University of Cape Town
Department of Computer Science

A File Server for the DistriX Prototype — a Multi Transputer UNIX System

by P Kuyper Hoffman

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.

Abstract

The DISTRIX operating system is a multiprocessor distributed operating system based on UNIX. It consists of a number of satellite processors connected to central servers. The system is derived from the MINIX operating system, compatible with UNIX Version 7. A remote procedure call interface is used in conjunction with a system wide, end-to-end communication protocol that connects satellite processors to the central servers. A cached file server provides access to all files and devices at the UNIX system call level. The design of the file server is discussed in depth and the performance evaluated.

Additional information is given about the software and hardware used during the development of the project. The MINIX operating system has proved to be a good choice as the software base, but certain features have proved to be poorer. The Inmos transputer emerges as a processor with many useful features that eased the implementation.

CR Categories: C.1.2 [**Multiprocessors**]; C.2.1 [**Network Architecture and Design**]; C.2.4 [**Distributed Systems**]; D.4.3 [**File Systems Management**]; D.4.4 [**Communications Management**]; D.4.7 [**Organization and Design**]; D.4.8 [**Performance**].

Key words: distributed systems, file servers, multiprocessors systems, operating systems, remote procedure call, satellite systems, transputer, UNIX.

Acknowledgements

Before thanking those directly related to the project, I must extend my heartfelt thanks to my mother and sister for putting up with and feeding me during my time beneath the ivy.

I am most grateful for the assistance provided by the following companies and individuals who contributed to the DISTRIX project. Firstly, Mr Hennie Le Roux of Westervoort Research and Development who proposed the original project and established the funding in conjunction with ICL(SA) who provided the equipment. Professor Pieter Kritzinger acted as a temporary supervisor for 1988. The Council for Scientific and Industrial Research provided funding for the duration of my research. The Department of Computer Science provided me with a Research Assistantship for 1989 and office space for the duration of the project.

My colleague, Paul McCullagh, with whom I have been working for some years now in the field of Distributed Systems, has always been ready to knock ideas around and many of the concepts introduced in this document were the result of joint discussions. Gavin Gray assisted in the design and hunting down some particularly elusive bugs.

Pieter Bakkes and Elize van der Walt at the Institute for Electronics at the University of Stellenbosch gave much of their time to the hardware design stages and advised on technical issues. Ralph Pina and Gunther Dörgelon at SED at the University of Stellenbosch patiently endured many technical queries and unscheduled 'adjustments' to our hardware.

Nick Curwell and the staff at the Teaching Methods Unit gave of their time to produce the photographs of the transputers.

Lastly, but primarily, I owe an extreme debt of gratitude to my supervisor, Dr Riël Smit. His time, comments and contributions to this project are above value — thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The DISTRIX project was undertaken by the Laboratory for Advanced Computing in the Department of Computer Science at the University of Cape Town during 1988 and 1989. It involves the design and implementation of a multiprocessor UNIX[1] system on a network of transputers. Its primary aims are:

- to investigate the feasibility of UNIX on transputers,
- to gain experience in the development of distributed operating systems, and
- to provide a workbench for future development.

In achieving these aims, our group ported the MINIX operating system to the Inmos transputer. MINIX is a modular operating system, based on a message passing model. It is compatible with UNIX Version 7, featuring most of the system calls present in Version 7.

Initially another goal formed part of the system requirements, namely that DISTRIX should comply with the X/OPEN specification for portability, based on UNIX System V. Consequently, the discussions that follow will refer to the X/OPEN specification.

The reason for choosing a multiprocessor architecture lies in the upper processing limit that one eventually reaches in a uniprocessor architecture. A solution is to lighten the load on the processor. This may be achieved by harnessing the power of more processors, each sharing the load. The arrangement of these processors is a study in its own, but for the purposes of this project, the simple satellite model has been chosen. Its choice is

[1] UNIX is a registered trademark of AT&T Bell Laboratories.

motivated not only by its simplicity, but also by the characteristics of the software and hardware at our disposal.

In our implementation of the satellite model, a number of user processes (clients) interact with a group of server processes. The servers consist of a central kernel and a file server. Between them, most of the system calls are serviced. The remaining calls are serviced locally by the user processor. The clients and servers interact over narrow channels by means of a Remote Procedure Call mechanism.

I was responsible for the port of the file server and creating device drivers for the new devices. My duties also included the exchange software and the specification of a system-wide communications protocol.

1.1 Advantages of distributed systems

The purpose of distributing the system over a number of processors is to increase the overall system throughput by offloading the work from a single processor onto many. To maintain the simplified view of a single machine, all file access is controlled by a central file server.

Whenever a process on a satellite processor performs a system call, the calling process is suspended until the server has completed the request. During this time, any other processes sharing the satellite with the calling process are able to continue processing. Multiple satellites may continue executing in parallel, only contending with other processes when accessing the central exchange or a central server processor.

The mechanism that provides this ability to communicate with remote servers has a cost attached. Individual processes execute slower than in a similar uniprocessor system, but when the system is placed under load, the throughput, as a whole, is increased.

1.2 The thesis

The thesis of this dissertation is that a modular operating system can be distributed over a network of communicating processors. The use of the Inmos transputer and the MINIX operating system in this format has not previously been documented. It is not necessarily particularly efficient, but raises some interesting problems. The solutions to these problems prove to be effective. The main topic of this study involves the relocation of a UNIX file

server to a remote processor.

In order to provide the service, the file server must be connected to the other processors in the system. This interconnection is another topic investigated. Using a remote procedure call interface, the user processes are able to communicate with the server without being aware that it has been removed to a remote processor. Their requests are routed by a central packet switching exchange.

The file server provides its service at the system call level and manipulates both ordinary files and directories, as well as devices. The devices all present a uniform interface by using device drivers. The device drivers in this implementation are particularly simple as they are attached to devices with a high interface level.

The efficiencies of both the interconnections and overall system throughput are measured and presented, together with suggestions for future improvement.

1.3 Thesis structure

Before embarking on the discussion of the file server, the necessary groundwork must be laid. To this end, the first three chapters each describe some aspect of the system.

Chapter 2 describes the concepts of multiprocessors and the operating systems that manage them. The various models are presented and contrasted. The satellite model is chosen as the model for DISTRIX. Particular attention is paid to the UNIX operating system in a multiprocessor environment.

Chapter 3 introduces the DISTRIX system in detail, motivating the choices made during the design of the system. The components of the system (and their functions) are outlined. The main purpose of this chapter is to place the various components in perspective.

Chapter 4 discusses aspects of the communications mechanisms used in DISTRIX. The layered protocol and the components that motivated its structure (the remote procedure call interface and central exchange) are described and the performance of the exchange is presented.

The emphasis of the thesis is on Chapter 5, in which the DISTRIX file server is presented. The design shows that the concept of *discrete* server processes on a single processor can be extended to cover discrete servers on separate processors. The Remote Procedure Call interface is described in the context of the file server, with particular attention

to the buffer concept applied to server stubs. The device drivers accessed by the file server are described and their high level interface outlined. Finally, the performance of the file server shows that advantages are not realized until the overall system is placed under load.

Several Appendices have been included. They serve to give the reader a background to the facilities available while developing DISTRIX.

The first Appendix is devoted to the MINIX operating system. MINIX was used as the software base for the DISTRIX project and the features of MINIX, together with their relative advantages and disadvantages are presented. A contrast between UNIX and MINIX is presented to gauge their similarity.

Appendix B gives a general introduction to the Inmos transputer, the processor used to implement DISTRIX. It is written from a software perspective, but some engineering facts are given to allow the more technically minded reader to place the transputer in perspective against similar processors.

Appendix C describes specific configuration issues regarding the DISTRIX project. In particular, the operations of booting and determining memory size are described.

The development history of DISTRIX is outlined in Appendix D. It explains some of the early configurations during the development stages. The early experiments formed the basis of the present system. Of particular interest are the first MINIX applications making use of the transputer. A device driver to assist in this process is described. The remainder of the Appendix describes some of the problems encountered in a concurrent environment without debuggers.

Appendix E contains the C code for the exchange processor.

In general, each chapter or section will give a brief overview of what follows. These points will then be expanded upon individually. All the points will be brought together at the end of the chapter with some concluding remarks.

Chapter 2

Multiprocessors and multiprocessor operating systems

The DISTRIX system has a multiprocessor architecture based on the satellite model. In this chapter this model is discussed and placed in perspective with other models.

The distributed nature of a multiprocessor system should not be visible to the user of the system. This requirement of transparency is discussed, introducing some of the mechanisms used to achieve it. Particular attention is given to the aspects pertinent to the UNIX operating system in a multiprocessor environment.

2.1 Multiple processors

In a single processor machine, the illusion of executing multiple processes simultaneously is achieved by dividing the processor's time amongst each of the processes. This technique is called multiprogramming or multitasking. Thus, pseudo-concurrency is achieved by means of fast process switching.

In order to achieve *true* concurrency, more than one processor must be harnessed. The class of multiple processor machines contains two major variations: parallel processors and multiprocessors.

Parallel processing is at a fine granularity, with multiple processors applied to one individual task, each processor operating on a separate portion of the problem with the intention of increasing the execution speed of subparts of the same logical task [Serl85]. Serlin classifies this level of parallelism as *medium grain*.

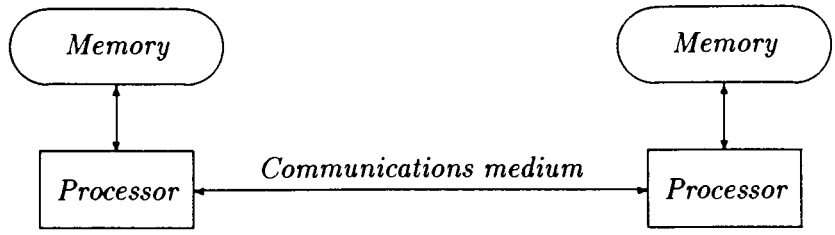

Figure 2.1: *Loosely coupled multiprocessors*

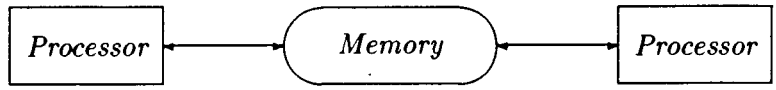

Figure 2.2: *Tightly coupled multiprocessors (Shared memory)*

Multiprocessing can be defined as the use of two or more processors applied to separate, unrelated tasks, each processor operating on an individual task with the intention of increasing overall system throughput.

Multiprocessor architectures are becoming more attractive as the price of microprocessors drops. The advantages of multiprocessing fall within two (possibly overlapping) categories:

- increased performance, or
- increased system reliability through redundancy.

The DISTRIX system takes advantage of the former, and no particular attention has been given to improving reliability at this stage. Multiprocessing itself has two broad categories [Bach84]:

- loosely coupled systems, where two or more processors, each having their own local memory, communicate by means of some networking facility (Figure 2.1), and
- tightly coupled systems, where two or more processors share a common memory by multiport (Figure 2.2), bus (Figure 2.3), or crossbar (Figure 2.4).

As the DISTRIX architecture (described Chapter 3) is based on a loosely coupled approach, it is the former that will be dealt with in depth, although shared memory models are also discussed.

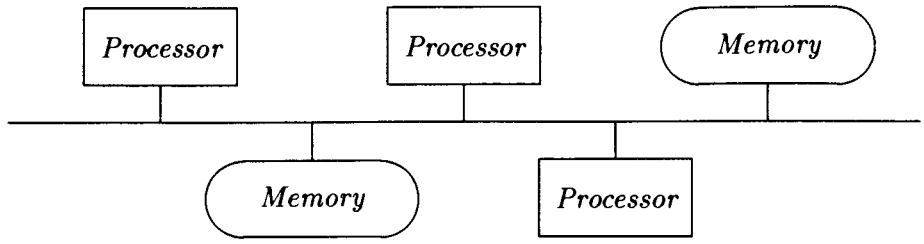

Figure 2.3: *Tightly coupled multiprocessors (Common bus)*

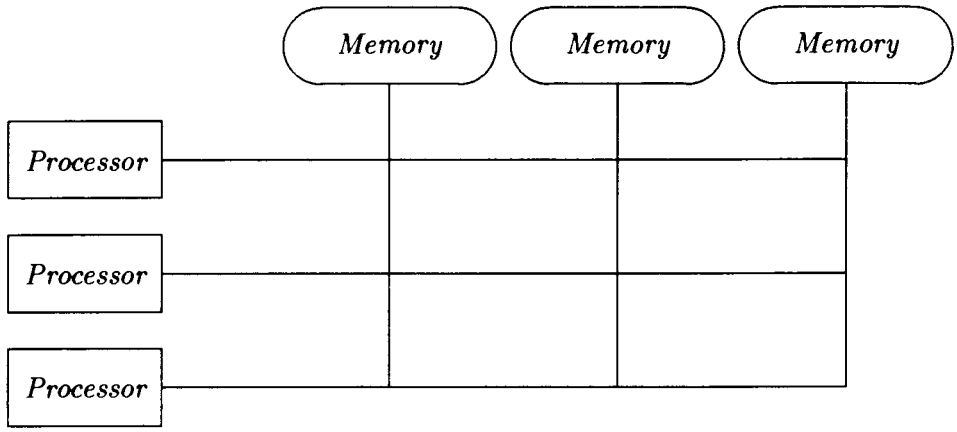

Figure 2.4: *Tightly coupled multiprocessors (Crossbar switch)*

2.2 The basic models

Two models for multiprocessor systems will be discussed:

- shared memory, including
  - multiport,
  - common bus, and
  - crossbar switched memories;
- local (private) memory, including
  - satellite (client-server),
  - "Newcastle" distributed systems, and
  - transparent distributed systems.

The shared memory model was introduced earlier in defining the requirements for a tightly coupled system and is illustrated in Figures 2.2–2.4. Early definitions of multiprocessor systems [Comt74, pp19, 26, 135] [Ensl77] made the requirement that memory be shared

by all processors a prerequisite in the definition of multiprocessor systems. This definition survived as late as the early eighties [Saty80, p2].

Subsequently, loosely coupled systems with local memory, such as the satellite model have provided an alternative architecture. They have only a bus or communications channel common to all processors. Such processors rely on local memory and use the bus or channel for the transfer of messages only.

2.2.1 Shared memory

Shared memory multiprocessor systems allow the processors to communicate with one another during the processing of their respective tasks. This allows them to either act on independent tasks, or to act cooperatively on subtasks of the same main task. Stenström suggests that this makes shared memory multiprocessors very versatile [Sten88].

Stenström goes on to point out the problems of shared memory. These are primarily related to degradation as the result of multiple, simultaneous memory accesses. As the memory can handle only one request at a time, subsequent requests must await completion of preceding ones.

The contention exhibited by shared memory systems can be alleviated by use of private, local memories, caching data structures to reduce memory traffic. In the light of cache consistency problems, research is being conducted to ensure efficient solutions [Cher86, Cher89].

Enslow describes some ways of effecting the actual interconnection of processors and memories [Comt74, Ensl77]. Figure 2.2 displays a shared memory multiprocessor making use of *multiported* memory. This has the most expensive memory components, but can display high throughput. As the porting circuitry forms part of the memory itself, the number of ports may be low.

One of the other ways of connecting the processors to the memories is to make use of a *common bus* (Figure 2.3). In this case, contention for the bus arises, although a bus protocol can allow several memory requests to be pipelined in the bus. This is the simplest and least costly technique. It is very flexible, but dependent on the reliability and bandwidth of the bus.

The connection of processors and memory using the *crossbar* (Figure 2.4) requires the most complex interconnection system, but displays the highest efficiency.

The first shared memory multiprocessor system recorded by Enslow [Comt74], is a Burroughs processor with 4 processors sharing 16 memory modules through a crossbar switch. The system is dated November 1962. More recently described systems [Sten88] such as C.mmp (crossbar) and CM* (local memory and shared memory with multiple busses) at Carnegie-Mellon University, and Cedar (multistage network — a variation on the crossbar [Sten88]) at the University of Illinois, Urbana-Champaign are also examples of shared memory multiprocessor systems.

2.2.2 Local memory

Whilst shared memory multiprocessor systems rely on their common memory to communicate, loosely coupled systems generally make use of a network or other communications mechanism to perform this task. The processors are arranged as discrete processing units, each with its own memory. This memory cannot be accessed by the other processors, except by arrangement with the processor itself.

The ways in which nodes may be connected are numerous: cubes, hypercubes, rings, trees, etc. The simplest and most useful way is to provide one connection between each processor in the system. However, for a collection of n processors, the number of connections is $\frac{n^2-n}{2}$, which grows rapidly as n is increased. Other ways to fully connect a network fully do exist, and a solution using a central switching exchange (similar to the crossbar introduced earlier) using only n links, is described in section 4.2.

The connections described thus far have been physical connections, dealing mainly with the problems of how to communicate. Within the general category of communicating, loosely coupled systems with local memory, Bach [Bach86] describes three systems.

The so-called "Newcastle" distributed systems feature distributed file systems. A file server is present on each processor, but the files are made globally available by introducing a super-tree. Each subtree rooted a child of this main root represents the file system of a different machine and uses the name of the machine. Thus, the distribution is explicit. The user is expected to know the location of the file when specifying the remote file name.

A refinement of the Newcastle idea is a *transparent* distributed system, in which the location of a file is hidden from the user. A global file system is presented as a single hierarchical structure, with files from multiple machines accessible to all users. An example of a transparent system such as this is the LOCUS system [Pope85].

One other system is described by Bach, namely satellite systems. As this is the approach taken in DISTRIX, it will be examined in more detail.

Satellite systems

Satellite systems have as their main aim, the task of increasing overall system throughput by distributing the workload to a number of processors, each able to execute in parallel with one another. The definition given by Bach [Bach84], relies on the presence of a central server. This server governs the satellites in most affairs.

Generally, the satellites only interact with the central server processes when assistance is required. Their autonomy is restricted to the way they handle 'local affairs' such as scheduling and memory management. The satellite model is also called the client-server model. Each process executing on a satellite is regarded as a client. The system provides certain services that are made available to the client (and other servers). The details of the mechanism allowing a client to request work from a server are hidden by the transparent nature of the system (described in 2.4).

Provided the communications can be kept low, the satellite model can allow for a marked performance increase. Processes executing on different satellites do not interfere with each other and can be allowed to use all the resources of their processor until such time as they require the use of a server.

2.3 Operating systems

Janssens *et al.* [Jans86] describe two approaches in developing multiprocessor operating systems:

- design a completely new operating system in either an existing or a completely new language, or
- adapt an existing operating system for the new hardware configuration.

The second option is more attractive, as the bulk of the original system can often be retained. For specialized systems that do not require compatibility with existing systems, the approach of designing a wholly new system can often produce a more efficient system, tailored to specific needs.

2.4 Transparency

One of the requirements of an operating system for a multiprocessor architecture is that the user must be unaware of the change from a uniprocessor. This is not only to ensure software compatibility with old software, but the problems that may be encountered due to truly concurrent events are best left to the systems programmer and a novice should not be burdened with such complexities.

A succinct way of describing the transparent nature of a distributed operating system, and also a multiprocessor system, is to regard the system as a *'virtual uniprocessor'* [Tane85]. This implies that regardless of the number of processors making up the system, the user's view is one of a single processor.

It is generally the task of the software to ensure this transparent view of the system. Any low level communications between the component parts of the system should be hidden by the compiler or libraries. This is explained more fully in sections 4.1.3 and 5.2.1 which explain the Remote Procedure Call (RPC) mechanism.

When seen in the light of a *Distributed File System*, the term transparency pertains more to the user's view of the files on the system. In such a file system, many file servers work cooperatively to create the illusion of a single file system [Stur80]. Users generally still execute processes on their local processor, but have transparent access to files located on physically different machines. The user does not need to know which server is responsible for the file. Such location transparency is displayed by LOCUS [Pope85]. LOCUS also supports remote execution of processes. This is particularly useful in a heterogeneous system, where a process may require to be run on a processor different to that to which the user is attached.

2.5 Requirements of UNIX on a new processor

Developed at Bell Laboratories in the early 1970's, UNIX has developed as a popular operating system for minicomputers. It encourages portability by providing users with a uniform view of the I/O system, identical system calls, subroutines and utility programs. The so-called 'portable' release [Joy83], Version 7 became available in 1978. The subsequent releases (System III and System V respectively) are 'upwardly-compatible' with earlier releases, allowing source code to be retained.

The UNIX operating system is written largely in C, with small portions of low-level code written in the machine language of the host machine [Bode84]. All the same, the following words by Johnson and Ritchie [John78, p2045] perhaps sum the task of porting up better than most:

> *'The definition of C suggests that some machines are more suitable for C implementations than others; likewise, the design of the* UNIX *kernel fits in well with some machine architectures and poorly with others. Once again, the requirements are not absolute, but a serious enough mismatch may make an implementation unattractive.'*

The remainder of this chapter discusses a few of the factors that must be considered in porting UNIX. Both uniprocessor and multiprocessor aspects are presented.

2.5.1 Uniprocessor

From an architectural point of view, the following properties in a processor are desirable when porting UNIX:

- memory segments that may be divided into 3 logical sections [John78],
- virtual memory management and therefore,
- restartable instructions [Tane87, p225],
- a minimum of two processor modes (kernel and user) [Morr85], and
- a clock that generates interrupts at 50-60Hz [John78] and [Tane87, p88].

These requirements are not essential, and (as will be seen in the case of the transputer processor being used in DISTRIX) a processor meeting none of these requirements can still host UNIX.

The requirement for memory segments, together with dual processor modes supply the major portion of kernel security in the UNIX system. Earlier systems such as MULTICS [Orga72] featured extensive protection, allowing each memory segment to be tagged with a different level of protection. This hierarchical approach is not necessary in UNIX, where a simple two-level system is sufficient [Morr85]. The reason for using the memory

segments and privileged (kernel) modes is primarily to allow user processes to execute in an isolated environment (errors should affect neither the kernel nor other user processes).

Less obvious, but equally important, is the ability to allow the user process to perform system calls. System calls allow the user processes to interact with the kernel. It is the operation of the dual processor mode that permits the rapid change of processor state between user mode and kernel mode. The process performing the system call informs the kernel of its intention to request work by means of a TRAP instruction, forcing the processor to switch to kernel mode. Parameters are placed in registers and the call carried out. When completed, the processor switches back to user mode and supplies the reply values in registers.

Interrupts other than those generated by the clock may be caused by terminals, disks, etc. Such interrupts generally preempt a user process and force the kernel to provide immediate service. Should the kernel already be executing, the interrupts are usually overridden by raising the processor priority [Bach84]. This priority must not be confused with the kernel and user modes mentioned earlier. The priority mentioned here is usually manipulated by an interrupt controller. This mechanism prevents corruption of system tables.

2.5.2 Multiprocessor

The mechanism preventing corruption of system tables while executing in kernel mode, described earlier, does not necessarily apply in the case of a multiprocessor implementation of UNIX. Should the kernel be distributed, different portions of the kernel may be interrupted without the knowledge of the other portions. Therefore, updates of system tables would have to be controlled by means of semaphores or some other mechanism to ensure single, system-wide updates. Such a system is implemented in the Sprite Network Operating System [Oust88]. Sprite has a multi-threaded kernel, making use of monitors to place individual locks on sensitive data structures.

Another method, not requiring semaphores, only allows one processor to update the kernel at any time. This is possibly simpler to implement, but as most UNIX processes spend around 40%-50% of their time performing system calls [Bach84], usually resulting in updates to the kernel tables, the performance is likely to suffer. Despite the potentially poorer performance, this approach has been taken in the DISTRIX file server which is

introduced in Chapter 3 and described fully in Chapter 5. The decision is motivated by the simplicity of such an approach. [1]

The third solution [Jans86], is to rewrite UNIX. The advantage of this approach is that the rewritten code could conceivably be written in a more modular fashion. Although not specifically done with a multiprocessor port in mind, this is one of the main features of MINIX as described in Appendix A.

2.6 Conclusions

Multiprocessor architectures are an effective way to increase overall throughput of a computer system. The operating system for a multiprocessor machine is considerably more complex than its uniprocessor counterpart, and may have features unique to the particular implementation. The operating system must provide a transparent view, giving the impression of a virtual uniprocessor. Remote Procedure Calls are a valuable tool in providing this view.

Of the two major models, namely shared memory and local memory models, the shared memory approach has been the more popular in the past. However its higher cost and complexity (particularly of hardware) weighs heavily against the simpler local memory model. The satellite model falls in the category of local memory models. Nodes in a satellite system communicate with one another by means of narrow channels. To provide full interconnection without physically connecting every node, a crossbar or its software equivalent, a switching exchange, may be used.

Written in the C language, UNIX displays high portability. Nonetheless, for the purposes of a port of UNIX, the features of the target processor play a predominant rôle in the actual porting effort. The architectural requirements for the processor are more desirable than they are essential and of the five requirements listed, a processor displaying none of the features may still target a UNIX implementation. Each feature generally facilitates an aspect of porting the operating system.

Multiprocessor implementations of UNIX are subject to the dangers of simultaneous updates of system tables. To overcome this, three solutions exist:

[1] In addition, due to the high processing power of the transputer (10 MIPS was still regarded as good in early 1988), we were prepared to sacrifice some processing time to simplify implementation. This was not necessarily a good decision as the cost was higher than anticipated.

- use semaphores to restrict updates of critical structures,
- limit kernel access to one processor in the network at one time, or
- rewrite the code.

Each of these solutions requires at best a large amount of examination of code, and at worst an entire rewrite. The option of limiting kernel access to one processor at a time appears to be the simplest to implement, at the possible cost of performance.

Chapter 3

Design of DistriX

This chapter provides a conceptual design of the DISTRIX system. In Appendix C, the actual hardware configuration is presented, including descriptions of specific hardware, such as transputer motherboards, and how to interconnect processors.

The structure of DISTRIX is an extension of the modular structure of MINIX (see Appendix A). A selection of the modules of MINIX are separated and placed on dedicated processors. The kernel and file server are each allocated a processor. A pool of processors is dedicated to the users. The processors are connected by means of a switching exchange. This is largely based on the satellite model outlined in section 2.2.2, but allows for *multiple* servers. The goal of increasing throughput remains unchanged.

The separation onto discrete processors is motivated not only by increasing processing power, but also by the poor memory model of the transputer. No memory protection exists and a user process is not restricted. All of memory is accessible by any process.

3.1 Transputer memory model

This section discusses the memory model of the transputer. The lack of protection, segmentation and virtual memory are investigated briefly.

The Inmos transputer (models include the T414, recently updated to T425 integer– and T800 floating point processor) is a 32-bit processor. It supports a linear address space of 4 Gigabytes. More commonly, however, the processor is packaged with more modest memory sizes ranging from 2 KB or 4 KB (using the built in static RAM) up to 8 MB of external dynamic RAM.

Unlike processors designed with memory segments in mind (Intel 8086 [Rect80] and Intel 80286 [Inte85]), the transputer presents the memory linearly. As a result, all memory addresses are accessible by all processes. This poses a particularly serious problem for implementations of UNIX on the transputer. To reiterate some of the points made in section 2.5, UNIX and indeed MINIX are *multiprogramming* operating systems, allowing many user processes to share the same processor. This results in a potential situation in which one user may overwrite the code or memory space of another user or even the kernel. Consequently, most UNIX implementations rely on the processor's ability to switch between kernel and user modes. Furthermore, to assist the processor, memory is divided into fixed size pages or variable sized segments. A user's instructions are then limited to operate within the particular page or segment. Segmentation not only protects users from one another, but also from themselves, by placing the program's code and data in separate segments. The UNIX system allows for three logical segments of a user's address space: code, data and stack [John78]. Paging is most useful for virtual memory systems.

The lack of paging and segmentation in the transputer has led to a different design approach. Each user is allocated a processor. The kernel processes are relocated to dedicated processors, each with their own memory. This provides both users and the kernel with immunity from attacks (malicious or accidental) by other users. The problem of forged messages is not yet addressed in DISTRIX.

A further specific problem relates to the *fork* system call, central to the design of UNIX. The purpose of this call is to create a new process in the process table and start execution of the code. The code and global data for the new process (the *child*) are copies of the calling process (the *parent*). Any changes that the child makes to the global data are limited to its own copy of the data. To support this, a base–offset system is used. All processes access data *relative* to a base pointer, usually the base of their data segment. As the transputer has no base pointer (all memory accesses are *absolute*), changes were applied to the compiler. The new code adds a base address to every global memory access, function call, and function return address. In this way the *fork* call is made possible. The code size overheads introduced by adding base pointers to each of the aforementioned memory accesses are reasonably high (40%–50%).

Often a process requires very large amounts of memory, particularly in scientific or engineering applications where large arrays are manipulated. Minicomputers and main-

frames deal with such processes by providing a virtual memory space that satisfies their requirements. This memory is commonly divided into a number of pages. A moderately small amount of Random Access Memory (RAM) is provided. The processor checks each address request made by the process. Whenever a request is made that falls outside the range of the actual memory, a page fault is generated and the process is suspended. The required area of memory is fetched from secondary store, usually a high speed disk or drum, and placed into actual memory, replacing some older page. Address translation is aided by a table that maps the virtual address to a real address.

All this work would place an extra burden on the processor. Consequently, a dedicated processor, called the Memory Management Unit (MMU), is used to perform the page fault detection, replacement and mapping. In many current processors, the MMU is being included on the main processor.

One implementation of a virtual memory system for the transputer is in use and is described by Bakkes *et al.* [Bakk88].

3.2 Structure of DistriX

The structure of DISTRIX is an extension of the modular structure of MINIX. In MINIX the major logical components of the operating system are each contained in discrete processes. In DISTRIX this has been extended to place a selection of the modules on dedicated processors. All the processors are connected by means of one or more switching exchanges. Remote Procedure Calls are used to communicate requests from the clients to the operating system modules. The structure of the DISTRIX prototype is illustrated in Figure 3.1.

Each of the major components are described in the remainder of this section.

The system code for the central kernel and user processor together form the original MINIX kernel and memory manager. They have been re-written for the transputer. The new code makes use of the concurrency features of the transputer and the modifications are described by McCullagh [McCu89].

3.2.1 Central kernel

The central kernel (CK) is a server. It accepts calls from the user processes executing in the processor pool. The calls for the CK are carried out by stub processes. A stub process

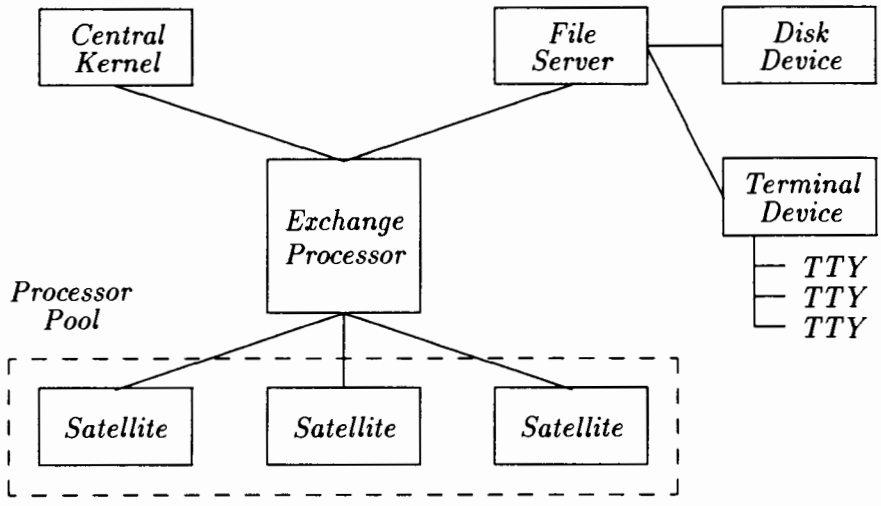

Figure 3.1: DISTRIX *prototype system architecture*

exists for each user process.

It also interacts with the file server, requesting the file server to transfer executable files. Signals generated by the user at the keyboard are routed by the file server to the CK.

3.2.2 Mini-kernel

The mini-kernel (MK) oversees the user processes. The MK filters and classifies each system call. Any calls that can be handled locally are carried out by the MK. The remaining calls are either sent to the central kernel or the file server.

A *system task* forms part of the mini-kernel. This process accepts requests from other server processors to access local memory. There are two such requests, namely to read from local memory and to write into local memory. In this way, large blocks of data can be copied in and out of a transputer's memory.

This model is different to that of the V Kernel [Cher88], in which each kernel module provides all the services (memory management, process scheduling, clocks, file access and devices). The poor memory model of the transputer will not permit such a scheme as no protection exists for the sensitive kernel modules.

Scheduling of processes is performed by the transputer scheduler. The tables maintained by the MK are used for memory allocation and network addresses of stub processes.

3.2.3 File server

The file server (FS) is a remote implementation of the MINIX file server. The interface to the FS is at the system call level. Requests are accepted from user processes and the kernel. To assist in buffering and translation of Remote Procedure Calls, a stub process exists for each user process.

The FS manages the visible resources of the system: disks, terminals, etc. In order to hide the low level details of the devices, device specific drivers are created for each device. These device drivers are discrete processes, each having the same form as the servers. The drivers may reside on dedicated processors, but at present, the device drivers all reside on the FS processor.

The FS is described more fully in Chapter 5.

3.2.4 Processor pool

All user processes execute on one of the processors in the processor pool. Ideally only one user is allowed per processor (see Section 3.1), although multiple users may share a processor. Additionally, one mini-kernel process executes on each user processor. Processes may migrate from one processor in the pool to another using the *move* system call [McCu89]. This is not a regular UNIX or MINIX system call but was added to DISTRIX. The *move* call is usually issued immediately after a *fork* call and is typically be followed by an *exec* call. The only parameter to *move* is *node*, which specifies the network address of the processor to which the process should migrate. The *move* call fails if there is insufficient memory on the target processor.

Both the CK and the FS must be informed of the *move* call in order to maintain their internal addressing tables.

3.2.5 Exchange

The exchange processor connects all nodes in the network. All messages distributed by the exchange are preceded by an address field. The exchange is described in detail in section 4.2.

3.3 Conclusions

DISTRIX is modelled on the modular MINIX structure. The concept of separate processes for logical modules of the operating system is extended to encompass separate processors. This separation not only shifts the load from one processor to many, but also provides the protection lacking on the transputer.

The interface used for communicating requests to servers is at the system call level. Remote Procedure Call mechanisms hide the distribution from the user processes. Process migration is achieved at run-time.

A central packet switching exchange allows the interconnection of all processors.

By making use of a central kernel, to which requests for work must be made, and a file server (accepting most other system calls) we have made a requirement that all but a few system calls are processed by a remote processor. This decision is well motivated by the transputer's poor memory protection, but leads to a potential performance penalty. Bach and Buroff, in their description of Multiprocessor UNIX systems [Bach84] point out that between 40% and 50% of process executions are spent performing system calls. They recommend that each processor be allowed to execute kernel code in order to allow maximum efficiency. This will be a considerating factor in future implementations, by which time better memory protection may be available. For the present this simpler model will be used, despite the potential penalties.

Chapter 4

The communications mechanism

In a distributed environment, the modules on different processors must be able to communicate simply and efficiently.

A four layer, end-to-end protocol is defined. This protocol satisfies the needs of the modular components described in section 3.2. The OSI reference model serves as a basis, but its seven layers are too 'top-heavy' for the specialized needs of DISTRIX. Remote Procedure Calls (RPC's) are the primary mechanism by which user processes communicate with servers. A switching exchange ensures that all nodes in the system are connected. The exchange emerges as a very convenient way to address the problem of system wide communications, but can lead to poor performance if not tuned correctly.

4.1 Protocol

4.1.1 Design goals

The DISTRIX communications protocol is designed to have the following characteristics:

- clear definition,
- support for Remote Procedure Calls,
- accommodate the exchange,
- sufficient flexibility for expansion, and
- efficiency within the constraints of the other characteristics.

Support for RPC's is achieved by ensuring that end-to-end communication is realized through careful layering.

4.1.2 Ties to the OSI reference model

The OSI reference model [Zimm80] makes use of seven layers to describe the protocol adopted by the International Organization for Standardization (ISO).

Layered protocols are *modular*, each layer is described by the *service* it offers, not how the service is provided. Thus the interface between layers is the only way of transferring information. Provided this interface is not altered, the entire lower layer (lower layers can be thought of as *servers*) may be changed without affecting the layer above it (the *client*). The DISTRIX protocol does not require all the services offered by the OSI reference model. As each layer adds not only service quality, but also overhead, the goal of efficiency forces us to consider a subset of the full model for the purposes of DISTRIX. The following is a list of the four layers implemented in the prototype:

0 - Physical Layer is responsible for bit-level transmission of data. On the transputer, the *link* provides this service (see Section D.2.1.) With an error rate of less than 1 per 10^{25} [Shep87], this is considered a reliable medium for the purposes of the DISTRIX prototype. The link is described further in section 4.2.1.

1 - Firmware Layer corresponds to the OSI data link layer and on the transputer is provided for at a machine instruction level with the IN and OUT instructions. These instructions perform blocking *receive*s and *send*s respectively.

2 - Network Layer places addressing information in a block header. The addressing information is discarded by the Network Layer at the destination, but is retained through the exchange.

3 - Buffer Layer is discussed at length in section 5.2.1. This is the highest layer in which specialized (unique to DISTRIX) messages are used. Above this layer, all message passing reverts to standard MINIX messages as described by Tanenbaum [Tane87, section 2.5.3, p89]. The Buffer Layer is not easily mapped onto any *one* of the OSI layers.

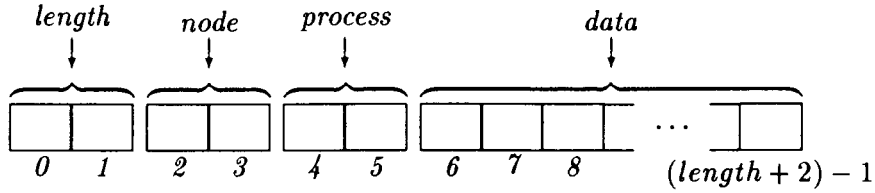

Figure 4.1: DISTRIX *message format — Byte level*

These layers provide sufficient service to support both the RPC mechanism and allow for the exchange. The resulting message structure is shown in Figure 4.1. The maximum length is determined by finding a good balance amongst the requirements of the RPC mechanism, the cost of local buffering and the header overhead. The choice is motivated in section 4.2.3.

The above description clearly identifies the function of each layer. As described, the protocol is still efficient, adding only 6 bytes to a message. With large messages, (¿ 1000 bytes), this accounts for less than 1%.

4.1.3 An introduction to remote procedure calls

The concept of Remote Procedure Call (RPC) allows for transparent communication between modules in a distributed system. Nelson defines RPC as:

> '*...the synchronous language level transfer of control between programs in disjoint address spaces whose primary communication is a narrow channel.*' [Nels81, Section 2.1.1]).

Let us examine this definition in some detail, with particular attention to the DISTRIX environment. The fact that the transputer allows for *synchronous* (or blocking) instructions, maps well onto the first portion of Nelson's definition. In the DISTRIX model described in 3.2 it is clear that processes on different nodes are in *disjoint address spaces.* Finally, the *narrow channel* characteristic is certainly applicable to the transputer link. Indeed, the link is not merely the primary communication medium, but in fact the only.

Thus, RPC provides a mechanism for remote communication. This mechanism appears to the user to be identical to the local procedure call mechanism. In the 'pure' model described by Nelson, RPC's are not only used to perform transfer of data to a remote process, but also the transfer of control. This concept of transfer of control is no exploited

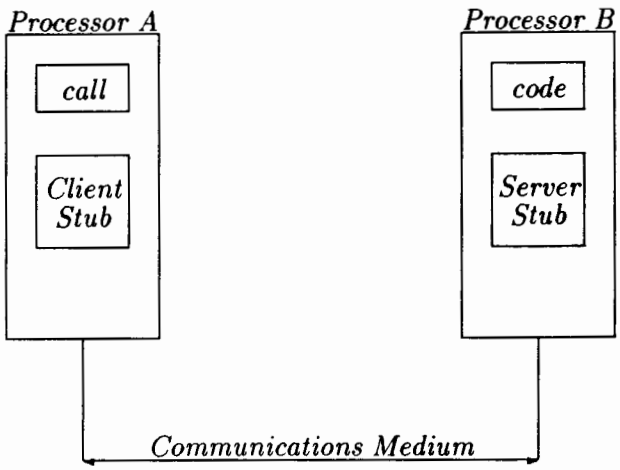

Figure 4.2: *Simplified view of a remote procedure call environment*

in DISTRIX.

Figure 4.2 shows a simple environment in which RPC's can be used to effect a call (on Processor A) of code (on Processor B). The stubs are responsible for ensuring that the message conforms to the protocol and for hiding the details of the calling mechanism from the user. The actual stubs may exist in one of two forms:

- inline code produced by the compiler, or
- library level procedures.

The former is a more general form, but requires a specialized compiler, able to identify remote calls. The latter requires hand-coded changes to the library code.

The condition of disjoint address space in RPC requires special attention for reference parameters. In a conventional local procedure call, the reference parameter is an ideal method for allowing the called procedure to alter a large memory structure without the need to pass each element in the structure explicitly. The reference parameter is just a pointer to the specified parameter's true occurrence in the memory space of the caller.

In the case of RPC, this does not apply. All reference parameters must be replaced by the actual structure to which they point. This expansion is termed *marshalling*. The message containing the (possibly expanded) structure and other parameters is presented to the communications sublayer. Once it has reached the destination, a reverse procedure

is carried out and the data representing structures are placed into local memory by unmarshalling the message block. The call is then carried out as a local procedure call, using reference parameters where applicable. This whole process is repeated for any replies that may be forthcoming.

As in Sprite [Oust88], the stubs (both client and server) are generated by hand and supplied as libraries. Automatic stub generators or inline code as proposed by Nelson would be convenient, but would be more general than is necessary in DISTRIX.

4.2 The central switching exchange

In the following sections the exchange is described. The exchange connects all nodes in the network. The hardware and software are described. As this point could easily become a bottleneck, particular attention was given to ensuring efficiency and the cost of the software overhead was determined by benchmarking. The results of these benchmarks are presented.

Each exchange has a finite number of links. Multiple exchanges may be cascaded (output of one exchange acts as input to another) to increase the number of usable links. Usable links are those links to which processing nodes may be attached and exclude links used to interconnect exchanges. It is possible to cascade n exchanges, each of k links, to yield $n(k-2)+2$ usable links.

Another reason for using multiple exchanges might be to provide multiple, parallel data paths, allowing data flow between processors in the pool and the central kernel and file server simultaneously.

4.2.1 Hardware description

The Inmos transputer provides 4 bidirectional, serial links [Home87]. These links are connected directly to the processor and memory of the transputer (Figure 4.3). The links are able to operate at data rates ranging from 5 to 20 Mbit/second. This is the standard method of communicating between transputers.

For the interested reader, the technical details of the inmos link are explained more fully in section B.4. The prototype exchange processor uses a T414, as do most other nodes. Furthermore, the link speed is set at 10 Mbit/second. Thus the *predicted* throughput is

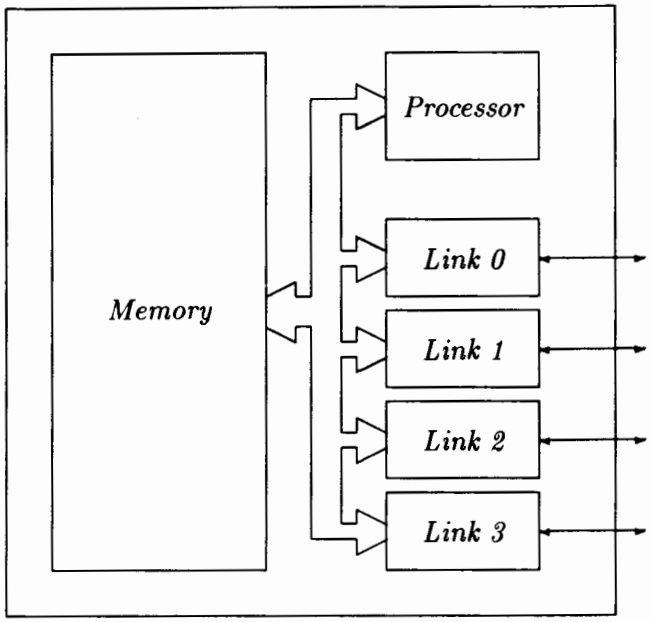

Figure 4.3: *The transputer links*

limited to 0.4 MByte/second.

In order to connect to other devices, link adaptors [Inmo88] may be used. For processors that have interrupt structures, these may also be generated by the link adaptor whenever the status registers are updated.

In order to use a transputer as the exchange, modifications were made to a transputer, providing it with additional links. These links take the form of link adaptors. One link adaptor is used for each *'pseudo-link'*. The link adaptor transforms an Inmos serial message into a stream of parallel bytes. This byte is made available in a register. Bytes placed in the outgoing register are sent across the link in a serial fashion. Additionally the link adaptor features two status registers that indicate when a byte has arrived or is ready to be sent. Each link adaptor is mapped onto an area of memory in the exchange transputer's memory.

The transputer does not support interrupts, but in addition to the 4 communication links has a special link known as the *event channel*. Software aspects in using the event channel are described below. The interrupt signals from the link adaptor may be fed to the event channel, causing a software detectable exception in the presence of an interrupt from the link adaptor.

4.2.2 Software description

The exchange software implemented in the DISTRIX prototype provides a store-and-forward exchange. Its purpose is to ensure the delivery of packets of data to another node in the system. The packets arriving at the exchange have the form of Figure 4.1 shown on page 24. The exchange must be given the packet length in order to accept the remainder of the packet. Each node is uniquely numbered with a 16-bit network address. The process number is not used by the exchange.

The exchange is implemented as a number of concurrent processes. Each process is allocated a link to which it listens for incoming messages. Each link has a dedicated buffer where the incoming message is stored. As soon as the entire message has been received, the exchange starts the sending sequence. In order to send a message, the process must gain exclusive use of the outgoing link. This is achieved using *semaphores*. A semaphore is allocated to each outgoing link. When a process has gathered a complete message, and the destination link has been determined, the semaphore is requested. Should the semaphore not be available at that time, the process will be suspended until such time as the semaphore has been released.

In order to discuss the mechanics of using the links, two groups of links are identified:

- hardware links and
- pseudo-links.

The hardware links are dealt with by the transputer machine instructions IN and OUT. The pseudo-links require somewhat more work. As mentioned in the hardware description of the exchange, the link adaptors offer the programmer some useful registers to communicate its status.

Provided the exchange operates on a dedicated processor, the most efficient form of status checking is simple polling of all links. Thus, the event channel is not used, but a brief description of how it could be used is presented.

The transputer machine instruction, IN may be used for the event channel. One process would be suspended on the event channel. When a byte arrives at one of the link adaptors, the event channel becomes active and the suspended process is revived. This process then examines the status registers of each pseudo-link to determine the source of the 'interrupt'. Thereafter, control is passed to the process associated with that pseudo-

link. This scheme wastes little processor time during the idle mode between messages as all exchange processes are suspended. It would be particularly useful if the exchange processor is shared with other processes.

The polling scheme is more wasteful of processor time, but has a slightly better response in a dedicated environment. Essentially, each pseudo-link process examines its own status register for incoming messages. Should the register be clear, it immediately places itself at the back of the scheduling queue allowing the next process to check its respective register. The processor time taken up by this polling scheme would potentially be a problem in a situation where the exchange processor is shared with other processes.

4.2.3 Exchange performance

As the exchange is unavoidably a point through which all messages must pass, it is necessary to determine the penalty being paid for the luxury of an 'unlimited' number of links. The choice of packet size is also optimized by measuring the performance under different conditions.

Method

In order to measure the throughput of the exchange, a dual processor 'bit-bucket' was created. It consists of a simple producer and consumer, each on separate processors. The producer is reproduced in an algorithmic form in Figure 4.4. A buffer (of increasing size) is repeatedly sent to the remote process. The connection between the processors is varied and the times recorded. Four methods of connecting the producer and consumer exist:

- Direct connection (no exchange),
- using two of the exchange's hardware links,
- using one hardware link and one pseudo-link, and
- using two of the exchange's pseudo-links.

The choice of packet size is also varied to reflect the effects of very small and very large packets on the efficiency of the connection.

The times for small transfers are difficult to record accurately, so the benchmark program ensures that the amount of data transferred is fixed. The packet size is inversely

proportional to the number of packets transferred. The fixed data transfer size is chosen to be 1 Megabyte (1024×1024 bytes).

Results

The packet sizes are varied from 16 bytes to 8192 bytes. The graphs (Figures 4.5 and 4.6) show the poor performance for small packet sizes. Note that the direct connections in both instances suffer only a minor decrease due to small packet sizes.

The reason why two similar graphs are shown, addresses the question of whether the network header information should be considered as data. Two sets of readings are taken. In the first, the 'real' data transfer, the header is *not* included in the 1MB fixed transfer. Consequently, small packets carrying only 16 bytes of useful or real data, are burdened with an additional 6 bytes of header data, in effect, resulting in a packet size of 22 bytes, transmitted 65536 times, or 1.4MB. In the other case, the 'raw' data transfer, the packet sizes indicated reflect the size of the actual, or raw transfer. The 16 bytes at the lower end of the scale consist of only 10 bytes of useful data and 6 bytes of header. The raw data transfers are only included for completeness, and for the purposes of the prototype discussions, only the real data transfers will be referred to in the future.

From the graphs we can see that the curves all tend to flatten out at a block size of around 256 bytes. Therefore, any block size (within memory constraints) greater than that will suffice. For the purposes of DISTRIX, in which disk blocks are frequently transferred from the file server to the satellite processors, the packet size is chosen to be at least as large as the disk block size of 1024 bytes. In addition to the 6 byte header, the transfer of disk blocks has a further 5 words (20 bytes) attached by the buffer layer. Thus in order to transfer a disk block using only one transfer, the packet size is set to 1050 bytes ($1024 + 20 + 6$). This is the largest message generated in DISTRIX. Thus the packet size should be no greater than 1050 bytes, as the additional buffer space would be wasted. Larger blocks may be transmitted, but must be split and sent as more than one network packet. This is the solution used by Sprite [Oust88] and is also supported in DISTRIX.

The effective throughputs of the various transfer types are presented in Table 4.3 for a block size of 1024 (the closest recorded figures to the chosen packet size). The theoretical entry is adjusted to account for real transfers which consist of 1024 transmissions of 1030 bytes each. The remaining entries are calculated by $1024/x$ where x is the time in seconds

```
MAIN:
begin
  bufsize    := 16
  iterations := 65536
  while (bufsize < 8192) do
    time := bench( bufsize, iterations )
    display( bufsize, iterations, time )
    bufsize    := bufsize * 2
    iterations := iterations / 2
  endwhile
end

FUNCTION bench ( size, iterations )
begin
  loop  := 0
  start := time
  while (loop < iterations) do
    xmit( buffer, size )
    loop := loop + 1
  endwhile
  stop := time
  return stop-start
end

PROCEDURE xmit ( buffer, size )
begin
  linkout( size+4 )
  linkout( node )
  linkout( process )
  linkoutchar( buffer, size )
end
```

Figure 4.4: *Algorithm to test the exchange for real data transfers*

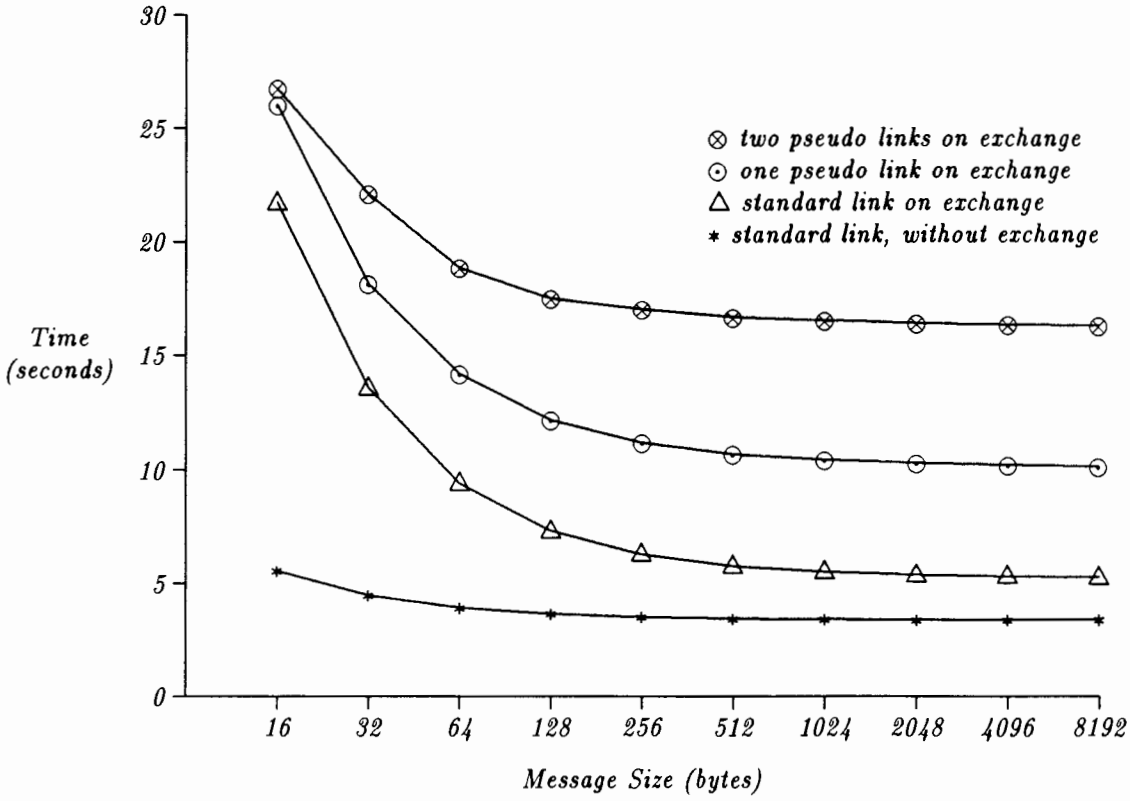

Figure 4.5: *Graph of time as a function of block size (Real data – see text)*

Block Size	Numer of Iterations	Time (μ seconds) Direct Link	Hard Link	Single Pseudo	Double Pseudo
16	65536	5557454	21705357	26030667	26738528
32	32768	4456449	13547600	18119032	22082712
64	16384	3905947	9384688	14155161	18806441
128	8192	3630696	7312669	12149258	17457799
256	4096	3493070	6278776	11160289	16984849
512	2048	3424258	5757068	10661827	16632232
1024	**1024**	**3389851**	**5497043**	**10407118**	**16477395**
2048	512	3372647	5365333	10276144	16364946
4096	256	3364046	5294391	10197308	16305860
8192	128	3359745	5248878	10134001	16258625

Table 4.1: *Timing results for real data transfers*

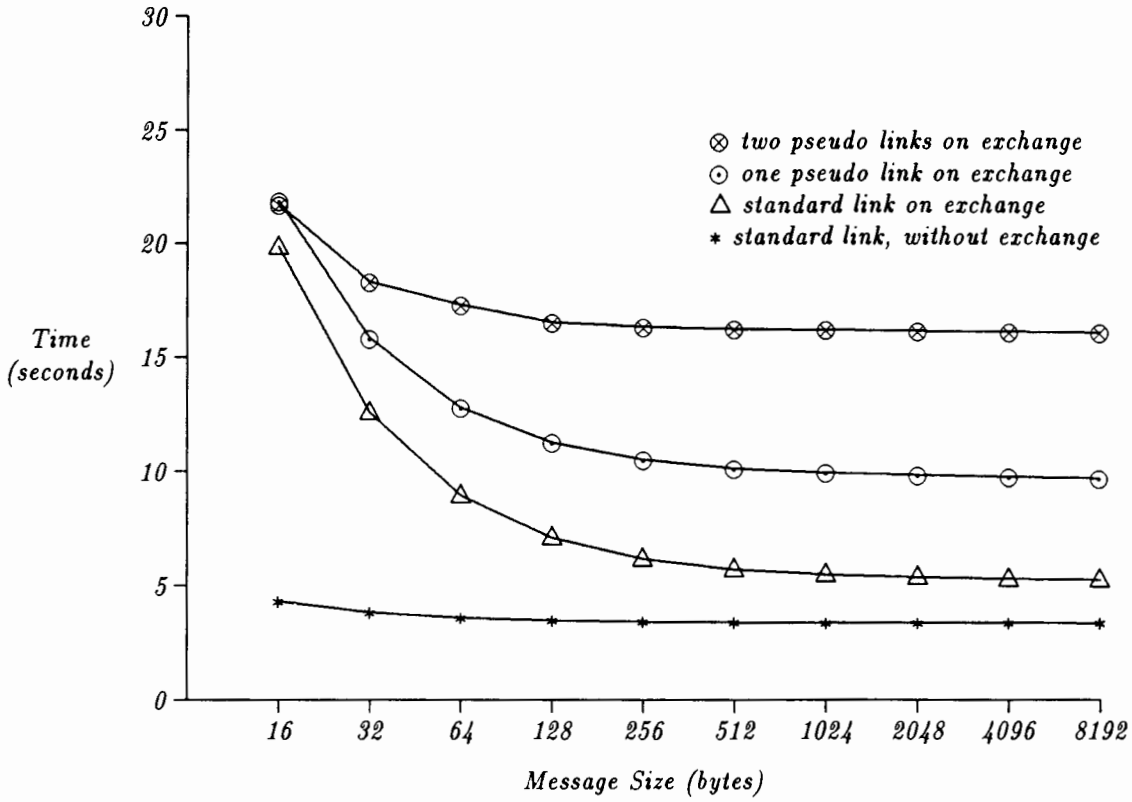

Figure 4.6: *Graph of time as a function of block size (Raw data - see text)*

Block Size	Numer of Iterations	Time (μ seconds) Direct Link	Hard Link	Single Pseudo	Double Pseudo
16	65536	4299163	19817941	21862625	21705343
32	32768	3827303	12553575	15806983	18284305
64	16384	3591373	8919168	12773357	17269546
128	8192	3473408	7073335	11244955	16514219
256	4096	3414427	6150231	10496886	16319401
512	2048	3384935	5693676	10111594	16223739
1024	**1024**	**3370188**	**5467105**	**9919748**	**16183060**
2048	512	3362816	5349557	9818076	16144396
4096	256	3359130	5286503	9755607	16123136
8192	128	3357287	5245004	9701411	16078365

Table 4.2: *Timing results for raw data transfers*

Method	Throughput (KB/s)
Theoretical	407
Direct	302
Hard Link	186
Single Pseudo	98
Double Pseudo	62

Table 4.3: *Throughput for transfers of packets of 1024 bytes of real data*

of the transfer.

Conclusions

The predicted throughput of 0.4 Mbyte/second is not achieved in the direct link transfer due to loop constructs, slow memory fetches and other timing overheads of some 25% in the benchmark program. In the other models, the overhead introduced is mainly due to the exchange software and the fact that polled links (the pseudo-links) are accessed by software, not firmware.

The use of the exchange is a flexible way of connecting the nodes in a distributed system of transputers. The cost of such flexibility is in performance. The complexity of the exchange software is low. Throughput could potentially be doubled on the hard links by using the T800 processor (which employs a more efficient overlapping protocol) for all nodes, including the exchange. The expense of the T800 over the T414 (approximately double) does not justify this at present, although the newly released T425, which replaces the T414, also supports the overlapped protocol. The doubling of throughput would only be present on the hard links, as the overlapping protocol is implemented in firmware. The pseudo-links make no use of the firmware routines for accessing the data and the link adaptor does not support the overlapping protocol at present.

It would be advisable to ensure that the traffic through the exchange is kept to a minimum to avoid overloading. This can be achieved by optimizing the data paths of objects such as disk blocks. In the present prototype, images of executable programs pass through the exchange twice, once while travelling to the central kernel and again while travelling to their respective satellite processor. The exchange traffic may also be reduced by using two exchanges, one for general traffic and another devoted to the file server.

4.3 Conclusions

A protocol based on the OSI reference model provides communication throughout the system. Remote Procedure Calls are used to hide the details of the communication from user processes. The initial goals are met.

A central packet switching exchange allows the interconnection of all processors. A modified transputer provides the hardware base. Simple software using a store-and-forward method routes packets between nodes in the network. The exchange is not as efficient as direct links between processors, but the penalty is minimized by correct choice of packet size. Multiple exchanges may be used, either to improve throughput by providing multiple paths, or provide for greater connectivity by cascading.

Chapter 5

The DistriX file server

This chapter devotes itself to the main goal of the work described earlier and brings us to the central theme of the thesis. File servers, in general, are introduced and the DISTRIX file server is discussed in depth. Its design goals are explained and motivated before re-introducing the communications mechanisms in the light of the file server. An outline of the implementation details follows.

A section devoted to devices in DISTRIX covers the concept of high level devices and provides details of this interface. Three specific device classes are discussed, namely the disks, terminals and real-time clock.

Finally, the performance of the completed file server is presented. The results are explained and possible improvements suggested.

5.1 Overview

The file server (FS) is that portion of the operating system '*having as its main purpose the storage and retrieval of bulk data*' [Birr80]. It manages the most visible resources in the system, typically handling all access to disks, terminals and printers. In UNIX-like operating systems, the user's view of all of these devices is similar; they are all treated as files that may be opened, closed, and possibly read from and written to.

Most UNIX devices are represented and manipulated as special files. A notable exception is the system clock. This device is not treated as a file, but is used by the FS to date-stamp files at creation or modification time.

Thus ordinary user files used to store data or programs are accessed by the same

mechanisms as devices such as a raw disk or terminal. Furthermore, directories (collections of files arranged hierarchically) are treated as files.

The unique identifier called an *i-node* [Ritc78] is the central access method for all files. The i-node contains all the information about a file. The FS in MINIX and DISTRIX retain a copy of the i-node table in memory as well as on disk. With the i-node, the FS can, amongst other actions, locate a file on the disk, examine its protection status or obtain its size.

The i-node is internal to the FS, and users do not have access to the files directly via the i-node. Instead, a *file descriptor* is used as the low-level identifier in system calls such as *read*, *write*, *lseek* and *close*. The file descriptor is obtained by the *open* system call. *File pointers* are a feature offered by most C libraries. File pointers offer buffering and are more portable than file descriptors, as they operate on *streams*. Access to streams is independent of the operating system.

Thus far we have used the term *file server* in place of the more common UNIX term *file system*. The former is used for two main reasons. Firstly, the generic term file system usually refers to both the software controlling the files and the actual disk image — this is confusing and the term file system will now be restricted to the disk image. Secondly, the DISTRIX file server is exactly that — a server. Requests for work arrive from remote processors by means of a communications network. The individual tasks are carried out by the server and the replies sent back via the network.

5.2 Design

One of the primary aims in moving the MINIX file server to a remote processor, was to maintain as much of the original file server as possible within the constraints of the architecture of DISTRIX. With this in mind, the environment was often adapted to suit the needs of the FS and sacrifices were made as a result, particularly in performance. The gain was realized in the ease of implementation.

The aspects of communications are examined with respect to Remote Procedure Calls and a fitting way to implement buffering of requests is presented.

The DISTRIX file server is not a *distributed file system*, but is a *remote file server* servicing a *distributed operating system*. A distributed file systems is defined as many file

servers working cooperatively to create the illusion of a single file system [Stur80, Coul88].

Furthermore, the current file server only services one request at a time, completing the request fully before proceeding with the next. This simplifies the design, but does not make full use of the processor.

5.2.1 Communications

This discussion on the communications mechanism will place emphasis on the aspects pertinent to the file server.

As a server, the FS must be *fair*, servicing requests on a first come, first served basis. Requests arrive and are queued until the FS can service them. To meet the fairness requirement and simultaneously take advantage of the scheduling features of the transputer, an infrastructure of communications software and agent processes was created. The agents correspond to the server stubs depicted in Figure 4.2 in Chapter 4 which introduced Remote Procedure Calls.

The agent processes act not only as marshals, but also as message buffers and, by means of the transputer scheduler, ensure fair scheduling of all work requests.

Several schemes are presented in order to effect controlled buffering of requests arriving at the FS:

- *Explicit Dynamic Buffering*: One process receives all work requests, allocates memory to store each request, and places a request identifier at the end of a queue which the FS services in finite time. This dynamic memory allocation implies that a system call may fail due to memory limitations. The failure may not be reliably predicted by a user. We expand on this problem later in Section 5.2.2.

- *Explicit Static Buffering*: One process receives all work requests, but the buffer slots for every possible client are allocated at compile-time. Each client already has an entry in the process table. This approach solves the unpredictable failure problem noted above, but may limit the number of possible clients allowed in order to accommodate buffer space for each agent process.

In general, the *explicit buffer* approach operates as follows. The Buffer layer (see section 4.1.2) must accept each incoming work request immediately, so as not to block the Network layer. Since the FS runs as a separate process, it has no knowledge of jobs queued,

and, in particular, has no knowledge of the order of the jobs. Thus, when the FS has completed its last job, it must indicate its willingness to accept work by means of a message to the Buffer layer. The Buffer layer, using the queue of requests, chooses the next job and submits this to the FS. Since replies are not buffered, but are written directly to the Network layer, this approach displays *asymmetry* with respect to requests and replies.

Hoare [Hoar85] describes a buffer as a series of concurrent process, accepting input data from their 'left' and producing output to their 'right'. In this spirit, DISTRIX allocates a process to each buffer.

The DISTRIX variant is called the *implicit buffer* approach and once again has two variations:

- *Implicit Dynamic Buffering*: As new processes are created elsewhere in the system, a corresponding agent process is created simultaneously on the FS. The sole function of this process is to accept work from, and carry out work on behalf of its client sibling on the mini-kernel. The downfall of the dynamic creation of processes is the same as for the explicit buffer approach: memory limitations may cause run-time failure.

- *Implicit Static Buffering*: The same as above, but an agent process is created for every possible client. Each client already has an entry in the process table. The actual process creation takes place at boot-time, but as the buffer space set aside for each process is declared at compile-time, the creation is guaranteed to succeed. This may limit the number of possible clients allowed in order to accommodate buffer space for each agent process.

This approach was chosen to allow the choice of which job to run next to be made by the transputer scheduler and not by the Buffer layer. However, inherent favouritism exists in the scheduling algorithms and the main program of the FS has to adjust an internal table to provide 'fair' servicing of all queued requests. This is described more fully in section 5.3.2. This approach displays *symmetry* with respect to requests and replies, both passing through the agent process (Buffer layer).

By means of this buffering, multiple requests for work may be stored simultaneously on the FS processor, but due to the original FS design, requests are carried out sequentially. Thus, while fetching a block of data from a slow secondary storage device, such as a disk,

no other work may be accepted despite the FS being idle during this time. This contrasts with a description [Tane85] of the file server in the V Kernel [Cher83]. The V Kernel's FS is described as a team of processes, with one member being able to continue processing whilst another is suspended on some task, such as disk I/O.

This scheme, in which the agents (or server stubs) provide buffer space, allows for the future use of a multi-threaded FS servicing the requests of any number of agents simultaneously.

5.2.2 Advantages of static buffering

Earlier it was mentioned that dynamic buffering can lead to run-time failure. Whilst it is possible to return an error code when the agent fails to allocate memory for buffering, this would require additional interaction with the central kernel. The central kernel is responsible for administering the process creation. Before allowing the new process to be created, it would thus have to obtain confirmation of the successful creation of the process (or buffer) from the FS.

For the prototype it was decided to eliminate this interaction and create an FS that can guarantee successful buffer creation by performing the allocation at boot-time. As this forms part a single startup routine, there is no further cost associated with static buffering. These processes are idle until work arrives. On the transputer, idle processes pose no processing overhead.

On the other hand, dynamic buffer creation requires not only kernel interaction, but also memory allocation, even though the time required to allocate memory is small (approximately 1 microsecond).

5.3 Implementation

This section will briefly discuss the options chosen. An explanation of the mechanics of work requests being processed by the FS is presented. As the development of any large piece of software requires readily available hardware and software the tools used to develop the file server are described. No large software port is without problems. A few of the more serious problems are discussed and their respective solutions presented.

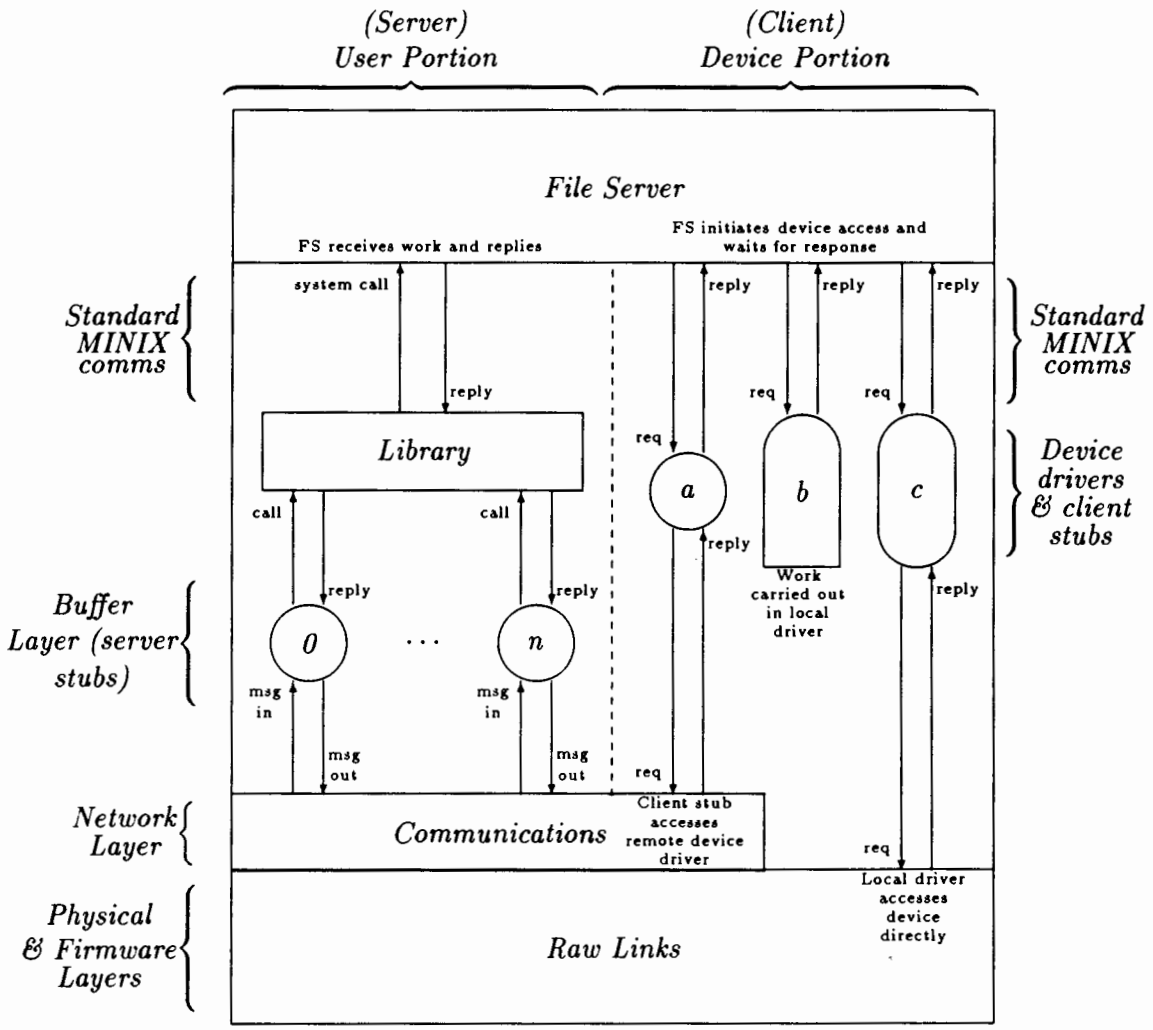

Figure 5.1: *Conceptual structure of the* DISTRIX *file server*

5.3.1 How it works

Figure 5.1 displays the DISTRIX file server structure. Notice that the FS is divided into two halves, each representing a way in which the FS may operate, the *server* being the normal way of handling user requests and the *client* in which the FS makes use of other servers.

The server stubs $0 \ldots n$ each act on behalf of a remote process. They are also referred to as *agents*. Their only form of communications with the client stubs is through the Network layer. They may not initiate any communications until a message is received from their respective siblings on the remote processor. The procedure that produced the message at the client stub (see Section 4.1.3) is now reversed. The message is unmarshalled, placing data structures into local memory and then performing a standard library call (once again

making use of reference parameters). The library translates the call into a standard MINIX message which is then passed on to the FS. The FS processes the request and either replies to the waiting agent, or requests further assistance from a device driver before replying to the agent.

The agents perform *implicit static buffering.* Each agent is entered into the FS processor's scheduling table at boot time and has its own local buffer space. The code for the agents is shared to conserve memory.

The client portion of the FS operates in reverse order to the server portion. The FS sends a request to one of the device drivers. The message is a standard MINIX message. In the diagram, three drivers are depicted, each representing one of the three options possible.

Driver *a* is the simplest form of device driver. It is no more than a client stub. It contains no device driver code, but merely marshals the request and then carries out a Remote Procedure Call. The message is sent to the Network layer for delivery to the remote driver. The remote driver performs the work and sends the reply and any resulting data back to driver *a*, now suspended. Driver *a* unmarshals the reply and in turn replies to the waiting FS. During the initial development of DISTRIX, all device drivers resided on a remote processor and the FS relied on drivers such as *a* to effect the invisible transfer of requests. In the current DISTRIX implementation the FS uses such a driver to communicate with the central kernel. Its primary purpose is to pass on interrupts generated by the user at the keyboard. This message is sent through the network to a corresponding file server stub on the central kernel.

Driver *b* represents a device driver that is contained wholly on the FS processor. No external assistance is required by drivers of this class. The message arrives, causing the driver to perform the requested task. This class of driver does not access any of the links and has no RPC mechanism, handling only local requests from the FS. In the current DISTRIX implementation only one such driver exists, namely the *clock* (discussed in section 5.4.4). The reply is sent back to the waiting FS.

Driver *c* is the most common form of driver in DISTRIX. The entire device driver is contained on the FS processor, however, communication with the external device is required. Any access to a device is direct, using the raw link and not passing through the Network layer. The protocol is specialized and is defined individually for each device to ensure optimal performance. In general, these drivers are not extremely complex, the

devices they control are intelligent. The disks and terminals in DISTRIX make use of these drivers and are described in sections 5.4.2 and 5.4.3.

5.3.2 Problems

This section will not expand further on the problems of the actual development environment, such as the lack of debuggers (see Appendix D), but will concentrate more on fundamental difficulties experienced during the porting of the MINIX FS.

MINIX was originally written for the Intel 80x86 range of processors. Under MINIX they operate as 16-bit processors, and the C compiler used to compile MINIX supports a 16-bit **short int** type. The LSC C compiler produces code for 32-bit transputers and the **short int** type, produces a normal **int** of 32-bits. By the nature of the FS, several data structures rely on 16-bits fields, to allow the structure as a whole to be some power of 2. Such structures had to be redesigned and fillers inserted, to pad the structure to a power of 2.

Whilst the Mock additions [Mock88] (see Section D.1.1) to the LSC C compiler proved invaluable, there exists an inherent problem in the scheduling features of the transputer. The scheduler is not fair. From a list of ready-to-run processes, the scheduler does not make a random choice, but always chooses the first in the list. This problem is also present in the Occam language [Jone89]. The format for the LSC **ProcAltList()** library function requires a list of pointers to channels. The function suspends the calling process until one of the channels in the list becomes active. If one of the channels already contains a message at the time of the call, the function returns immediately, providing an index value indicating which channel was active. The documentation does not make clear the expected result should more than one of the channels be active. In practice, the function always returns the index of the active channel nearest the head of the list. Clearly, this could lead to a situation in which lower numbered processes are shown favour by the FS. As the FS is supposed to be fair, a different scheme must be used.

The solution lies in re-ordering the list before each call to the library function **ProcAltList()**. If $p_0, p_1, p_2, \ldots, p_n$ are the channel pointers for agents $0, 1, 2, \ldots, n$, the initial list presented to **ProcAltList()** is:

$$L : p_0, p_1, p_2, \ldots, p_{k-1}, p_k, p_{k+1}, \ldots, p_n, 0$$

Assume that agent k is the next to request work from the FS. After completion of the request from agent k, the list is re-ordered to look as follows:

$$L' : p_{k+1}, \ldots, p_n, p_0, p_1, p_2, \ldots, p_{k-1}, p_k, 0$$

The time to perform such a re-ordering is negligible, although more memory is required. Ordinarily, the list L requires only $n + 1$ elements. In order to allow the re-ordering to take place without physically rotating the list, a double list is used (containing $2n + 1$ elements). By moving the head and tail of this double list, a different ordering may be presented.

This solves the problem of perpetual lockout, but is still not entirely fail-safe. As can be seen in list L', agent $k - 1$ has moved near to the tail of the list and may be served *after* agent $k + 1$.

5.4 Devices

Device drivers allow UNIX systems to treat peripherals as files. The low level details of the actual device are hidden by a layer of software that presents a uniform view of the device. A device driver exists for every peripheral in a UNIX system. In MINIX and DISTRIX, the clock is also accessed by means of a device driver.

In DISTRIX, the device drivers have been simplified by making use of so-called high level devices. The devices present in the DISTRIX prototype are the disks, terminals and a clock.

5.4.1 High level device interfaces

Intelligent device controllers are becoming more prevalent. Device drivers interfacing to such controllers are no longer concerned with the register or interrupt level access to a device, but communicate at a more logical level.

So, for instance, a disk controller might no longer require the motor to be turned on before allowing access, instead the controller itself decides whether the motor needs to be turned on when a request for disk access is received. The access may no longer require the client to describe the location of the data by head, cylinder and sector, but rather by a logical block number. In the case of a terminal, the details of interrupts and transmitter

registers are removed and the only details required may be the number of the terminal together with the respective command or data.

The removal of the mundane details of interrupts and other low-level details results in much device drivers, again extending the idea of modular components on modular processors further.

5.4.2 Disks

The disk system consists of a high level device called a Mass Storage Controller (MSC) manufactured by Parsytec [Mula88]. An MSC consists of a transputer connected to a SCSI controller. Thus, the transputer is able to issue control commands to the devices attached to the SCSI controller and transfer data between its local (4MB) memory and the attached disk, diskette or tape.

The only interface between the MSC and the high level device driver on the FS processor, is through the Inmos link. The software on the MSC accepts the commands issued by the device driver and reinterprets them into low level commands for the SCSI controller.

Four logical channels are used to pass commands, results and blocks of data for reading and writing. There are, at present, seventeen commands supported by the MSC. A separate channel returns (for each command) a corresponding set of result values indicating the nature of any errors found in either the command itself, or in the execution of the command. These channels may be multiplexed over a single link as the transmissions are not duplex.

In the DISTRIX prototype, an MSC was not available. An emulator replaces the hardware. The emulator is written in C and executes on a PC/AT. Provision is made for two fixed disks and two floppy diskettes. These are emulated by MS-DOS files, each representing a device. The emulator polls the link adaptor connecting it to the high level device driver until work arrives. The request is interpreted and the relevant operation carried out. A request for a logical block results in the emulator seeking to a corresponding offset in the respective MS-DOS file. Any reply data are then sent back out on the link to the waiting device driver.

The library is used with the device driver and hides the details of communications with the MSC. The interface is consistent with the MSC documentation and should thus not need alteration should a real MSC replace the emulator.

5.4.3 Terminals

The terminal system consists of a high level device called a Terminal Interface Module (TIM). The TIM is a proposed piece of hardware consisting of a transputer connected to several Universal Asynchronous Receiver/Transmitters (UARTs).

The UART is responsible for the parallel-to-serial (and reverse) translation of data. Registers are used to hold newly arrived data or data to be sent to the serial device. Additional registers contain status information and control the operation of the UART.

Each UART may control one terminal (or more in some cases). Terminals are a general class of serial devices and may include the usual keyboard and screen combination, as well as punch card readers, printers and graphics plotters.

In the DISTRIX prototype, a TIM emulator is implemented as a C program on a PC/AT. This program accepts (over a standard inmos link) commands from the high level device driver on the FS processor. The commands are preceded by an identifying sequence to describe the exact terminal to which the remainder of the command applies. The commands supported at present allow the device driver to *read* from, *write* to or change the control state of a terminal (*ioctl*). Data arriving from a terminal are buffered on the TIM until the FS device driver requests them in the form of a *read* command.

The emulator's interface to the serial ports is at the BIOS level, using the built-in features of the PC/AT to avoid the troublesome task of handling interrupts. Since the TIM emulator executes on a dedicated processor, the use of expensive polling schemes do not pose a threat to system performance.

One important point worth noting is the absence of the concept of *process groups* in MINIX. The process group is only explicitly explained in System V [Bach86, XOPE87], but the mechanisms must have existed implicitly in Version 7 in order to allow multiple concurrent users. A process group identifies any number of processes related by a parent process (such as *login*). It is important that all of (and only) these processes receive a signal when the user causes an interrupt from the keyboard or when the parent process terminates. In System V it is possible for a process to become the leader of a new process group that may or may not have a terminal associated with it, by means of the *setpgrp* system call.

In the DISTRIX file server, the terminal device driver determines when a new process group leader has been formed by keeping track of the number of times that a terminal line

has been opened by the user. Whenever a terminal is opened[1] for the first time, the process issuing the *open* call is identified and the central kernel informed. Correspondingly, each *close* system call reduces the counter until it drops to zero. Thereafter no signals will be sent until the terminal again has a group leader.

5.4.4 Clock

The clock provides the FS with the ability to get the time (in seconds since January 1, 1970) or to set the clock. The clock must be set by user input during the boot. This is common amongst many UNIX systems.

A global, 32-bit **long** is used to represent the number of seconds since boot time. This is added to the number of seconds since January 1, 1970. The global variable representing the time since boot is incremented every second using the transputer scheduler. This scheduler allows any process (here the clock ticking process) to sleep until a specified time has passed. This time is set by the clock ticking process to be the sum of the current time and one second. Should the process not be scheduled within any particular second, the clock's accuracy will momentarily drift. However, as the clock ticking process is one of the few high priority processes, when it is next scheduled it will perform several ticks simultaneously until it has caught up any lost seconds and accuracy will be restored.

One more anomaly remains to be explained. As the clock ticking process runs at a high priority, its view of the transputer clock is at the level of microseconds. Consequently, one second of real time consumes one million transputer ticks. With a 32-bit, signed integer counter for the transputer clock, the largest value possible is 2^{31}, or 2 000 million. So in just 2 000 seconds (about 36 minutes) the transputer clock overflows. This results in one second lost and is accounted for by the clock ticking function. The overflow is well explained by Packer [Pack88].

5.5 Performance

The DISTRIX file server has been operational since the beginning of 1989. It has thus far only been used to perform benchmarks and test the remainder of the system. The benchmarks performed to date show that the FS is faster than the original MINIX system,

[1] This is carried out by a device opening routine which previously had no purpose in MINIX.

System Call	Explanation
read	read block from file
write	write block to file
lseek	move to position within file
fstat	obtain file statistics (given file descriptor)
stat	obtain file statistics (given file name)

Table 5.1: FS *system calls chosen for benchmarks*

but when timed across the network, performance for single users is considerably worse than for MINIX.

This is caused by the latency experienced by messages passing through the exchange. Under load (multiple processes per processor), the advantages of remote processing are realized.

Method

A selection of FS system calls was made. The chosen calls, displayed in Table 5.1, were used in a skeleton benchmarking program and the times for each call measured.

In order to obtain accurate times under the MINIX environment, care had to be taken due to the low granularity of the clock. Since the only available clock (the one offered by the MINIX FS) is only accurate to 1 second, the number of repetitions of a particular system call had to be high (10 000–25 000). This allowed the time for an individual system call to be quoted with microsecond accuracy. With such a high number of iterations, the overhead of the loop was itself a factor. The time of a null loop of equal length was therefore subtracted from the total time

Under DISTRIX it was possible to gain access to the transputer clock that has a much finer granularity. This was done by embedding machine language instructions into the benchmark program.

Times for the original MINIX system were recorded on an 8MHz PC/AT equipped with a fixed disk drive. The times for DISTRIX were recorded on 20MHz T414 transputers. The FS processor was a 20MHz T800 transputer. The exchange processor was used with both the FS and processor pool attached via hardware links. In order to determine the

cost of a Remote call versus a Local call, the benchmarks were also carried out on the FS processor, using no RPC.

These times were measured under two different conditions. In the first instance, the user processor (the 80286 for MINIX and a processor from the processor pool for DISTRIX) were not loaded with any processes unrelated to either the benchmark itself, or the system tasks. The second set of readings were taken under a condition of CPU load. The user processor was loaded with a background process before running the benchmark suite. The background process was merely an infinite loop calculating sums and differences of integers. To avoid compiler optimizers discarding the loop, the variables were continuously altered.

For the *read* and *write* system calls, an *lseek* call was performed each time to reset the file pointer to the head of the file to prevent the file from growing out of control. This also ensured that all accesses were local only to the FS cache and did not require any disk access. This was desirable as the DISTRIX disk is an emulator whose performance is limited and might taint the results.

Results

Figures 5.2–5.4 show the results of the benchmarks under conditions of *light load.*. Clearly, DISTRIX is significantly slower than MINIX, particularly with regard to FS operations such as reading and writing of large blocks of data. For the system calls presented, DISTRIX is, on average, three times slower than MINIX from the user's perspective. This is weighted rather heavily by the particularly poor performance displayed by the *read* system call for block sizes of 2048 and 4096 bytes, with penalizing factors of 6 and 10 respectively.

Cheriton and Zwanepoel [Cher83] achieved RPC in the V Kernel with a performance penalty factor of 4. Compared with MINIX on and 80286, DISTRIX has achieved RPC within these bounds.

The times for the calls carried out locally on the FS processor are more encouraging, being on average three times *faster* than MINIX. This is as a result of the greater processing power of the transputer.

Figure 5.3 represents the difference between the times recorded under DISTRIX as measured by the user process in the processor pool, versus times measured on the FS. It clearly shows that the weakness lies in the remote procedure call overhead. The times for the large *read* and *write* system calls confirm this fact. The need to transfer large blocks

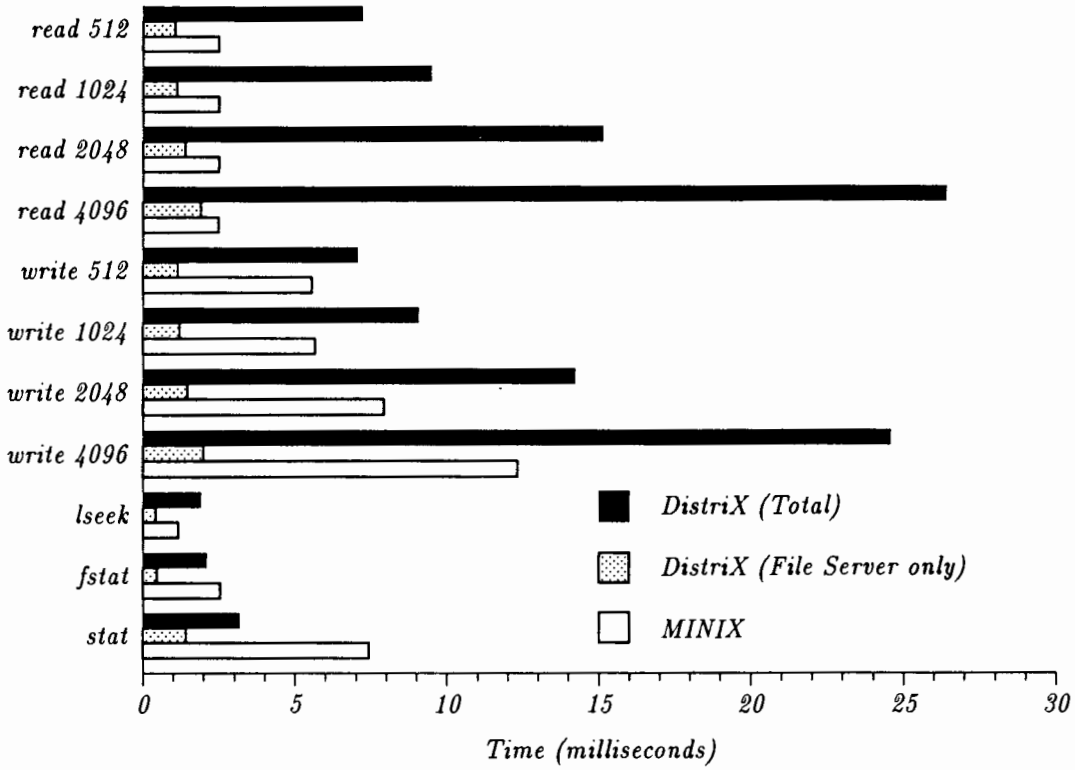

Figure 5.2: *A comparison of the times to perform selected* FS *system calls*

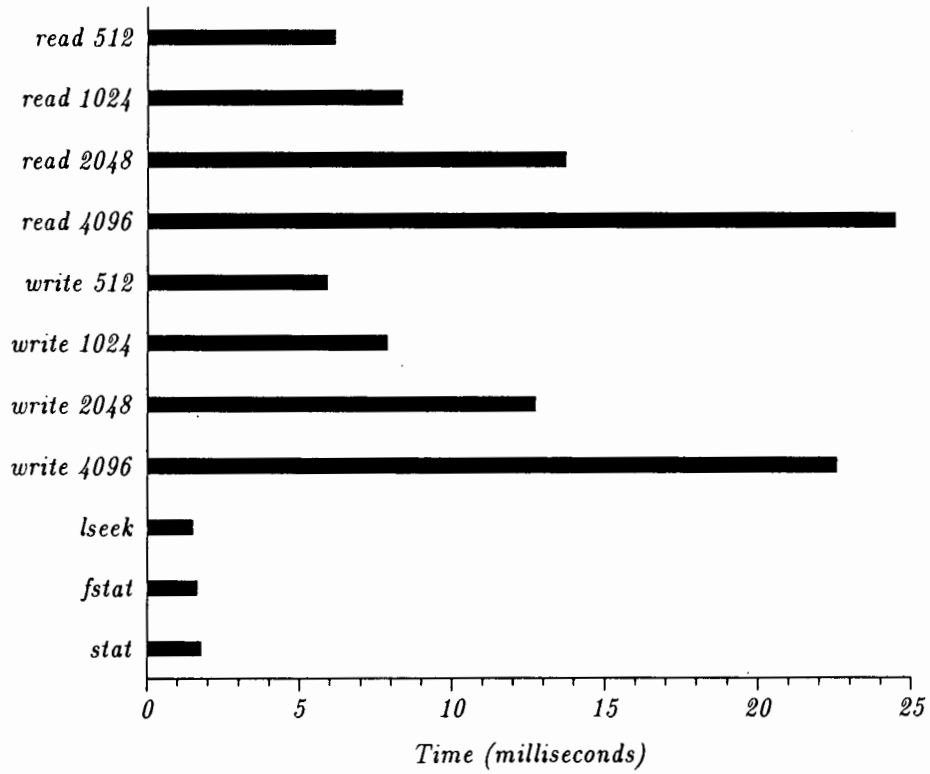

Figure 5.3: *Graph of* DISTRIX *remote procedure call overhead*

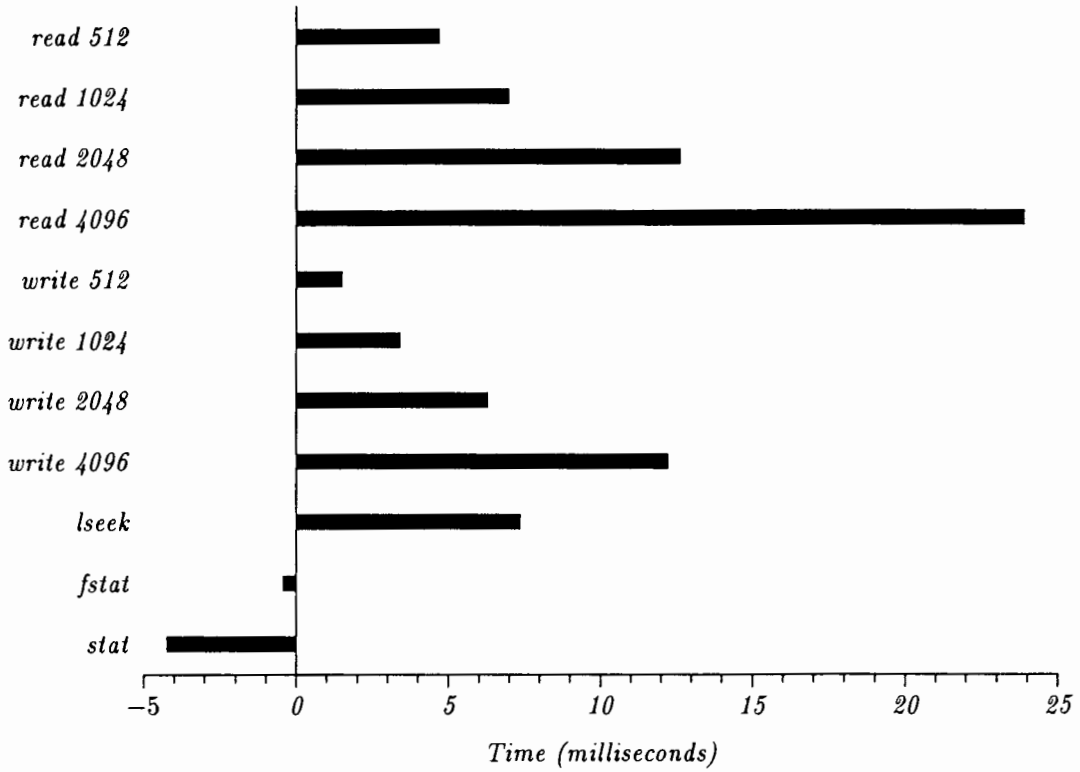

Figure 5.4: *Graph of* DISTRIX *penalty over* MINIX *per* FS *function*

of data between processors is expensive. On average, 85% of the time measured by the user processor is spent by the RPC mechanism. When measured purely on transputers, we must conclude that the DISTRIX performance penalty is now higher than a factor 4. This is not acceptable.

Under conditions of *heavy load*, however, the situation is reversed. Due to the amount of time spent in moving data via RPC's, the DISTRIX times suffer only mildly under load. The time consumed by the background load process does not affect the exchange or FS processor. Conversely, in MINIX, the background load process is contending with both the benchmark process and the operating system, slowing both down considerably. The graphs are not presented as the factors are so large as to make comparisons meaningless. MINIX executes approximately 20 times slower than DISTRIX. Naturally the times recorded on the FS processor are not expected to change as the load on the user processor is unrelated. The DISTRIX times measured on the user processor increase by an average of 6.6% when the load is added. The corresponding increase under MINIX was of the order of 3500% (or 36 times slower).

Conclusions

The cost of RPC for system calls requiring low network traffic is low (around 2 milliseconds latency through the exchange). The cost of RPC is high for block transfers and alternative measures need to be considered. The presence of hardware level memory protection might allow each user processor in the pool to have a local file server. All such servers may be connected to form a distributed file system as described earlier.

The poor performance relative to MINIX is eliminated in a situation in which the 80286 is loaded by a CPU-bound job. In the DISTRIX model, with a degradation of only 6.6%, such a situation brings to fore the advantages of a multiprocessor system. While the user processor is kept busy with the CPU-bound job, the FS processor and exchange may continue processing the system call unhampered.

The severe penalty for MINIX under load is brought on by MINIX being designed as a small system, ideally managing single processes. The scheduling is simple round-robin, with fixed time quanta for all processes. Time quanta are not adjusted according to the work profile.

One way of improving the performance of file server calls, involves the use of local caches on each user processor. This method is used very successfully in Sprite [Nels88, Oust88], but it is unlikely that it would be implemented in DISTRIX, as the poor memory protection offered by the transputer (discussed in section 3.1) can result in the cache being tampered with by a user process. In addition, the mini-kernel would have to interpret the FS calls, filtering out calls that can be satisfied locally, before passing them on to the FS. This goes against the modular design. Furthermore, cache consistency then becomes a major problem, although hardware solutions to this problem may be applied [Cher86, Cher89].

The cache local to the FS is safe from user processes and is able to reduce device access sufficiently. A larger memory on the FS processor will allow a larger cache, displaying a better hit ratio. As has been mentioned, the benchmarks used ensure that the cache is always used, thus the figures presented are independent of cache size.

5.6 Conclusions

The file server is responsible for the management of all storage resources in the system. It is also the point at which the user interfaces with the operating system by means of terminals. Designed as a remote file server, the FS accepts requests in the form of system call level RPC's. It is not a distributed file system. To service all requests fairly, a system of agent processes is created, each process acting on behalf of a remote client and as a buffer for requests and replies. The buffers should not prove to be restrictive in future implementations of a multi-threaded FS.

The FS not only serves users, but in the process may become a client of another server such as a device driver. Device drivers are designed to give the user a uniform view of all devices and files. The device drivers may reside in part or in whole on the FS processor and usually make use of an intelligent device controller with a high interface level. Fixed disks, floppy diskettes, terminals and a system clock are the devices supported at present.

Due to the communications and RPC overhead, the overall system performs poorly under light load. Under heavier loads it suffers a significantly smaller penalty than do equivalent uniprocessor systems.

Local caches would improve performance on the user processes, but the current design and processor limitations do not permit such a scheme. For the present, the cache on the FS will suffice.

Chapter 6

Conclusion

DISTRIX is a multiprocessor, UNIX system based on MINIX. it has been operative since the beginning of 1989. Using a network of 5 transputers, the system has been tested and benchmarks have been performed on it.

The satellite model for multiprocessor systems was chosen because it displays simplicity, but additionally fits in well with the modular design of MINIX. The transputer is ideally suited to communicating with other transputers by means of a narrow channel, namely the link.

The use of the satellite model has required the creation of a central switching exchange in order to allow all satellites to be connected to the central servers. This exchange has had a large influence on the specification of the communications protocol. It has not only affected the protocol itself, but for reasons of efficiency, the size of the data packets too.

The four layer, end-to-end protocol, designed to accommodate the remote procedure call mechanism and allow traffic to be routed by the central switching exchange is a balance between flexibility and efficiency.

The file server is based on the MINIX file server, and the changes applied to it were kept to a minimum. Shortcomings in MINIX, such as the lack of process groups, had to be overcome by modifying the function of the terminal device driver.

The file sever consists of a number of server stub processes that buffer requests. The requests are passed on to the file server itself, as standard MINIX messages. The server then carries out the request, possibly becoming a client in order to obtain assistance from another server or a device.

In meeting the original goals, we have discovered that UNIX can be implemented on

transputers, albeit at the loss of some of the potential power of the transputer.

I have personally gained a great deal of experience in distributed operating systems in general and more particularly in UNIX and the transputer.

The workbench created out of the project has much scope for future development as the present performance of DISTRIX has been disappointing. The possible future work is outlined next.

Future work

There are many ways in which our initial system can be improved. Performance appears to be a major bottleneck at present, particularly with regard to I/O. In this area, the exchange must be seen as the weak point.

An immediate benefit may be gained by lowering the exchange traffic. This can be achieved in two ways, firstly by optimizing the route travelled by data. The executable code for all processes executed on the user processor travels from the file server to the central kernel and then to the satellite, thus travelling through the exchange twice. Secondly, it is possible to use two exchange processors, one dedicated to the central kernel and the other to the file server. In this way, the traffic flow is split and the processes making use of the central kernel exchange will not be penalized by the high volume of traffic generally associated with the file server exchange.

As has been mentioned, there is currently a project under way in which the file server is being parallelized. This will bring it in line with the redesigned central kernel. The advantages to be gained from this will be realized under conditions of high device access. While the present file server is attending to a device access, it is suspended until the device replies. During this time, a multi-threaded file server would be able to continue executing another request, making more efficient use of the file server processor.

Work in this field is still relatively new, particularly when dealing with UNIX. The last decade has seen a rapid growth of UNIX and the need for a solution to poor performance of heavily loaded UNIX systems exists now more than ever. The emerging technologies of self-contained computers such as the transputer promise to open the way to alternate architectures.

Appendix A

The MINIX operating system

In an attempt to provide students with a more concrete, practical view of operating systems in university courses, Andrew Tanenbaum of the Vrije Universiteit at Amsterdam in The Netherlands produced an operating system that, together with its companion text [Tane87], could be used in teaching.

Using UNIX Version 7 (V7) [Bell83] as a model, MINIX was written (using C and some assembler) for the IBM PC range of microcomputers. It provides a multiprogramming environment with a shell based on the Bourne shell [Bour83] and includes a compiler and a subset of the usual UNIX utilities (*cat, grep, make,* etc.). In future references to UNIX, it will be assumed, unless stated otherwise, that the same applies to MINIX.

A.1 Reasons for choosing MINIX

MINIX was chosen as the starting point for the DISTRIX operating system to simplify the development of the prototype. The choice was motivated by the following characteristics displayed by MINIX:

- availability,
- small size, and
- highly modular design.

As MINIX is intended as an academic tool, it is sold at a low price and includes all the source (save the compiler for which source is available elsewhere). Thus, no source licensing problems exist.

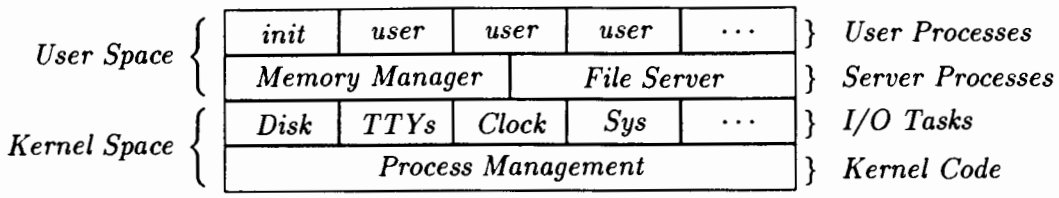

Figure A.1: *Modular structure of* MINIX

MINIX is a small operating system. Together with the book, the entire operating system code can be read, and a cursory understanding of the system as a whole can be gained in a short time.

One unusual feature of MINIX is that a very well defined modular structure has been imposed on the system. This modularity is essential for the distribution of the component parts of the operating system and is described more fully in the following section. The merits of porting MINIX are outlined by Lewis [Lewi89].

A.1.1 The MINIX modular structure

MINIX is structured as a number of discrete processes, each running in their own address space, with no access to global data and only communicating with other processes by means of *message passing*. This structure is illustrated in Figure A.1. The kernel is responsible for process management and the message passing mechanisms. The I/O Tasks are device drivers, providing the low-level services needed to communicate with hardware devices. The Servers are responsible for handling system calls to manipulate files and memory. Most of the devices (excluding the clock) are treated as files.

The structure of MINIX differs significantly from that of UNIX, in particular the handling of system calls and the placement of device drivers in the code is unusual. The structural and mechanical differences are described next. Other differences, particularly those pertaining to the supported system calls and commands are described more fully in section A.4.

For the purposes of performing system calls, most implementations of UNIX rely on the fact that the processor can operate in at least two modes [Bach86]. The first is generally an unprivileged *user mode* in which restrictions are placed upon certain instructions, either preventing the user from issuing them, or limiting the memory space in which they may operate. In the second mode, often referred to as *kernel mode*, the limitations are

removed, and any instruction may be issued, in any region of memory. In UNIX only the kernel or central core of the operating system is permitted to perform these instructions.

When a user process wishes to perform a privileged task, it must do so by means of a *system call*. The UNIX kernel is generally one large, monolithic program performing system calls by means of kernel traps.

MINIX is based on the client-server model and relies on its message passing mechanism to allow user processes (clients) to perform system calls. The parameters are placed in specific positions in a message block. The message block is transferred to the specific module (File Server, Kernel or Memory Manager) by means of the blocking SEND instruction. Replies from the kernel are sent in a similar fashion. The message block is limited in capacity and larger items such as disk blocks are not passed through the message block itself, but are copied directly to or from the memory buffer specified by the user process.

The MINIX server processes manipulate files and memory. Each have a single point of entry through which work requests are accepted. The server processes are essentially user level processes, having entries in the normal process table.

The file server provides all the file manipulation facilities (creation, deletion, etc.) as well as providing access control. The term 'files' includes devices such as the disks or terminals, and it is often necessary to physically access these devices. To maintain uniformity, all such access is provided through device drivers, hiding the specifics of the device and providing a uniform interface for all devices.

The device interface also relies on message passing. Messages conforming to the standard MINIX message format [Tane87, section 2.5.3, p89] are sent from the file server to the device driver process. Each device driver is a discrete process with a similar structure to the server processes, having a single entry point for work requests.

It is particularly this structure of discrete, communicating processes that led to the use of MINIX in the DISTRIX project. The fact that the two server processes and all the device drivers could, in theory, be relocated to remote processors with relative ease was a major reason for choosing MINIX.

A.1.2 Disadvantages of MINIX

Unfortunately, designing MINIX as a small operating system for small processors has had disadvantages. The cost of modular design has primarily been in performance.

One of these costs is that although MINIX is able to support multiple processes, the support for multiple, *concurrent* users is not built in. No support for multiple terminals exists in the early versions of MINIX, but device drivers for serial terminals have been written. When we first tested our own serial device driver, we were interested to notice that although the terminals could operate independently in most respects, signals were a problem area. Whenever a user at one terminal caused an interrupt (by pressing the DEL key), both the process that he was interrupting and the processes of the other terminal received a signal. This was caused by the lack of *process group* manipulation in MINIX, and more information on how this was overcome is given in section 5.4.3.

At a less obvious level, the file server is only capable of servicing one request at a time, and generally such a request is serviced to completion before accepting the next request for work. Consequently, multiple requests to a device such as a disk may not be queued in order to perform optimizations on disk head movement. The current memory manager does not offer support for virtual memory. There is presently another project in progress, attempting to parallelize the file server, taking advantage of the concurrency facilities of the transputer.

A.2 The MINIX file server

The file server (FS) is designed to provide the user with simple, efficient access to files (both conventional files and devices). The term *files* may be a little misleading when considered out of context. Files in any UNIX environment refer not only to disk files, but also directories (collections of files), terminals, pipes (special files allowing separate processes to communicate with one another) and raw devices (such as floppy diskette drives or tape streamers).

The directory concept allows the user, at their discretion, to group files logically. Taking this further, groups of directories may also be collected together in a hierarchy, hence the UNIX *hierarchical structure*. File names need only be unique within a particular directory. Each file (ordinary file, directory or device) has one or more *'i-nodes'* that uniquely identify the file.

The FS must also provide a protection mechanism. All UNIX file servers provide access control as follows:

- files belong to one *owner* and belong in one *group*;
- three distinct classes of users are discernible:
  - the file owner,
  - members of the group to which the file belongs and
  - all other users;
- access within each of the three classes can be restricted to:
  - reading,
  - writing and
  - executing the file.

The actual locations of the files on the medium are recorded by the FS and it is its function to store and retrieve the data on behalf of the user. As files may be stored on different media (and indeed these devices are themselves files) the FS records the position of the files as a *logical block number*. It is the responsibility of the individual device drivers to translate these block numbers into addresses on the specific device.

A.3 Structure and function of MINIX device drivers

The devices in UNIX systems fall into two broad categories, *block* and *character* devices. Block devices include media such as floppy diskettes, fixed disks and magnetic tapes. Character devices include terminals and printers. Each driver can usually handle several instances of a particular device. Thus, the floppy diskette device driver is responsible for all the floppy diskette drives, but not necessarily for fixed disks.

MINIX device drivers are each discrete processes, interacting with the servers only by means of message passing. The processes are all similar in design, accepting a request for work, carrying it out and then replying to the source.

Within each device driver there exist two portions, the device independent portion providing a uniform interface to the server and the device dependent portion operating at the lower level of the actual device.

The lower level operations often rely on interrupt mechanisms to signal the driver in the event of a change of state of the device.

System Call	Explanation
acct	turn accounting on or off
execn	present only in MINIX, not in Version 7
indir	indirect system call
lock	lock a process in primary memory
mpx	create and manipulate multiplexed files
mpxcall	multiplexor and channel interface
nice	set program priority
phys	allows a process to access physical addresses
pkon, pkoff	establish packet protocol
profil	execution time profile
ptrace	process trace

Table A.1: *Version 7 system calls not present in* MINIX *[Bell83]*

A.4 Comparison to UNIX — System calls and commands

The X/OPEN specification Portability Guide [XOPE87] is used as a yardstick by which to measure DISTRIX's compatibility to other UNIX systems. MINIX is based on UNIX Version 7, but the X/OPEN specifications are based on System V. The changes between each of the systems is presented next.

Tanenbaum states that MINIX has '*the same system calls as Version 7* UNIX *(except for the omission of a small number of unimportant ones).* ' [Tane87, p xiv]. Table A.1 lists the Version 7 system calls omitted from MINIX. Table A.2 shows the additions subsequent to Version 7 in the later System V release of UNIX as put forward by the X/OPEN Portability Guide [XOPE87].

The remaining calls that were retained between Version 7 and System V essentially stayed the same, but calls such as *open* were modified to do away with the need for the *creat* system call by incorporating a creation field in the list of status flag bits.

MINIX contains some 50 *commands* (utility programs provided with the system that run as user processes) as opposed to the 143 listed in the X/OPEN guide. This guide appears to be far from complete or exhaustive, as the original Version 7 programmer's manual [Bell83] lists 153 commands.

System Call	Explanation
execlp *execvp*	execute the file pointed to by *file*
_exit	exit without cleaning up
fcntl	file control
indir	removed in System V
lock	removed in System V
mpx	removed in System V
mpxcall	removed in System V
phys	removed in System V
pkon, pkoff	removed in System V
plock	lock a process, text, or data in memory
setpgrp	set process group ID
ulimit	get and set user limits
uname	get name of current system
ustat	get file system statistics

Table A.2: *Differences between System V and Version 7 system calls [XOPE87]*

A.5 Conclusions

MINIX is an academic tool that has proved to be useful in real-world spheres, even though DISTRIX is presently still an academic exercise. The highly modular design and structure of discrete, communicating processes facilitate the relocation of parts of the system to remote processors.

MINIX is not designed for large applications, and its performance in a larger (possibly multi-user) environment is unlikely to be optimal. Lack of virtual memory also restricts its usefulness for large applications. The advantages of modularity are, however, expected to outweigh this disadvantage.

The MINIX file server provides all services at the system call level. It presents a hierarchical directory structure. Access to files may be restricted by the file owner.

The file server provides an interface to the attached devices by means of device drivers that present a uniform view of devices. The device drivers are structured in a similar fashion to other servers and only communicate by message passing.

There are thirteen system calls from the X/OPEN specification for System V absent in

MINIX. The set of commands is far less complete.

Despite the latter disadvantages, MINIX proves to be a good choice for the base of a multiprocessor operating system by virtue of its modular design and built-in communication mechanisms.

Appendix B

The Inmos transputer – An introduction

Although many articles have been written about the transputer, this thesis would be incomplete without its own description of some of the key features of the transputer. This Appendix supplies this description for the lay-person and acts as a refresher and reference source to those already familiar with the processor. As, I am essentially a software person interested in hardware, the descriptions are not likely to stand up to the rigours of an engineer, but should serve to lay some conceptual groundwork. Where relevant, throughput figures are presented. They are generally rounded for convenience and their original sources should be consulted if very accurate data are required.

The history and concepts behind the transputer are outlined before detailing the architecture and the microcoded features such as process scheduling and communications.

B.1 History

The transputer is designed as a single-chip computer. It is intended to be used with other transputers, each processor being assigned a task to complete. The tasks may run in parallel and may communicate with one another by means of high speed transputer links. Thus, the transputer can be used as a stand-alone processor, or several may be combined to form a multiprocessor system.

In most cases, the designers of a processor do not design their product with any regard for the languages eventually to be implemented on it. This is not meant to imply any

indifference on the part of the designers, but rather to point out what has become the norm. The team at Inmos in Great Britain worked somewhat differently, designing the transputer and its host language Occam concurrently [Tayl86]. The transputer supports concurrency and both interprocess and interprocessor communication at the machine instruction level.

Occam takes its name from a 14th century philosopher, William of Occam. He is credited with a statement now known as Occam's Razor: *'Entia non sunt multiplicanda praeter necessitatem'*[1] which dictates simplicity over complexity. Based on the CSP notation [Hoar85], Occam is the natural language for the transputer. The transputer supports channels and process concurrency as machine language level primitives and the code can be executed directly by the transputer hardware. The channels are extended to include the hardware links mentioned earlier, but may also be used between processes on the same processor. This allows a parallel program to be tested on a single processor before separating the parallel components and distributing them to their own processors.

One of the primary design goals [Barr83] was to ensure that although Occam is the preferred language for the transputer, its instruction set should also be suitable for the compilation of other high-level languages such as C. In general, these languages support the transputer's features by means of libraries.

B.2 Architecture

The transputer is available as either a 16-bit or 32-bit processor. The 32-bit varieties have been used more commonly and no further mention will be made of 16-bit transputers.

The 32-bit transputer is currently available in two forms, namely the T414 or T425 integer processors, and the T800 floating point processor. Both are built with small (2KB–4KB) Static Random Access Memory (SRAM) units incorporated on-chip. This RAM may be accessed in one cycle (50 nanoseconds at 20MHz) with off-chip Dynamic RAM of up to 4 Gigabytes available at lower speeds, depending on the RAM used. The transputer has the ability to manipulate processes and incorporates a microcoded scheduler to perform the task switching. Each processor also features 4 bidirectional hardware links. These links generate serial signals, compatible with other transputers and operate at speeds of between 5 and 20 Megabits per second [Inmo88].

[1] Roughly translates to *'entities should not be multiplied beyond necessity'*

The instruction set is similar to that of a Reduced Instruction Set Computer (RISC). The 16 *direct* (most common) instructions can be encoded using only one byte. This is divided into 4 bits identifying the function code and 4 bits of data. This limits the size of the operand on which the instruction may operate. A PREFIX instruction may be used to operate on larger operands. The remaining *indirect* instructions operate on values present on the stack and are not coded with the operand.

The transputer differs significantly from a RISC architecture in three important areas [Inmo87]. Firstly, only about 30% of the instructions that are common to both the T414 and T800, complete in one processor cycle. Secondly, the transputer has over 100 instructions, somewhat contrary to the implication of the name, reduced instruction set. Many of these instructions are quite complex, performing more than just a simple operation. Finally, it features a *small* number of registers. The transputer has the following 6 registers:

- workspace pointer to local variables of current process;
- instruction pointer for current process;
- operand register; and
- evaluation stack registers: A, B and C.

The registers A, B and C, are arranged and operated upon like a stack. Moving a value into register A, causes its previous value to be stored in B, and B's previous value to be stored in C. The reverse operation is also possible.

B.3 The microcoded scheduler

The transputer has one of its most important features implemented in microcode. The user is given the ability to create and destroy processes at the machine instruction level. The processes may be created at one of two priority levels.

The scheduler shares the processor amongst the processes in a round-robin fashion. Each process must supply a workspace for local variables. A process may execute until one of three events occur:

1. The process attempts to communicate (using a channel or hardware link) and is unable to because the process at the other end of the transmission is not yet ready.

2. The process is suspended awaiting completion of a time delay.

3. The time quantum allocated for the process has expired.

The scheduler ensures the completion of expression evaluation. This is achieved by only descheduling (suspending) the current process whilst executing one of 12 instructions, known as descheduling points [Inmo87]. By their nature (the instructions include channel operations, jumps, etc.), these instructions are not used in expression evaluations.

Low priority processes are subject to all three conditions and are given a time quantum of approximately 1 millisecond and are then suspended by the scheduler at the next descheduling point. Naturally they may be descheduled prior to that for one of the other reasons, such as blocking on a channel operation.

High priority processes are not affected by time quanta. They are only descheduled for reasons 1 and 2 listed earlier.

Furthermore, in choosing the next process from the process queue, the scheduler always picks high priority processes first, executing any low priority processes only when no high priority processes are ready.

The time needed by the scheduler to switch between processes is less than 1 microsecond ($1\mu s$).

A clock is available to each process class. For high priority processes the clock is calibrated in ticks of $1\mu s$. A similar clock for low priority processes is calibrated in ticks of $64\mu s$. These clocks may be accessed in a similar fashion to channels (discussed next), suspending a process until a given time has passed. The value of the clock may also be read as a 32-bit value.

B.4 Channels and links

The transputer provides machine instructions that allow processes to communicate with one another by means of a *channel*. The processes make use of an agreed upon address (the channel pointer) through which to communicate. The first process to reach a communication instruction (say OUT, to send a message) suspends until the other process reaches the corresponding instruction (in this case IN, to receive a message), the transfer then takes place. This is an important synchronization technique used by concurrent processes.

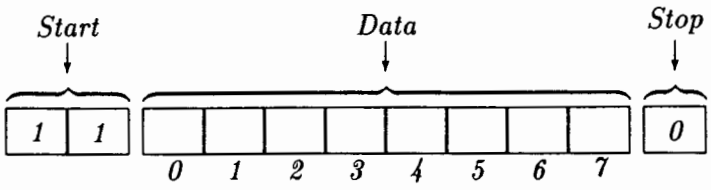

Figure B.1: *Transputer message format — Bit level*

When the communicating processes are not resident on the same transputer, they may make use of special channels called *links*. These links are mapped to specified addresses in the transputer's memory space and are accessed in exactly the same way as a conventional channel.

From a hardware perspective, the link allows a byte or sequence of bytes to be transmitted along a wire to a link on another transputer. The link sends data at speeds ranging between 5 and 20 Megabits per second (Mbps). The links can all operate in parallel, using direct memory accessing to read or write the transputer memory. This also frees the processor to continue executing other tasks requiring no link I/O.

In their 1987 paper [Home87], Homewood and others describe the exact mechanisms of the Inmos link. In particular, they highlight the differences between the T414 and T800 transputers. During the transfer of a message, the protocol of the sending transputer requires an acknowledgement from the receiver before sending the next byte. In the case of the T414 acting as a receiver, the processor only emits the acknowledgement after the entire packet (consisting of two start bits, eight data bits and a stop bit, see Figure B.1) has been buffered. This leads to an effective throughput of 0.8 MByte/second at a link speed of 20 Mbps.

To improve throughput, the T800 and newer T425, use an overlapping protocol. By using extra buffers, they can emit the acknowledgement as soon as they have detected a data packet. The throughput rises to 1.8 MByte/second. As the links are bidirectional, acknowledge packets may be interspersed with data packets travelling in the reverse direction, allowing a better throughput (2.4 MByte/second for bidirectional traffic obeying the overlapping protocol).

Appendix C

Hardware configuration of DistriX

This Appendix describes the hardware configuration of the DISTRIX system. The concepts and modular structure were introduced and motivated in Chapter 3 and it should be read first, to give the reader an understanding of the structure. In particular, this Appendix presents the transputer products used in DISTRIX. They were built by the Institute for Electronics at the University of Stellenbosch and include a board compatible with, but with more features than, the original Inmos B004 transputer motherboard. They also developed a number of computing modules, in particular, the transputer exchange processor.

The components used in DISTRIX are presented before explaining the interconnection mechanism. The booting procedure is explained together with a short explanation of how memory size may be determined by the process after booting has completed.

C.1 Components

The transputer, although a complete computer on one chip, is not very useful on its own. The links are generally its only form of communication with the outside world, but cannot be summarily attached to a terminal to accept work.

Furthermore, the transputer does not usually operate with a resident operating system, but rather a process is downloaded by another operating system. The program is then executed with or without assistance from the host operating system, and the result returned to the I/O server on the host.

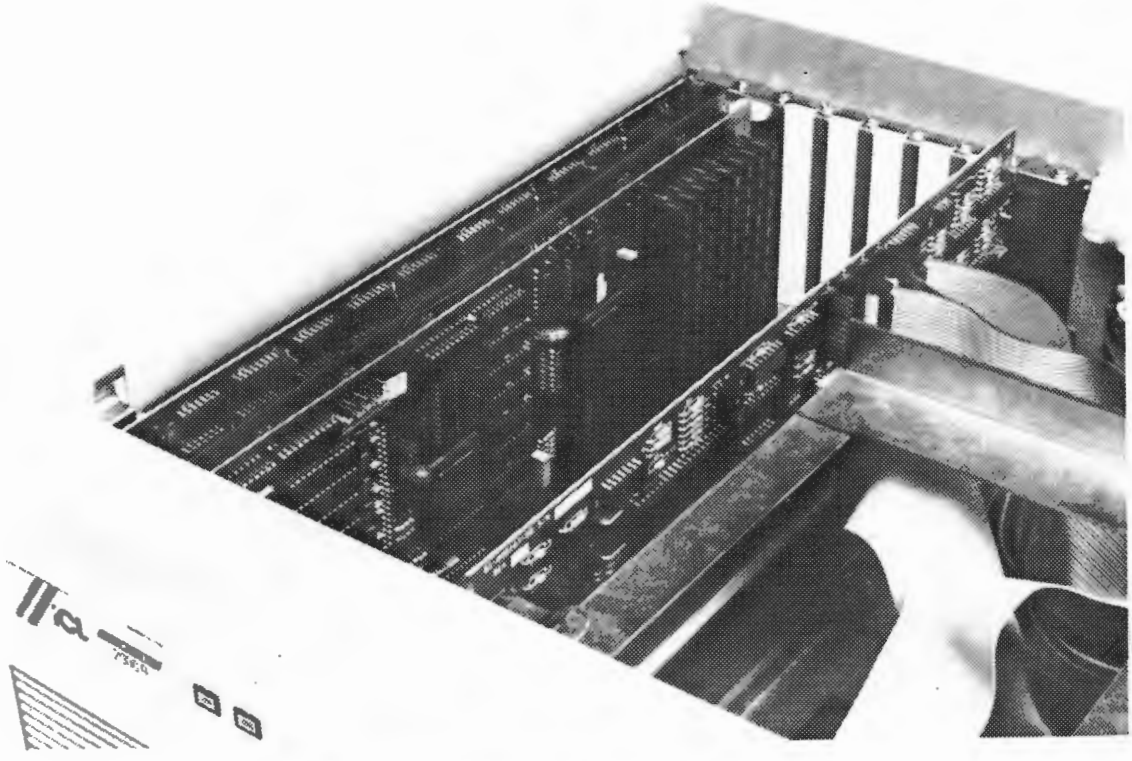

Figure C.1: *Transputer card inserted in a PC/AT*

For the DISTRIX prototype, the host environment is MS-DOS running on PC/AT compatible microcomputers. The interface is in the form of a board that may be connected to the PC via the bus edge connectors present in all PC compatible microcomputers (see Figure C.1). These boards, known as *transputer motherboards* contain all the circuitry necessary to interface with a PC bus, including a C012 link adaptor [Inmo88], bus driver chips, status LED's (light emitting diodes) for each *station* and connectors for remote link interfacing. Transputers are collected on *daughterboards*. Each such board contains its own local memory for the transputer and a 40-pin connector to the transputer motherboard. Presently 8 stations are present on each motherboard (see Figure C.2). A daughterboard may occupy from 1 to 4 stations, depending on its physical size. Figure C.3 shows a motherboard with one daughterboard occupying 4 stations. One station may support one or two transputers. Thus for small daughterboards, containing two transputers, one motherboard may support 16 transputers.

The status and transfer registers on the motherboard's link adaptor are accessible by the PC, allowing the PC to download and upload data from a transputer on one of the

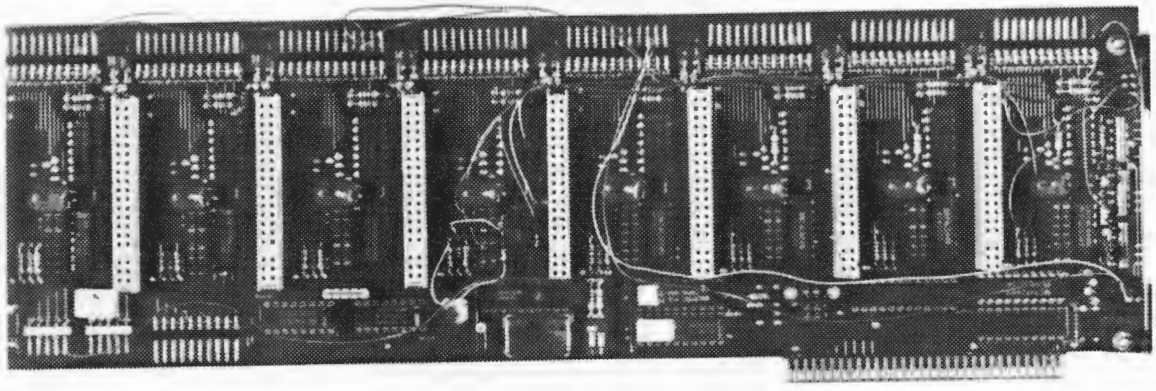

Figure C.2: *Unpopulated transputer motherboard*

daughterboards.

A major concern of the designers was that of reconfigurability. It is possible to configure these boards in many ways by means of *'wire-wrapping'*. This allows connections to be made between points on the board that have been deliberately left unconnected. When the eventual hardware configuration has been decided upon, a new board may be designed, with the final connections permanently applied during the board's manufacture. The visual aspect of wire-wrapping is seldom æsthetically pleasing, albeit colourful, but the reward of instant changes to hardware is worth the mess.

DISTRIX makes use of 3 types of daughterboards. The smallest (TR2) each occupy 2 stations, and contain a transputer and 256 KB of RAM. The transputer exchange processor (TR8 – see section 4.2.1) occupies 4 stations and houses a transputer, 256 KB of RAM and 8 C012 link adaptor chips. The TR6 is a 4-station module with one transputer and 2 MB of RAM.

The daughterboards are placed on the motherboards which are then each inserted into one of the free bus expansion slots in the PC. Using short cables, the transputers on each

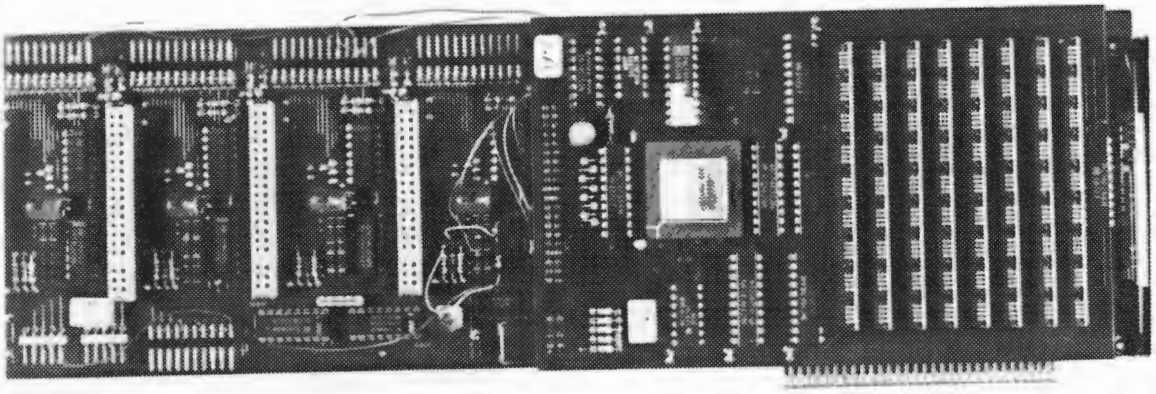

Figure C.3: *Transputer motherboard with one daughterboard*

daughterboard are connected to the exchange.

C.2 Transputer interconnection

The physical connection between transputers makes use of the link hardware. Each of the 4 links require two wires, the output of one link serving as the input for another transputer.

The connection is not limited to transputers on the same motherboard, or even the same PC. For the wire connecting links, Inmos specifies a maximum length of 300mm (approximately 12 inches). In reality, one can operate at distances in excess of 3m in an electrically quiet environment. For links extending beyond the shielded environment of the PC's case, a third wire is wrapped around the two signal wires and connected to the ground pin of the motherboards at either end of the connection. This shields the signal wires from stray electromagnetic interferences. Over the longer cables (3m), we did experience trouble when switching equipment such as fluorescent lighting on and off, despite the shielding wire.

The electrical connection is made by means of a miniature connector. The connectors

slide over access pins arranged along the top of each motherboard.

The current DISTRIX prototype (as displayed in Figure 3.1 on page 19) consists of three transputer motherboards, housed in two PC/AT computers. The first PC contains all the processors on 2 motherboards (the DISTRIX motherboards):

- Board 1:
  - the filer server (TR6-T800)
  - the exchange (TR8-T414)
- Board 2:
  - the central kernel (TR2-T414)
  - 2 mini-kernel processors
    - primary user processor (TR6-T414)
    - secondary user processor (TR2-T414)

This PC contains the operating system code in the MS-DOS partition on the fixed disk. At system boot time (described next), the code is downloaded to the various modules. The mass storage controller (MSC) emulator code is then executed, and becomes ready to process all disk requests from DISTRIX.

The second PC (used as a terminal interface module (TIM) emulator) contains an empty motherboard (the TIM motherboard), only making use of its link adaptor to accept data from the remote DISTRIX processors. The DISTRIX file server is connected to this motherboard via a standard Inmos link. The PC, which executes the TIM code, sends and receives messages through the bus.

On the DISTRIX motherboards, each of the processors are connected to the exchange processor. The file server processor is connected directly to the PC bus (accessed by the MSC) and (via a longer cable) to the TIM motherboard.

This is the current configuration for a two-user system. More user processors may be added to additional motherboards in the future, and the system has been run with 3 user processors. The initial configuration and connection of the system, during testing stages, was considerably different and a description of the early system is given in Appendix D.

C.3 Booting procedure

The development environment, used to create the components of DISTRIX, was MS-DOS. After careful evaluation of several compilers, the choice fell on the Logical Systems Transputer Toolset. The toolset consists of:

- a preprocessor,
- a C compiler targeted for the transputer,
- an assembler,
- a linker,
- a library management program, and
- various downloading programs and MS-DOS interface drivers.

In addition, a library of system calls, subroutines and concurrency control functions is included. All of these programs and libraries were available in source form.

It is particularly the downloaders on which we will now focus our attention. As mentioned in section C.1, the transputer has no resident operating system. All programs that execute on the transputer are downloaded, execute, possibly interact with the PC or peripherals, and then terminate.

The process of downloading is performed by one of two programs, `ld-b004` or `ld-net`. The former (deriving its name from the Inmos B004 transputer motherboard) is used to download a single program to a transputer. The latter is able to download many programs to a network of transputers.

The interaction with the PC is possible by the inclusion of a library and host I/O driver. The driver is executed on the PC as soon as the downloader has placed the transputer programs into the transputer's memory. The library presents the process on the transputer with a set of standard operating system calls (such as for manipulating files or the screen). The calls are made locally on the transputer, but the library sends a message to the I/O driver, where the work is actually carried out. Thus the transputer process is given an interface to MS-DOS.

For the DISTRIX prototype, the I/O driver is not used. Instead, the MSC executes on the PC, giving the file server transputer an interface to a high level disk device.

To download the DISTRIX operating system code, the `ld-net` program is used. The connections between the processors is described in a network information file.

C.4 Determining memory size

The mini-kernel processors in the processor pool are responsible for the user processes. Part of their function is to allocate the user memory when requested by a system call such as *brk*. In order to allocate memory (and to fail if too much is requested) the mini-kernel must have knowledge of the size of physical memory. It can determine its own code and data size by means of pointers to the heap. The top of heap is the highest address used by the mini-kernel program, all memory from the top of heap to the end of physical memory is thus available.

As described in section C.2, the prototype makes use of more than one type of daughterboard for the user processors. In particular, both 2MB and 256KB boards are used.

Naturally, it is possible to determine beforehand what memory configurations exist and to create a mini-kernel image for each possible occurrence. This is not considered as viable, and an automatic scheme for dynamic memory calculation was devised.

The simplest way is to search from the top of heap (a known address) until an error is generated by reading past the end of physical memory. However, in the transputer, the memory map is arranged cyclically and the first test program failed. When a reference is made that is above the physical top of memory, the memory mapping references from the bottom of memory. With this knowledge, the next version of the program inserted a unique string (of alphanumeric characters) into its processor's memory. This string is searched for and its location recorded. The program then searches for it again, starting at a location just above the first reference and scans memory until the string is located a second time. This second reference returns an address that is beyond the top of physical memory. The exact size of memory is calculated as the difference of the two references.

This method worked well, but the standard *strcmp* routine proved to be very slow, requiring some 15 seconds to determine the size of a 2MB transputer. Clearly the algorithm needed some work. The current memory counting routine makes use of the Boyer-Moore [Boye77] algorithm with a C function based on the algorithmic code presented by Smit [Smit82]. This algorithm is particularly efficient for long patterns (more than 4 char-

acters) not occurring early in the search area. This is certainly the case for the DISTRIX memory counter, the memory areas are at least 256K in length, currently to a maximum of 2048K. The pattern searched for is in excess of 64 characters allowing the algorithm to step through memory in steps of up to 64 bytes between comparisons.

The memory counting routine has been incorporated into the mini-kernel code. It is executed at system boot time and the result is recorded for use by the memory allocation routines.

Appendix D

Development history of DistriX

Based on the MINIX operating system, DISTRIX is a multiprocessor, distributed UNIX operating system for a network of transputers. As described in Chapter 3, DISTRIX consists of a number of processors, each responsible for a portion of the work in the complete system.

In the prototype, the components consist of transputers and PC/AT compatible computers. As explained in section C.2, the transputer requires a host processor and operating system. One host may support multiple transputers, but for development purposes this may be undesirable. In this section the initial machine configuration is described to give a perspective on how the system grew.

The major problem discovered during implementation was the lack of debuggers. This is outlined, together with the problems caused by events happening in a truly concurrent fashion.

D.1 Early configurations

The first versions of DISTRIX were, in a sense, more distributed than the final prototype. Each module displayed in Figure3.1 on page 19 was housed in a separate PC compatible.

Prior to the first actual DISTRIX implementation, this was only a logical distribution, as the file server and kernel both resided as separate processes on the same processor (they were, in fact, the original MINIX operating system modules). The user processes communicated with their respective agents by means of a device driver running under MINIX. This device driver and the hardware supporting it are more fully explained in

section D.2.

This configuration was retained until the completion of tests of the mini-kernel. Thereafter, the file server and kernel were relocated to transputers. The satellite processors, each executing a mini-kernel and the user processes, made system calls that were routed to the two server processors.

Whilst testing the FS, the MINIX processor was retained. This was necessary as none of the device drivers had yet been ported to the transputer, and the FS still relied on their presence. The device drivers were accessed by means of remote procedure calls, with an agent for each driver running as a user level process on the PC/AT executing MINIX.

As the device drivers were moved onto the FS processor, the need for the MINIX PC was removed. The addition of more user processors and the arrival of the exchange hardware prompted the incorporation of the exchange. The final stage required moving all the processors into the same PC/AT and attaching the relevant devices or device emulators via link adaptors.

During testing, the processors were kept in separate PC's for reasons of debugging. Only once all debugging had been completed could the processors be brought together.

D.1.1 Development environment

The development environment for the DISTRIX project was initially MINIX. Several other C compilers were examined, including two from Inmos, the company that builds the transputer. The major drawbacks of most compilers examined were:

- no support for the *fork* system call (see section 3.1),
- no source code for the compiler, and
- no source code for the libraries.

Source code of a compiler is not generally made available. With the situation in which we found ourselves, the availability of source of the compiler was the only means by which we would be able to add the support for the *fork* call without developing a new compiler. Having access to library source is a more common occurrence, and was regarded as a very desirable property.

The choice finally fell on the Logical Systems Toolset. The toolset operates under MS-DOS and provides a preprocessor, cross-compiler, linker and programs to download the

resulting executables to networks of transputers. The sources of both the entire toolset (including the compiler) and the libraries were supplied.

The libraries contain many of the standard UNIX library routines, but for the purposes of the DISTRIX file server, most of the Logical Systems C (LSC) library routines were discarded. Certainly, all the system calls made by agents on the FS processor, use modified versions of the original MINIX library, recompiled using LSC.

The most important LSC library routines used in the FS are those introduced by Jeffrey Mock [Mock88]. The routines provide functions that interface to the hardware level channel I/O, process creation and destruction and timing features of the transputer. Semaphores are also provided to control the simultaneous access of data structures or channels.

Unfortunately no debugger was available. Thus the only way to ferret out errors in the system was to use selective tracing statements in the code. These statements produced screen messages displaying values of variables and other state information. This lack of good debugging tools presented one of the most serious problems during development.

The original MINIX file server consists of some 5 000 lines of C code. After the addition of code to support the communication layers and the debugging code (much of which remains embedded but inoperative in the code), the FS has grown to about 12 000 lines of C code.

D.2 Interface with MINIX

The Inmos link adaptor allows a transputer to communicate with other computers that do not support the Inmos link interface. The link adaptor is described with reference to its usefulness in the early DISTRIX development environment. A special device driver allowing MINIX processes to communicate with the transputer is described.

D.2.1 The Inmos link adaptor

In order for agent processes to communicate with user processes executing on a remote transputer, an Inmos link adaptor (IMSC012) [Inmo88] is used. This processor allows for the conversion of Inmos serial link messages to a parallel stream, suitable for an 8-bit PC bus (and vice-versa). The link adaptor (introduced in the hardware description of

the exchange in section 4.2.1) was used to allow MINIX processes to exchange data with a transputer. To simplify the interaction, a device driver was written for MINIX.

Using the status registers and interrupt facilities of the link adaptor, the device driver is able to send a given block of data out on a link to a transputer or accept a block of data from a transputer into a buffer set aside by a MINIX process.

D.2.2 The device driver

MINIX device drivers are designed as server processes. Each runs as an independent process. Their main body performs the task of accepting requests for work from (remote) client processes. The requests are made via the standard MINIX message passing mechanism using the blocking send and receive routines.

Each driver is further broken down into two portions. The upper half is responsible for the interface with the user and is, in theory, device independent. The lower half is concerned with the actual interface to a specific device in the general class of devices handled by the driver as a whole. It performs all the critical tasks related to interrupts and/or polling of the device.

The MINIX Link Adaptor Device Driver (LADD) was written in this way. At boot time, the low level portion of the driver clears the status registers and enables interrupts. The upper half then awaits one of three messages:

- send buffer to transputer,
- receive from transputer into buffer, or
- service an interrupt received from the link adaptor.

In general, this method proved to be too inefficient. While interrupts are enabled, each byte generates an interrupt that in turn results in a MINIX message sent from the lower half of the driver to the waiting upper half. As a standard downloadable file was of the order of 10–60 KBytes in length, the number of interrupts led to a transfer time of between 30 seconds and 3 minutes, clearly unacceptable.

A solution was found in allowing the driver to use a mixture of interrupts and polling to effect transfers efficiently. As the messages conform to a protocol, the driver is easily able to establish the size of the message. By the nature of the transmission software on

either side of the link, once a message has started crossing the link, the entire message is sent at the maximum rate at which the link can accept it.

Taking advantage of this pre-knowledge, the LADD turns off the interrupts as soon as the first interrupt has been received. Thereafter, the LADD sends or receives the remainder of the message without the assistance (hindrance) of interrupts. For example, while sending from the PC/AT to the transputer, the LADD accepts the message from the user process. The message contains the length of the message, its destination address (this was included for future expansion when the PC/AT was no longer used as a server), and the address in memory of the data portion. The LADD places the first byte into the transmission register and then awaits the interrupt indicating that the byte has been accepted by the link hardware. The LADD then enters a send–poll loop until all the bytes have been sent and the transmission status register is cleared. Interrupts are re-enabled and the driver reverts to an idle state.

This process consumed all the CPU time and did not allow user process to execute during transmission. However, it was justified for the following reasons. The system was only configured in this way during early development and testing stages and there was only one user active. Secondly, performance of the scheme using interrupts was not much different as the link was able to clear its buffer almost immediately upon receiving the byte, causing an interrupt to be generated. Such interrupts take precedence over user processes. Lastly, and related to the previous point, the new transfer times were in the order of 10 seconds for a 60 KByte file, an improvement of some 2 000%. The changes to the device driver were simplified by the modular design of device drivers in MINIX, and the changes were completed in a morning.

One additional change was made to the LADD due to a feature of the interrupt structure of the hardware used. The interrupts generated for incoming and outgoing data were split and one driver was used for each direction. The split did not alter the performance of the LADD. The last version of the LADD makes use of the PC interrupts, IRQ3 and IRQ5. The latter, IRQ5, is not essential in an AT, as it is reserved for the second parallel printer port. In a PC, however, this is the interrupt from the fixed disk controller. Thus, the current hardware configuration is limited to operation on a PC/AT.

D.3 Debugging

A major hurdle in our development of DISTRIX was the lack of good debugging tools. Even with access to the full source of most of the development software, errors in our own code were difficult to trace.

The method finally adopted involved acquiring additional hardware, in particular more PC compatible machines. As mentioned earlier, a transputer motherboard (described in section C.1) was placed into each machine. Each motherboard contained only one transputer. Consequently, it became possible to allow each processor to display debugging information about the internal state of the program on the display screen of each PC. Although extremely laborious (the debugging statements are inserted manually), this method became the primary way to trace errors in the system.

The features of a proper debugging tool for distributed systems (such as Pilgrim [Coop88]) should include support for both development and target environments. The debugger should be entirely transparent, so as not to cause the error being traced to disappear just by the debugger's presence. The debugger should be interactive and allow the user to examine the state of a process at any time. Furthermore, the debugger should provide these features at a source level, allowing the user performing the actual debugging to observe the progress in terms of the original source code.

For many reasons, the above features are often unavailable. Cooper explains that the obstacles most often in the way of a debugging environment are related to the compiler support for debuggers. Such support includes source-object mapping and other run-time debugging information. This information is often not included into the object file by the compiler, making the task of source-level debugging impossible. This is certainly the case for the present Logical Systems C Compiler used by DISTRIX.

D.4 True concurrency

The introduction of true concurrency provided its fair share of headaches during development. Whereas events *appear* to happen simultaneously on a uniprocessor, multiprocessor systems must take account of events *actually* happening simultaneously on multiple processors.

This proved to be particularly problematic when dealing with unpredictable events,

such as user interrupts. The DISTRIX file server is based on MINIXand obeys a similar technique for the handling of interrupts. In particular, when a user process is interrupted during *read* from a terminal, the FS informs the central kernel of the interrupt and then terminates the *read* with an error. However, it is essential that the interrupt reaches the user process before the reply to the *read*. In MINIX this is always the case because there is only one processor; the kernel receives the request to interrupt from the FS and then performs the interrupt immediately. By the time the FS is again able to run, the user process is already signalled.

With DISTRIX, this mechanism would be subject to intermittent failure, determined by network traffic and the availability of the central kernel. To overcome this problem (and other similar features of concurrency), the structure of the call-reply mechanism was adjusted. The user's view of the system remains unchanged, but the internal mechanism has been changed to take cognisance of the simultaneity of events.

These changes are akin to the required changes discussed in section 2.5.2 in which the possibility of complete rewrites of UNIX is suggested as a solution. Luckily, in DISTRIX, only a few such changes were eventually required.

Appendix E

The exchange program

This Appendix is included to provide the interested reader with an example of program code used in DISTRIX. The code listed in Figure E.1 is the C program that is downloaded to the exchange processor during the booting procedure. The library calls are part of the Logical Systems Transputer Toolset, and in particular make use of the Mock extensions [Mock88]. The header file is also part of the Toolset.

Details such as the addresses of the links are included but are not essential to the understanding of the program.

```
#include <conc.h>

#define NR_LINKS        12
#define BUF_SIZE        1050    /* Largest message possible     */
#define LSTN_WS_SIZE    2048    /* Workspace size in bytes      */

#define LINK4IN         0x80
#define LINK4OUT        0x84
#define LINK4INSTAT     0x88
#define LINK4OUTSTAT    0x8C
#define LINK4RESET      0x10

Semaphore       *out_sem[NR_LINKS];
Semaphore       prt = SEMAPHOREINIT;
Channel         *LinkInList[NR_LINKS+1],
                *LinkInStat[NR_LINKS],
                *LinkOutList[NR_LINKS],
                *LinkOutStat[NR_LINKS];
char            buffer[NR_LINKS][BUF_SIZE];
```

```
void            listen();
Process         lstn_proc[NR_LINKS];
int             lstn_ws[NR_LINKS][LSTN_WS_SIZE];

main()
{
  int           i;

  ProcToHigh();
  reset_links();

  for (i=0; i<NR_LINKS; ++i)
  {
    out_sem[i] = SemAlloc();
    ProcInit(   &lstn_proc[i],
                (int (*) ()) listen,
                lstn_ws[i],
                sizeof(lstn_ws[i]),
                1,
                i );
    ProcRunHigh(&lstn_proc[i]);
  }
  ProcToLow();
  ProcStop();

}

void listen(p, num)
Process *p;
int num;
{
  int           dest, length;

  for (;;)
  {
    buffer[num][0] = PseuChanInChar(num);
    buffer[num][1] = PseuChanInChar(num);
    length = (int) (buffer[num][0] + (buffer[num][1] << 8));
    length &= 0xFFFF;
    PseuChanIn(num, &buffer[num][2], length);
    dest = (int) (buffer[num][2] + (buffer[num][3] << 8));
    dest &= 0xFFFF;
    SemP(*out_sem[dest]);       /* Acquire exclusive use of this link.  */
    PseuChanOut(dest, &buffer[num][0], length+2);
    SemV(*out_sem[dest]);       /* Release the rights on this link.     */
  }
}
```

```
void reset_links()
{
  int   i;

  ChanReset(LINK0IN);
  ChanReset(LINK1IN);
  ChanReset(LINK2IN);
  ChanReset(LINK3IN);
  ChanReset(LINK0OUT);
  ChanReset(LINK1OUT);
  ChanReset(LINK2OUT);
  ChanReset(LINK3OUT);

  LinkInList[0]=LINK0IN;
  LinkInList[1]=LINK1IN;
  LinkInList[2]=LINK2IN;
  LinkInList[3]=LINK3IN;
  LinkOutList[0]=LINK0OUT;
  LinkOutList[1]=LINK1OUT;
  LinkOutList[2]=LINK2OUT;
  LinkOutList[3]=LINK3OUT;

  /* Reset the Link Adaptors for the remaining 8 pseudo-links   */

  for(i=4; i<NR_LINKS; ++i)
  {
    LinkInList[i]  = (Channel *)  (int) (LINK4IN      + 16*(i-4));
    LinkInStat[i]  = (Channel *)  (int) (LINK4INSTAT  + 16*(i-4));
    LinkOutList[i] = (Channel *)  (int) (LINK4OUT     + 16*(i-4));
    LinkOutStat[i] = (Channel *)  (int) (LINK4OUTSTAT + 16*(i-4));
    la_reset(LINK4RESET + (4*(i-4)));
  }
  LinkInList[NR_LINKS]=0;
}

void la_reset(lr)
int *lr;
{
  *lr=0;
  ProcWait(2);
  *lr=1;
  ProcWait(2);
  *lr=0;
  ProcWait(2);
}
```

```
void PseuChanOutChar(chan, ch)
int chan;
char ch;
{
  int   outval;

  if (chan >= 0 && chan <= 3)
    ChanOutChar(LinkOutList[chan], ch);
  else
  {
    while (!(*LinkOutStat[chan] & 0x01)) { ProcReschedule(); }
    outval = (int) ch;
    outval &= 0xFF;
    *LinkOutList[chan] = outval;
  }
}


void PseuChanOutInt(chan, number)
int chan, number;
{
  int   outval, i;
  char  ch;

  if (chan >= 0 && chan <= 3)
    ChanOutInt(LinkOutList[chan], number);
  else
  {
    for (i=0; i<4; ++i)
    {
      while (!(*LinkOutStat[chan] & 0x01)) { ProcReschedule(); }
      outval = (int) (number >> (i*8));
      outval &= 0xFF;
      *LinkOutList[chan] = outval;
    }
  }
}
```

```
void PseuChanOut(chan, buf, bytes)
int chan, bytes;
char *buf;
{
  int   outval, i;
  char  ch;

  if (chan >= 0 && chan <= 3)
    ChanOut(LinkOutList[chan], buf, bytes);
  else
  {
    for (i=0; i<bytes; ++i)
    {
      while (!(*LinkOutStat[chan] & 0x01)) { ProcReschedule(); }
      *LinkOutList[chan] = (int) (buf[i] & 0xFF);
    }
  }
}

char PseuChanInChar(chan)
int chan;
{
  if (chan >= 0 && chan <= 3)
    return (ChanInChar(LinkInList[chan]));
  else
  {
    while (!(*LinkInStat[chan] & 0x01)) { ProcReschedule(); }
    return ((char) *LinkInList[chan]);
  }
}
```

```
int PseuChanInInt(chan)
int chan;
{
  int   inval, i;
  char  ch;

  if (chan >= 0 && chan <= 3)
    return (ChanInInt(LinkInList[chan]));
  else
  {
    for (i=0; i<4; ++i)
    {
      while (!(*LinkInStat[chan] & 0x01)) { ProcReschedule(); }
      ch = (char) *LinkInList[chan];
      inval = (ch << (i*8));
    }
    return (inval);
  }
}


void PseuChanIn(chan, buf, bytes)
int chan, bytes;
char *buf;
{
  int   outval, i;
  char  ch;

  if (chan >= 0 && chan <= 3)
    ChanIn(LinkInList[chan], buf, bytes);
  else
  {
    for (i=0; i<bytes; ++i)
    {
      while (!(*LinkInStat[chan] & 0x01)) { ProcReschedule(); }
      buf[i] = (char) *LinkInList[chan];
    }
  }
}
```

Figure E.1: *Exchange software — C code*

References

[Bach84] BACH, M.J., AND BUROFF, S.J. 1984. The UNIX system: Multiprocessor UNIX operating systems. *AT&T Bell Laboratories Technical Journal.* Vol 63, 8, October, pp1733–1749.

[Bach86] BACH, M.J. 1986. *The design of the* UNIX *operating system.* Prentice-Hall, Englewood Cliffs, New Jersey, USA.

[Bakk88] BAKKES, P.J., PINA, R., AND DU PLESSIS, J.J. 1988. Transputer enhancement: Virtual memory management for the transputer. In *Parallel Processing '88, Symposium Record, Part 1.* 26–28 October, Johannesburg, South Africa.

[Barr83] BARRON, I., CAVILL, P., MAY, D., AND WILSON, P. 1983. Transputer does 5 or more MIPS even when not used in parallel. *Electronics.* Number 56, November 17, pp109–115.

[Bell83] BELL LABORATORIES. 1983. UNIX *time-sharing system:* UNIX *programmer's Manual.* Holt, Rinehart and Winston, Saunders College Publishing, USA.

[Birr80] BIRRELL, A.D., AND NEEDHAM, R.M. 1980. A universal file server. *IEEE Transactions on Software Engineering.* Vol SE-6, 5, September, pp450–453.

[Birr83] BIRRELL, A.D., LEVIN, R., NEEDHAM, R.M., AND SCHROEDER, M.D. 1980. Grapevine: Two papers and a report. *XEROX PARC Report number CSL-83-12.*

[Bode84] BODENSTAB, D.E., HOUGHTON, T.F., KELLEMAN, K.A., RONKIN, G., AND, SCHAN, E.P. 1984. The UNIX system: UNIX operating system port-

ing experiences. *AT&T Bell Laboratories Technical Journal.* Vol 63, 8, October, pp1769–1790.

[Bour83] BOURNE, S.R. 1983. *The* UNIX *system.* Addison Wesley, UK.

[Boye77] BOYER, R.S., AND MOORE, J.S. 1977. A fast string search algorithm. *Communications of the ACM.* Vol 20, 10, October, pp762–772.

[Cher83] CHERITON, D.R. AND ZWANEPOEL, W. 1983. The distributed V kernel and its performance for diskless workstations. In *Proceedings of the 9th ACM Symposium on Operating Systems principles. Operating Systems Review.* Vol 17, 5, Special Issue, 10–13 October, pp129–140.

[Cher86] CHERITON, D.R., SLAVENBURG, G.A., AND BOYLE, P.D. 1986. Software-controlled caches in the VMP multiprocessor. *Computer Science Department, Stanford University.*

[Cher88] CHERITON, D.R. 1988. The V distributed system. *Communications of the ACM.* Vol 31, 3, March, pp314–333.

[Cher89] CHERITON, D.R., GOOSEN, H.A., AND BOYLE, P.D. 1989. Multi-level shared caching techniques for scalability in VMP-MC. *Communications of the ACM.* Vol 32, pp16–24.

[Comt74] COMTRE CORPORATION, ENSLOW, P.H. JR., ED. 1974. *Multiprocessors and parallel processing.* John Wiley & Sons, USA.

[Coop88] COOPER, R.C.B. 1988. *Debugging concurrent and distributed programs.* PhD Dissertation, Technical Report 128, Computing Laboratory, University of Cambridge, UK.

[Coul88] COULOURIS, G.F., AND DOLLIMORE, J. 1988. *Distributed systems: Concepts and design.* Addison Wesley, UK.

[Deit84] DEITEL, H.M. 1984. *An introduction to operating systems.* Addison Wesley, USA.

[Ensl77] ENSLOW, P.H., JR. 1977. Multiprocessor organization — A survey. *Computing Surveys* Vol 9, 1, March, pp103–129.

[Hoar85] HOARE, C.A.R. 1985. *Communicating sequential processes*. Prentice-Hall International, UK.

[Home87] HOMEWOOD, M., MAY, D., SHEPHERD, D., AND SHEPHERD, R. 1987. The IMS T800 transputer. *IEEE Micro*. Vol 7, 5, October, pp10–26.

[Inmo87] INMOS LTD. 1987. *IMS T800 transputer data sheet, ref 42-1082-00*. April, Inmos Ltd, UK.

[Inmo88] INMOS LTD. 1988. *Transputer reference manual*. Prentice-Hall International, UK.

[Inte85] INTEL CORP. 1985. *iAPX 286 Programmer's reference manual*. Intel, USA.

[Jans86] JANSSENS, M.D., ANNOT, J.K., AND VAN DE GOOR, A.J. 1986. Adapting UNIX for a multiprocessor environment. *Communications of the ACM*. Vol 29, 9, September, pp895–901.

[John78] JOHNSON, S.C., AND RITCHIE, D.M. 1978. UNIX time-sharing system: Portability of C programs and the UNIX system. *The Bell System Technical Journal*. Vol 57, 6, July–August, pp1021–2048.

[Jone89] JONES, G. 1989. Carefully scheduled selection with ALT. *Occam User Group Newsletter*. Inmos Ltd, UK.

[Joy83] JOY, B. 1983. Evolution of the UNIX operating system. *Electronics*. Vol 56, 15, July 28, p115.

[Lewi89] LEWIS, H. 1989. A critique of MINIX. *Technical Report number ITR-89-03-01*. The Institute for Applied Computer Science, University of Stellenbosch, South Africa.

[McCu89] MCCULLAGH, P.J. 1989. DISTRIX: *An implementation of* UNIX *on transputers*. Submitted as MSc Thesis, Department of Computer Science, University of Cape Town, South Africa.

[Mock88] MOCK, J. 1988. Processes, channels and semaphores (Version 2). *Logical Systems C Compiler — Users manual*. Logical Systems, Corvallis, USA.

[Morr85] MORRIS, D., THEAKER, C.J., PHILLIPS, R., AND LOVE, W.R. 1985. Virtual memory for microcomputers. *IEE Proceedings.* Vol 132, Part E, 6, November, pp323–332.

[Mula88] MULARSKI, W. 1988. *Mass Storage Controller, Technical documentation.* Version 1.2, April. Parsytec GmbH, West Germany.

[Nels81] NELSON, B.J. 1981. *Remote procedure call.* XEROX PARC Report number CSL-81-9 also published as a PhD Dissertation, Technical Report CMU-CS-81-119, Carnegie-Mellon University, USA.

[Nels88] NELSON, M.N., WELCH, B.B., AND OUSTERHOUT, J.K. 1988. Caching in the Sprite network file system. *ACM Transaction on Computer Systems.* Vol 6, 1, February, pp134–135.

[Orga72] ORGANICK, E.I. 1972. *The* MULTICS *system.* MIT Press, USA.

[Oust88] OUSTERHOUT, J.K., CHERENSON, A.R., DOUGLIS, F, NELSON, M.N., AND WELCH, B.B. 1988. The Sprite network operating system. *IEEE Computer.* Vol 21, 2, February, pp23–35.

[Pack88] PACKER, J. 1988. Simpler real-time programming with the transputer. *Inmos Technical Note 51, 72-TCH-051-00.* March, Inmos Ltd, UK.

[Pope85] POPEK, G.J., AND WALKER, B.J. 1985. *The* LOCUS *distributed system architecture.* MIT Press, USA.

[Rect80] RECTOR, R., ALEXY, G. 1980. *The 8086 book.* McGraw-Hill, USA.

[Ritc78] RITCHIE, D.M., AND THOMPSON, K. 1978. The UNIX time-sharing system. *The Bell System Technical Journal.* Vol 57, 6, July–August, pp1905–1929.

[Saty80] SATYANARAYANAN, M. 1980. *Multiprocessors: A comparative study.* Prentice-Hall, Englewood Cliffs, New Jersey, USA.

[Serl85] SERLIN, O. 1985. Parallel processing: Fact or fancy? *Datamation.* Vol 31, 23, December 1, pp93–105.

[Shep87] SHEPHERD, R. 1987. Extraordinary use of transputer links. *Inmos Technical Note 1, 72-TCH-001-00.* February, Inmos Ltd, UK.

[Smit82] SMIT, G. DE V. 1982. A comparison of three string matching algorithms. *Software Practice and Experience*. Vol 12, 1, January, pp57–66.

[Smit89] SMIT, G. DE V., HOFFMAN, P.K., AND MCCULLAGH, P.J. 1989. DISTRIX: A multiprocessor UNIX workbench. *Technical Report number CS-89-04-00*. Department of Computer Science, University of Cape Town, South Africa.

[Sten88] STENSTRÖM, P. 1988. Reducing contention in shared-memory multiprocessors. *IEEE Computer*. Vol 21, 11, November, pp26–37.

[Stur80] STURGIS, H., MITCHELL, J., AND ISRAEL, J. 1980. Issues in the design and use of a distributed file system. *Operating Systems Review*. Vol 14, 3, July, pp55–69.

[Tane87] TANENBAUM, A.S. 1987. *Operating systems: Design and implementation*. Prentice-Hall, Englewood Cliffs, New Jersey, USA.

[Tane85] TANENBAUM, A.S., AND VAN RENESSE, R. 1985. Distributed operating systems. *Computing Surveys*. Vol 17, 4, December, pp419–470.

[Tayl86] TAYLOR, R. 1986. Transputer communication link. *Microprocessors and microsystems*. Vol 10, 4, May, pp211–215.

[Walk85] WALKER, P. 1985. The transputer: A building block for parallel processing. *Byte*. Vol 10, 5, May, pp219–235.

[XOPE87] X/OPEN. 1987. X/OPEN *Portability Guide*. Elsevier Science Publishing Company, Netherlands.

[Zimm80] ZIMMERMANN, H. 1980. OSI reference model — The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*. Vol COM-28, 4, April, pp425–432.