is received by the **mmcall** process in the mini-kernel.

   (a) The user process, currently idle, is sent a message containing initial stack and instruction pointers.

   (b) The user process begins execution of the new program.

   (c) A reply is sent to the central kernel agent.

6. The agent receives the reply.

   (a) No reply is sent to conclude the EXEC call.

   (b) The agent prepares to receive the next system call or a signal.

This completes the procedure for the EXEC system call.

4.4.2 The FORK System call

The procedure to perform the FORK system call is as follows:

1. The mmpass process in the mini-kernel receives the FORK system call message.

   (a) A check is done for a free slot (i.e. idle user process), which will become the child process.

   (b) An attempt is made to allocate a block of memory equal to that of the calling process (parent).

   (c) The parent process information is copied to the child.

   (d) The new memory map information is stored as part of the child process information.

   (e) If all the above steps were successful the FORK call, including the child satellite slot number, is sent to the parent agent.

2. The FORK call message is received by the parent agent in the central kernel.

   (a) An attempt is made to allocate a global slot (i.e. user agent) to the child process.

   (b) If unsuccessful the mini-kernel is told to cancel the FORK and deallocate the memory and slot it has just allocated.

   (c) If successful the global parent data is copied to the global child data area.

   (d) The child's satellite slot number, parent global slot number and process identity are updated.

(e) A message is sent to the mini-kernel to start the child process.

3. The start message is received, containing the child process's agent number as allocated by the central kernel.

   (a) The parent process image is copied to the child memory area.

   (b) The instruction and stack pointers for the child are calculated from those of the parent.

   (c) Request and reply channel addresses are patched into the child processes static data area.

   (d) The child process's execution is started by forcing it to perform the operation of receiving a message as if it had just executed a system call.

   (e) A reply is sent to the waiting agent.

4. The reply is received by the user agent.

   (a) The file server is informed of the new process and its details.

   (b) A special signal is sent via the signal passing layer which causes the child agent to awake and send a reply to its waiting user process.

   (c) A reply is sent to the parent user process containing the child process identifier.

   (d) The agent prepares to receive a message or signal.

4.4.3 The EXIT System call

The EXIT system call is handled by the mini-kernel and central kernel as follows:

1. The EXIT call message is received by the **mmpass** process in the mini-kernel.

   (a) The exiting user process is forced to terminate execution of the program and enter the idle state.

   (b) The EXIT call along with the exit status is passed to the central kernel.

2. The user agent receives the EXIT call. The *general procedure for an exiting process* is followed:

   (a) If the parent is not waiting the agent is placed in the em zombie state.

(b) Otherwise, if the parent is waiting the file server is informed of the termination of the process.

(c) If the exiting process has children they become children of *init*[2].

(d) The waiting parent is woken by sending a reply message containing the exit status of the child to the parent user process.

(e) No reply is sent to the user process that made the EXIT call.

(f) The agent slot is freed, the agent enters an idle state, waiting for an activating signal (see section 4.4.2 4b).

4.4.4 The WAIT System call

The user process that does a WAIT system call but has no exited children is suspended by the agent. This is done simply by not sending a reply message. The reply message is, in fact, only sent later when a child exits (see section 4.4.3 2d above).

1. The WAIT call is received and is forwarded directly to the central kernel.

2. The message is received by the user agent.

   (a) If the calling process has any children in the zombie state the *general procedure for exiting processes* is followed (section 4.4.3 2).

   (b) If the calling process has children but none have exited the process is suspended by not sending a reply.

   (c) If the process has no children an error code is returned in a reply message.

   (d) The agent waits for the next call or signal.

4.4.5 The ALARM System call

The ALARM system call causes a SIGALRM signal to be sent to the calling process after a certain amount of time. The DISTRIX implementation is as follows:

1. ALARM system calls received are passed on directly to the central kernel.

2. The agent receives the alarm call containing the number of seconds still to elapse before the alarm signal is to be sent.

[2] *Init* is the initial process in UNIX. One of its tasks is to collect all orphaned children in the system.

(a) The agent either sets the alarm (section 4.3.2) or cancels a previous alarm call if the number of seconds indicated is equal to zero.

(b) A reply is sent to the user agent.

(c) The agent prepares to receive another system call or signal.

4.4.6 The PAUSE System call

A process that executes a PAUSE system call is suspended until it receives a signal. The procedure is simple:

1. The PAUSE call is forwarded to the central kernel as soon as it is received.

2. The PAUSE system call message is received by the user agent.

   (a) The agent does not send a reply message to the waiting user process.

   (b) The agent waits for a signal from the signal passing layer.

4.4.7 The SIGNAL System call

Using the SIGNAL system call a process indicates which signals it wishes to ignore and those it wishes to trap. The default action is to terminate the process on receipt of a signal.

1. The SIGNAL system call is received by the mini-kernel.

   (a) The address of the signal handling routine is stored.

   (b) The message is sent to the central kernel.

2. The SIGNAL message is received by the user agent.

   (a) Depending on the value of the parameter the signal will either be caught, ignored or terminate the process on receipt.

   (b) A reply message is sent to the user process.

   (c) The user agent prepares to receive the next system call message or a signal.

4.4.8 The KILL System call

In this section the procedure for sending a signal as well as the actions performed on receipt of a signal are given. Figure 3.4 in chapter 3 illustrates how a system call message from a user process can be received before the ***interrupt process reply*** message is received. Two solutions to this problem were suggested. The solution used in the MINIX-based system was to have the mini-kernel only send a reply to the agent when the agent received the signal during the processing of a system call (solution 2 in section 3.5.4).

This solves the problem in the case of the agents running under MINIX but does not work for the agents in the central kernel. This is due to the fact that an agent in the central kernel does not know if its corresponding user process has issued a *file* system call. Therefore the agent always receives a reply to an ***interrupt process*** request. Since a system call may arrive before the reply is received, a receive is not specifically issued for the ***interrupt process reply*** message, instead it is received at the same point system call messages are received (see 6 below).

This solution corresponds to a slight variation of solution 1 in section 3.5.4. Instead of buffering the system call it is actually processed, regardless of whether the ***interrupt process reply*** has been received yet. The procedure for the KILL system call is given below. Note that the procedure on receipt of a signal from the signal passing layer starts at 3.

1. The mini-kernel receives the KILL system call message.
   (a) The message is sent on to the user agent on the central kernel.
2. The user agent receives the KILL call.
   (a) The agent sends signals to other agents as required by the call via the signal passing layer.
   (b) A reply message is sent back to the user process.
   (c) The agent waits for a system call message or signal to arrive.
3. An agent receives a signal from the signal passing layer.
   (a) Depending on the previous SIGNAL calls the agent decides whether its user process should catch the signal, ignore it, or be terminated.
   (b) If the user process is to be terminated:

i. A message is sent to tell the **mmcall** process on the mini-kernel to terminate the user process (see 4 below).

ii. The agent waits for the reply from **mmcall** ignoring any messages from the user process[3].

iii. The agent slot is freed, the agent enters the idle state and waits for an activating signal (section 4.4.2 4b).

(c) If the signal is to be ignored the agent returns to waiting for a system call or signal.

(d) If the user process is to catch the signal:

i. The **mmcall** process on the appropriate mini-kernel is sent a message telling it to interrupt the user process (see 5 below).

ii. The agent does not wait for a reply but will receive a reply as it does a system call message.

iii. The process returns to waiting for a system call but will not receive any more signals until ***interrupt process reply*** (see 6 below) is received.

4. The **mmcall** process on the mini-kernel receives the ***terminate process*** message.

   (a) The user process is forced to stop execution of the user program and enter the idle state.

   (b) The program memory and slot are freed.

   (c) A reply is sent to the central kernel.

5. The mini-kernel receives the ***interrupt process*** request.

   (a) The instruction pointer of the user process is changed, forcing it to execute its signal handling routine.

   (b) A reply is sent to the user agent.

6. The ***interrupt process reply*** message is received from the **mmcall** process on the mini-kernel.

[3]A system call message may arrive from the user process but is thrown away since the process is about to be terminated.

(a) The agent checks to see if the user process is paused and, if so, a reply is sent.

(b) If the user process is suspended on a WAIT call a reply is sent.

(c) If none of the above then a message is sent to the file server telling it to reply to the user process if it was suspended on a pipe read or terminal read, etc.

(d) The agent prepares to receive the next system call message or signal.

4.4.9 The BRK System call

The BRK system call is handled exclusively by the mini-kernel.

1. The BRK call is received.

   (a) The break address is checked to see that it is within the heap area of the user process.

   (b) A reply is sent to the user process.

4.5 The Problem of Relocating Processes

In DISTRIX, as in all distributed operating systems, a method of distributing the workload amongst the available processors had to be found. Initially it was necessary to decide whether the task of process distribution would be performed by the operating system or whether it would be done by the user.

Since automated process distribution is a complex and vast aspect of distributed operating systems it was decided not to place DISTRIX process distribution completely in the kernel. Leaving process distribution to the user (or **superuser**) in UNIX requires (at very least) the addition or enhancement of a system call. An example of this would be to add a parameter to the FORK system call telling the system where to place the child process. Alternatively a system call could be added which, given a process identity and destination processor, would move that process to the specified satellite.

In DISTRIX, not wanting to change the specification of existing UNIX system calls and, hoping to maintain simplicity, it was decided that processes would only be able to relocate themselves. This is done by a system call named MOVE which takes one parameter, the destination processor number. This call proved to be fairly simple to implement and, in fact, flexible in its use. Both the initial process and shells etc., can *move* child processes

by performing a MOVE after the FORK but before the EXEC call. In addition, any process can be moved if it is forced to execute a MOVE call following the receipt of a certain signal.

The procedure followed by the mini-kernel and central kernel on receiving a MOVE system call is detailed below:

1. The mini-kernel receives the MOVE call message.
   (a) Information such as, instruction pointer, stack pointer, memory map and destination satellite for the move are packed into a message.
   (b) The message is sent to the user agent.
2. The agent receives the message including other details on the moving process.
   (a) A request is sent to the mini-kernel on the destination satellite to allocate a slot.
3. The destination mini-kernel receives the slot allocation request.
   (a) A slot (idle user process) is allocated if one is available.
   (b) The result of the allocation attempt is returned.
4. The reply is received by the central kernel.
   (a) If unsuccessful an error reply is sent to the user process that made the call.
   (b) A memory allocation message is sent to the mini-kernel on the destination satellite.
5. The destination mini-kernel receives the allocation message.
   (a) An attempt is made to allocate the required memory to the new user process.
   (b) If successful the user process's memory map is updated.
   (c) The result of the operation is returned, including memory map details.
6. The result of the allocation is received by the user agent.
   (a) If unsuccessful an error reply is sent to the user process.
   (b) The program image is copied from the source user process to the destination user process using the **sys** tasks on each satellite.

(c) A message is sent to the source mini-kernel to terminate the old user process.

7. The source mini-kernel receives the message.

    (a) The user process is forced to stop execution of the user program and enter the idle state.

    (b) Memory and slot occupied by the user process is freed.

    (c) A reply is sent to the central kernel.

8. The reply is received by the user agent.

    (a) A message is sent, including process information such as the stack pointer and instruction pointer, to enable the destination mini-kernel to start the user process at the correct point.

9. The destination mini-kernel receives the message.

    (a) The user process is forced to do a receive to place it in a state identical to that of the original user process.

    (b) A reply is sent to the central kernel.

10. The reply is received by the central kernel.

    (a) A message is sent to the file server to inform it of the new location of the user process.

    (b) A reply to the MOVE system call is sent to the new user process to inform it that the call was successful.

This ends the procedure for the MOVE system call.

4.6 The Boot Procedure

The DISTRIX boot procedure is conducted by the central kernel. The process is according to the usual UNIX convention for system boot.

1. The various processes of the central kernel begin execution.

    (a) The user agent of *init* sends a message to the file server to open the *init* executable image file.

   (b) The header of the file is read and size information extracted.

   (c) A message is sent to a mini-kernel to allocate a slot.

2. The **mmcall** process on the mini-kernel receives the allocate slot request.

   (a) A slot is allocated.

   (b) A reply containing the slot number is returned.

3. The reply is received by the agent.

   (a) A memory request for the initial process is sent to the mini-kernel.

4. The request to allocate memory is received by the **mmcall** process.

   (a) The central kernel attempts to allocate the required amount of memory to the new user process.

   (b) The result is returned to the central kernel.

5. The reply is received.

   (a) The initial process image is read from the file server and downloaded using the **sys** task on the mini-kernel.

   (b) Effective user and group identities of the initial process are set to **superuser**.

   (c) A message is sent to the file system informing it of the existence of a new process.

   (d) The mini-kernel is told to start the initial process, given the instruction pointer.

6. The start message is received by the mini-kernel.

   (a) The user process is sent a message telling it to begin execution of the program and the user process becomes active.

   (b) A reply is sent to the central kernel.

7. The central kernel receives the reply.

   (a) The user agent prepares to receive the first system call or signal.

From this point the initial process takes control of the booting process and, following the procedure set out in the *inittab* file, configures the system by forking and moving various processes.

Chapter 5

Conclusion

In the previous chapters all aspects of DISTRIX have been described. This includes: the design methodology, the specific transputer related issues concerning the implementation of UNIX on transputers, the development of the mini-kernel and the MINIX-based system in chapter 3, and finally, the overall design of DISTRIX. Most of the implementation details have been placed in the appendix following this chapter. The product of implementation (i.e. the exact state of DISTRIX at this time) is described in the following section.

DISTRIX has great potential as a vehicle for further investigation and research. Section 5.2 takes a look at the future of DISTRIX.

5.1 The Current Status of DISTRIX

Approximately a year has been spent in the development of DISTRIX. This section describes the prototype DISTRIX operating system in terms of hardware and usability.

5.1.1 DISTRIX Hardware

DISTRIX is currently running on a network of up to 5 transputers. Figure 5.1 illustrates the configuration of DISTRIX. The central processor, file system processor and between 1 and 3 satellite processors are all connected to a single exchange transputer by means of the transputer links.

The DISTRIX file server is connected to two PC server machines by means of link to PC bus adaptors. Both PCs run hardware simulators, one is a disk interface known as the *mass*

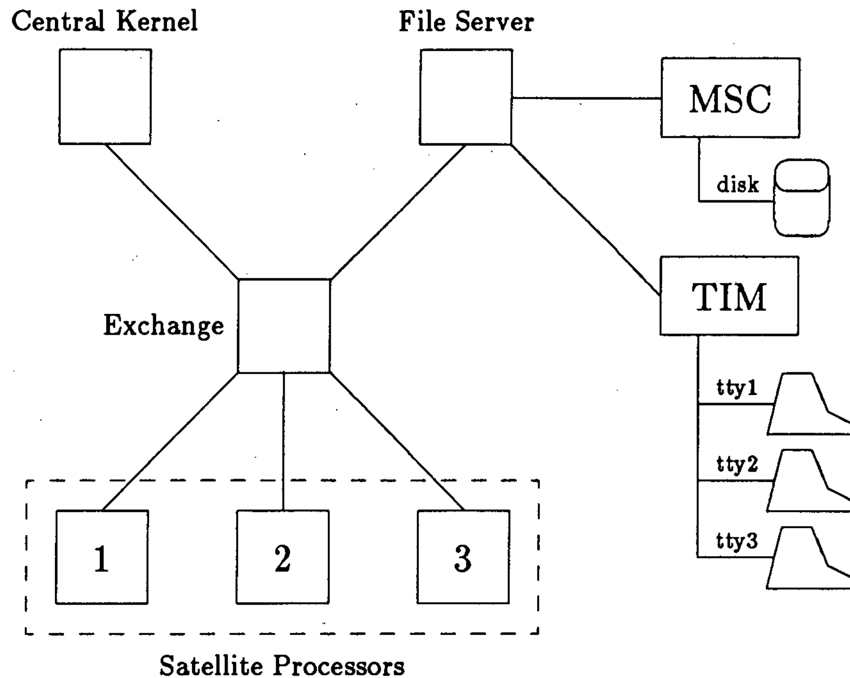

Figure 5.1: Current configuration of DISTRIX.

storage controller (MSC[1]) and the other a terminal interface called the *terminal interface module* (TIM[2]). The mass storage controller allows DISTRIX to access a simulated fixed disk and floppy disk drive. The terminal interface module supports a maximum of 3 terminals and as a result DISTRIX is limited to a 3 user configuration.

Each of the transputers is mounted on a PC board which is slotted into the PC bus connector. At the moment the entire system, consisting of 3 boards, is housed in the MSC server machine.

5.1.2 Booting the DISTRIX Operating System

The DISTRIX operating system is stored on the host PC. It consists of a number of concurrent *C* programs [Kerni78], written with the Logical Systems C compiler and libraries [Logic88a] (see section A.3 for details). To boot DISTRIX a network loader program is executed on the PC. The loader examines a configuration file which describes the network and which program runs on each of the transputers. After the programs have been downloaded the network loader program executes the mass storage controller simulator

[1] The hardware version of the MSC is produced by Parcytec [Malur88], West Germany.
[2] The hardware version of the TIM was under development at the IE at the University of Stellenbosch.

program. By this time the PC running the terminal interface module simulator must be running, so that it is ready to receive the first command from the file server.

Once running, the central kernel makes a call to the file system to read the *init* processes executable file. The executable is downloaded to one of the satellite transputers and the first user level process is started. The *init* process then opens and reads the system configuration file which tells it how many terminals are active and where (i.e. on which satellite processor) it is to run each of the *login* processes. It then forks the login processes and 'moves' them to the appropriate satellites.

The *init* process is currently the only process in DISTRIX that performs the MOVE system call. As a result, no processes are moved after *init* has placed the *login* processes on various satellites. In this way DISTRIX can be configured to run the processes of each user on different satellites, or all on the same satellite depending on the configuration file.

5.1.3 Using DISTRIX

At the moment, using DISTRIX is a bit tedious due to the lack of a native *C* compiler. Programs must be compiled under DOS using the cross-compiler. The DISTRIX root file system is also constructed under DOS. Programs are imported into DISTRIX by placing the executables and other files in the file system prototype file used to build the file system. At this stage DISTRIX is unable to check and correct a file system disk image if it becomes corrupted and, as a result, the file system is often rebuild.

Making DISTRIX even more difficult to use is the fact that file system calls are even slower than those of MINIX from which DISTRIX was derived. For example, an interactive editor is almost unable to keep up with a normal rate of typing under DISTRIX. According to Kuyper Hoffman [Hoffm89] DISTRIX file system calls are approximately 3 times slower from the user's point of view than the same calls under MINIX. A number of factors account for this:

- The data throughput rate achieved over the transputer links is a slim 25% of the theoretical 400 Kbytes/sec link transfer rate.
- Data transfer accounts for 85% of the time taken by the file system calls.
- The DISTRIX devices are simulated rather than being true hardware devices.

- The file server is connected to the devices over PC link adaptors which achieve an even slower data transfer rate than the transputer link-to-link connections.

Further details on the file system call evaluation and DISTRIX data transfer rates may be found in Kuyper Hoffman's thesis [Hoffm89].

Although file system calls are slower than under MINIX, DISTRIX performs much better than MINIX under load. In fact, when two processes are run on the same satellite, one I/O bound and the other CPU bound, DISTRIX file system calls are about 12 times faster than those in MINIX under the same conditions. This is largely due to the fact that the system calls are being serviced by a remote server under DISTRIX, thus an I/O bound job consumes no processing time at all once a system call has been issued. In addition, the CPU bound job does not affect the service time of the system call.

CPU bound jobs are, in general, 3.1 times faster under DISTRIX than they are under MINIX as a result of the high speed transputer processor. In addition, DISTRIX experiences no degradation at all when running two CPU bound jobs on different satellites, while the same load under any uniprocessor system causes the run time to double.

5.2 The Future of DISTRIX

DISTRIX, although limited as an implementation of UNIX, has shown that a transputer based UNIX is feasible. DISTRIX also demonstrates a linear increase in user processing power by the addition of satellite processors. Perhaps the greatest advantage of DISTRIX is that it may be used as a platform on which to base research into many aspects of distributed computing.

The basic implementation of DISTRIX may be improved by data protection, better data routing and satellite recovery mechanism. Research topics that may be addressed in the context of DISTRIX are process distribution, distributed file systems and parallel programming and debugging. Each of these topics are discussed in the sections that follow.

5.2.1 Data Protection

The encryption of messages passed by the communications network in DISTRIX is essential to ensure the security of the system as a whole. Although DISTRIX includes the standard UNIX file protection facilities which control read, write and execution permissions for

owner, group and public, this can be circumvented by a few forged messages to the file server.

Sending a forged message in DISTRIX is quite difficult due to the fact that the sender will have to share a channel or link with another process at some stage. If a channel is shared this could have some unexpected results unless carefully timed. For example, assume process A communicates with process B along channel c. Process C wishes to send a forged message to B along channel c (wanting the message to be mistaken as one from A). There are three possibilities:

1. If C sends the message when B is receiving on the channel the message will be received by B correctly.

2. If C sends the message while A is waiting to send a message along the channel, A will actually receive the message.

3. If C sends the message when nobody is waiting on the channel, the message will either be sent correctly if B does a receive or C will receive a message from A if A does a send.

This strange behavior is due to the nature of the transputer's `in` and `out` machine code instructions which control the receiving and sending (respectively) of messages along channels. The second channel instruction (`in` or `out`) performed on a channel determines the direction of the message exchange. If 2 above were to happen, it will probably cause both process A and process C to hang and possibly disrupt the operation of the mini-kernel. This problem is discussed further in section 5.2.3.

Since finding the addresses of internal channels is tricky, processes that send forged messages will probably send them directly to the transputer hardware links. The sharing of hardware links also causes problems but they are less well defined then those listed above. It has been observed that if two processes send a message to a hardware link at the same time, the result is a larger message of some mixture of the two. Garbage messages such as this should be detected by the communications network and some form of recovery initiated.

As a result, data protection on the transputer includes a lot more than simply encryption. However, if a forged message is sent correctly by a process, the only means of

detecting such an occurrence is by encryption. For example, a process can change its effective user identity by forging a message from the central kernel to the file server. This can be prevented by having the file server send the central kernel an encryption key at boot time[3], or make use of a public key encryption system[Needh78]. All messages sent from the central kernel to the file server should then be encoded using this key. For further information on encryption in general, books by Denning [Denni82] and Kahn [Kahn67] are recommended. The American National Bureau of Standards produced the DES encryption algorithm [Natio77] (described in [Tanen81], chapter 9) which would be a good starting point for an encryption system in DISTRIX.

5.2.2 Improved Data Routing

In DISTRIX data transfer accounts for a large percentage of time taken to service a system call (file system calls approximately 85% and for the central kernel about 83%). For some calls such as EXEC, READ and WRITE this overhead is even larger. In these cases it may be possible to short circuit the route followed by data in the present system. For example, when the central kernel performs an EXEC call it reads the executable file from the file server and then downloads the information to the satellite. If a special protocol could be established with the file server, the executable data could be transferred directly to the satellite. The READ and WRITE calls on the other hand transfer data from devices to DISTRIX processes and vice versa via the file server machine. Although this was the simplest method from an implementation point of view a more direct route is possible.

A great deal of work needs to be done in the analysis of data flowing in DISTRIX as a whole. Alternative hardware should be found (or existing hardware improved) and software optimized in order to reduce the large overhead that the transmission of data places on system calls.

5.2.3 Satellite Recovery

In order to make DISTRIX more usable than it is at present it should be possible to reboot a satellite processor without disturbing the rest of the system. This would by a useful facility due to the fact that an incorrect or malicious user process can destroy kernel code on the satellites. If this situation can be detected by the central kernel, the satellite

[3]Before any user processes (that may be interested in the key) are running.

transputer can be reset and restarted by downloading the kernel code as is done at boot time.

The processes that were running on the corrupted satellite should *exit* correctly from the point of view of the rest of the system. In this way, if a *shell* is destroyed, the user is logged out or if a *login* process is terminated, the *init* process will create another. This approach does not help if the satellite processor running the *init* process itself is corrupted. The only way to protect *init* is not to allow any user processes to run on the same satellite. In fact, it is quite probable that under normal usage DISTRIX would use one satellite exclusively for system processes such as *init*, *update*, etc.

The main problem with satellite recovery is *when* to do it. Certainly a good deal of work would go into deciding how the central kernel could detect a defective satellite and even how a satellite kernel would decide for itself whether it has been corrupted. One possibility would be for each satellite to have an independent *check self* process which wakes up every second or so, and runs a number of integrity checks on the kernel. The most important test to be made by *check self* is that every transputer process is accounted for and *legal* and running correctly within its program area. If all is well a message is sent to the central kernel. The central kernel will expect an *OK* message from each satellite. If one does not arrive within a certain amount of time the satellite is rebooted. This amount of time should certainly be greater than the frequency of checks done on the satellite, but if it is too long, system response to a satellite crash will be slow. The time waited for an *OK* must be set carefully due to the fact that a delayed *OK* would cause an uncorrupted satellite to be rebooted.

5.2.4 Process Distribution

At present process distribution in DISTRIX is done at the user level, at boot time only. In this way DISTRIX is configured so that all the processes initiated at a certain terminal remain on the same satellite node. A more intelligent form of process distribution would require a measure of the process load on each satellite. One possibility is to count the number of processes running on a satellite periodically and take the average of the last n measurements as the load [Tanen85]. Another method suggested by Bryant and Finkel [Bryan81] uses an estimation of the residual running times of all processes to determine the load. Processes created (by the shell for example) on one processor could be moved

to a processor under less load. It is also possible to allocate a number of transputers to each user, or to have a means of requesting available processors. On initiating a process, the user could specify where the process is to be run, by typing, for example:

```
$ program &2
```

The shell would then run `program` in background on the second processor belonging to the user. To do this the shell would execute the following code segment:

```
if ((pid = fork()) == 0) {
  /* Child process */
  move(user_processor[x]);
  execv(file, argv, envp);
}
```

Where `x = 2` and `file = "program"`.

Another possibility is to take the task of process distribution out of the hands of the user entirely. In this case when a process is created, the central kernel would decide where the process is to run according to the load of each satellite. The MOVE system call would then be unnecessary.

5.2.5 Process Migration

Process migration is distinguished from process distribution in that the latter occurs at process start time while the former may occur at any time during the life of a process. Process migration may be achieved in DISTRIX by a process voluntarily executing a MOVE system call at some point during its execution. The MOVE call that occurs before the EXEC call in the case of process distribution above is not considered voluntary.

The main objective of process migration is to balance processing loads in a more flexible and dynamic manner to that of process distribution. A good deal of research has been done in this area [Barak85, Efe82, Stank84]. Process migration is usually controlled by the system, or a system process. In DISTRIX this can be done with very few modifications by adding to each program an interrupt routine that executes a MOVE on receipt of a certain signal. A process could be written that continually gathers information about the processing load of the entire system and moves processes about to maintain a balance.

If process migration algorithms are written into the operating system itself, greater efficiency could be achieved and some interesting options explored. For example, each

satellite processor could be connected directly to a number of other satellites via spare transputer links. Each satellite kernel could gather process load information about its neighbours [Barak85, Smith79]. Processes can then be moved directly over the links to reduce the difference in processing loads.

Unfortunately, there is an inherent problem with process migration on the transputer. As mentioned in chapter 2 the transputer is devoid of any form of memory protection. Process migration may place the processes of one user on the same processor as those of another, thus negating the only means of protecting one user from another in the distributed transputer environment. However, it is still of interest to investigate process migration under DISTRIX. Besides the obvious academic value of such an investigation, INMOS may enhance the protection capabilities of future generations of transputers.

5.2.6 A Parallel-based File Server

The DISTRIX file server itself may be the source of a great deal of research. One possibility is to parallelize the operation of the file server. At present, calls are serviced in the order in which they are received. This is sufficient when the load on the file server is low. A parallel file server would perform much better under load. For example, a request for a cached block would not be held up by calls requiring disk access.

5.2.7 Distributed File Server

Perhaps the only solution (in the case of DISTRIX) to the problem of an overloaded file system is to create some form of distributed file system. The creators of such a file system should have three objectives in mind:

1. To divide the work of servicing the file system calls.
2. To place peripherals closer (in terms of hardware) to the processes that access the peripherals.
3. To make the file server independent of the central kernel.

To achieve these objectives and to maintain the typical UNIX file server interface is a difficult task, but most desirable. Firstly, in achieving objective 1 not only will it relieve the bottleneck created by a single processor having to service all calls, but it will also relieve the pressure on the communications medium connected to the file server.

By placing peripherals closer to the processes that access them, messages containing data would have less distance to travel and in some cases may not even have to cross machine boundaries. For example, a terminal may be connected directly to each satellite processor.

A file server that is entirely independent of the central processor is probably essential in terms of implementing a distributed file server. As mentioned before, the present DISTRIX file server relies on information from the central kernel concerning the creation, termination and location of processes in DISTRIX. If the file server was distributed, passing such information would become a significant overhead. Instead, processes should identify themselves to the file server when they first make a call. A capability system [Denni66, Levy84, Pasht82] for determining file access permissions should be used in place of the current system in which the central kernel informs the file server of a particular process's access permissions.

Lastly, and certainly not any less important is for the file server to maintain the interface of the UNIX file system. Which, naturally, works against the above mentioned three objectives. Well, who said life was easy?

5.2.8 Distributed Programming and Debugging

Although DISTRIX has been implemented on the transputer it does not use the parallel processing capabilities of the transputer as they were originally intended. The user programs are conventional single threaded processes, but it would be a fairly simple task to write a library of routines compatible with DISTRIX giving a process multi-threaded capabilities.

The satellite kernel would have to take into account the fact that a number of transputer processes may be associated with a single user process. When a process exits, or is terminated for some other reason, all of the transputer processes created by the original DISTRIX process would have to be terminated.

Although the foundations have been laid by the creation of DISTRIX, there is still much work to be done and great opportunities for further research.

Appendix A

Implementation

This appendix contains details concerning the implementation of DISTRIX. Topics discussed are: the transputer microprocessor in the following section, the MINIX operating system in section A.2, the DISTRIX implementation language (section A.3), the implementation of DISTRIX processes (section A.4), DISTRIX communications (section A.5), signals on the central kernel (section A.6) and the central kernel clock in section A.7.

A.1 The Transputer Microprocessor

The transputer microprocessor was specifically designed for large-scale computational problem solving, and it is in this area that most work on transputers is done. The full potential of the transputer may be realized if the problems are of a parallel nature and can therefore be divided amongst a number of transputers. Transputers have been used for applications such as realtime processing, hardware simulations, graphics and image processing, speech recognition, signal processing and fast fourier transforms.

Many unique facilities provided by the transputer hardware were used in the implementation of DISTRIX. This includes the multi-processing capabilities, the transputer message passing functions and transputer hardware links. The following sections provide details of the various aspects of the transputer hardware.

A.1.1 The Transputer CPU

The transputer CPU consists of an *instruction pointer*, a *workspace pointer*, an *operand register* and three general purpose registers, A, B and C, which operate as a three word

stack. The transputer has a 32 bit address and 32 bit data bus; consequently all registers are 32 bits long. The operand register is used to construct constant values and complex instructions. The register stack is used to hold the arguments and result of an instruction and to pass parameters[1] and return values of a function call. Besides absolute addressing memory can be addressed as constant offsets from the workspace pointer. This makes the workspace pointer the most natural choice for the stack pointer in a *C* program[2] which enables easy access to local variables and parameters.

A.1.2 The Instruction Set

A transputer instruction is one byte long. The high order 4 bits contain the instruction code while the low order 4 bits contain data. The execution of an instruction is as follows:

1. The operand register is shifted left by 4 bits.
2. The 4 bit data is loaded into the operand register.
3. The instruction code is decoded and executed.
4. The operand register is cleared.

Larger constants can be loaded using the `pfix` and `nfix` instructions which do not perform step 4.

Although the transputer only has 16 direct operations, more complex indirect operations can be performed using the opr instruction. When this instruction is executed the value in the operand register is considered to be an instruction code which is decoded and executed. Indirect operations always take their arguments from the register stack and, if the operation produces a result, it will be pushed onto the stack.

A.1.3 Transputer Processes

The transputer provides hardware support for the running of concurrent processes. Transputer processes may run at one of two priority levels, high or low priority. High priority processes are always run in preference to low priority processes. Once running, a high

[1]Of course, only three parameters can be passed in this manner and the rest must be placed on the *stack* maintained by the program in memory.
[2]*C* was used in the implementation of DISTRIX, see section A.3.

priority process cannot be pre-empted and will thus run until it performs channel input or output (see section A.1.5). As a result high priority processes are able to share data without requiring additional means of mutual exclusion.

Low priority processes run until they perform I/O or until they exceed their timeslice. If a high priority process becomes ready to run when a low priority process is running, the low priority process is interrupted. When this occurs the register values of the low priority process are stored at a fixed point in memory until the high priority process is descheduled. When the high priority process is descheduled, the register values are restored and the low priority process continues to execute.

A.1.4 The Process Scheduler

The transputer scheduler maintains two process queues of ready to run processes; one for high priority processes and the other for low priority processes. Each queue consists of a linked list of workspace pointers. The scheduler has registers which point to the front and back of these queues, thus enabling it to add processes to a queue as it becomes ready to run and to determine the next process to be run.

When a process is descheduled, a number of words at a negative offset from the workspace pointer are used to store information about the process. This information includes:

- The instruction pointer.
- The message buffer address, if the process was descheduled on I/O (see section A.1.5).
- The workspace pointer of the next process in the queue, if the process is on a queue.

The information, such as the instruction pointer, can be altered but this should not be necessary under normal circumstances. Strictly speaking, if this data is modified (by anything but the transputer hardware itself) the transputer is being made to do something it was not designed to do.

A.1.5 Communication Channels

Transputer processes communicate by passing messages via entities known as channels. Channels are the only means of I/O available to a transputer process. Channel I/O is

point to point (i.e. only two processes should attempt to transfer data along a channel at any time) and blocking (i.e. if a process sends or receives a message along a channel the process is descheduled until another process is sends or receives along the channel). The messages passed along channels consist of a length and a data field of *length* bytes. In other words, messages are of variable length.

In order for two processes to communicate both must have the address of the same channel. A transputer channel is simply a word in memory. The transputer instructions in and out perform input and output respectively along a channel. They take as arguments, the channel address, the message buffer address and the number of bytes to be transferred.

When a process performs an in or out instruction, the transputer deschedules the process, storing the workspace pointer in the word indicated by the channel address. Note that only the message buffer address of the communicating process is stored (section A.1.4); the number of bytes and whether input or output was done is discarded. This information is determined by the second process to do I/O on the channel. When a second process performs an I/O instruction, the transputer copies the number of bytes indicated from the one message buffer to the other. Because the direction of data transference is determined by the second process to perform I/O on a channel it is recommended to only use a channel for uni-directional communication between two processes.

A.1.6 Hardware Links

The transputers only means of communication with the outside world is via eight uni-directional hardware channels. The channels are grouped in pairs to create bi-directional connections, known as links, to other transputers. The in and out instructions are also used to transfer information along the links. This is done by specifying the link hardware address instead of the channel address. The data transfer rate may be as fast as 20 Mbits/s over a link.

Each link has its own processor which enables DMA transfer of data to occur after an in or out instruction has been executed. The process is, however, suspended while the transfer takes place as in the case of the channels. The transputer can be connected to a conventional parallel bus via a *link-adapter*. A server running on the *host* machine can enable the transputer to access the peripherals available to the host. However, peripherals are being developed that connect directly to the transputer link (for example [Malur88]).

A.1.7 The Transputer Clock

The transputer has an onboard clock which is accessible to all transputer processes. Two timer values are maintained, one for each priority level. The high priority timer is incremented every 1 microsecond while the low priority timer is incremented every 64 microseconds.

The `ldtimer` instruction places the value of the current priority level clock on the top of the register stack. The instruction `tin`, timer input, causes a process to be descheduled until the time counter has exceeded a certain value. In this way a process can be suspended for a certain fixed amount time.

A.2 DISTRIX and MINIX

DISTRIX is derived from the MINIX operating system written by Tanenbaum [Tanen87] for the IBM compatible PC. MINIX was written with two goals in mind: modularity and readability. This is due to the fact that Tanenbaum intended MINIX to be used as a teaching aid. In the next section a brief overview of MINIX is presented and in section A.2.2 the reasons for using MINIX are explained.

A.2.1 An Overview of MINIX

Figure A.1[3] illustrates the design of the operating system. The operating system is built on a process management layer which enables the PC to execute a number of independent sequential processes. A message passing mechanism is also implemented at this level as the means of inter-process communication. All interrupts are handled by the process management layer, where they are turned into messages and sent to the appropriate processes.

The only system functions available to processes in MINIX are *send* and *receive*. System calls are done by sending messages to one of the two server processes, the *memory manager* or the *file system*. The memory manager carries out system calls that require memory management, such as FORK, EXEC, EXIT and BRK. The file system carries out all the file system calls (READ, WRITE, MOUNT, CHDIR, etc.).

This division of work was maintained in creating DISTRIX. The central kernel agents process almost all system calls processed by the MINIX memory manager (BRK being the

[3] Reproduced from [Tanen87].

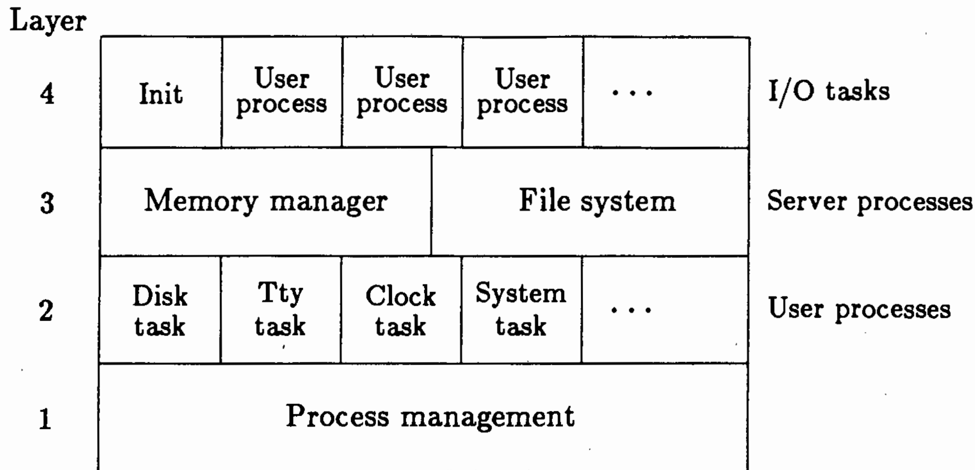

Figure A.1: The structure of MINIX.

only exception), and the MINIX file system was ported to the transputer to create the DISTRIX file server. Creating the central kernel agents by duplicating certain functions of the MINIX memory manager was simplified by having the MINIX source which is clearly written and well commented.

The I/O tasks are special system processes that handle the various devices available to the PC. One task is required for each major device type. The process driver, used by the MINIX-based system, was implemented as a task. Only the server processes send and receive messages directly to and from the I/O tasks. This restriction is imposed by the process management layer in order to prevent processes from directly accessing the hardware and thus bypassing the file system protections. In the MINIX-based system the restriction was removed in order to allow the agents to communicate directly with the process driver.

The user level processes run in layer 4 as illustrated in A.1. This includes shells, editors, compilers and all user-written processes. The user processes in MINIX correspond to the user processes on the satellite processors in DISTRIX. As a result, a large amount of the MINIX memory management functions are performed by the mini-kernel. The combination of the mini-kernel and the central kernel perform the same work as the MINIX memory manager.

A.2.2 Why MINIX?

As mentioned before, DISTRIX has an advantage over UNIX in being clearly written and well documented. In contrast, little documentation exists on any particular implementation of UNIX. In addition to this, UNIX source code is a large monolithic, interrupt-driven program with many optimizations. Not only does this make it difficult to read, but very little of the code would be directly portable to a network of transputers. There are two main reasons for this:

- There is no clear division of function in the UNIX code in order that it may be divided amongst a number of transputers.
- The transputer hardware does not support interrupts.

On the other hand, MINIX solves the first problem by dividing the functions of the operating system into a number of processes and the second problem by not having the processes deal directly with interrupts.

Another major advantage of using MINIX to create DISTRIX was having the source to the MINIX commands, utilities and library functions and the ease with which these programs were ported to DISTRIX. The library was altered to send a transputer message instead of a MINIX message. This was simplified by maintaining the fixed length message format used to make system calls in MINIX (as described in section A.4.4 on the system call interface). Once the library had been altered and compiled most of the MINIX commands and utilities were compiled for DISTRIX using the new library. This includes the MINIX shell, *login*, *update*, *init*, *ls*, *cat*, *cp*, and others. Only very few utilities, such as the editor, were obtained from different sources.

A.3 Implementation Languages

Some problem were encountered in finding a suitable implementation language for DISTRIX. This section describes the DISTRIX implementation language and outlines some of the issues concerning the transputers and high-level languages.

A.3.1 The OCCAM Language

An early version of the mini-kernel was implemented in OCCAM [Inmos84b, May84], the 'native' language of the transputer. This is largely due to the fact that the OCCAM compiler was all that was available at the time. OCCAM proved inadequate for the implementation of the mini-kernel mainly because of the lack of an inline assembler facility[4] or the ability to link in assembler programs.

As a result it was difficult to perform the low level process signaling, termination and creation functions. It was not impossible, however, because INMOS did provide a function called `KERNEL.RUN` which allowed a process to execute a piece of program data. Small machine code routines required could therefore be written by hand and placed in arrays. Work done in creating the early OCCAM version of the mini-kernel was not a waste. Firstly, it proved that what had to be done was possible, and secondly, implementation methods and 'tricks' learnt were reused in the later version.

A.3.2 C Compilers for the Transputer

As a result of the transputer's architecture there is a slight difficulty in the implementation of the high level languages in general. The problem manifests itself when trying to create *load time* relocatable code. This makes it difficult to access global or static variables.

The transputer has no spare registers which can be used to point to the base of a static data area. This is not a problem if the compiler produces code that relies on the absolute positioning of global variables in the transputer memory. However, this is not acceptable to a dynamic system like UNIX which has to be able to place a program in memory wherever there is space.

To make up for the lack of a base register the transputer provides a special instruction called `ldpi` which loads the address of a memory location at a fixed offset from the end of the `ldpi` instruction itself. For example, if x is the address of the end of the instruction `ldpi b` the value $x - \mathtt{b}$ is pushed onto the top of the register stack. Using this instruction programs can locate their global data without the compiler having to make any assumptions as to where the program will be located at run time. All the compiler has

[4]This is not completely true. OCCAM has a construct called `GUY` which allows selective machine code mnemonics to be written directly into the code. However, this proved to be a great source of frustration because none of the useful instructions like jumps, process creation and processes queue register access instructions are allowed.

to know is the value of b (in the above example), which is simply the difference between the address of the end of the ldpi instruction and the address of the global variable[5].

This scheme, however, prevents processes from sharing code, unless they share global data as well. In performing the FORK system call, therefore, the child process cannot share code with the parent process which would be a significant optimization.

Another solution, used by the 3L C compiler [Inmos88], is to pass a pointer to the static data area as a 'hidden' parameter of every function. This method still does not solve the shared code problem unless the pointers are changed when the stack is copied.

With *C* compilers there is a further problem with initialized pointers. Since the initial value of such pointers may be unknown at compile time, code must be generated to initialize such variables after the program has been loaded.

Although the load time relocation problem was solved by the compilers encountered, *none* came close to solving the problem presented by the FORK system call (see section 2.3.2). To solve this problem a new compiler had to be written or an existing one had to be altered.

A.3.3 Logical Systems C

A *C* compiler [Kerni78] for the transputer was required for two main reasons:

1. The MINIX file system (written in *C*) had to be compiled for the transputer.

2. MINIX commands and utilities and other user level programs needed to be compiled for use under DISTRIX.

Considering the problems outlined in the previous section it was fortunate that a *C* compiler and source code became available from Logical Systems [Logic88a]. Having the source to the compiler, assembler and linker proved invaluable to the successful completion of the project. Appendix B describes the changes that were made to the compiler. This version of the compiler was used to compile the DISTRIX user level programs.

DISTRIX itself was implemented using an unaltered version of the compiler[6]. The compiler is, in fact, a cross-compiler that runs under DOS on an IBM compatible PC.

[5]This value can be calculated at compile time, of course.

[6]The facilities provided by the altered version are not required by DISTRIX, and furthermore, the altered version produces both slower and larger code.

A library of functions is supplied with the compiler that enables the programmer to make use of the parallel processing capabilities of the transputer. Both the mini-kernel and the central kernel are parallel *C* programs.

The parallel functions used in the implementation of the mini-kernel and central kernel can be divided into three categories: those functions which manipulate transputer processes, those that enable the use of transputer channels and the timer functions. Full details of the concurrency features may be found in a paper by Jeffrey Mock [Mock88]. Outlined below are just those functions used in the implementation of DISTRIX.

Transputer process functions

Transputer process functions are provided to create, run, deschedule and terminate transputer processes:

1. `void ProcInit(process,function,workspace,WS_SIZE,nparam,p1,...,pn)`

   The `ProcInit` function creates a transputer process. A pointer to a process structure, `process`, must be provided in which the process details are stored. `function` is the address of a function which will be executed by the process. `workspace` is a pointer to a block of memory, of length `WS_SIZE` bytes, set aside as workspace for the process. p1, p2, and so on, are parameters to be passed to the process itself. `nparam` specifies the number words of memory required to store p1 to pn, allowing a variable number of arguments and types to be passed to the process.

2. `void ProcRunHigh(process)`

   This function starts the specified process, `process`, at high priority level. The process begins execution when it is scheduled to run by the transputer micro-code scheduler.

3. `void ProcRunLow(process)`

   This function is identical to `ProcRunHigh` except that it starts a process at the low priority level.

4. `void ProcStop()`

   If a process calls `ProcStop` it is descheduled and removed from the process queue. As a result the process will never run again.

5. `void ProcReschedule()`

   This function causes a process to be descheduled and placed at the back of the *ready to run* queue. The function is most often used to perform a busy wait on a certain resource.

Transputer channel functions

The Logical Systems library include the following functions to perform I/O on channels:

1. `void ChanIn(channel,message,length)`

   The `ChanIn` function performs channel input. `channel` is the address of a channel, `message` is the address of a message buffer and `length` is the number of bytes to be transferred.

2. `int ChanInInt(channel)`

   This function receives a transputer word from a specific channel. The word is transferred in the form of a 4 byte message.

3. `void ChanOut(channel,message,length)`

   `ChanOut` is identical to the `ChanIn` function except that channel output is performed.

4. `void ChanOutInt(channel,value)`

   The `ChanOutInt` function sends a word of data along `channel`.

5. `int ProcAlt(ch1,ch2,...,chn,0)`

   `ProcAlt` allows a process to poll a number of channels for input. It takes as arguments a variable number of channel addresses: `ch1,ch2,...,chn`. The function causes the process to block until one of the channels in the list is ready for input. On completion, the routine returns an index into the parameter list for the process ready for input.

6. `int ProcAltList(channel_list)`

   `ProcAltList` performs a similar function to `ProcAlt`. `channel_list` is a pointer to a zero terminated list of channel addresses that the calling process wishes to poll for input. `ProcAltList` returns an index into the list of the first process ready. This

function and `ProcAlt` favor the channels that are first in the list and consequently some care must be taken in their use.

Transputer timer functions

The timer functions allow a *C* program to easily use the transputer timer. Only two functions were used in the implementation of DISTRIX.

1. `void ProcWait(ticks)`

   This function suspends a process for a specific number of clock *ticks*. If the calling process is a high priority process then `ticks` is the delay in microseconds. For a low priority process `ticks` is the delay in 64 microsecond intervals. For example, to wait for a second[7], a low priority function would issue the call `ProcWait(15625)`.

2. `void ProcAfter(value)`

   `ProcAfter` suspends a process until the timer has exceeded `value`. If a process is not scheduled immediately after `value` has been exceeded, it will be scheduled as soon as possible. If the function is called with a value that is already less than the current timer value, then the process is not descheduled.

In addition to the functions listed above, it was necessary, in the case of the mini-kernel, to write and perform certain functions in assembly code. Fortunately, the Logical Systems compiler allows inline assembly code and therefore these functions were written directly into the source.

A.4 The Implementation of DISTRIX Processes

A process in UNIX is an independent thread of control that passes from user code to system functions and back as system calls are issued and completed. In DISTRIX, the operating system itself consists of a number of processes. These are called system processes. The equivalent of the UNIX process, the logical DISTRIX process, is implemented by the operation of a number of system processes. Some system processes perform the system functions while others execute the user programs. The following sections describe how

[7] Note that this is real time, not processing time.

logical DISTRIX processes (or DISTRIX processes for short) are created and run by the system processes. In section A.4.4 the system call interface from the user's point of view is discussed.

A.4.1 The mini-kernel user process

The user processes are the system processes that have the task of executing user programs. The user processes run in the user level of the mini-kernel (see figure 4.1 in chapter 4). Each user process can communicate with the kernel level via two channels, namely: *request* and *reply*. Messages are sent to the kernel along the *request* channel and messages are received via the *reply* channel. A user process may be found in one of two states:

1. **Active** In the active state the user process executes a user program.
2. **Idle** In the idle state the user process does not execute a user program, but waits to be activated by the kernel.

Active user processes are the user level part of DISTRIX processes, while the other system processes in DISTRIX (particularly the agent processes on the central kernel) constitute the system level part of a DISTRIX process.

There are a fixed number of user processes in the user level. Therefore there is a limit to the number of DISTRIX processes that may be run on a satellite processor. When a user process is started by the kernel it is in the idle state. In the idle state a user process waits on a message from the kernel and therefore represents absolutely no overhead to the transputer. The transition from the idle to active state is as follows:

1. Receive the message consisting of the initial instruction and workspace pointers of the user program from the kernel.
2. Adjust the workspace pointer.
3. Jump to the start of the user code.

Transputer assembler code was used to perform the last two steps, adjusting the workspace pointer and jumping to the start of the user program. In practice it was found that it is necessary to place a short delay between step 1 and step 2 in order to prevent the user process from starting before the kernel has had a chance to reply to the user agent that initiated the execution of the user program (refer to chapter 4 section 4.4.1 5c).

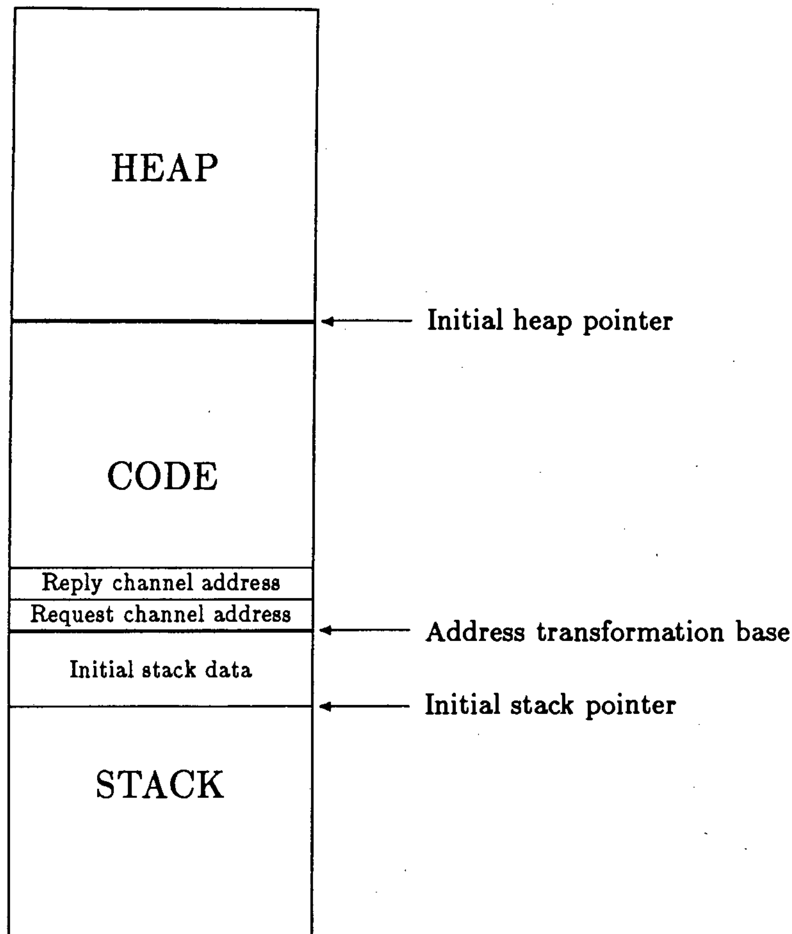

Figure A.2: The DISTRIX process memory map.

A.4.2 The user program image

All DISTRIX processes occupy a fixed amount of memory on a satellite. The memory is divided into three sections: code, heap and stack. The code section includes all of the program and initialized data. Program instructions and data are interspersed by the compiler, therefore no distinction is made by the operating system. The size of the heap and stack areas are placed in the header of the executable file. The address of the start of the code segment is used as the base for address transformation as described in appendix B section B.6.1.

Figure A.2 indicates the position of the various data areas of a program in memory. At the lowest point in memory is the end of the stack segment. Just above the stack segment is the code segment and at the top is the heap area. The heap pointer starts at the base of the heap area. When memory is allocated by the BRK system call the heap pointer is

moved upwards. The stack pointer is initially set to a position just below the start of the code segment. Memory is allocated to the stack by moving the stack pointer downwards in memory.

The first two words of the code segment are used to store the addresses of the *request* and *reply* channels. The channel addresses are filled in by the operating system before the user process begins execution of the user program.

A.4.3 Running the user program

The first step in the creation of a DISTRIX process is the allocation of memory. The amount of memory to allocate is determined by the central kernel which reads the header of the executable file. Following this the central kernel downloads the program image to the satellite processor, placing the information (code and data) in the program code segment.

The mini-kernel then copies the initial stack of the new process to the start of the program stack area. The pointers in the initial stack are adjusted before the stack is copied. The initial value of the workspace pointer (stack pointer) is calculated to be the *base of the code less* the size of the initial stack. The initial value of the instruction pointer is obtained from the central kernel which reads the value from the executable file.

The mini-kernel then fills in the addresses of the communication channels at the base of the code segment. Finally the kernel sends a message containing the workspace and instruction pointers to the user process and execution of the user program begins.

A.4.4 The system call interface

Unlike more conventional system call interfaces, such as the hardware trap, DISTRIX system calls are done by sending a message to the kernel. The message is sent along the *request* channel to the kernel and the result of the call is received on the *reply* channel.

The DISTRIX user library functions handle the transformation of a function call into a message. The general procedure followed by the library functions in performing a system call is:

1. Pack the function parameters into a fixed length message.

2. Send the message along the *request* channel.

3. Wait for a message to arrive on the *reply* channel.

4. Extract the returned data from the reply message.

5. Update reference parameters.

6. Return the result of the system call.

The format of messages to the kernel is identical to that used by MINIX. The DISTRIX library was created from the MINIX library by changing the low level message passing functions. An additional change was made, however, in placing the system call message buffer on the stack instead of in static data, where it is in MINIX. If the message buffer is static data, a problem may arise if a signal occurs during a system call. This problem occurs if the interrupt routine (that is called immediately after the completion of the system call) issues a system call itself. The issuing of a system call overwrites the previous contents of the message buffer, thereby loosing the reply to the previous call.

A.5 DISTRIX Inter-process Communications

The overall design of DISTRIX illustrated in figure 1.2 shows how the central and satellite processors are connected by a communications network. It is necessary to distinguish the central and satellite processors (which are also called nodes) from the processors of the communications network. The distinguishing factor is that nodes are the source and destination of all messages in the system. In order for the communications network to pass a message from one node to another correctly, every node in the system is given a unique number. In turn, every process on a node that can send and receive messages to and from processes on other nodes is given a unique process number. In this way a process can be identified by its node and process numbers.

Each node has a communications layer which serves as the interface between the kernel code running on that node and the low level communications network (see figure 4.1 and 4.2 in chapter 4). The basic components of the communications layer are the **linkin** and **linkout** processes and the basic component of the communications network is the exchange processor.

A.5.1 The linkin and linkout processes

There is a **linkin** and **linkout** process for every active link on a processor node. The **linkin** process is responsible for the input hardware channel and the **linkout** process is responsible for the output channel of the link. Messages that are to be sent over the network are passed to the **linkout** process.

Processes wishing to send a message to a remote process send the **linkout** process 3 integers, a length, a node number and a process number, followed by the message data. The **linkout** process creates a network message from this information. The network message consists of a data field proceeded by 2 bytes indicating the length of the data field. The first 2 bytes of the data field contain the destination node and the second 2 bytes indicate the destination process. Following this the network message is sent out along the link.

The **linkin** process receives network messages from the link. From the message it extracts the destination process number and the message contents. From the process number it determines which process in the kernel is to receive the message. It then send a message consisting of 2 integers, a length and a process number, and the message contents along the appropriate internal channel.

A.5.2 The DISTRIX exchange processor

An exchange receives a network message along one link and forwards it along another depending on the destination node number. There must be enough exchanges in the communications network to fully connect all satellite processors to the central processors[8]. If the number of links on the exchange is n and the number of satellites in the system is greater than $n-2$ then more than one exchange is needed to connect all the satellites to the central kernel machines. Extra exchanges may be used in order to create alternative routes for messages passing from one node to another.

The exchange software

Currently DISTRIX only uses one exchange. This makes the exchange software fairly simple. Each link is numbered from 1 to m. Each node in the system is connected to the exchange. The node number of each processor is defined to be the link number to which

[8] Of course, DISTRIX has two central processors, the central kernel and the file server.

the node is connected. In this way the exchange simply sends a network message out along the link corresponding to the destination node number it finds in the message.

If more than one exchange is required, the message routing procedure is slightly more complex. Each exchange has a routing table which maps a destination node number onto an exchange link number. On receiving a message the exchange looks up the destination node number in the routing table and then sends the message out along the link indicated.

The routing table is supplied to each exchange when the system is booted. The routing tables must be constructed by the central kernel based on the system configuration. In calculating the routing tables the central kernel must ensure that when a message is passed on it moves closer[9] to its destination.

The exchange hardware

The four physical links of the transputer proved to be insufficient for the requirements of the DISTRIX exchange. As a result a number of 'pseudo' links were added to the existing hardware. The pseudo links are memory mapped and as a result are not accessed using the conventional in and out instructions. Instead a process must read or write the appropriate memory locations in order to transfer data. Unlike the hardware links, data can only be transferred across the pseudo link byte for byte. The exchange transputer was developed at the Institute for Electronics at the University of Stellenbosch. The exchange currently in use has eight pseudo links.

A.6 Signals on the Central Kernel

When a DISTRIX process performs the KILL system call, a message is sent to the user agent on the central kernel. The user agent will send a *central kernel signal* to other user agents who then arrange with the mini-kernel to have a software interrupt sent to their corresponding DISTRIX processes. The *central kernel signal* is the subject of this section.

The signal sent from one agent to another is, in fact, a transputer message. However, user agents are not connected directly to one another by means of channels as this could cause deadlock. Instead, the message is sent via the central kernel signal passing layer

[9] The distance between a message and its destination is defined to be the minimum number of exchanges on a route to the destination.

(see chapter 4 section 4.3.3). When a user agent is not servicing a system call it waits for a message to arrive from its corresponding DISTRIX process *or* from the signal passing layer. Since an agent is not able to receive a signal message when it is servicing a system call the messages are buffered by the signal passing layer and only sent when the agent is ready. Each agent has two channels connecting it to the signal passing layer. One is for sending signal messages and the other for receiving messages.

The signal passing layer consists of two transputer processes. The first process, known as **sigin**, receives signal messages and the second, called **sigout** sends them. The **sigin** process uses the `ProcAltList` function to check for a message on any of the input signal channels. Signal messages received from a user agent consists of two integers: an agent slot number and a signal number.

The signal passing layer has a data structure consisting of one queue for each user agent. Each queue consists of signals for the agent in the order in which they occurred. When **sigin** receives a signal message it places the signal number at the back of the agent's queue.

The **sigout** process performs the following procedure:

1. Check each agent queue:
    (a) If the queue is not empty and
    (b) if the user agent is ready to receive, then
    (c) send the first signal in the queue to the agent.
2. Deschedule using the `ProcReschedule` function.
3. Go to step 1.

To check if a user agent is ready to receive a message (in step 1b) the **sigout** examines the value stored at the channel address. If the value is the workspace pointer of the user agent then the agent has done a receive. If the user agent has not done a receive, the value stored at the channel address will be MININT (the most negative word that can be represented on the transputer). The procedure followed by the **sigout** processes would be simpler if the transputer supported output polling of channels, as it does input polling (in the form of the functions `ProcAlt` and `ProcAltList`).

A.7 The Central Kernel Clock

In order to implement the UNIX ALARM system call a clock is required. As mentioned before the transputer has a very accurate real time clock which is easily accessible to *C* programs. The `ProcAfter` function was used to implement a clock process which keeps track of time in increments of one second. This clock is completely independent of the clock used by the file system which is used to date files etc. Since it is only used to implement the ALARM system call, actual time of day is of no interest.

Besides keeping track of the time, the central kernel clock has the task of *signaling* various user agents after a certain number of seconds has elapsed. To do this a list of the pending alarm calls is maintained and checked every second. This is the procedure followed by the clock process:

1. Wait for the next second to pass.
2. Increment the clock timer.
3. Check pending alarm call list.
4. For each time on the list less than or equal to the current clock time:
   (a) Send a SIGALRM signal to the agent.
   (b) Remove the alarm call from the list.
5. Go to step 1.

The alarm call list is accessible to all processes in the central kernel and the alarm calls are added to the list by the central kernel agents themselves. This list is kept in order of increasing timer values. To add an alarm call which will occur in `sec` seconds time, a user agent performs the following procedure:

1. If the agent already has an alarm call pending the call is removed from the list.
2. If `sec` is less then or equal to zero then end of procedure.
3. Add the current clock value to `sec`[10].

[10]Practically speaking, this value can never overflow. The clock value (in seconds) is 32 bits long, representing 136 years.

4. Scan the alarm call list until a time is found that is greater or equal to sec.

5. Insert sec and the user agent slot number into the queue just before this time.

This ends the procedure to add an alarm call to the list.

Appendix B

The Compiler

UNIX creates the illusion of a number of concurrently and independently running programs known as processes. A mechanism is provided for creating and destroying processes. When a process is created, memory is allocated for the process and when it is terminated the memory is released by the system. In UNIX creating a process is known as *forking*. A forking process creates an independently running, identical copy of itself known as the *child* process.

The UNIX fork system function caused a great deal of difficulty in the implementation of DISTRIX on transputers. This is largely due to the lack of memory management support on the transputer. Most micro-processors either have a memory management unit (MMU) on-chip or a support chip is available, providing memory management functions. On other processors a simple base register or segment register system is used. Address transformation, paging and segmentation facilities are absent from the transputer hardware.

For reasons outlined below a software solution to the lack of hardware memory management facilities had to be found. This involved a number of changes to the *C* compiler used to compile programs that run under DISTRIX. This chapter discusses the problem, the solution, how the solution was applied to the problem and finally the implementation of the solution.

B.1 The Memory Allocation Problem

Whenever a block of memory is allocated by the operating system, the requesting process is given a pointer to the allocated block. This pointer is valid until the allocated block is moved. There are two reasons for moving an allocated block:

1. Memory blocks must be packed by the garbage collection routine in order to reduce fragmentation of memory.
2. Code and Data blocks must be copied to create a child process.

Both of these introduce a problem with pointers. In the first the pointers referring to moved memory blocks are no longer valid and in the second the pointers contained in the copied memory blocks are no longer valid. Addresses stored in the child's memory would incorrectly refer to the parent's memory.

The operating system can solve these problems with the aid of hardware memory management capabilities. Since no such facilities exist on the transputer a software solution had to be found. In the search for a software solution, existing hardware solutions were examined and the most suitable of these solutions in terms of implementability and economical usage of space and time was applied.

B.2 The Framework for a Solution

Looking at the problem of the allocated block, a simple solution would be to make the operating system aware of the location of the pointers and then update these values when a block is moved. The only problem with this is that a user may make several copies of a pointer and the operating system would have to adjust all of them. The overhead of this would quickly become overwhelming.

All practical solutions involve some form of address transformation. The *virtual* pointer stored by the program is mapped onto a physical memory address. The process of transformation is either done or controlled by the memory allocation software.

A simple example of a virtual pointer is known as a handle and was used in early systems as a software solution to the problem. A handle is a double indirect pointer. Each handle, stored by the user, points to a master pointer which in turn points to a

block of memory. The master pointer is set by the memory manager when a block of memory is allocated or moved.

The example illustrates the transformation of the virtual pointer (the handle) into a physical pointer. The transformation mechanism used is indirection. This method does not solve the problem of the case in which a block is copied. As a result it is not the answer to the problem which stems from the creation of child processes in UNIX.

B.3 Virtual Memory

The solution is embodied in a comprehensive system known as virtual memory [Baer80, Denni70, Doran76]. Virtual memory is a method for making the amount of memory that a program uses independent of the actual amount of RAM in the machine. In virtual memory the disk effectively becomes the main memory [Shiel86]. To do this virtual memory uses the idea of a virtual address, that is, all addresses are virtual pointers. The result is a virtual address space that maps onto a physical address space by whatever method is used to map a virtual pointer onto a physical pointer. Each process in the system has its own virtual address space created by the memory manager when the process is created. The mapping mechanism makes it independent of the physical means of storage so it is possible to move and copy the memory blocks as required by the memory manager. In particular:

1. The move works because when the physical block is moved, the transformation mechanism is adjusted to take this into account.
2. The copy works because when a memory block is copied it is placed in a newly created virtual address space at the same virtual address as it was in the original. As a result, the virtual pointers stored in the block are still valid.

As mentioned before, the virtual memory system is most famous for freeing a system from having to place an entire program in primary storage. This is due to the fact that the address transformation mechanism can be extended to identify data in secondary storage. However, UNIX can function without virtual memory but cannot function without some form of address transformation. Address transformation mechanisms used can be divided into three categories: paging, segmentation and both [Tompl85]. Some of the information for the following overview was obtained from a case study of the memory management

schemes of various microprocessors [Alper83, Schmi83], the Intel iAPX 286 and iAPX 386 [Wells84, Wells86], Motorola MC68020 and MC68030 [MacGr84, Ruhla87] and the Zilog Z80000 [Phill85].

B.3.1 Paging

In a paging system, allocation of memory is done in fixed length blocks. Memory is viewed as a number of contiguous blocks called pages. Pages are numbered from 0 to n in physical memory. In general a virtual pointer in a paging scheme consist of a page index and an offset. A physical pointer consists of a page number and the offset.

The index is translated into a physical page number using a lookup table. Each process has its own lookup table which describes its virtual address space. Page tables often have a hierarchical structure in order to reduce the amount of memory required to store the table. This is done by only holding parts of the page table in memory at any particular time.

The size of a page determines the length of the offset, and the number of pages in the virtual address space determines the size of the index. The advantage of having large pages is that page tables are smaller, requiring less memory and a faster lookup time. However, large pages lead to what is known as internal fragmentation which occurs when an entire page must be allocated to a program that only requires a small amount of additional memory.

B.3.2 Segmentation

Segmentation eliminates the need for large lookup tables and is based on the fact that, logically speaking, a program may be divided into a number of distinct blocks of information. For example, all programs may be divided into code and data sections. These blocks are called segments. In segmentation the number of segments per process is fixed, but the size of the segments may vary.

A virtual pointer in the segmentation system consists of two independent values: the segment descriptor and the segment address. The segment descriptor identifies a block in memory while the address is the offset into the block. Every process has access to its segment descriptors. When memory is referenced, the appropriate segment descriptor is selected and transformed into a base physical address. The segment address is then added

to the base physical address to get the actual memory point referenced. The length of the segment address determines the maximum segment size of the system. A segment may be of any length up to the maximum segment size.

One advantage of segmentation over paging is that a process can be allocated the exact amount of memory it requires. However, memory fragmentation occurs as segments are allocated and freed in random order. This leads to a situation where sufficient memory may exist to satisfy a request but not in a contiguous block.

B.3.3 Segmentation and Paging

In this method both systems are used in order to reduce internal fragmentation and eliminate external fragmentation as described before. A virtual pointer is first regarded as a pointer in a segmentation system and translated as such. The result is then translated as a paging system pointer.

B.4 Software Implementation

Implementing virtual memory using software is expensive in terms of time and space. The cost in space includes both the additional code required and the memory required to store translation tables.

In providing a software solution to the memory management problem it is not necessary to do address transformation on every memory access as is done by hardware. This would in fact be impossible. Instead, it is only necessary (and sufficient) to perform address transformation on addresses actually *stored* by the program (pointers). The address transformation code should be placed in programs by the compiler. At any later stage the distinction between addresses and data in a program is less apparent. This also means that the user of a high level language need not even be aware of the fact that the program is doing address transformation.

Address transformation will probably be done frequently by a program and it is therefore important the operation is as quick and efficient as possible. Ultimately the hardware available must be considered in deciding which scheme to implement. The following sections outline the strong and weak points of a software implementation of the various memory management schemes.

B.4.1 Software Paging Schemes

To implement a software paging scheme all pointers are considered to consist of an index and an offset. Each process must have a separate page table which maps the index onto a physical address. The process must have a pointer to the start of the table in memory. This pointer should be stored in a CPU register to ensure fast access. If no register is available, placing the table at a fixed place in memory for each process may be considered.

Transformation of the virtual pointer must be done before each memory reference. This prohibits the page table from being greater than one level in most cases and the entire page table should be in memory while a process is running.

A large page size should be used to keep the page table as small as possible. For example, using 16K pages and a 256 entry page table produces a 4 Mbyte virtual address space. The address translation method is as follows:

1. The virtual pointer is shifted right to access the index part.
2. An indexed memory reference is done to obtain the base address of the page.
3. The offset is masked and added to the base address.
4. The physical pointer is used to reference memory.

The main disadvantage of this scheme is that it is very much more expensive than other schemes in terms of space, i.e. a page table per active process. In addition there is a large amount of internal fragmentation due to the large page sizes.

A software paging scheme does not allow complete flexibility in the allocation of memory as code must still be placed in contiguous physical memory blocks. This is due to the fact that there is no way of preventing the instruction pointer from containing a physical address. If the processor has a stack pointer then the stack memory must be handled in a similar manner or the stack must be implemented in software.

An inherent advantage of the paging scheme is that it provides some degree of memory protection provided the virtual to physical memory address translation is not bypassed.

B.4.2 Software Segmentation

There are a number of choices when implementing a segmentation system in software. The most important is the number of segments per process. The segmentation virtual pointer

consists of a segment descriptor and an address. The fastest mechanism would be to have the segment descriptor refer to one of a number of CPU registers which contain the base address of a segment. The number of segments available could be decided according to the number of CPU registers available. If there are insufficient registers for this approach then a table containing the base pointers must be placed in main memory and some mechanism found for a process to access the table. In this case the address translation becomes identical to that of the paging scheme.

There are three possibilities as to how a virtual address may be stored.

1. The segment descriptor may be omitted. In this case each process would have only one segment, or the segments represent specific classes of data, e.g. code and stack, and the segment to be used is implied by the instruction.

2. The segment descriptor and address are stored as two separate words. In this way segments can be as large as the processors physical address space will allow. The disadvantage is that extra memory is required to store each pointer.

3. The last alternative is to store the segment and address as part of the same machine word. The main disadvantage of this is, in doing pointer arithmetic the two values must be separated.

B.4.3 Software Segmentation and Paging

The use of the segmentation/paging combination as a software solution to the memory management problem is probably far too expensive in terms of processing time and, therefore, is not considered in the following section.

B.5 A Transputer Software Solution

The unconventional CPU of the transputer further limits the above mentioned possibilities. The transputer CPU consists of an instruction pointer, an operand register, a workspace pointer and three registers that form a stack. All of the registers have specific purposes:

- The instruction pointer contains the absolute address of the next instruction word in memory.

- The operand register is used to build large operands and complex instructions.
- The workspace pointer is used to access memory in an indexed manner (as offsets from the workspace pointer).
- The register stack is used to store the arguments and results of particular operations.

Further details on the transputer micro-processor may be found in appendix A section A.1. In *C* programs the workspace pointer is most naturally used as a stack pointer. This enables quick and easy access to the local variables and parameters of a function. It is possible to use the workspace pointer in a different manner. However, the stack pointer would then have to be stored in memory. A program has no flexibility in how it uses any of the other transputer registers.

The workspace pointer is, however, very limited in its usage as it can only be used in conjunction with the operand register which, in turn, can only be loaded with constant values. As a result, memory can only be accessed by constant offsets from the workspace pointer. The workspace pointer is therefore not suitable for storing the base of a page table or for storing the base of a segment.

A comparison of the possible transputer software address transformation methods is made below. Assume the operation to load a virtual pointer takes x cycles. In the code segments below the `.LDC` instruction represents a combination of `pfix` and `nfix` commands to load a constant value into the operand register. The code segments are written in Logical Systems transputer assembler [Logic88b]. It is otherwise assumed that the reader is familiar with the transputer instruction set [Inmos87b].

A software paging scheme is very expensive. Address transformations from virtual to physical address takes over 32 transputer cycles.

```
                       ;                               cycles:
                       ; load virtual pointer          x
      .LDC   14        ; number of shifts              1
      shr              ; virtual pointer shifted right 16
      .LDC   _ptable   ;                               4
      ldpi             ; load page table base          2
      add              ; add                           1
      ldnl   0         ; load page base address        2
                       ; load virtual pointer again!   x
      .LDC   0x3FFF    ; offset mask                   4
      and              ; extract offset                1
```

```
add                 ; add to base                    1

                    ;                                32 + 2x cycles
```

In the implementation of software segmentation only one segment should be used as no registers are available to store the segment base values. Under this assumption calculation of a physical address from a virtual address is as follows:

```
                    ;                                cycles:
                    ; load the virtual pointer       x
.LDC    __base__    ;                                8
ldpi                ; load the segment base          2
add                 ; add                            1

                    ;                                11 + x  cycles
```

So the segmentation scheme takes less than half the time of the software paging scheme. Using more than one segment will cause the time for segmentation address transformation to tend towards paging address transformation time. For example, if two words are used to store a segment address, the first word indicating the segment to be used, and the second containing the segment offset:

```
                    ;                                cycles:
                    ; load the segment descriptor    x
.LDC    _stable     ; load the segment table base    8
ldpi                ;                                2
add                 ; add                            1
ldnl    0           ; load segment base              2
                    ; load the segment offset        x + 1
add                 ; add to segment base            1

                    ;                                15 + 2x cycles
```

Not only speed but also increase in program size must be considered in selecting an address transformation mechanism. In this case the fastest method is also the shortest.

B.6 The Transputer C Compiler

A number of existing transputer *C* compilers were considered for DISTRIX. All compilers considered would have to be modified to do address transformation in programs running under DISTRIX. The best solution would have been to write a compiler for DISTRIX from

scratch, however, due to time and manpower constraints this was not possible. Instead, a *C* cross-compiler, assembler and linker including the programs source were obtained from Logical Systems [Logic88a].

The compiler was then modified to perform address transformation. This did, however, prove to be a fairly difficult task as the *C* compiler is a 20,000 line program for which no implementation details were available.

B.6.1 Adjustment Strategy

The address transformation mechanism used was the single segment, software segmentation method described in the proceeding section. The start of code was defined as the *base*. The actual value of *base* is obtained by loading a ***pointer to an instruction*** using the `ldpi` transputer instruction (as was seen in the code segments above).

A variable called **__base__** was defined in the compiler to represent *base*. The declaration of **__base__** is placed in the library[1]. The Logical Systems linker groups sections of code, in order, according to a *segment number*. To ensure that **__base__** is placed at the start of code it is placed in segment 0 and the rest of the code is grouped in segment 1 by the compiler.

As compilation occurs additional code is generated to do address transformation. Code is generated to add the value of **__base__** to all pointers before they are used to access memory. The value of **__base__** is also subtracted from all absolute addresses calculated and stored by the compiler.

B.6.2 The Abstract Syntax Tree

Most modifications done to the compiler were done at the abstract syntax tree level. This is mainly due to the fact that we had no documentation or description of the compiler code 'diversions' used by Logical Systems compiler during code generation.

The modifications took the form of a slight adjustment of the abstract syntax tree. The first change involved an adjustment to an array pointer (see figure B.1). Since an array is an absolute address, the value of **__base__** is subtracted to convert it to a virtual address. In figure B.2, the second change is illustrated. The indirection operator indicates that a pointer is about to be used and as a result the value of **__base__** is added. In

[1]Note, **__base__** is an address and no data is specifically stored at that address.

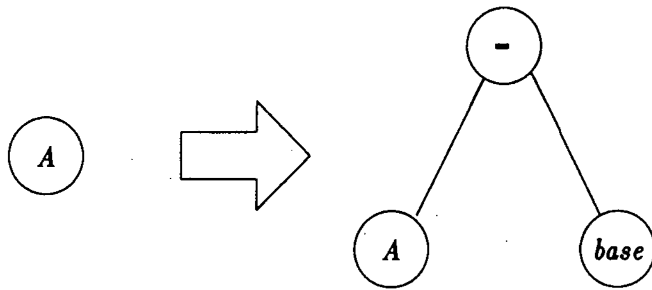

Figure B.1: Conversion of arrays.

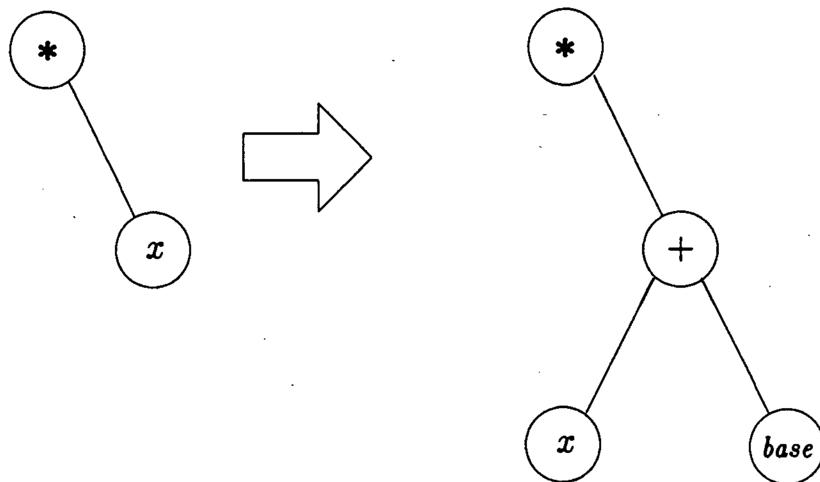

Figure B.2: Conversion of a virtual address to a physical address.

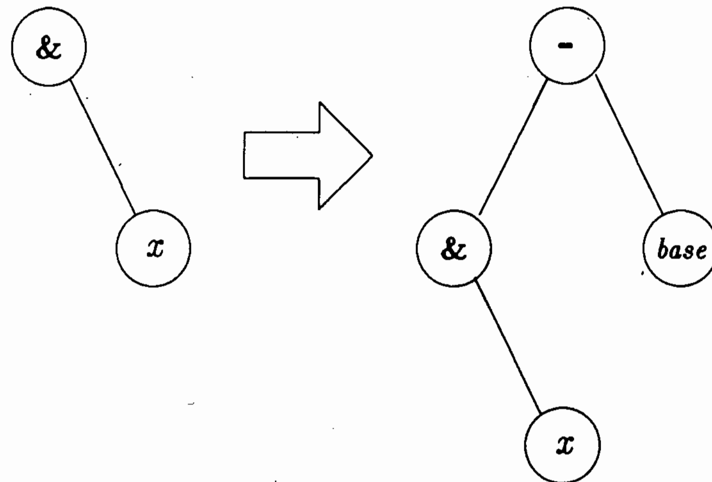

Figure B.3: Conversion of a physical address to virtual address.

the illustration x represents the operations that actually loads the address. In the third change (figure B.3) the address operator, **&**, loads an absolute address, so an adjustment is made to convert this to a virtual address.

B.6.3 Procedure Calls

When a function is called, the transputer `call` instruction places the return address on the stack. The return address is, of course, absolute and must therefore be adjusted. The compiler was changed to add code to the start of every function to load the return address, subtract the value of `__base__` and store it. Later, when a function returns the value of `__base__` is added to the return address before it is used to return control to the calling code.

This may seem like a pointless exercise, but consider a child process that has just been created: if the return addresses on its stack have not been adjusted all of the return addresses refer to the parent's code! As a result, when a function returns, the child would suddenly start executing the parent's program.

B.6.4 Memory Allocation

One of the major problems with the Logical Systems compiler is that of memory usage. The abstract syntax tree and code generation methods all use large amounts of computer memory. This is one reason why the Logical Systems compiler is still a cross-compiler —

it is *too large* to compile itself.

The Logical Systems compiler is compiled under DOS using the MSC (Microsoft C) compiler. A program compiled by MSC is allowed a maximum heap of 64K. The Logical Systems compiler allocates all the memory used to store the abstract syntax tree from the heap. The adjustments made to the compiler increased the size of the abstract syntax tree by a large amount. As a result, the new LSC compiler soon ran out of memory even when compiling small programs. To solve this problem additional memory was allocated from large statically declared buffers.

Bibliography

[Ahuja83] AHUJA, S. 1983. S/Net: A high-speed interconnect for multiple computers. *IEEE Selected Areas in Communication,* November, page 751-756.

[Alper83] ALPERT, D. 1983. Powerful 32-bit micro includes memory management. *Computer Design,* October, page 35-42.

[Almes85] ALMES, G.T., BLACK, A.P., LAZOWASKA, E.D., NOE, J.D. 1985. The Eden system: A technical review. *IEEE Transactions on Software Engineering,* January, page 43-59.

[Bach84] BACH, M.J., BUROFF, S.J. 1984. Multiprocessor UNIX operating systems. *Bell Systems Technical Journal,* Vol. 63, No. 8, page 1733-1749.

[Bach86] BACH, M.J. 1986. *The design of the UNIX operating system.* Prentice-Hall, Inc.

[Baer80] BAER, J. 1980. *Computer Systems Architecture.* Computer Science Press, Rockville, Md.

[Bakke88] BAKKES, P.J., PINA, R., DU PLESSIS, J.J. 1988. *Transputer enhancement: Virtual memory management for the transputer.* Parallel Processing '88, October, Part I.

[Barak80] BARAK, A.B., SHAPIR, A. 1980. UNIX with satellite Processors. *Software—Practice and Experience,* Vol. 10, page 383-392.

[Barak85] BARAK, A., SHILOH, A. 1985. A distributed load balancing policy for a multicomputer. *Software—Practice and Experience,* Vol. 15, page 901-913.

[Bell79] BELL LABORATORIES. 1979. *UNIX time-sharing system—programmer's manual.* CBS College Publishing.

[Birre84] BIRREL, A.D., NELSON, B.J. 1984. Implementing remote procedure calls. *ACM transactions on computing systems,* Vol. 2, No. 1, page 39-59.

[Blair85] BLAIR, G.S., MALONE, J.R., MARIANI, J.A. 1985. A critique of UNIX. *Software—Practice and Experience,* Vol. 12, No. 12, page 1125-1139.

[Brown82] BROWNBRIDGE, D.R., MARSHALL, L.F., RANDELL, B. 1982. The newcastle connection—Or UNIXES of the world unite! *Software—Practice and Experience,* Vol. 12, page 1147-1162.

[Brown84] BROWN, C.M., ELLIS, C.S., FELDMAN, J.A., LEBLANC, T.J., PETERSON, G.L. 1984. Research with the Butterfly multicomputer. *Computer Science Engineering research review,* University of Rochester, 1984-1985.

[Bryan81] BRYANT, R.M., FINKEL, R.A. 1981. A stable distributed scheduling algorithm. *Proceedings of the 2nd International Conference on Distributed Computing systems,* IEEE, New York, page 314-323.

[Carri86] CARRIERO, N., GELERNTER, D. 1986. The S/Net's Linda kernel. *ACM Transactions on Computing Systems,* Vol. 14, No. 2, page 110-129.

[Cheri83] CHERITON, D.R., ZWAENAPOEL, W. 1984. The distributed V kernel and its performance for diskless workstations. *Proceeding of the 9th Symposium on Operating System Principles,* ACM, New York, page 128-140.

[Cheri84] CHERITON, D.R. 1984. The V kernel: A software base for distributed systems. *IEEE Software,* April, page 19-42.

[Corne88] CORNELL THEORY CENTER 1988. *Facts about the Trollius Operating System.* Cornell University.

[Coulo88] COULOURIS, G.F., DOLLIMORE, J. 1988. *Distributed systems - Concepts and design.* Addison-Wesley Publishers Ltd.

[Curra87] CURRAN, L. 1987. Here comes high-powered UNIX for multiple CPUs. *Electronics,* October 29, page 77-79.

[Denni66] DENNIS, J.B., VAN HORN, E.C. 1966. Programming semantics for multiprogramming computations, *Communications of the ACM,* Vol. 9, No. 3, page 143-155.

[Denni70] DENNING, P.J. 1970. Virtual Memory. *ACM Computing Surveys,* Vol. 2, No. 3, page 153-189.

[Denni82] DENNING, D. 1982. *Cryptography and data security*. Addison-Wesley, Reading, Mass.

[Doran76] DORAN, R.W. 1976. Virtual Memory. *Computer,* October, page 27-37.

[Dijks65] DIJKSTRA, E.W. 1965. *Cooperating Sequential Processes*. Technological University, Eindhoven, Netherlands. (Reprinted in *Programming Languages,* Academic Press).

[Efe82] EFE, K. 1982. Heuristic models of task assignment scheduling in distributed systems. *Computer,* June, page 50-56.

[Farbe75] FARBER, D.J. 1975. A ring network. *Datamation,* Vol. 21, No. 2, page 44-46.

[Garne87] GARNETT N.H. 1983. HELIOS - An Operating System for the Transputer. *7th OCCAM User Group Int. Workshop on Parallel Programming of Transputer based machines,* September 14-16.

[Glaze83] GLAZER, S. 1983. Enhanced version of Bell Lab's UNIX servers fault-tolerant multiprocessor system. *Electronics,* November 3, page 145-149.

[Hoare78] HOARE, C.A.R. 1978. Communicating Sequential Processes. *Communications of the ACM,* Vol. 21, No. 8, page 666-677.

[Hoffm89] HOFFMAN, P.K. 1989. *A File Server for the DISTRIX Prototype — A Multi Transputer UNIX System*. Masters thesis, UCT.

[Inmos84a] INMOS LIMITED. 1984. *IMS T424 Transputer Reference Manual*. INMOS Ltd, 1000 Aztec West, Bristol, UK.

[Inmos84b] INMOS LIMITED. 1984. *OCCAM Programming Manual*. Prentice-Hall International.

[Inmos87a] INMOS LIMITED. 1987. *The transputer family 1987*. INMOS Ltd, 1000 Aztec West, Bristol, UK.

[Inmos87b] INMOS LIMITED. 1987. *The transputer instruction set - a compiler writer's guide*. INMOS Ltd, 1000 Aztec West, Bristol, UK.

[Inmos88] INMOS LIMITED. 1988. *3L C compiler*. INMOS Ltd, 1000 Aztec West, Bristol, UK.

[Janss86] JANSSENS, M.D., ANNOT, J.K., VAN DE GOOR, A.J. 1986. Adapting UNIX for a multiprocessor environment. *Communications of the ACM,* Vol. 29, No. 9, page 895-901.

[Jense78] JENSEN E.D. 1978. The honeywell experimental distributed processor: an overview. *Computer,* Vol. 11, No. 1, page 28-38.

[Kahn67] KAHN, D. 1967. *The Codebreakers*. MacMillan, New York.

[Karsh83] KARSHMER, A.I., DEPREE, D.J., PHELAN, J. 1983. The New Mexico State University Ring-Star system: A Distributed UNIX environment. *Software—Practice and experience,* Vol. 13, No. 12, page 1157-1168.

[Kerni78] KERNIGHAM, B.W., RITCHIE, D.M. 1978. *The C programming Language*. Prentice-Hall, New Jersey.

[Levy84] LEVY, H.M. 1984. *Capability-Based Computer Systems*. Digital Press, Maynard, Mass.

[Logic88a] LOGICAL SYSTEMS. 1988. *Transputer Toolset*. Logical Systems, P.O.Box 1702, Corvallis, USA.

[Logic88b] LOGICAL SYSTEMS. 1988. *TASM Transputer Assembler User Guide*. Logical Systems, P.O.Box 1702, Corvallis, USA.

[Lyckl78] LYCKLAMA, H., CHRISTENSEN, C. 1978. A minicomputer satellite processor system. *The Bell System Technical Journal,* Vol. 57, No. 6, page 2103-2113.

[MacGr84] MACGREGOR, D., MOTHERSOLE, D., MOYER, B. 1984. The Motorola MC68020. *IEEE Micro,* August, page 101-118.

[May84] MAY, D., TAYLOR, R. 1984. OCCAM. *Microprocessors and Microsystems,* Vol 8, No. 2.

[Mock88] MOCK, J. 1988. *Processes, Channels and Semaphores (version 2).* Logical Systems, P.O.Box 1702, Corvallis, USA.

[Mulle86] MULLENDER, S.J., TANENBAUM, A.S. 1986. The design of a capability-based distributed operating system. *Computer Journal,* Vol. 29, No. 4, page 287-300.

[Natio77] NATIONAL BUREAU OF STANDARDS (US) 1977. Data encryption standard. *Federal Information Processing Standards,* No. 46, Washington DC.

[Needh78] NEEDHAM, R.M., SHROEDER, M.D. 1978. Using encryption for authentication in large networks of computers. *Communications of the ACM,* Vol. 21, page 993-999.

[Nelso81] NELSON, B.J. 1981. *Remote procedure call.* Technical Report CSL-81-9, Xerox Palo Alto Research Center, California.

[Ouste80] OUSTERHOUT, J.K., SCELZA, D.A., SINDU, P.S. 1980. Medusa: An experiment in Distributed Operating System Structure. *Communications of the ACM,* Vol. 23, No. 2, page 92-104.

[Pasht82] PASHTAN, A. 1982. Object oriented operating systems: An emerging design methodology. *Proceedings of the ACM National Conference,* October, page 126-131.

[Malur88] MALARSKI, W. 1988. Mass Storage Controller MEGAFRAME Transputer Module with SCSI and Floppy Bus. *Parsytec Technical Documentation,* April, 1988.

[Phill85] PHILLIPS, D. 1985. The Z80000 Microprocessor. *IEEE Micro,* December, page 23-36.

[Popek85] POPEK, G.J., WALKER, B.J. 1985. *The LOCUS Distributed System Architecture.* The MIT Press.

[Pount84] POUNTAIN, D. 1984. Microprocessor Design. *Byte,* August, page 361.

[Ritch78] RITCHIE, D.M., THOMPSON, K. 1978. The UNIX time-sharing system. *The Bell System Technical Journal,* Vol. 57, No. 6, page 2103-2113.

[Ruhla87] RUHLAND, M. 1987. *MC68030: The second generation 32-bit microprocessor*. National Computer Conference, 1987.

[Russe87] RUSSEL, C.H., PAMELA, J.W. 1987. Variations of UNIX for parallel-processing computers. *Communications of the ACM,* Vol. 30, No. 12, page 1048-1055.

[Schmi83] SCHMITT, S. 1983. Virtual Memory for Microprocessors. *BYTE,* April, page 86-98.

[Shiel86] SHIELL, J. 1986. Virtual Memory, Virtual Machines. *BYTE,* Extra Edition, page 111-121.

[Smit88] SMIT, G. DE V., GRAY, G.T. 1988. Concurrent Distributed Operating Systems. *International Symposium on Parallel Processing, Technology and Applications,* October, Johannesburg.

[Smit89] SMIT, G. DE V., MCCULLAGH, P.J., HOFFMAN, P.K. 1989. *DISTRIX: A Multi-Processor UNIX Workbench*. Research report CS-89-04-00, Department of Computer Science, University of Cape Town.

[Smith79] SMITH, R. 1979. The contract net protocol: High level communication and control in a distributed problem solver. *Proceedings of the 1st International Conference on Distributed Computing systems,* IEEE, New York, page 185-192.

[Stank84] STANKOVIC, J.A., SIDHU, I.S. 1984. An adaptive bidding algorithm for processes, clusters and distributed ups. *Proceedings of the 4th International Conference on Distributed Computing systems,* IEEE, New York, page 49-59.

[Swan77] SWAN, R.J., FULLER, S.H., SIEWIOREK, D.P. 1977. CM*—a modular multi-microprocessor. *AFIPS Conference Proceedings, National Computer Conference,* Vol. 46, page 637-644.

[Tanen81] TANENBAUM, A.S. 1981. *Computer Networks*. Prentice-Hall, Englewood Cliffs, New Jersey.

[Tanen85] TANENBAUM, A.S., VAN RENESSE, R. 1985. Distributed Operating Systems. *Computing Surveys,* Vol. 17, No. 4, page 419-470.

[Tanen87] TANENBAUM, A.S. 1987. *Operating systems—Design and implementation.* Prentice-Hall, Inc.

[Tompl85] TOMPLAIT, C. 1985. Memory management boosts efficiency of powerful micros. *Computer Design,* July 1, page 105-109.

[Verhu89] VERHULST, E. 1989. TROS - A Real time kernel for a fault tolerant multiprocessor system. *Occam User Group - newsletter,* July, No. 11.

[Walke85] WALKER, P. 1985. The Transputer. *Byte,* Vol. 10, No. 5, May, page 219.

[Welch88] WELCH, P.H. 1988. Report from the UNIX SIG. *Occam User Group - newsletter,* January 1988, No. 8.

[Welch89] WELCH, P.H. 1989. Transputers, UNIX, and future support environments. *A workshop of the OUG UNIX SIG and OUG operating systems SIG,* 21 February, Canterbury.

[Wells84] WELLS, P. 1984. The 80286 Microprocessor. *BYTE,* November, page 231-241.

[Wells86] WELLS, P. 1986. The 80386 Architecture. *BYTE,* Extra Edition, page 89-106.

[Wilke79] WILKES, M.V., WHEELER, D.J. 1979. The Cambridge digital communications ring. *Proceedings of the Local Area Network Symposium,* May, page 47-61.