Explicit Network Scheduling

Richard John Black

Churchill College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

December 1994

Abstract

This dissertation considers various problems associated with the scheduling and network I/O organisation found in conventional operating systems for effective support for multimedia applications which require Quality of Service.

A solution for these problems is proposed in a micro-kernel structure. The pivotal features of the proposed design are that the processing of device interrupts is performed by user-space processes which are scheduled by the system like any other, that events are used for both inter- and intra- process synchronisation, and the use of a specially developed high performance I/O buffer management system.

An evaluation of an experimental implementation is included. In addition to solving the scheduling and networking problems addressed, the prototype is shown to out-perform the Wanda system (a locally developed micro-kernel) on the same platform.

This dissertation concludes that it is possible to construct an operating system where the kernel provides only the fundamental job of fine grain sharing of the CPU between processes, and hence synchronisation between those processes. This enables processes to perform task specific optimisations; as a result system performance is enhanced, both with respect to throughput and the meeting of soft real-time guarantees.

To my parents

John and Marcella

Preface

Except where otherwise stated in the text, this dissertation is the result of my own work and is not the outcome of work done in collaboration.

This dissertation is not substantially the same as any I have submitted for a degree or diploma or any other qualification at any other university.

No part of this dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Trademarks

Alpha AXP, DECstation, TURBOchannel, Ultrix and VAX are trademarks of Digital Equipment Corporation.
Archimedes is a trademark of Acorn Computers Ltd.
ARM is a trademark of Advanced RISC Machines.
Ethernet is a trademark of the Xerox Corporation.
MIPS is a trademark of MIPS Technologies Inc.
TAXI is a trademark of Advanced Micro Devices Inc.
UNIX is a trademark of AT&T.
Windows NT is a trademark of Microsoft Corporation.
Xilinx is a trademark of Xilinx Inc.

Other trademarks which may be used are also acknowledged.

Acknowledgements

I would like to thank my supervisor, Derek McAuley, for his encouragement and advice during my time at the Computer Laboratory. I would also like to thank Roger Needham, Head of the Laboratory, for his support and for encouraging me to spend one summer getting a new perspective on research by working as an intern at DEC Systems Research Center in Palo Alto.

I am grateful for the help and friendship of the members of the Systems Research Group, who have always proved ready to engage in useful discussions. Joe Dixon, Mark Hayter, Ian Leslie and Eoin Hyden deserve a special mention.

I am indebted to Paul Barham, Mark Hayter, Eoin Hyden, Ian Leslie, Derek McAuley, and Cosmos Nicolaou who read and commented on various drafts of this dissertation. I would also like to thank Paul Barham for artistic advice, and Robin Fairbairns for typographical guidance. The world famous Trojan Room coffee machine deserves a mention too for its stimulating input over the years.

I would like to take this opportunity to congratulate Martyn Johnson and his staff on the exceptionally high quality of the systems administration at the Computer Lab., and in particular to thank him for the cheerful and indeed encouraging way he has reacted to my all-too-frequent requests to do peculiar things to his system.

This work was supported by an XNI studentship from the Department of Education for Northern Ireland.

Contents

List of Figures

List of Tables

Glossary

AAL	ATM adaptation layer
ADU	Application Data Unit
ANSA	Advanced Networked Systems Architecture
ARM	Advanced RISC Machine
ATDM	Asynchronous Time Division Multiplexing
ATM	Asynchronous Transfer Mode
AXP	DEC Alpha project logo
BISDN	Broadband Integrated Services Digital Network
BSD	Berkeley Software Distribution
CFR	Cambridge Fast Ring
CPU	Central Processing Unit
DAN	Desk Area Network
DEC	Digital Equipment Corporation
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
ECC	Error Correcting Code
EDF	Earliest Deadline First
FDL	Fairisle Data Link
FIFO	First In First Out
FIQ	Fast Interrupt Request (on ARM processors)
FPC	Fairisle Port Controller
FPGA	Field Programmable Gate Array
FRC	Free Running Counter
I/O	Input and Output

IP	Internet Protocol
IPC	Inter-process communication
JPEG	Joint Photographic Experts Group (a video compression standard)
LED	Light Emitting Diode
MIDDL	Mothy's interface definition language
MLS	Meta-Level Scheduler
MMU	Memory Management Unit
MRU	Most Recently Used
MSDR	MultiService Data Representation
MSNA	MultiService Network Architecture
MSNL	MultiService Network Layer
MSRPC	MultiService Remote Procedure Call
MSSAR	MultiService Segmentation and Reassembly
NFS	Network File System
NTSC	Nemo Trusted Supervisor Code *or* National Television Standards Committee (the American TV standard)
OSF	Open Software Foundation
PAL	Phase Alternate Line (the European TV standard)
PCB	Printed Circuit Board
PDU	Protocol Data Unit
QOS	Quality of Service
RISC	Reduced Instruction Set Computer
RMM	Receive Master Mode
ROM	Read-Only Memory
PTM	Packet Transfer Mode
RPC	Remote Procedure Call
SAS	Single Address Space
SRAM	Static Random Access Memory
SRC	Digital Equipment Corporation Systems Research Center
STM	Synchronous Transfer Mode
SVR4	Unix System Five Release Four

TAXI	Transparent Asynchronous Transmitter / Receiver Interface
TC	Timer Counter
TCP	Transport Control Protocol
TLB	Translation Lookaside Buffer
TMM	Transmit Master Mode
UDP	Unreliable Datagram Protocol
USC	Universal Stub Compiler
VCI	Virtual Circuit Identifier (also Virtual Channel Identifier)
VPI	Virtual Processor Interface
XTP	Express Transport Protocol
XUNET	Experimental University Network

Chapter 1

Introduction

Over the past decade there have been many advances in the field of computer networking ranging from improvements in fibre optics to the wide-scale interconnection of heterogeneous equipment. In particular, developments in high speed packet switching technology have created the likelihood of a universal paradigm for all types of user communication – the multi-service network.

During this time the traditional exponential growth of computing performance has continued unabated and many changes have been made in the area of operating systems to capitalise on this growth in workstation power. In the recent past there has not, however, been an equivalent rise in the performance of network systems which the market has demanded from workstations.

Recent networking technology promises not only enormously improved performance, but also Quality of Service guarantees; the ability to realise true multi-service communication to the workstation has become essential, and the deployment of this high performance network technology has vastly reduced the ratio of workstation power to network interface bandwidth.

From the author's experience of the Fairisle high speed network, using both traditional and micro-kernel operating systems, substantial deficiencies have been uncovered which threaten the latent utility of the available hardware.

This dissertation presents scheduling and I/O structures for ***Fawn***, a general purpose multi-user operating system designed to maximise the service to applications from such hardware.

1.1 Context

Previous research has been done in Cambridge in the field of Continuous Media applications. Much of this has been practical with the implementation of the Pandora's Box [**Hopper90**], a continuous media peripheral for a workstation. Such facilities are typically used in a distributed environment [**Nicolaou90**]. Storage and synchronisation services have been implemented for the Pandora system which used an early ATM network known as the Cambridge Fast Ring [**Temple84**].

The Fairisle project [**Leslie91**] was an investigation of switch based ATM networks. This project designed and constructed a test-bed switched ATM infrastructure in order to investigate the management and behaviour of real traffic streams.

Use of the Pandora system revealed the problems of operating with first generation multimedia equipment, which supports the capture and display of continuous media streams but not their direct manipulation by applications. The observation that bus-based workstations (where data traversed the bus many times) were not ideal led to a prototype second generation multimedia workstation replacing the bus with an ATM based interconnect – the Desk Area Network [**Hayter91**, **Hayter93**, **Barham95**].

Operating system support for continuous media streams has also been under investigation; the creation of the Pegasus project [**Mullender92**] was intended to develop such a system. At the time of writing, the Pegasus project has begun an implementation of some of the low levels of such a system.

1.2 Outline

Background ideas to the work are introduced in the next chapter together with a discussion of the research environment in which this investigation took place. Consideration of previous or related work occurs in context in subsequent chapters where the structure of the relevant parts of the Fawn design is examined.

Chapter 3 studies the quality of service issues of scheduling in a multimedia multi-service networking environment. The methods used by Fawn for inter-process scheduling, inter-process communication, virtual processor interface, and intra-process synchronisation are generated.

Subsequently, chapter 4 develops the higher levels of the communication model and considers trust, binding and naming.

In chapter 5, the schema for bulk data transportation in Fawn is presented after surveying the merits of previous designs.

An experimental implementation, including performance measurements is examined in chapter 6.

Suggestions for further work and extensions to the prototype system are made in chapter 7. These fall broadly into Desk Area Network and Multiprocessor concerns.

Finally chapter 8 summarises the main arguments of the dissertation and makes some concluding remarks.

Chapter 2

Background

This chapter discusses the areas of research which form the background to the work described in the rest of this dissertation. Some of the previous research is considered at the point at which its relation to the equivalent systems proposed in this work is more directly assessable.

2.1 Networking Technology

2.1.1 Asynchronous Transfer Mode

Asynchronous Transfer Mode (ATM), which was originally called Asynchronous Time Division Multiplexing (ATDM), has been in use for approximately 25 years [**Fraser93**] as a technique for integrating isochronous and bursty traffic in the same data network. This technique uses small fixed sized data transfer units known as cells (or mini-packets) each of which includes a header which is used throughout the network to indicate the lightweight virtual circuit over which the data is to travel. During this time Bell Labs has produced four ATM networks: Spider, Datakit, Incon and XUNET [**Fraser92**].

At Cambridge, the Computer Laboratory developed ATDM networks based on slotted rings beginning with the Cambridge Ring [**Hopper78**]. More recently the Fairisle project has developed a general topology switch based ATM network.

In Synchronous Transfer Mode (STM) information from multiple circuits is multiplexed together in a deterministic way into different time slots within a larger frame. The time slot is allocated for the duration of the circuit at the time of its establishment. For example, 32 telephony circuits of 64Kbit/sec may be multiplexed onto a single trunk of 2Mbit/sec. The granularity of information from each circuit is typically very small (e.g. 8 bits). Since the position of each octet within the frame is fixed for each link in the network only a fixed size buffer is needed at each switch. The advantage of such a network is that the data stream arrives at the recipient with no jitter[1] and that the switching requirements are deterministic. The disadvantage is that it is inefficient with bursty data traffic sources and cannot take advantage of fine grain statistical multiplexing.

In Packet Transfer Mode (PTM) information is packaged in large units usually measured in tens of thousands of bits known as packets. Multiplexing is performed asynchronously; when a network node has a packet to send it acquires the entire bandwidth of the channel for the duration of the packet. This involves some form of media access protocol. Contention between nodes introduces variable delays in accessing the channel, leading to jitter. The advantage of PTM is that the bandwidth can be very effectively shared between bursty sources, each using the full channel when required. At switching nodes, queues build up for output ports for which packets are arriving from multiple input ports. This leads to large buffering requirements and adds additional jitter on the data streams.

In Asynchronous Transfer Mode the allocation of time to each circuit is also not predetermined. Instead a dynamic algorithm is used to decide which of the circuits should be permitted to make use of the bandwidth next. To reduce jitter, and to ensure fine grained sharing of the available bandwidth, the allocations are not as large as with PTM (e.g. 32, 48 or 64 octets), and are of a fixed size to ease their handling in hardware. Since the allocation is dynamic ATM networks can accommodate bursty traffic sources much better than STM, and can take advantage of statistical multiplexing. The disadvantage is that the buffering requirements at the switching points is nondeterministic and there is some jitter in the arriving data stream. The header or tag containing the Virtual Circuit Identifier (VCI) is kept as small as possible and only has local significance; it is replaced or *remapped* at each stage. It is the remapping tables in the switches along the route that represent the state for the circuit. The circuits carried by ATM networks are usually lightweight [**Leslie83**] which means that they do not have any hop by hop error control and may be disbanded by intermediate nodes at any time. ATM is designed to be a compromise between PTM and STM which includes sufficient control over jitter and responsiveness to bursty sources that it

[1] Other than the marginal effect of the clock drift of the underlying transmission medium.

can be used for inter-operation between all types of equipment.

A detailed discussion of multiplexing techniques and the advantages and disadvantages of ATM networks may be found in [**McAuley90**].

The author's experience of ATM includes the implementation of an ATM protocol stack within Unix [**Black94c**].

2.1.2 Fairisle

The Fairisle project (on which the author was employed for a period of a year before beginning research) has built a switch based ATM network for the local area [**Leslie91**, **Black94d**]. Since this network was to be used to investigate the management of quality of service in a local area multi-service network, one of the key features of the design was flexibility and programmability.

Field programmable gate arrays (FPGAs) are used extensively throughout the design, which allows the behaviour of the hardware to be changed, and many decisions are made in software. The firmware also supports a great deal of telemetry.

The overall design of the switch is based around a space division fabric with a number of port controllers[2] grouped around it. This is shown in figure 2.1. The switch is input buffered.

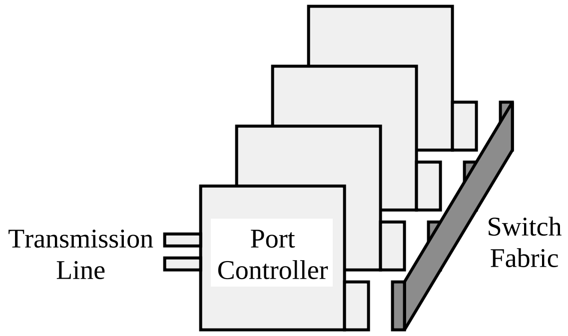

Figure 2.1: Fairisle switch overview

The switch fabrics are available as 4 by 4, 8 by 8, or 16 by 16. These are made up of self routeing round-robin priority 4 by 4 crossbar elements based on the design presented in [**Newman89**] and implemented on a Xilinx 3064. The 16

[2] Also known as line cards.

by 16 fabric uses 8 of these elements arranged in a delta network using two interconnected PCBs. A single PCB gives an 8 by 8 fabric with no use made of the potential additional paths. The arrangement for an 8 by 8 fabric is shown in figure 2.2. For a 16 by 16, the unconnected paths are inter-connected with an identical board.

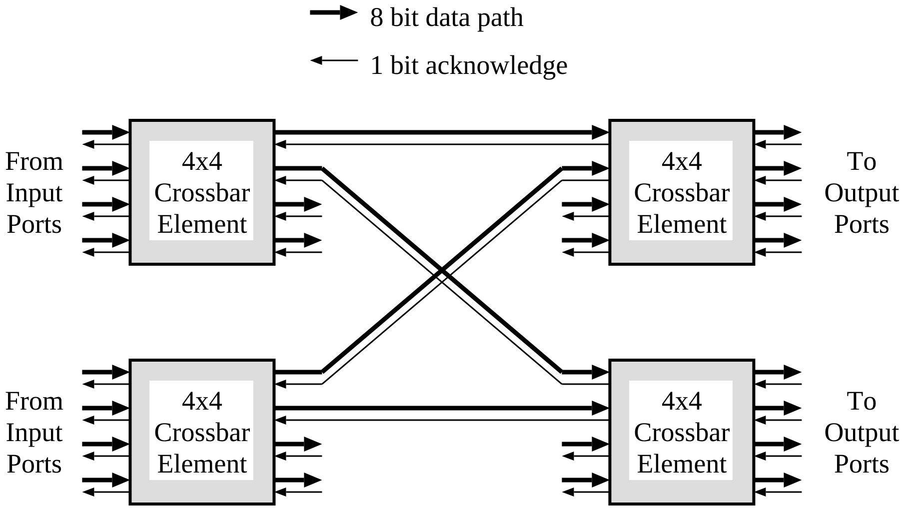

Figure 2.2: Schematic of an 8 by 8 fabric

Each port controller card contains one input and one output port which together make up one line. On early versions of the port controller the transmission system was a daughter-board, but on later versions this was included on the main PCB.

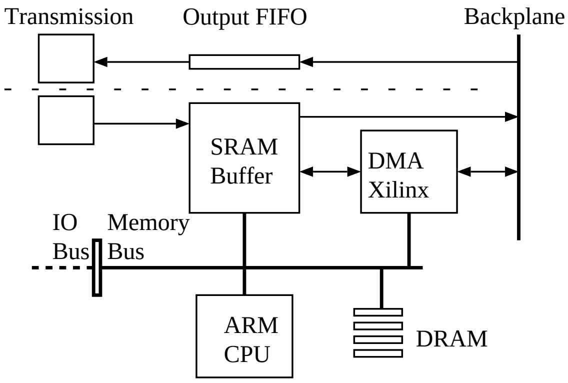

Figure 2.3: Port controller overview

The Port Controller itself consists of a standard Arm based microcomputer including DRAM and ROM together with the Fairisle network section. The network section is comprised of 128Kbytes of SRAM and a Xilinx 3090[3] FPGA. The Xilinx chip is responsible for reading the arriving cells into the buffer SRAM and transmitting cells into the switch fabric itself. Software is in control of these operations dictating the cell scheduling and retry policies. After early experimentation, the management of the free queue, the header remapping, and a small number of automatic retries in the event of contention, were committed to the firmware.

The Arm processor runs the Wanda micro-kernel (see section 2.2.1) together with a hand-crafted assembler interrupt routine which is responsible for handling the cell interface. The Arm processor provides a special low-latency interrupt class known as FIQ which has access to a private bank of registers. Within the Wanda system this FIQ interrupt is never disabled and it interacts with Wanda using conventional interrupts. In this way the Fairisle hardware is presented as a *virtual device*. The full details of the operation of the Fairisle interface may be found in [**Hayter94a**, **Hayter94b**].

Since the port controller includes the standard I/O bus for the Arm chip set it is possible to inter-operate with other networks by plugging in the appropriate adaptor (e.g. Ethernet or CFR).

The Fairisle port controller was used as the experimental platform for the work described in this dissertation and further discussions of its features occur throughout the text.

2.2 Operating System Research

2.2.1 Wanda

The Wanda micro-kernel (loosely derived from Amoeba [**Tanenbaum81**]) is a locally developed operating system designed as an aid to networking, multimedia and operating system research. It has been implemented on both uni- and multi-processors including the VAX Firefly, the 68000, Arm and MIPS R3000. It was ported to the Arm range of processors and the Fairisle hardware by the author.[4]

Wanda provides a multi-threaded environment using kernel threads. The kernel implementation is multi-threaded and includes the virtual address system, the

[3] A 3064 was used on earlier versions.

[4] Mike Roe of the Computer Laboratory assisted in the original port to the Archimedes.

IPC system, device drivers, and the Multi-Service Network Architecture protocol family [**McAuley90**]. The scheduler uses a static priority run-to-completion policy. Synchronisation uses counting semaphores which have separate implementations at user and kernel levels to aid performance. A user space emulation library for the Posix threads primitives exists.[5]

The system provides (logically) separate virtual address spaces for each process but its focus on embedded network or operating system services precludes virtual memory. On many platforms there is no MMU hardware and Wanda has a special Single Address Space (SAS) mode for these platforms. It was partially experience with the Wanda SAS system which led to the consideration of a single virtual address space for the Pegasus project (see section 2.2.3).

There is usually no multiplexing of user threads over kernel threads since the user level synchronisation primitives reduce the cost of kernel level scheduling. An exception is in the ANSA Testbench [**ANSA89**], which is used as one of the available RPC systems; ANSA threads are multiplexed over tasks, with the tasks being implemented using Wanda threads.

The principal distinguishing feature of this system is that it uses the MSNA suite of protocols as its only built-in means of communication (both intra- as well as inter-machine). The original design is presented in [**Dixon91**], subsequently the networking code was re-written by the author in a similar methodology to the original as a requirement to support the Fairisle network. The I/O scheme of the Wanda micro-kernel is considered in detail in section 5.1.2. Other protocol families such as the internet protocol are provided through user space server processes, as is common on many micro-kernels.

2.2.2 The *x*-kernel

The *x*-kernel [**Hutchinson91**] is a kernel that provides an explicit architecture for constructing and composing network protocols. A protocol is viewed as a specification of a communication abstraction through which a collection of participants exchange a set of messages. Three primitive objects are provided: protocols, sessions and messages.

Protocol objects represent the static passive parts of conventional network protocols (i.e. the object code). Their relationship is statically configured. Session ob-

[5]This was implemented by Robin Fairbairns of the Computer Laboratory.

jects are also passive but they are dynamically created. A session object contains the data structures that represent the local state of some network connection. Messages are active objects which move through the protocol and session objects in the system.

The system also provides buffer management routines, a map manager (for sets of bindings between identifiers) and alarm clocks. Messages flow from one protocol to another by making invocation operations on the protocols. The protocols may use the mapping function to track the related session information using fields in the headers etc.

The kernel contains a pool of threads which are used to shepherd arriving packets up from the device driver through the various protocols. For the sake of uniformity, user processes are required to present a protocol interface which consumes or generates packets as required. The kernel threads shepherding a message make an upcall into the user process to deliver data. To prevent a process capturing kernel resources the number of threads which can be active concurrently in a user process as a result of an upcall is limited.

The main feature of this system is the ease with which protocols can be manipulated, experimented with and tested. The measured performance overhead is very low compared to monolithic implementations. The *x*-kernel does, however, presume that all protocols are implemented in the kernel; recently support for device drivers external to the *x*-kernel has been added to allow the *x*-kernel to operate as a single Mach server.

2.2.3 Pegasus

The Pegasus project [**Mullender92**, **Mullender94**] is a joint project funded by the European Community ESPRIT programme involving the University of Cambridge Computer Laboratory and the University of Twente Faculty of Computer Science.

The aim of the project is to develop an architecture for a distributed multi-media system and thereby to construct an operating system which supports multimedia applications. The architecture calls for special purpose multimedia processing servers connected to Unix platforms. The Unix platforms are used for the out-of-band control of multimedia applications, and for non multimedia applications such as compilers.

The operating system support is comprised of the *Nemesis* micro-kernel, which supports multiple protection domains within a single virtual address space. System services are viewed as objects. When object and invoker are in the same process then invocation is via procedure call. Otherwise remote procedure call is used (taking advantage of local machine transports where possible).

The benefits of a single address space include ease of sharing between processes and performance enhancements on machines with virtually addressed caches. The costs are principally that processes must be (re-)linked when loaded, and that sharable code must use some method other than static addresses to access data.

The architecture also calls for split level scheduling to allow the applications to have control over their own scheduling requirements without compromising the kernel's control of application resource usage.

The Nemesis system is currently under development at Cambridge; where it is referred to in this work the reader must understand that this refers to the situation at the time of writing (mid 1994); substantial changes in both design and implementation are likely in the future.

2.2.4 Nemo

The Nemo system [**Hyden94**] was an early prototype of a system in the Pegasus Architecture (and has been used as a starting platform for it). This work principally considered the inter-process scheduling required to promote Quality of Service. Hyden considers real-time problems in detail.

A real time system is one in which the correct operation of the system is dependent not only on the logical correctness of some computation, but also on the time that it is completed. A hard real-time system is one in which temporal deadlines must never be missed. A soft real-time system is one in which a small *probabilistic* chance of missing deadlines is acceptable. A soft real-time system may operate successfully under much greater load than a hard real-time one.

Hyden contemplates the feasibility of using hard real-time scheduling technology for solving the problems of such a soft real-time system which is subject to overload. He concludes that in such a system (where the demand may not be known in advance) a negotiated share of the CPU over a regular time period is more appropriate.

In the Nemo system the processes are guaranteed some number of milliseconds of processor time over some longer per-process period. Unallocated (or unused) time is given to processes desiring more on an environment specific basis. Scheduling decisions are made by a privileged system process known as the "kernel". The Nemo Trusted Supervisor Code (NTSC) (which represents the code to which the term kernel is more usually applied) is responsible for dispatching all system events to the kernel for consideration and for executing various processes on demand.

The key feature of the Nemo system is that the applications are always aware of the amount of CPU they have been allocated and are informed (via an upcall) when they are given the CPU. Use of layered processing is shown to improve the results of computations if more time is available (e.g. decoding a JPEG image).

The Nemo system does not include device drivers, consideration of inter-process communication, or intra-domain synchronisation. Nevertheless it is a seminal work in this field and forms the basis for the lowest levels of the Fawn system presented in this dissertation.

2.3 Related Scheduling Work

2.3.1 Sumo

The Sumo project at Lancaster University [**Coulson93**] is addressing multimedia support by modifying the Chorus micro-kernel. This work has the advantage that Chorus already supports Unix applications through the provision of a Unix sub-system. The features which are being revised include, the introduction of stream based rather than message communication primitives, support for using quality of service to exert control over communications parameters, and changes to the existing coarse grained relative priority scheduling.

Like the Pegasus architecture, this work adopts split level scheduling primitives, multiplexing user process threads over kernel provided threads. To avoid priority inversion the maximum number of kernel threads active within a process is limited to the underlying number of CPUs on the machine. This is also achieved using non-blocking system calls and upcalls (or software interrupts) to indicate various events.

The Sumo project uses earliest deadline first scheduling although no absolute guarantee is made that the deadlines will be met; the QoS parameters are treated as forming a soft-real-time system. The real-time deadline scheduling co-exists with the Chorus system using the standard scheduling classes mechanism. Each multimedia process presents the earliest deadline of its user-level threads to the kernel level scheduler.

The kernel level scheduler always runs the process with the earliest deadline. The user level schedulers are expected to run the thread with the earliest deadline. The system relies on the user level processes not to abuse this mechanism to take more of the CPU than the quality of service they have negotiated. The deadline of each virtual processor is read on every rescheduling operation by the kernel to compute the next kernel thread to schedule. This has a cost which is linear in the number of processes in the system.

Device drivers are implemented entirely within the kernel and no multiplexing over kernel threads occurs. The bulk data I/O mechanism relies on using the virtual memory system to remap buffers from the device driver to the receiving process. Applications are frequently finished with the buffers before the device driver needs to use them again. In the event that they are not the system relies on Chorus' copy-on-write. A scheme to compromise between the driver's requirement always to have buffers available and minimising the need to copy is being researched.

2.3.2 Meta-Level Scheduler

A variant on the idea of the split level scheduler is proposed in [**Oikawa93**]. In that system a layer known as a Meta-Level Scheduler (MLS) is inserted between kernel functionality and the processes. This layer consists of a page of memory for each process which is used by that process to store its intra-process scheduling requirements. A process does not have write access to any other processes' area.

The Meta-Level contains the thread control data structures for events and timeouts, along with information about priorities and policies for scheduling. When a kernel event occurs, the kernel calls into the MLS and the MLS makes the scheduling decision. The MLS may then instruct the kernel which process to run, changing protection domain if necessary. Various different policies may be implemented within the MLS and the system is considered to be easy to experiment with as only the MLS code need be altered.

In essence this design abstracts the kernel-scheduling and virtual processor interface from the kernel and user processes and incorporates it into an abstract object which is constructed in such a way that methods may be invoked from the user processes and the kernel concurrently.

The MLS is a meta-level in the sense that it is designed to encourage modularity of thought when considering the scheduling decisions of the system. Clearly the implementation will include a kernel / user level split.

2.4 Summary

It is now accepted that multimedia systems are a soft real-time problem. Various scheduling mechanisms are being used to try and provide quality of service support within such a system. Some of these (e.g. Sumo) are adapting technology derived for the hard real-time environment. Most systems are using split level scheduling to separate the CPU resource usage of the various applications from the internal scheduling requirements of each application.

High performance ATM networks are becoming available for the transport of voluminous multimedia data. Recent hardware and operating system research has been based on improving the throughput from the network interface to the application, rather than the traditional CPU to memory concerns of systems engineers.

Chapter 3

Process Scheduling

Priority is normally considered to be the solution to the problem of providing quality of service support to multimedia applications. This chapter will consider some of the problems with priority based scheduling and then move on to consider the solution in the Fawn system – derived from the Nemo system.

Inter-process communication and the virtual processor interface of the Fawn kernel are then described. Finally the intra-process synchronisation mechanisms are described.

3.1 Priority

First, note that a simple priority scheme is rarely sufficient. Simple priority schemes suffer from priority inversion, leading to the priority inheritance protocol, and the priority ceiling protocol [**Sha87**] which requires static analysis (the determination of the complete call graph at compile time). In fact Namakura argues [**Nakamura93**] that even more complex schemes are required, again requiring static analysis.

Static analysis of code, however, is an operation which can only be applied to code with a static call graph. This is most untypical of all but the most esoteric of executives. In a system using dynamic binding of services, closures [**Saltzer78**] for library routines, and especially RPC (even if restricted to same machine), it simply cannot be applied. Furthermore in a general purpose operating system it is impossible to reconcile the claims of different competing user processes. Even in a single user, single purpose system the strict use of priority is not always helpful.

3.1.1 Priority in the Internet

In [**Jacobson93**] Jacobson reports on the effect of routers prioritising the processing of routeing updates over forwarding duties on the internet. The observation is that the probability of packet loss during this (periodic) processing is much higher. It is important that a very high packet arrival rate does not delay the processing of routeing updates indefinitely. To avoid this, routers give the processing of routeing updates priority over data packet processing. The resultant gap in processing has been measured at over 300ms [**Jacobson93**] for a single router. The effect on a multimedia stream is made worse by the synchronisation of routeing updates which causes multiple routers to cease processing at the same time. This effect is particularly severe on audio streams as regular drop-outs are very perceptible to the human ear.

3.1.2 Priority in Wanda

Using the Wanda system, Dixon found that jitter over a local area connection could be significantly reduced if the Wanda threads running at each end had a higher priority than any other thread in the system [**Dixon91**]. The improvement gained must be weighed against the cost - crucial kernel housekeeping threads were no longer guaranteed to run, also the number of streams to which this technique can be applied is clearly limited.

Additionally, the fact that interrupts caused by arriving data can indefinitely starve the system of any thread-based activity leads to an implementation where the entire signalling stack is executed as an upcall from the interrupt service routine rather than in a kernel thread. This requires that the signalling stack be entirely non-blocking adding significantly to the protocol's implementation complexity. Even for the extremely lightweight ATM signalling in Wanda this in-band processing adds jitter to the streams passing through the machine; for the elaborate signalling protocols required for BISDN (e.g. Q.2931 / Q.SAAL) the jitter introduced would be enormous, assuming that these protocols could be implemented in this manner at all.

Whilst upcalls are a very efficient way of dispatching arriving data, the time for which such a call captures the CPU should be strictly curbed to ensure that effective control of CPU scheduling is not lost. [**Dixon91**] also notes that pervasive use of upcalls can have detrimental effects in certain environments.

3.1.3 Priority between Applications

In [**Nieh93**] a comprehensive study is made of the effects of various scheduling schemes and parameters on a mixture of interactive, multimedia and batch applications. This work was performed on Solaris 2.2 (an SVR4 Unix). They found that the performance was unacceptable without a great deal of tweaking of the priorities of the various processes; the resulting configuration being highly fragile with the job mix of the machine. Further, the use of the "real time" priority based scheduler did not lead to any acceptable solution. This is a direct quotation from [**Nieh93**]:

> Note that the existence of the strict-priority real-time scheduling class in standard SVR4 in no way allows a user to effectively deal with these types of problems. In addition, it opens the very real possibility of runaway applications that consume all CPU resources and effectively prevent a user or system administrator from regaining control without rebooting the system.

Their solution was to develop a new timesharing class which allocates the CPU more fairly and attempts to ensure that all runnable processes do make steady progress.

3.1.4 Priority within Applications

In [**Sreenan93**] Wanda is used as a platform for implementing a local area synchronisation service. A number of problems with priority scheduling in Wanda arose during this work. It was found necessary to split the clock interrupts into two separate pieces reminiscent of the *hardclock* versus *softclock* distinction in Unix [**Leffler89**]. The high priority interrupt (which can interrupt other device drivers) ensures that no time skew occurs. The lower priority interrupt is used to examine the scheduler data structures to determine if any threads need to be awoken. This split solved the problem of high network load causing missed timer interrupts but implementation of this dynamic priority change in timer handling necessitated an (interrupt) context switch. [**Dixon91**] reported that the cost of timer interrupts in the common case was 14μs.[1] The cost of split timer interrupts is not reported but a likely estimate is double at 30μs or 3% of the machine for his 1ms ticker.[2] Despite using the machine in single application mode it took considerable experimentation to configure acceptable priorities.

Another example of an application where priority caused problems is described in [**Jardetzky92**]. This application used a dedicated Wanda machine as a networked continuous media fileserver. Two problems were encountered in the design of this system. Firstly, the problem of selecting the priority for control threads (i.e. the threads handling out of band control via RPC). These threads do not require immediate execution since that would introduce jitter on the data streams. They do, however, require timely execution and must not be delayed indefinitely by the data streams which they control.

Secondly, a similar problem occurs with the disk write threads which wish to run infrequently writing clusters of buffered data to the disk. These should be of lower priority than the disk read and network threads, whose scheduling is important for observed application jitter, but they must get some processing time otherwise the buffers in the system will become exhausted. This particular concern is examined thoroughly in [**Bosch94**]. In that work (concerned with conventional file I/O) the interruption of disk write requests by disk read requests is recommended. In addition writes are delayed altogether for as long as 1000 seconds or until the buffer cache becomes more than 25% dirty. This, together with a very large cache, is used to reduce fileserver latency. That work makes the assumption that the read requests are sufficiently bursty that the disk eventually becomes idle and writes can occur.

For the Pandora system a set of guiding principles was developed to help the system behave reasonably under overload, and to ensure that the user was best placed to notice and take corrective action. These were:[3]

[1] On 20MHz 68020 / 68030 powered machines.

[2] One of the devices on this platform has a feature that if the device handler is interrupted at an unfortunate moment the device has a small probability of crashing the machine. Theoretically such (short) sections of the device driver should also change their dynamic interrupt priority to avoid this problem. Although this is a peculiar device similar requirements do occasionally present themselves in the form of cache ECC errors and the like.

[3] This list is abridged from [**Jones93**].

- Outgoing data streams have priority over incoming ones.[4]
- Commands have priority over audio which in turn has priority over video.
- Newer data streams have priority over older ones.
- Where a stream is being split (multicast) bottlenecks should not affect upstream nodes.

3.1.5 Priority in Devices

Problems also occur in using priority for handling devices. In Unix network device drivers operate at a high interrupt priority known as *splimp* with protocol processing being performed at the lower interrupt priority *splnet*. This leads to performance loss due to the context switches (dispatching overhead and register file saves and loads) and reconfiguration of device interrupt masks. One discussion of this performance cost can be found in [**Black94c**]. A related cost is due to the effect of *live-lock*, where excessive data arriving can hinder the progress of requests already in the system. Live-lock in Unix has been reported in [**Burrows88**] and [**Mogul**]. This particular live-lock problem is now so severe with high performance network devices that some hardware [**Rodeheffer94**] includes special support which allows the device driver to disable high priority interrupts from the adaptor so that low priority protocol processing can be performed [**Burrows94**]. In effect the scheduling of the system has been moved from the scheduler into the device driver code, with the resultant potential for pathological interactions between multiple such device drivers.

Another problem with giving device interrupts priority over processes is that the interrupt dispatching overhead can consume substantial amounts of the CPU. An extreme example of this is the Fairisle port controller, reported in [**Black94d**], where an ATM device generates an interrupt for every cell received. The overhead of the interrupt dispatching in the system is such that no process execution occurs once the system is approximately half loaded; the extra CPU resources being lost. Tackling this problem is a specific aim of this dissertation.

It is interesting to note that mainframe computers avoid the problems associated with devices by having a large number of dedicated *channel processors*. In the case of overloading a particular channel processor may suffer from these problems,

[4] Except at a fileserver.

but the rest of the processors in the system will continue to make progress. This amounts to a limit (albeit a physically imposed one) on the amount of the total CPU power of the machine which can be consumed by a device. That is, the device does *not* have priority over the computations of the system as a whole.

3.1.6 Periodicity

As a result of these problems with priority many hard-real-time control systems use the Rate Monotonic algorithm presented in [**Liu73**]. This provides a method for scheduling a set of tasks based on a static priority calculated from their periods. It relies on the following assumptions:[5]

(A1) The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.

(A2) Deadlines consist of run-ability constraints only – i.e., each task must be completed before the next request for it occurs.

(A3) The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.

(A4) Run-time for each task is constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

(A5) Any non-periodic tasks in the system are special; they are initialization or failure-recovery routines; they displace periodic tasks while they themselves are being run and do not themselves have hard, critical deadlines.

Of these assumptions, thread independence, constant period, and constant requirements, are considered inappropriate for a network based fluctuating service. Further consideration of the inappropriate application of hard real-time solutions to soft real time systems can be found in [**Hyden94**].

3.1.7 Earliest Deadline First

The EDF algorithm also presented in [**Liu73**] is a dynamic priority scheduling algorithm which also relies on the assumptions of section 3.1.6. The deadline of

[5] Quoted directly from [**Liu73**].

a task is considered to be the time at which the next request for that task will occur. It is shown that this scheme will permit a feasible schedule where the CPU utilisation is less than or equal to 100%.

This also suffers from inapplicability of the assumption of independence. In an environment where processes make use of services provided by others, a "task" is not associated with a "process"; simply knowing what the current deadline *is* can be very difficult. An equivalent of the priority inheritance protocol would be required. Every interaction with every other process could involve recalculation of the deadlines of the processes. Furthermore the servers must be trusted to perform work only for the task with the earliest deadline, and to pass on the deadline correctly to any other services.

Even if such a scheme were conceivable for a single machine it fails completely in a distributed environment, where remote services must be invoked (where a deadline carries no authority), and unpredictable network delays exist. In general a device driver may have to exert a substantial amount of effort when a packet arrives before it is even aware which process is the recipient, and therefore which deadline is appropriate.

3.1.8 Summary

Priority based mechanisms are ideal for certain applications, specifically those for which an explicit precedence of well-defined tasks exists. Such an absolute ordering does not frequently occur, however, in general purpose systems; forcing such a system into the priority mould leads to instability, unmaintainability and loss of flexibility.

3.2 Sharing the CPU

As noted earlier, Fawn uses a split level scheduling policy. This section considers the lowest (kernel) level scheduling of processes. First it will be constructive to examine this process scheduling within the Nemo and early Nemesis[6] systems, considering some of the infelicities.

[6] Mid 1994

3.2.1 Interprocess scheduling in Nemo

In the architecture of [**Hyden94**], system processes which have guaranteed access to the CPU do so by negotiating a certain amount of CPU time (called[7] a *slice*) in another larger timer period (called[7] an *interval*). For example a process displaying NTSC video may require a contract for a *slice* of 10ms of CPU every *interval* of $33\frac{1}{3}$ms.

The scheduler keeps three different run queues; Q_c is for processes with contracted time, Q_w is for processes which have consumed all their contracted time but wish for more CPU if it is available, and Q_f for best effort (background) processes.

The kernel keeps an alarm timer for every process in order to allocate that process's contracted amount. Whenever this alarm occurs the appropriate process has its contracted amount set to its negotiated value and is made runnable.

The design calls for the use of an EDF algorithm for scheduling processes on Q_c. Whilst Q_c is empty, the kernel runs processes from Q_w, and then from Q_f, according to an environment specific algorithm (which was left for further study).

The principle of this design is that processes acquire a fair share (as deemed by the resource manager) of the CPU, with regular access to it. Remaining time is allocated in a way which attempts to maximise the possibility of it being used effectively.

This design suffers from the use of EDF scheduling which was discussed in section 3.1.7. Also the number of clock interrupts and context switches involved can be quite large, because allocation of each slice to each process potentially requires a separate clock interrupt, a context switch, and a further context switch later to finish the slice of the interrupted process.

A related problem comes from the transfer of resources between clients and servers. When a client requires some service from another process it may arrange for some of its CPU allocation to be transferred to the server in order to perform the work. If the client and server have different *intervals* it is not clear exactly how the server's scheduling parameters should be altered.

[7] [**Hyden94**] does not use this terminology; I introduce it here for clarity only.

3.2.2 Interprocess scheduling in Nemesis

The mid 1994 version of the Nemesis system adopted a particular prototype of Nemo which used differing scheduling algorithms to those described above.

For Q_c the Nemesis scheduler context switches to a process as soon as it is allocated a slice. For Q_w and Q_f it allocates in a first-come first-served fashion (i.e. they are run to completion unless some contracted time is subsequently allocated).

There are a number of problems with this design. The primary problem is related to the beating of processes with different intervals. Consider the scenario of figure 3.1 where process **A** is allocated a slice of 4 over an interval of 8 ticks (i.e. 50% of the CPU over a short interval). Processes **B** and **C** are allocated a slice of 8 over a much longer (unimportant) interval.

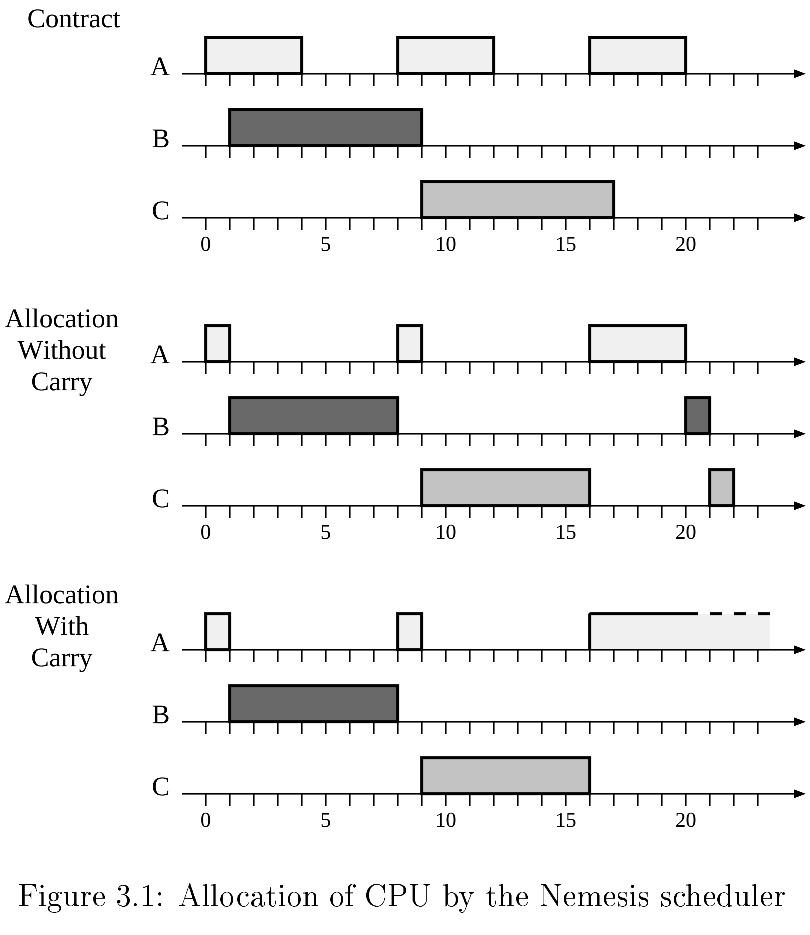

Figure 3.1: Allocation of CPU by the Nemesis scheduler

For Nemesis, the allocation of the CPU will be as shown for “Without Carry”.

When **B** gets allocated its slice (shortly after **A** has started running), the scheduler will defer **A** in favour of it. Likewise **A** will regain the CPU at the beginning of its next interval. Shortly after that **C** gains the CPU from **A**. The result is that over two intervals **A** has had only two very small periods of access to the CPU. If there were more processes in the system (**D**, **E**, ...) this starvation could be much worse.

Obviously the long term starvation can be addressed by adding the slice to the contracted time of a process at the beginning of each of its intervals rather than assigning it. This is shown in figure 3.1 by the "With Carry" case. Over a long time scale **A** will now be guaranteed its 50% share. It is still true, however, that processing by **A** can, in the worst case, be delayed by the sum of the slices of all of the other processes in the system, potentially a very large amount in comparison to **A**'s interval.

For a hard real-time system these problems would be resolved by simply applying the Rate Monotonic algorithm. We have already seen (in section 3.1.6), though, that in a general purpose system the rate monotonic algorithm is inappropriate.

Note also that the long term starvation is exactly the case that will occur in an EDF scheduled system (under consideration for possible addition to Nemesis in late 1994) when the applications lie about their deadlines in order to try and access the CPU as soon as possible. EDF was also shown to be ill-suited in section 3.1.7 above.

3.2.3 Interprocess scheduling in Fawn

The Fawn system is based on the Nemo design and also uses a single virtual address space for all processes, with multiple protection domains.

In the Nemo system, as described above, allocation of CPU resource is made to processes at a frequency of their own choice, with no correlation between the points at which this crediting is made.

For the Fawn system EDF scheduling was rejected and it was decided instead to perform the reallocation of CPU resources to processes simultaneously at a single system-defined frequency. To avoid overloading the word epoch the period between these allocations is known as a *Jubilee*. A jubilee has a period derived from the minimum significant time period for the platform. This will typically be a few tens of milliseconds, depending on the performance and the exact nature of the expected applications.

The use of the jubilee system ensures that processes get a fair share of the CPU with regular access to it. In effect all processes in the system have the same *interval*. Access to the CPU at a finer granularity than the jubilee is probabilistic rather than guaranteed. Their use also reduces the number of interrupts and context switches in comparison with the Nemo system.

Like the Nemo system, the kernel has a number of scheduling levels which represent a strict priority order. The first level is for guaranteed CPU; the others for successively more speculative computations. Each process in the system is allocated an amount of CPU at zero or more levels by the Quality of Service manager.

Allocations are measured in *ticks* which is the base unit of time on the system. The guaranteed (i.e. first level) allocations of all processes must sum to *less* than 100% of the number of *ticks* per *jubilee*. Since the kernel overheads are accrued against the currently selected process, allocations should not be made too small. The kernel maintains a record, per process, of the number of ticks consumed at each level; and also the total number of ticks consumed in the system by processes in general at each level. These measures provide feedback to the QoS manager[8] which may consequently alter the allocations.

Associated with each scheduling level is a queue which contains all the processes for which that level represents their highest current entitlement. A process only resides on a single queue.

The kernel's scheduling algorithm is very simple. It always selects the first process from the highest scheduling level. An alarm is initialised to interrupt the system after the number of ticks currently remaining to that process in that level have elapsed. In this way the CPU used by each process can be strictly controlled. If a process blocks it is moved off the level's run queue and its remaining time is decremented by the time for which it did run. If it is subsequently unblocked it will be returned to that queue. If a process is still running when the alarm occurs then it is moved to the next lower queue.

At the beginning of each jubilee all processes are made runnable, their allocations at each level are reinitialised and they are restored to the highest queue for which they have an allocation.

[8]The QoS manager is, of course, expected to provide itself with some guaranteed time.

Processes are thus guaranteed to have been given a certain share of the CPU over the interval of a jubilee without the complexity of EDF scheduling.

One of the advantages of the jubilee scheme is that it becomes trivial to transfer resources from a client to a server as a result of some contract; this merely requires the subtraction from the client's allocation and addition to the server's.

3.3 Inter Process Communication

In an operating system one of the requirements is to provide a mechanism for the various processes to communicate. There were a number of important considerations for the design of the inter process communication mechanism.

- For efficiency reasons, processes in the system will transfer the large part of their data using memory addresses in the single address space to which they both have access.
- The mechanism must be sufficiently flexible to permit processes to construct arbitrary communication protocols in such memory.
- The mechanism must not force synchronous behaviour on processes which would find an asynchronous mechanism more convenient.
- It must be possible to communicate in a non-blocking manner. This is important for device drivers or servers which are QoS conscious. In Unix and Wanda, device drivers have a special set of library routines which provide non-blocking access to the facilities of the system.
- Invocation of the mechanism must convey sufficient semantics about any potential communication that it need be the only point at which explicit memory synchronisation need be performed in a loosely-coupled multiprocessor system. This requirement is to enable portable use of partially ordered memory systems such as an Alpha AXP multiprocessor, or the Desk Area Network.
- Termination of a communication channel by the process at one end should not necessitate synchronous recovery on the part of the process at the other end.

- A thread scheduler within a process can map communications activities to scheduling requirements efficiently; this necessitates that the communications primitives be designed in conjunction with the concurrency primitives.

These requirements dictate a solution which is asynchronous, non-blocking, and can indicate an arbitrary number of communications which have occurred to the receiver. It must never lose communication notifications nor require large structures to be stored.

For these reasons the design adopted a scheme based on *events*. An event is a monotonically increasing integer whose value may be read and modified atomically by the sender, and which can be forwarded to the recipient by a kernel system call. The recipient has a copy of the event whose value will be updated some time later by the kernel. The relationship between these *events* and event counts and sequencers [**Reed77**] is discussed in section 3.5 below.

This propagation is called an *event channel*. For each process, the kernel has a protected table of the destinations of the event channels originating at that process. A management process is responsible for initialising these tables and thereby creating communication channels.

3.4 Virtual Processor Interface

The virtual processor interface (VPI) is similar in some ways to that in the Nemo system. These features are:

- The kernel system calls are non-blocking.
- The kernel upcalls the process when it is given a time slice rather than resuming it. The process is then able to make its internal scheduling decisions. This is called an *activation*.
- The process may disable activations when operating within its own critical regions. When disabled the kernel will resume a process instead of activating it. This avoids the need to write re-entrant activation handlers, and greatly eases thread scheduling. This is the corresponding operation in the virtual processor to disabling interrupts in the physical processor.

It has also been extended as follows:

- A mechanism for the kernel to indicate which event channels have been active is provided.
- There is support for virtualising hardware interrupts for device drivers.
- System time is unified with event handling.

3.4.1 Activations

The concept of activations is derived from some of the ideas of [**Anderson90**]. When a process is given the CPU after some time without it, the process is upcalled or activated by a special means rather than simply being resumed at the point where it lost the CPU. This allows the process to take scheduling actions based on the real time at which it obtains CPU resource.

Most of the time a process will wish to be activated when it is given the CPU. Sometimes, however, it will be operating within a critical section where such an activation would be difficult to cope with, entailing re-entrant handlers. The process must be able to control whether it is activated or merely resumed. This control over activations is implemented using two event counts, known as *disable* and *enable*, and three context save pointers, *save*, *restore*, and *init*.

When a process is to be given the CPU, the kernel compares the values of *disable* and *enable*. If the value of *disable* is the greater then the kernel will resume the process by loading the context from *restore*, otherwise it will activate the process by increasing the value of *disable* (preventing re-entrant activations) and loading the context from *init*. When a process loses the CPU involuntarily, if the value of *disable* is the greater then the context will be stored in *restore*, otherwise it will be stored in *save*. The purpose of *init* is discussed below.

The context save pointers point into a special area of memory which is mapped both kernel and user read/write, and which is non pageable. The thread scheduler within a process will typically change the *save* pointer to point to the context of the currently running thread. The activation handler will ensure that the processor state pointed at by *save* is preserved before re-enabling activations. A system call exists for restoring the processor state from the *save* pointer and increasing the value of *enable* atomically.

The user process can disable activations at any time by incrementing the event *disable*.[9] Such a critical region can be ended by incrementing the event *enable*. These indivisible regions can therefore be nested if required. The kernel also provides system calls to give up the CPU voluntarily (both with and without increasing the value of *enable*) if the process decides that it has no more work to do.

If a process is concerned about starvation of activations (e.g. because it frequently has them disabled when it happens to lose the CPU) then it can detect this by reading the value of the jubilee (or incoming events) counters after increasing *enable*. These are discussed in the next two sections.

When a process is to be activated (rather than resumed) it is necessary for security reasons for the kernel to clear the processor registers before performing the upcall. This ensures that no privileged information may leak from the kernel nor private information leak from other processes. When the process is activated it must make intra-process scheduling decisions (see section 3.5). To make these decisions it must load some scheduling state into the processor registers.

In Fawn these two operations are combined. In the virtual processor interface, each process must supply an *init* context which is stored in the same format as the saved or resumed context. When the process is to be activated the kernel loads this context which includes the program counter, stack, etc. Thus the process begins immediate execution of the intra-process scheduler with the full context necessary. This means that the cost of clearing the registers can be eliminated.

	Losing the CPU	Gaining the CPU
$disable > enable$	*restore*	*restore*
$disable \leq {}^{a}enable$	*save*	*init*[b]

[a] *Disable* less than *enable* is not likely to be useful.
[b] This represents an activation, which will automatically increase the value of *disable*.

Table 3.1: Summary of VPI context area usage.

As a further optimisation it can be noted that the kernel must have already examined the *disable* and *restore* meta events when saving a process's context. Thus at this time it can calculate whether the process will be subsequently resumed or

[9] This may necessitate a kernel trap on platforms with no support for appropriate atomic primitives.

activated and store a single pointer in the process' kernel data structure.[10] When that process is next given an allocation of CPU time the kernel need merely load the register file, there is no case analysis required.

A summary of the use of *init*, *restore* and *save* is given in table 3.1.

3.4.2 Events

The VPI contains a pointer to the process's event table which is written by the kernel when delivering an event, and also two meta-events called *pending* and *acknowledged* which are used in the delivery of normal events to a process. *Pending* is incremented by the kernel when it updates an event value and *acknowledged* is incremented by the process when it has taken due consideration of the arrival of that event. These meta-events are used to prevent a process losing the ability to act on an event arriving just as it voluntarily relinquishes the CPU. When a process gives up the CPU, the kernel compares the value of the two meta events and if *pending* is the greater then the process is considered for immediate re-activation.

The VPI also includes an implementation-specific mechanism for indicating to the process's scheduler which of its events have been updated. Two mechanisms have been considered:

3.4.2.1 Circular Buffer

The two meta-events are used as a producer / consumer pair on a small circular buffer. The kernel writes into the buffer the index of the event that it is modifying. If the buffer becomes full the kernel writes a special code which indicates overflow, in which case the process must check all the incoming events.

3.4.2.2 Bitmask

A bitmask is used to indicate which events have been updated by the kernel. The kernel sets the bit when updating the event, and the process clears it before considering the event. The process must scan the bitmask when activated to

[10] Incrementing *disable* if necessary as described above.

determine which events must be checked to determine if waiting threads should be unblocked.

A variant on this scheme is to have one bit in the bitmask for a set of events which share the same cache line. A set bit indicates that at least one of the events in the set has been changed and all should be examined by the intra-process scheduler. This would reduce the size of the bitmask without changing the number of cache lines of the (potentially large) event table which must be loaded into the cache during the scan.

3.4.2.3 Discussion

Of these two schemes the circular buffer is preferred on two counts. First of all the bitmask scheme introduces a requirement for an atomic operation which was not otherwise needed, whereas the circular buffer scheme relies on the atomic update already present for events. Second, the expected CPU and cache performance of the circular buffer will be superior as even a buffer of a few cache lines will rarely overflow. Hybrid schemes are also possible, where the bitmask could be used both to reduce the cost when the circular buffer does overflow, and to avoid the same event being placed in the circular buffer more than once.

As an example, figure 3.2 shows the value of event number **n** in process **A** being propagated (by a system call) to process **B** where it is event number **m**. The mapping for event channels from **A** has the pair **B**,**m** for entry **n** so the kernel copies the value from **A**'s event table to the appropriate entry in **B**'s event table and places **B**'s index (in this case **m**) into **B**'s circular buffer fifo.

3.4.2.4 Growing the event tables

When a process needs to extend the size of its event table it must also potentially extend the size of the kernel's event channel table. Neither of these operations is problematic if the table may be extended in place; relocation is more difficult.

A process may extend its own event table by allocating a new area of memory and copying the contents of the old table into it. When this is complete the change can be registered in the VPI by performing an RPC (see later) to the management process responsible for managing event channels. During this time the process could have been the recipient of new events which may have been placed in either

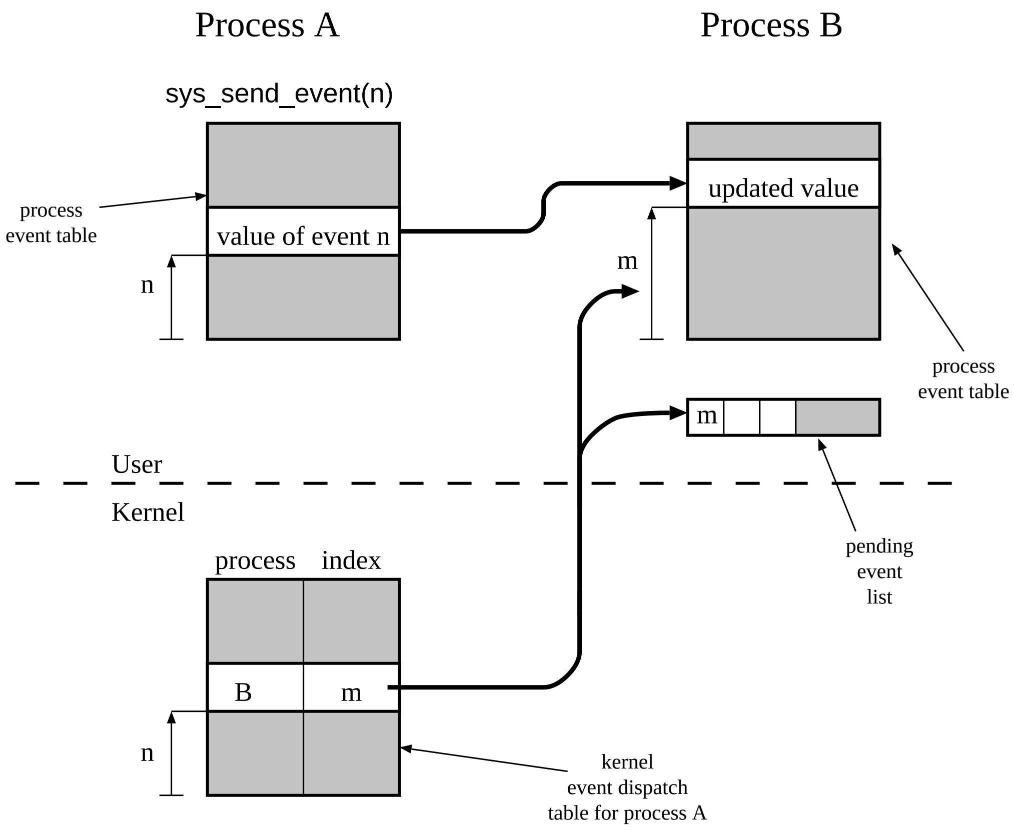

Figure 3.2: Example of sending an event update

the new or the old areas. The process can use the event notification mechanism to determine which events may potentially need to be re-copied. The fact that the events are monotonically increasing allows it to determine easily which is the correct value.

If the process determines that the kernel's event channel table requires extension then it must make an RPC call to the management process. The management process has authority to manipulate the kernel's table and so can copy the table to a new area. It sets a flag so that if during this action it performs any event channel management for that process it can make the changes to both the new and the old copies. This RPC may be amalgamated with the one for the process's own table.

During both of these operations all processes may continue as normal; the explicit notification of updates permits the change to proceed asynchronously.

3.4.3 Time

One remaining important part of the virtual processor interface is the aspect of time. A process needs to have some idea of the current time, since it needs to schedule many of its activities related to time in the real world. This is particularly important in a multi-media environment. A process may also need to know when activated whether it is getting an additional slice of CPU over and above its guaranteed amount or not.

In the Nemo architecture, an accurate hardware clock value is available read only to all processes. Unfortunately such a hardware register rarely exists and typical implementations use the kernel's clock value giving a time value accurate to the time of the last interrupt. Each process is expected to keep track of the number of times that its own *interval* has elapsed by counting the number of occasions that a particular bit is set in a status register when it is activated or by regularly reading the clock register. This mechanism was considered inappropriate for the Fawn system for multiple reasons:

- The process could occasionally be in a critical section when it is given a new time allocation. This could lead to the indication being missed – equivalent to missing a timer interrupt on a physical processor. As discussed in section 3.1.4 this possibility is, from experience, carefully avoided in current systems.
- There is no homogeneous way of incorporating timeouts or alarms for a process with the event mechanism.
- There is no active notification of the fact that time is passing, a process must include regular checks to detect it.
- The available time would be a wall clock time and would be subject to adjustments to keep it synchronised with other systems (e.g. by using NTP [**Mills89**]) rather than the time which is being used for local scheduling. Since adjustments may add as much as a 10% skew the locally based clock which is likely to be accurate with respect to instruction execution is more useful to many applications.

In the Fawn system, scheduling time is indicated to each process by using event zero to denote the number of elapsed jubilees. The event is updated by the kernel on the first occasion within a jubilee that the process is given the CPU. Within the process, code can read the current time (in jubilees) by reading the value of event zero. The system makes no guarantees about how the CPU is allocated with smaller granularity than a jubilee, but a method is provided for reading a (platform dependent) high granularity time of day clock. The implementation of this may vary from a single instruction access to a processor cycle counter register, to a reasonably costly system call. Of course the result may be wrong as soon as the value is obtained due to context switches. Providing scheduling time as an event means that time is now easily integrated with the event mechanism in a way suitable for thread primitives (see section 3.5.1).

At of mid 1994 the Nemesis system does not have a well defined concept of time, the current implementation has adopted Fawn's method of describing time without having adopted the jubilee concept. Instead the value which is updated on each activation represents the total number of system tick interrupts; Nemesis assumes that scheduling is based on a high frequency regular clock.

3.4.4 Interrupts

Finally, processes which manage devices may need to be informed about underlying device interrupts. These interrupts, which may be enabled or disabled via privileged system calls, are delivered to the process via an event. The initiation of this interrupt event channel is handled by the *Basic Resource Allocator* (see section 4.4.5).

The system permits (but does not require) the multiplexing of hardware interrupt conditions onto events. This is because platforms frequently have devices which raise different hardware interrupts on different conditions but which have a single driver. The kernel contains a table which maps from hardware interrupt bits to an event number in a particular process. Exactly one entry in the table exists which refers to an enabled interrupt.

When an interrupt occurs, the corresponding entry in the table is found and all the interrupts listed in that entry are disabled (this list is usually a bit-mask of the system's interrupt mask register). This operation requires the absolute minimum of code to make these hardware interrupt conditions cease (usually a write to the system's interrupt mask register). The listed event is sent to the process, and the kernel uses a heuristic (based on the length of time that the current process has been running) to determine if the process receiving the interrupt should be given the CPU. This heuristic reduces interrupt dispatching latency at the potential cost of slightly more context switches. Thus a process which has a sufficient allocation of CPU, and which always handles interrupts quickly, is likely to get a very low latency.

The index into the kernel interrupt table may also be used by the process managing the associated device to request the enabling or disabling of the individual hardware interrupts in the system interrupt mask register. The kernel checks that the process making the request is the one listed in the process field of the interrupt table, providing security.

3.5 Intra-Process Scheduling

The intra-process scheduler is the code which sits above the virtual processor interface. The code is not privileged and can differ from process to process. It may be very simple (in the case of a single threaded process), or more complex. Although four different thread schedulers have been written, the interface they provide to the rest of the process is the same.

Since there is exactly one intra-process scheduler for a process it is not necessary to pass a closure (see section 4.2) to this scheduler between functions in a process or within the shared libraries. Instead a special, machine dependent, method exists to access the equivalent closure for the "current" thread and process. This method is typically to indirect on some especially reserved register. This allows library implementations to build on the functions in the virtual processor interface and those of section 3.5.1 to utilise any (or all) of the synchronisation schemes presented in section 3.6. The functions which are provided by the intra-process scheduler are known as the process methods.

The interface that was adopted within processes was to extend the use of events already present for interprocess communication, and provide primitives similar to [**Reed77**] namely event counts and sequencers. The events which are used in this way are entirely local to the domain and so are called *local* events. This distinguishes them from *outbound* events (those which can be propagated to another process using a system call) and *inbound* events (those which change asynchronously as a result of some other process issuing such a system call).

3.5.1 Event counts and sequencers

There are three operations available on an event count **e** and two on a sequencer **s**. These are:

read(e) This returns the current value of the event count **e**. More strictly this returns some value of the event count between the start of this operation and its termination.

await(e,v) This operation blocks the calling thread until the event count **e** reaches or exceeds the value **v**.

advance(e,n) This operation increments the value of event count **e** by the amount **n**. This may cause other threads to become runnable.

read(s) This returns the current value of the sequencer **s**. More strictly this returns some value of the sequencer between the start of this operation and its termination.

ticket(s) This returns the current member of a monotonically increasing sequence and guarantees that any subsequent calls to either **ticket** or **read** will return a higher value.

In fact there is little difference between the underlying semantics of sequencers and event counts; the difference is that the **ticket** operation does not need to consider awaking threads, whereas the **advance** operation does (therefore it is wrong for a thread to **await** on a sequencer). The initial value for sequencers and event counts is zero; this may be altered immediately after creation using the above primitives.

These event primitives may be used for many sorts of synchronisation and communication both intra-process and inter-process; this is shown below.

The representation of time as a monotonically increasing event count makes it very easy for intra-process schedulers, already operating in terms of events, to provide time based control of events. Two additional operations are supported. These are **await_until(e,v,t)** which waits until event **e** has value **v** or until time (event zero) has value **t**, and **sleep_until(t)**.

By convention an **advance** on an outbound event will cause the new value to be propagated by issuing the requisite system call. Only **read** and **await** should be used on incoming events as their value may be overwritten at any time.

3.6 Concurrency primitives using events

In contrast to many other systems where implementing one style of concurrency primitives over another set can be expensive [**Birrell93**, **Fairbairns93**] it is very efficient to implement many schemes over event counts.

3.6.1 SRC threads

In SRC threads [**Birrell87**] concurrency is controlled by two primitives. These are mutexes and condition variables. A mutex is held for a short period of time to protect the examination of some state. A thread may then decide it needs to block while waiting for something to happen. It does this by blocking on a condition variable (which atomically releases the mutex). When some other thread has changed the condition, then it signals the waiting thread to wake up. A thread must always check the condition again inside the mutex as some other thread could have made the condition false again in the mean time.

3.6.1.1 Mutex

An implementation of a mutex over event counts and sequencers has two fields **m.s** and **m.e**. These are used as a sequencer and an event count respectively. A mutex initially has the sequencer with value zero, and the event count with value one. Then the implementation becomes:

```
lock(m):
        await(m.e,ticket(m.s))

release(m):
        advance(m.e,1)
```

3.6.1.2 Condition Variables

The operation of condition variables is almost as simple. I will first give the implementations and then argue their correctness. A condition variable contains the same fields as a mutex, namely an event count **c.e**, and a sequencer **c.s**.

```
wait(c,m):
        t = ticket(c.s)
        release(m)
        await(c.e,t)
        lock(m)

signal(c):
        advance(c.e, 1)

broadcast(c):
        t = read(c.s)
        f = read(c.e)
        if (t > (f + 1))
                advance(c.e, t-f-1)
```

Note that, in **wait**, since the sequencer value is obtained inside the mutex, threads will always be awoken in the order in which they were blocked. This means there is no out of order race when there is a simultaneous **wait** and **signal**; no thread could be re-started in preference to the one already blocked. Also the wake-up-waiting race is handled simply because if a blocking thread is interrupted before the **await** but after the **release** then when it is restarted it will still **await** for the same value and hence not block.

Finally there needs to be some special discussion of **broadcast**. There may be some cause for concern due to the unlocked reading of both **c.s** and **c.e**. The important point to remember is that by the semantics of condition variables threads may be woken up unnecessarily. This means it is only necessary to show that *at least* the correct set of threads gets woken up. Any thread which has (already) got a ticket value in **wait** of less than the current value will be woken because the event counter will be incremented to at least that value. If any other thread **signal**s in parallel between the **read** of **c.s** and the **read** of **c.e** then **c.e** will be **advance**d and the **broadcast** will perform one less increment, but note that the **signal** will itself have incremented **c.e** once and hence unblocked one thread. Also note that as far as can be observed the **signal** could have happened before the **broadcast** (since no action of the **broadcast** is observable at that point). Note that once **c.e** is **read** the required set of threads is guaranteed to be started and even if they, or other threads, perform other **signal**s or **broadcast**s the only action is for the unnecessary wakeups to occur in the future (since **c.e** may exceed **c.s**).

3.6.2 Posix threads

Posix mutexes are similar to SRC mutexes except that Posix mutexes may be locked recursively. This makes them a little more complicated to implement. First, there must be a way (provided in the process methods) to find out a unique identifier for the current thread. Secondly, each mutex must have an "owner" field which is only updated under the mutex itself. One observation helps: if some particular thread holds a mutex then the owner value will be set to that thread and will be stable. If, however, a particular thread does not hold that mutex then the owner value, though potentially changing, cannot be set to the identifier for that thread. This leads to the following implementation:

```
plock(m):
        if (m.owner == me)
                m.refs ++
        else
                lock(m)
                m.owner = me

prelease(m):
        assert(m.owner == me)
        if (m.refs > 0)
                m.refs --
        else
                m.owner = NULL
                release(m)
```

Note that since release (actually the underlying advance) must ensure memory consistency at the point that mutual exclusion is terminated the **m.owner** field is guaranteed to be observable as set to NULL before any other CPU can attempt to grab the mutex. Thus this implementation works on a multiprocessor with a partially ordered memory system.

3.6.3 Wanda threads

Wanda uses Dijkstra counting semaphores for its concurrency primitives. As well as the **P** and **V** operations, Wanda supports timed waiting - both relative

TimeP and absolute **UntilP**. These return an indication of whether they were **V**'ed or whether they timed out. They can also be implemented efficiently over event counts again using a pair of local events, one as a sequencer **s** and one as an event counter **e**, viz:

```
V(sem):
        advance(sem.e, 1)

P(sem):
        await(sem.e, ticket(sem.s))

UntilP(sem, time):
        t = ticket(sem.s)
        await_until(sem.e, t, time)
        if (t > read(sem.e))
                advance(sem.e, 1)
                return TimedOut
        return 0

TimeP(sem, delta):
        t = ticket(sem.s)
        await_until(sem.e, t, read(0) + delta)
        if (t > read(sem.e))
                advance(sem.e, 1)
                return TimedOut
        return 0
```

These work in the obvious way; when the thread returns from blocking, it checks to see if it has been **V**'ed (i.e. if the event counter has reached the required sequencer). This may have occurred while it was returning from the **await_until** due to the time being reached. Under Wanda's semantics this is not considered to be a timeout. If it really did time out, then the value of the event count must be corrected by incrementing it.

Wanda also has a function called **Broadcast** which unlocks all the threads waiting on a semaphore. Unfortunately the semantics of this are not well defined – what is "all" in the context of a simultaneous **P** operation? Thus no code ever written for Wanda uses **Broadcast**[11] and so it is not necessary to provide an emulation for it.

One further problem with providing the Wanda interface to concurrency is that in typical Wanda code semaphores are not explicitly destroyed; they are allocated space on the stack, initialised, used, and then implicitly destroyed by returning from the scope in which they were declared. Such programming methodology is correct in Wanda because native Wanda semaphores, whilst idle, do not utilise any resources. When a Wanda semaphore is synthesised over local events, then each semaphore which is discarded consumes two such events permanently. A deallocation function would have been a null-operation on current Wanda platforms, but would have made the porting of Wanda code much easier.

3.6.4 Priority

The three synchronisation schema described all provide a sequential service to synchronising threads. A criticism might be, despite the efforts of section 3.1, that there is no support for priority.

Note, however, that any of the above could be recoded such that a waiting thread listed itself and its priority and then waited on an individual (private) local event. The waker could consider the set of threads waiting and advance the chosen thread's event.

In general this is how thread packages are implemented in order to avoid race conditions where the underlying scheduling control primitives are to block oneself or wake a particular thread. Examples include user space Wanda semaphores over the Wanda scheduler system calls, SRC threads over the Taos system calls [**Birrell87**], Posix threads implemented using Wanda threads [**Fairbairns93**], and SRC threads over Windows NT primitives [**Birrell93**]. Again it is interesting to consider how this complexity is not required when implementing over event counts and sequencers.

[11] There is in fact a single use in the Internet library, but it is adjacent to a comment which notes that it shouldn't be being used because of the problem noted!

3.7 Summary

This chapter has considered the problems with using priority to control the allocation of CPU resource within an operating system. Instead, a mechanism based on a guaranteed share of the CPU over a fine time scale has been proposed. Interprocess communication using events has been recommended as an efficient solution to many problems of asynchrony and non-shared memory multiprocessors.

A virtual processor interface has been described which allows a process to make its own internal scheduling decisions based on information about the availability of the CPU, the events occurring as a result of inter-process communication, and, where appropriate, hardware interrupts. Events have also been shown to be effective for intraprocess synchronisation and efficient implementations of traditional concurrency paradigms have been demonstrated.

Chapter 4

Inter-Process Communication

Fawn is a micro-kernel and uses a single virtual address space. As a result the sharing of code, and the invocation of services in different processes are both highly relevant. This chapter considers these issues.

The system makes considerable use of closures [**Saltzer78**] to access state rather than addresses built into the code. This allows for sharing of code. Inter-process communication uses shared memory for data and events for signalling.

Interfaces of services within the system, either in a shared library or in another process, are specified with machine-generated *interface references* which can be passed between processes. A binding function exists for initiating inter-process communication with a service, and a trading [**ANSA89**] function exists for naming interface references with human-readable names, types and properties.

4.1 Language

On setting out to design a new operating system the question of which language to use is always a consideration. In addition to the usual factors there were a number of others particular to this work.

- concurrency: The language must not make assumptions about how concurrency control operates, or (worse) that it doesn't exist. Nor must it embed concurrency primitives in output code which were not present in the source. It must be possible to implement concurrency control as deemed fit by the

Fawn system and not as the designers of the language guessed.

- exceptions: The Pegasus project had determined to standardise on the use of exceptions for failure conditions so exceptions were a requirement for inter-operation.
- closures: Since the operating system uses a single address space, the language must permit the use of shared code with closures used for accessing state and performing operations.
- comparability: Finally it must be possible to compare this work with other systems such as Unix or Wanda without language issues masking the performance impact of the design features.

Modula-3 [**Cardelli89**] was rejected due to lack of availability. The number of processor architectures supported is extremely limited, it did not support the platform intended for practical work, and showed no signs of being more available in the future - the most recent port took three years despite the authors of the language working for the company producing the new hardware.[1]

The C++ language [**Stroustrup91**] was rejected due to problems with the stability of the g++ compiler, and the tendency for the language to hide the performance and memory costs of the underlying operations.

The C language [**ISO90**] was adopted together with a portable exception mechanism using **setjmp**.[2]

Without compiler support, the possibility of exceptions contributes a certain amount to the costs of service invocation. Their advantage is that they localise the blame for erroneous code; an exception will halt the thread if ignored. As a result, within Fawn, error codes are usually returned for code within a self contained module and exceptions used across module boundaries.

[1] Recently much greater platform availability has become available, but this change was too late for this work.

[2] The exception mechanism was implemented in conjunction with David Evers of the Computer Laboratory.

4.2 Shared Libraries

If a service is sufficiently trusted by the client then it is possible that it can be implemented as a shared library. A shared library is defined as one consisting of *pure* code, i.e. it contains exclusively (read-only) fully-resolved **text** segment or fully-resolved references to other shared libraries.

All state manipulated by the library (even if opaque to the client) is passed into functions using closures [**Saltzer78**] except the closure for process and thread specific functions as described in section 3.5.

Some library routines may operate entirely without state and have well known names (such as `strlen`). Other examples of such functions are ones which create closures for subsequent use. For these stateless functions location in a shared library with early binding (i.e. direct linking with the textual name) is used in Fawn rather than the later binding of [**Roscoe94a**].

A closure is a pair of pointers. One of these pointers points to a table of methods, the other points to the state record on which these methods operate. The state record may contain any private or public state including other closures.

[**Roscoe94a**] gives a full discussion of the use of closures and shared library modules within the Nemesis operating system. Within the Fawn system a slightly different scheme was used. This was due to this work preceeding the implementation of the necessary support tools in the Nemesis system, and also due to various bugs in the binary tools for the chosen platform (the Arm processor).

When a shared library is built all the separate object files are linked together at the appropriate virtual address and then a tool called `makeabs` is run over it which converts the public symbols[3] to absolutes. It can then be used as input for either the image builder `mkimage` (a boot image must contain the shared libraries used by the processes in the image) or the stub generator `makestubs`. The latter creates a static library archive with one stub object file per public symbol in the shared library. The code for each stub consists of a "far jump" so that the library may be placed anywhere in the address map without necessarily being in reach from client code with a processor branch instruction. Base addresses for shared libraries are allocated by a (human) administrator; code which is under development and likely to change frequently is placed in static rather than shared libraries.

[3] Also known as external symbols.

This shared library mechanism may be permitted to include code which requires external routines, provided that those routines are in another shared library, and that there is a partial order of shared libraries (any library with no dependencies on other libraries is a bottom element, libraries with dependencies are higher than the libraries on which they depend). An example is the marshalling code and stubs libraries for various interfaces which relies on the basic C library.[4] Programs should be linked against shared libraries in order from lowest to highest[5] to avoid indirecting on stubs more than once; this is the opposite requirement to Unix.[6]

As a result of this construction it is possible for a program to have a routine which has a name clash with something in a shared library which it uses, and still be able to use the shared library. Other uses of that function within the shared library are not affected.

4.3 IPC Model

The mechanism for local (i.e. same machine) RPC has been the subject of a great deal of research and experimentation over the last few years. For the Topaz micro-kernel, [**Bershad89**] reports that almost 95% of RPC is local, so in a highly decomposed (micro-kernel) system its performance can be crucial to the overall performance.

4.3.1 Trust

In a conventional monolithic kernel such as Unix, the various different parts make assumptions of trust about others, and user processes make assumptions of trust about kernel services. Within the kernel, servers make only rudimentary checks on arguments, and trust clients with shared memory. Similarly, clients trust the server to return the CPU. Likewise non-kernel clients trust kernel services with their memory, and that they will return the CPU.

[4] Some of the code for the basic C library was ported from the Wanda library which was written by various authors including Joe Dixon, Sai Lai Lo and Tim Wilson of the computer laboratory.

[5] In the partial order sense.

[6] This is because stubs for a lower level library which have been included in a higher library will be public symbols, and so will themselves have stubs in the stub archive for the higher library.

One of the advantages of a highly decomposed system is that the modularity improves the isolation of faults and debugging. Frequently, however, it is reasonable to consider re-introducing some of the trust found in a monolithic system for crucial or stable code. This is similar in concept to compiling a module without debugging code. For example, if a client is willing to trust a server, which itself only needs read-only access to non-secret state to implement the operation, to return the CPU, then the server may be implemented as a shared library. There are potentially many different types of trust between parts of the system and it is the responsibility of the trading and binding services to ensure that the same interface is presented irrespective of the exact implementation.

In most systems designed to achieve high performance, local RPC is implemented using pairwise shared memory which is allocated at bind time. This may be a single block of memory mapped read/write to both processes, or two areas of memory each mapped writable by only one process. The size of the memory is determined at bind time. Of course, although this may be the defined semantics the implementation may differ; all such buffers may be writable by all processes, and some platforms may lack protection hardware.

4.3.2 Migrating model

In [**Bershad89**] a migrating model for local RPC is described. In this model (also used by Spring [**Hamilton93**] and in some versions of Mach [**Ford94**]) the thread of control which makes the call in a client is transferred via the kernel and up-called into the server process. The client loses its thread until the server returns control. If the server terminates, or fails to return control after some long period, then the kernel creates a new thread to handle the fault condition in the client process. This scheme works best with kernel threads. In some systems the designers note that the remaining cost can be significantly reduced by avoiding the kernel scheduler on this path (charging the resource usage of the server to the client on whose benefit it is working), thus reducing the overhead still further. For Spring this path was reduced to 123 instructions[7] in the common case.

This model does have impressive performance statistics (especially given kernel threads), but has a number of problems. First, the client thread may be blocked for a substantial period of time. Second, the server may not use the upcall solely

[7] 102 in the kernel and 21 in user space, however this cost does not include the delayed effect of register window traps estimated at 300 cycles each in [**Sun94**].

for performing work for that client; the implicit billing may be highly inaccurate. As was noted earlier the use of upcalls can lead to loss of control over and accountability of scheduling. Third, there may be difficulties in the event of a failure somewhere in a long chain of interprocess calls; the various calls may not be timed out in the optimal order, preventing intermediate servers from performing intelligent recovery on behalf of their client. After careful consideration of this resource recovery problem, the designers of the ARX operating system [**McAuley94**] prohibited nested RPC by adding the constraint that no thread which had already made a migrating RPC call could make a further such call; equivalently replacing n kernel context switches with $n-1$ user ones. This problem of resource recovery is also found in equivalent capability based systems where services exist independently of processes and are invoked using "enter capabilities". A full discussion of the complex ramifications of this model may be found in [**Johnson81**].

Additionally, and particularly on a multiprocessor system, the cache contents on the migrating processor may be adversely affected. Finally, the model of a blocking RPC is inappropriate for the implementation of various real time activities and device drivers, which require non-blocking means of communication without thread overhead for hardware or performance reasons.

4.3.3 Switching model

The switching model is found in more conventional systems such as Unix and Wanda. In this model, the client signals to a blocked server thread that it should wake up and perform an operation. The client thread may or may not block under its own control. In a variant called scheduler activations [**Anderson90**], the thread blocks but control of the CPU is returned to the process' internal scheduler. The client may then at some later date block on or poll for the indication from the server that the operation is complete.

This model is recommended in [**Bershad91**] for multiprocessor machines. That work also notes that this model is superior when threads are provided at the user level (either over a non threaded or a kernel threaded base) due to the lower synchronisation costs. Although this model has a higher cost for local RPCs when measured simplistically, it does give the client much greater flexibility, and can in fact improve performance by batching up multiple requests to a server so that when it runs it will have better cache and TLB performance. Various examples of similar performance improvements on a multiprocessor machine may be found in [**Owicki89**].

The switching model is used in Fawn. Section 7.1.4 discusses a potential performance enhancing heuristic where the kernel scheduler keeps a hint of the last process which has been the recipient of an event, and when re-allocating the CPU, it will favour that process if it has CPU allocation remaining in that jubilee.

4.4 IPC Operation

4.4.1 Architecture

The architecture of the IPC system follows that of [**ANSA89**] and [**Evers94b**] and is connection oriented; clients must explicitly bind to servers before operations on interfaces can be invoked. Interface definitions are written in the MIDDL language [**Roscoe94b**]; the mapping onto the C language is known as MiddlC. Stubs are generated automatically by a stub compiler[8] whose back end is written in python, making for a quick prototyping language. The marshalling is procedural and the stubs are generated per-transport so they can take advantage of the local data and call mechanisms. The data representation is based on that used in MSDR [**Crosby94**].

4.4.2 Calling conventions

The calling conventions for MiddlC interfaces are summarised in table 4.1. The C language only supports returning a single value directly so, for operations with more than one result, pointers to the variables to receive the results are passed as additional arguments.

Only the creator of some information can allocate the correct amount of space for it, thus the client must allocate space for arguments and the server for results. What is more important is the consideration of when that memory gets released. For arguments, the client may release (or alter) the memory any time after the server returns, thus the server must copy any part of the arguments which it

[8] The front end of the stub compiler was written by David Evers and Timothy Roscoe of the Computer Laboratory.

wishes to keep in internal state. On the other hand, any memory used by results returned from the server becomes the responsibility of the client, so the server may need to allocate memory and copy its internal state into it in order to return it to the client. In both these cases there is potential for an unnecessary copy of data. Also in the latter case there is the inherent problem of the client knowing the correct heap closure to free the memory on.

	Size	Memory Allocated	Freed
argument	Small	(by value)	
	Large	Client	Client
result	Small	(client's stack)	
	Large	Server	Client

Table 4.1: Calling conventions for MiddlC

This is particularly inefficient in the RPC case, because the server stubs must allocate memory to hold the unmarshalled arguments which will then be copied by the server. Likewise the server will allocate memory to put the result into which will immediately be freed by the server stub.

These problems would be eliminated if sharing semantics were adopted between client and server with memory being reclaimed by garbage collecting methods as is used in the Network Objects distributed system [**Birrell94**]. In such cases sharing of the memory is reasonable since if one trusts the server or client to be in the same process then it is reasonable to trust it to share correctly the information; in the remote case it is reasonable to trust the stubs. This problem is similar to the problems with **mbufs** discussed in section 5.1.1.

As a corollary of these complications, it is necessary in MiddlC, that all exceptions are shallow - which means that none of the arguments to an exception must need memory to be allocated.

4.4.3 IPC Low Level

An IPC channel is implemented using a pair of memory areas. One, for arguments, is mapped read only into the server; the other, for results is mapped read only into the client. To signal that a set of arguments or results has been placed in the appropriate buffer there is also a pair of event channels between the two processes. When a client makes a call on a server the arguments are put in the argument area and the outgoing event is incremented. The thread then, typically, waits for the incoming event to be incremented, with some per-binding timeout. When it is awoken it reads the results from the other buffer.

ANSA and MIDDL have the concept of an announcement, as distinct from an operation. An announcement is a call which has no results and which can be considered to have at-most-once semantics. For announcements, the server uses the returning event to indicate when it has finished reading the arguments. The caller need not block unless it tries to re-use the same IPC channel before the server has read the arguments. Since announcements and operations are supported on the same interface, the low level IPC primitive to prepare the argument buffer also waits for the acknowledgement of the previous call. If the previous call was an operation then this is satisfied trivially.

The IPC low level provides an encapsulation of the IPC channel, its buffers, and event counts, with three operations. These are **prepare_tx**, for preparing the marshalling state record and ensuring that the server has finished with the previous call's arguments; **prepare_rx**, which waits for an indication that new data has arrived and initialises the unmarshalling state record; and **send**, which increments the outgoing event. Note that at this low level there is no difference between the active (i.e. client) and passive (i.e. server) sides of an IPC channel, they are entirely symmetric. A **send** is used by the higher levels to send arguments, to send results, or to acknowledge an announcement.

4.4.4 IPC Stubs

The machine generated stubs operate on public fields in the IPC channel state record. These are the current marshalling pointer, the remaining space and a closure for a heap to be used for memory for out-of-line structures. The marshalling fields are initialised as described above. Marshalling of simple types is done by macros which operate on the IPC connection state record. Marshalling for standard complex types is provided in the standard IPC shared library; the stub compiler generates procedural marshalling code for user defined complex types.

4.4.4.1 Client stubs

The client stubs provide a stub closure for their associated interface. The state record of this closure contains the IPC channel information which is used to forward any function requests to the server. The methods record contains function pointers to the individual stubs for each of the operations or announcements in the interface. A client invoking methods on an interface need not know whether the implementation is in the same process or not.[9] Since the stub operations operate on closures the code is pure and and can be shared between all clients of interfaces of that type.

4.4.4.2 Server Stubs

The server stubs consist of an IPC connection closure, and the closure of the actual implementation. A thread (created by the server at bind time) waits inside the stubs for the event which indicates that a new argument has been marshalled. When the thread awakes it dispatches on the first integer in the buffer to determine which operation number in the interface is being called. Individual functions exist for each operation which unmarshall the arguments, call the server's implementation via its closure, and then marshall and send the results. The structure of the outer code is a loop with exception handlers outside it. A further loop surrounds this. Such an arrangement means that only one instance of the code for handling a particular exception exists per interface. Also the cost of the **TRY** is not executed in the common case. This is illustrated in figure 4.1. Stubs whose server implementations may raise exceptions and which have allocated memory from the IPC's heap use a **TRY FINALLY** block to ensure that the storage is reclaimed.

Since the stubs invoke all operations via closures, they are shared between all implementations of a particular type of interface.

4.4.5 Binding

There is a special process in the system known as the Binder. For convenience this process is combined with the Basic Resource Allocator. When a client wishes to make an IPC connection to a server it calls the binder specifying the interface

[9] Except that in the remote case the stubs may raise RPC engineering failure exceptions.

```
loop
        try
                loop
                        wait for ipc
                        dispatch to function specific stub
                endloop
                catch exception 1
                catch exception 2
                ....
        endtry
endloop
```

Figure 4.1: Server stub dispatcher pseudo-code

reference of the interface that it wishes to connect to and the indices of a pair of events it has allocated. The interface reference contains a process identifier which the binder uses to call back on an IPC connection registered by that server process. As well as passing the interface reference, the binder passes in the sizes and addresses of the memory areas allocated. The call back code at the server has the option to vet the client, or the sizes of the memory buffers chosen before returning the event numbers that it has allocated. If the call-back returns successfully then the binder will update the kernel event tables to connect the pairs between the endpoints and return to the client. The IPC connection is then established.

4.4.6 Service Management

Within each process is a module known as the **object table**[10] which mediates between that process and the binder. It is this with which all services in a process register, and which all clients use to request connection to such services.

When a server wishes to export some interface it registers the interface with the object table together with a closure for a call-back function. The object table returns an interface reference which the server may then publish. The call back is activated when a request from a client occurs. The call back function has the last word in vetting the client, and may create any per-binding state required

[10] This name comes from [**Evers94b**].

within the service. It returns the actual closure for the service to the object table which is subsequently used by the server side stubs. All the other work of setting up the IPC channel is done by the object table.

The object table keeps internally a note of all the services which exist within its process. When a client requests an IPC connection to a particular service then the object table checks to see if that service is in fact local. If so then it will return the actual closure directly to the client. If the service is remote then the object table makes the request via the binder, and instantiates a set of client stubs of the appropriate type (a closure for the interface for creating stubs of the correct type is handed in by the client requesting to connect to the server of that type).

4.4.7 Trading

What needs to be named in an operating system? In what respects are the interface naming system and the file naming system integrated? How much control does any particular process have over its own view of the name space?

Considering the Unix name space for a moment, we can see that the file system name space is used, not just to name files but also to name interfaces. Interfaces named within the Unix name space are character and block special devices (these are a particular class of interfaces which have operations that are similar to the operations that can be used on a file, but they are interfaces nonetheless) and named pipes which explicitly name interfaces behind an IPC channel. Furthermore, in more recent versions of Unix, files such as */proc* again name things which in some ways are closer to interfaces than files.

Again considering Unix, even for files themselves, it is considered normal to have files of different types (data type, directory type, and symbolic link type) named in the same context, with a *stat* operation being used to distinguish them.

Further, it is normal for NFS fileservers to prohibit the creation of character and block special files[11] by clients of exported file systems. This is for security reasons. In other words there may need to be a restriction (in general) on the sorts of things which can be published in any given context.

Finally, Unix is not clear on differentiating the type of an interface from an

[11] named pipes are usually permitted since they do not affect security but don't work remotely.

interface reference (i.e. a particular instance of service of that type). This is one of the reasons for the somewhat esoteric naming used on Unix sockets.[12]

Following the ANSA model, the system described here provides a traded name space for both services and files. Any number of **Traders** or Name Servers exist in which appropriately authorised clients may store interface references for objects, and various ancillary information such as the type. Traders may also implement directories and symbolic links. A trader service may choose to satisfy requests any way it wishes, and may in fact itself consult various other traders. In the ANSA model this is known as federation.

Each mount point which a trader is prepared to export to clients desiring a naming context is represented by an IPC endpoint. The context within the name space rooted at that point is used as an argument to operations. This keeps the number of IPC endpoints exported by a trader to a manageable amount. Also with the exception of a symbolic link which needs to be looked up in the client (to allow crossing of mount points) this permits a trader to resolve an arbitrary number of components in a path name in a single operation.

4.4.8 Name Spaces

The system provides for processes to configure their name spaces as they wish. Various traders may be mounted at various points in the name space of the process as the process sees fit. This is similar to Plan 9 from Bell Labs [**Pike91**].[13]

Merging of file names provided by separate traders within a single directory (as provided in Plan 9) may be implemented using a proxy trader.

[12] Another example is the instantiation muddle in the Unix packet filter code, where the special device open function modifies the minor device number of the inode of the file handle which has been opened from the minor device number of the special file in order to ensure that it is informed about subsequent opens by other processes. This is required because the packet filter code must keep per-file-descriptor state.

[13] The Plan 9 implementation is rather restricted by a lack of the intermediate interface reference concept. This leads to an inability to export a configured name space from one process to another; their system relies on a per-user configured name space generated from the `.profile` file. This is likely to be addressed in a forthcoming system named Brazil [**Trickey93**].

4.4.9 Restriction of Name Space

In some systems it can be the case that the owner of a process may wish at the time of its creation to restrict the name space which is visible to that process. In Unix this is known as the **chroot** operation. In Fawn there are a number of ways of performing this operation.

Any process being created must inherit some form of name space from its parent. The parent could populate this name space with proxies which ensure that the lookups do not move above some particular point. Alternatively some traders could, on request, create a mount point for this purpose.

All of these rely on the interface references for objects in the system being unguessable. This is because the system does not enforce any special mechanisms on the passing of interfaces between processes. This is in contrast to systems such as Spring [**Hamilton93**] where the passing of *doors* between processes requires special consideration on the part of the operating system kernel. Even if an unprivileged (**chrooted**) process had a proxy-binding service interposed between itself and the real binder this does not permit a solution. Consider the case of a server which the process does have access to. The server may create a new interface reference for use by that process. There is no computationally efficient way for the proxy binder to decide whether that interface reference should be blacklisted or not.

The Pegasus project has now adopted unguessable interface references in order to permit name space restriction.

4.5 Bootstrapping

A number of special operations occur when the system boots to allow normal operation to begin. A boot image is constructed from a kernel and a number of processes. A special tool known as `mkimage` constructs an image which is capable of running. By convention the first two processes in the system are the console daemon and the binder respectively. Mkimage initialises the kernel's event tables so that every domain in the system shares an event pair with both of these special processes.

4.5.1 Binder

On system startup the Basic Resource Allocator (or Binder) uses a very basic simplex channel over the pair of events setup by `mkimage` to transmit the addresses and sizes of the memory areas it has allocated for IPC to each process. Once this information has been sent the same event pair is used for controlling a normal IPC channel between the process and the Binder as described in section 4.4.5. This is initially used by servers to initialise the call-back channel needed by the object table.

4.5.2 Trading

Two of the operations provided by the binder's interface are to set and get the interface reference of the bootstrap trader. Any process requesting the interface reference is blocked until it has been set.

4.6 Other communication

This chapter has discussed the provided IPC mechanism, the next chapter discusses high performance I/O support. The use of events as inter-process synchronisation primitives, however, enables many other types of communication. Examples include circular buffers (which may be used for a `stdio` stream) and versioning.

Versioning is when a process makes a tree data structure publicly available and performs updates to that tree by providing altered records from any change upwards as far as the root of the tree. A similar data structure was used for support of some atomic operations in the Cambridge File Server [**Needham82**]. Such a server can then send an event to inform the clients to change to the alternate root pointer. Clients acknowledge the change by using an event in the other direction, freeing the server to make subsequent updates. Such a mechanism may be most suitable for name servers.

In any communication scheme devised the event channel is used to indicate the synchronisation point between the users. This localisation of the memory system coherence and synchronisation ensures efficient access to the shared memory. This is in stark contrast to systems such as Wanda, where to obtain consistency on a multiprocessor, code must be compiled with the `-f volatile` option which forces the compiler to consider all memory accesses as referencing volatile variables.

4.7 Summary

This chapter has considered the impact that programming language can have on IPC and RPC systems. The C language together with an exception mechanism was adopted. The MIDDL interface definition language and MiddlC calling conventions were adopted for compatibility with the Nemesis system.

Closures were adopted for efficient operation in a single address space, and a shared library mechanism was described which uses automatically generated stubs for shared functions. The shared libraries may make use of other such libraries in a partial order manner.

A model for an IPC system in the context of the previous chapter was developed. Consideration was given to both the migrating and the switching models. The migrating model was considered inappropriate because it led to problems in accounting for CPU, poor cache locality, and an unacceptable interface for device drivers (or other processes which require non-blocking communication primitives). The transport uses pairwise shared memory and event channels.

The binding and naming schemes were described; based on interface references and trading concepts from the standard literature. Finally some issues to do with bootstrapping and efficiency of calling conventions were discussed.

Chapter 5

Input Output

The inter process communication mechanism described in the previous section fits the needs of RPC quite well, but is not appropriate for stream bulk transfer of data. Before presenting the scheme adopted in Fawn some other methods will be presented.

5.1 Previous Schemes

5.1.1 Unix

In Unix I/O is handled differently depending on whether the data is disk based or network based. Inherent in the system is that the paging code must deal in blocks that are the same size as file system blocks.

For disk based activity data is stored in blocks whose sizes are powers of two. The size of the disk "sectors", is not larger than the size at which allocation and requests are done on the block special device which is in turn not larger than the page size or the size of buffers in the buffer pool. In BSD derived Unix (specifically Ultrix) these are 512, 2048, 4096 and 8192 bytes respectively.

For network activity, memory buffers are allocated using **Mbufs** [**Leffler89**]. Mbufs are a two level hierarchy of linked lists. One list is used to indicate a sequence of packets on some queue, the other to indicate a sequence of mbufs which store the data for one particular packet. An mbuf usually contains a controlling header and a small number of bytes (e.g. 108) but for larger amounts can also be "extended" in one of two forms in which case a normal mbuf is used to administer the larger area of memory. The first form is a cluster mbuf in which the data area is a page allocated from a pool managed by the virtual memory system. The second form, known as a loaned mbuf, is a kludge added for NFS support where the data part of the mbuf is actually pointing into some private memory (usually the disk buffer cache). In that case the header mbuf contains a function pointer which must be called when the mbuf is freed.

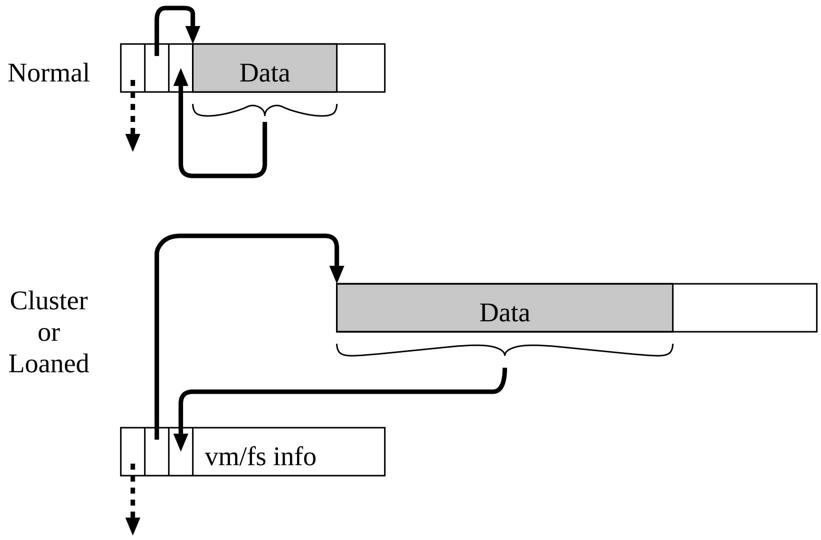

Figure 5.1: Unix Mbuf memory arrangement

Mbuf control fields include a length and an offset into the data where the logical start is. For normal mbufs the data may be modified, but for cluster or loaned mbufs the data is read only. Both the length and the offset of the data are specified in bytes and there are no alignment constraints.

When data is being transmitted, the semantics are defined such that the receiver of an mbuf may alter it as it sees fit (subject to the above constraint) and frees or reuses the mbuf when transmission has taken place.

In Unix all data is copied between user space memory and kernel buffers. On system calls the user process gives the kernel a variable length array of pointer length pairs which make up the request. The kernel copies these into or out of disk blocks or mbufs appropriately. When copying into a chain of mbufs the socket level implementation makes a heuristic decision about whether to allocate normal or cluster mbufs. The heuristic is that if more than 1024 bytes[1] remain

[1] This value is given for Ultrix, other systems may use different heuristics.

to be copied then a cluster mbuf will be allocated. This copying of the data into kernel memory is quite expensive. It is not the only copy operation however...

Since the device driver may alter the mbufs, any transport protocol which may need to resend the data, must copy the mbuf chain before giving it to the device driver. For normal mbufs this means allocating a new mbuf and copying the contents. For cluster mbufs this additionally entails entering the virtual memory system to increase the reference count on the page frame underlying the cluster. For loaned mbufs there is no way to increase the reference count as only a freeing method is stored in the normal mbuf header. Instead a block of memory is allocated by the virtual memory system (from the NFS pool) and the entire data area copied into it. This gives a loaned mbuf with the memory loaned by the virtual memory system rather than the file system; further copies of this new mbuf will still require further copies of the actual data.[2]

Protocols add headers to the chain by placing the header in an mbuf which is inserted at the head of the list.

Furthermore, on many modern platforms, many devices (excepting the most expensive and higher bandwidth) are not able to interface well with wide memory busses [**DEC91**, **DEC92**] and so data is copied again in the device driver into a format which can be accessed by these devices.

This means that data is potentially copied in software three times between user space and being acceptable to device hardware.

On the receiving side device drivers typically must copy the data into mbufs before passing it up to higher level software to decide whether it should be accepted or not. Further, this higher level software runs at a lower (software) interrupt priority and thus the mbufs are placed on a queue by the device drivers. Under periods of overload this queue can overflow leading to wasted effort. Additionally, the device driver has no way of knowing what proportion of each packet is headers and what proportion is user payload - leading to some additional copying. Finally, the data will be copied from the kernel buffers into the user space memory.

Various research work has considered the feasibility of using page remapping

[2]In OSF1 there is an additional "external" structure (allocated in an area of memory which is neither part of the mbuf header nor the cluster/loaned data) which contains the free method for the external memory and a doubly linked list of all the headers which refer to this data area. This reduces these costs somewhat with the tradeoff of the additional allocation and management of the external structure.

between user space and kernel buffer for data which happens to be page aligned. This can be complicated due to the asynchronous nature of transmission and the synchronous nature of acknowledgement for data moving operations on common transport protocols such as TCP. Copy on write schemes and/or the pinning of pages in the virtual address space, can be used if the complexity overhead is low enough. Unfortunately the cost of this page flipping is frequently greater than the cost of the copy. This is especially the case on a machine with a virtually addressed cache or on a multiprocessor; [**Dixon91**] notes that coherence of TLBs in a multiprocessor is usually not addressed at all (unlike the concern for memory coherence), necessitating an inter-processor interrupt whenever page remapping takes place.

One final problem with Unix is that for protocols with many small packets (e.g. audio) a system call is required for the process to collect each such packet. This can be a major performance penalty for such protocols.

5.1.2 Wanda

Buffer management in Wanda picks up many attributes of the Unix system and some ideas from the Topaz system [**Thacker87**, **Schroeder89**]. Communication primitives in Wanda are discussed more fully in [**Dixon91**].

Wanda has a set of kernel managed buffers called IOBufs. Each IOBuf is of fixed size; various sized buffers are available. An IOBuf is some multiple of the page size of the machine.[3] Applications may acquire and free IOBufs. The **send** operation takes as an argument a single IOBuf and the **recv** operation returns a single IOBuf. The semantics of IOBufs define that the IOBuf is mapped into the address space of the owner and unmapped when the IOBuf is sent or freed. The implementation, however, is that all of the buffers are globally accessible all of the time.

IOBufs are identified by their number, which is an index into a read only table of pointers. Each pointer points to a read only IOBuf control structure which records the actual address of the data area, the owner and (if the IOBuf is being manipulated within the kernel IPC system) the offset and length of the data. Within the data area the first two words are special, containing the user process's view of the offset and length. When an IOBuf number is passed from user to kernel, the kernel checks that that process owns the buffer and that the offset and

[3] Wanda is vaxocentric in assuming that the page size is 512 bytes.

length fields are valid. When an IOBuf is passed from kernel to user the users fields are updated.

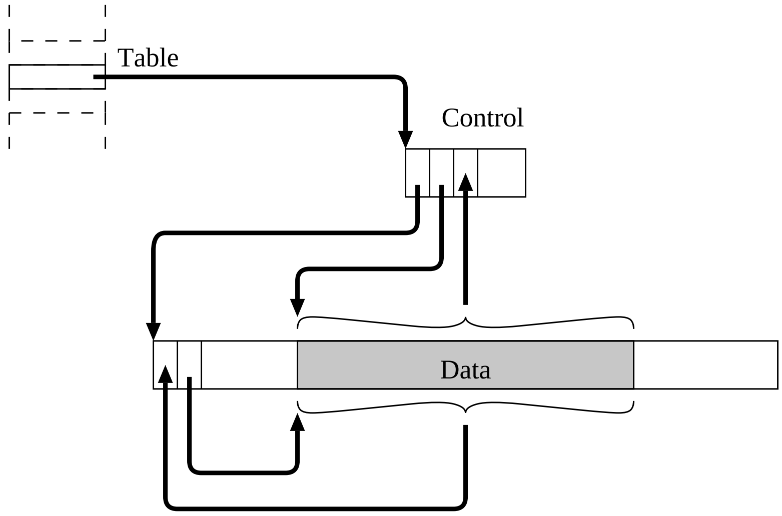

Figure 5.2: Wanda IOBuf memory arrangement

Although IOBufs are of fixed size, support for headers is enabled by leaving an area unallocated at the front of the buffer when it is requested. Thus headers may be added to the front of a contiguous area of data by reducing the initial offset. On Wanda the amount of this initial offset is 128 bytes - enough for substantial headers.[4]

Like Unix, Wanda also suffers from the problem that any transport protocol which may need to send the data more than once must copy the IOBuf before it is sent. This is slightly mitigated by the fact that the data is guaranteed to be aligned (in the worst case on Unix it may not be). Also RPC is the normal paradigm for programming on Wanda rather than use of transport protocols; RPC systems prefer to be in control of their own retry policies depending on the semantics of the operation.

On reception Wanda's performance relies on two aspects. One is a generalisation of a result noted in [**Bershad89**], that RPC programmers tend to ensure that interfaces are such that arguments and results fit in a single Ethernet packet. Since Wanda machines are usually used for specific services the sizes of the buffers present in the system may be configured appropriately. For example, the Pandora video file server [**Jardetzky92**] configured the kernel to have many IOBufs of the

[4]The "fat cell" packet format on the Ethernet uses 106 byte headers.

same size as video and audio segments.

The other aspect is that for cell based networks, the use of the MSSAR protocol [**McAuley90**] allows the size of the packet to be determined from the header of the first cell; Wanda allocates an IOBuf of sufficient size. Recently, to support AAL5 [**Forum93**], where this indication is not available, the Wanda MSNA implementation has been enhanced to allow an application to hint to a device driver the expected size of arriving packets.

Like Unix, Wanda can experience live-lock problems if data is arriving faster than the application can consume the IOBufs. Also there is no resource control on the number of IOBufs that can accumulate for a process; an occasionally observed fault condition on Wanda machines is where data is arriving for a process which is not reading from that socket (due to a bug or lack of priority) which eventually consumes all of the available buffers. This is a similar problem to priority inversion.

5.1.3 Fbufs

Fbufs [**Druschel93**] are another example of operating system research on buffer management. An Fbuf is a page of memory allocated from a reserved area of the virtual address space of the machine which is common across all processes. The system notes the difference between Protocol Data Units (PDUs) and Application Data Units (ADUs); thus a transfer from one domain to another is in the form of an aggregation of Fbufs each of which can have configurable length and offset. Figure 5.3[5] shows an example aggregation. This aggregation is most similar to a Unix mbuf chain consisting solely of cluster mbufs. Like IOBufs, when an Fbuf is sent it is unmapped (or reduced to read only access) from the sending domain and added to the receiver.

The designers suggest a number of optimisations to improve performance. First they note that Fbufs are likely to have locality in the paths they travel between various domains in the system. If this path is converted to a ring (i.e. Fbufs are returned to the originator) then some free list management has been avoided and also the buffer need not be cleared; security is not compromised by returning data to the domain that created it.

Another potential optimisation is to declare Fbufs to be volatile; the process

[5] This figure is based on a diagram in [**Druschel93**].

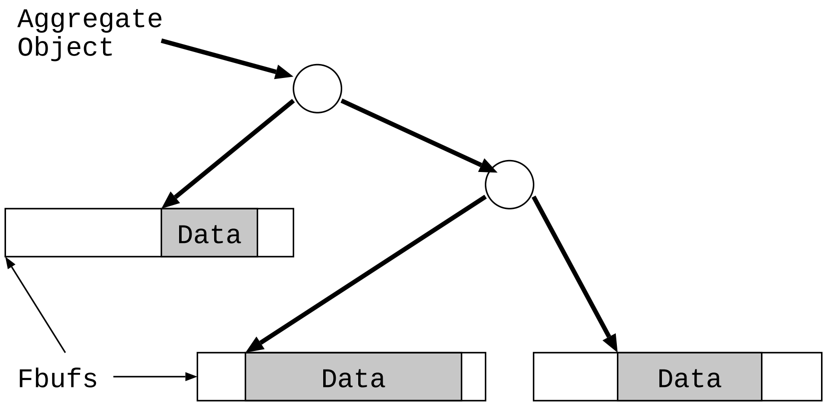

Figure 5.3: Fbuf memory arrangement

which initially created the data retains write access to the Fbufs and so they may conceivably change whilst being read by a receiving process. In many cases this is not a problem - for example since a process must trust the device driver to correctly give it the data which arrives for it, trusting the same device not to corrupt the data after sending the Fbufs to it is reasonable.

Fbufs provide high performance access to buffer memory for network I/O as exemplified in [**Druschel94**]. However there are a number of disadvantages. First there is no resource control on the number of Fbufs a process may collect as a result of I/O activity. The Fbuf area is pageable and hence an access may result in a page fault which may be unacceptable for device drivers (especially if it is the disk device driver which takes the fault). Furthermore there is no support either for applications to specify the way in which they want received data to be partitioned between Fbufs, or for an Fbuf aggregate to be given to multiple receivers (e.g. for multicast packets).

5.1.4 Application Data Unit Support

There are a number of previous schemes for support of ADUs within device drivers. The general idea is that the device driver will arrange for the data to arrive in a manner that will reduce the cost of its presentation to the receiving user process.

5.1.4.1 IP Trailers

The IP trailers [**Leffler84**] scheme found in some versions of BSD Unix used a set of special Ethernet type codes to indicate that the "headers" of the IP packet[6] had been placed at the end instead of at the start. A range of Ethernet type codes was used which indicated the number of 512 byte blocks of user data in the packet. The purpose of this mechanism was to allow the receiving machine to use page-flipping instead of copying. Unfortunately it was heavily vaxocentric assuming that all machines had 512 byte pages and that user processes' requests were page aligned. For many machines it was inappropriate since the cost of reorganising the virtual memory map far exceeded the cost of copying.

The principal reason for moving the headers to the end was that they were variable sized (the TCP header is 12 bytes longer than the UDP header). However it would have been much simpler merely to pad the IP header of UDP packets with 12 bytes of dummy IP options (appropriate padding options already exist). This would have converted the variable sized headers into fixed size ones, allowing the receivers to configure their receive buffers appropriately. This would have avoided the implementation costs in many receivers for whom the trailers protocol was no benefit in any case. This padding scheme is to be used in the network interface for the Desk Area Network [**Pratt94b**] where an internal message may need to be converted into an external IP datagram. The IP (and UDP/TCP) headers are padded to fill a 48 byte ATM cell so that the user data remains cell-aligned.

5.1.4.2 Optimistic Blast

The Optimistic Blast protocol [**Carter89**] tried to avoid data copying by assuming that the transmission medium is likely to be idle. The data packets are sent back to back on the network; when the first packet arrives the receiver assumes that the blast protocol is being used and changes the descriptor rings for the Ethernet chip to put the following packets in memory in such a way as to avoid copying. When contention arises the data must be copied. This scheme has the problem that it only works for fixed sized packets, the device driver must know which protocol is in use (in the implemented case only one protocol was supported) and the first packet of a blast must still be copied.

Another problem with this implementation was that the device driver attempted

[6] In theory it could be used for any higher level protocol.

to redirect the Ethernet descriptors concurrently with data arriving. Regularly interleaved accesses by the driver and the hardware would cause the hardware to lock up; reinitialisation was costly and the entire blast would be lost.

A final concern is related to security. If processes are permitted to perform non-blocking I/O, then it is possible that when a packet interrupts a burst and is accidentally placed (temporarily) in the wrong area of memory, that the process that owns that area of memory may be able to access data which it should not be able to. For example, this would allow a rogue process on one machine to deliberately attempt to cause burst transfers when a message from a fileserver was expected, in order to gain unauthorised access to data for other processes on that machine.

5.1.4.3 Transparent Blast

An improved transparent blast protocol was proposed in [**O'Malley90**]. In this scheme a control packet is sent in advance of the blast. This contains enough information to allow the receiver to reprogram the interface chip in order to put the blast data in the correct place. The Ethernet chip is disabled while the reprogramming takes place which leads to a "deaf" period of approximately 150μs. Data packets are either padded to ensure that the headers are all the same length, or else the length of the headers is indicated in the control packet.

This implementation suffers from a different problem which is that the bus latency caused by accessing the descriptors used to break the received packet up into the header and data components can cause the Ethernet chip to overrun.

5.1.4.4 The Cambridge Ring

The Media Access Protocol on the Cambridge Ring [**Hopper78**] allows a receiver to indicate a *Not Selected* indication to a transmitter. This allows the construction of a very simple interface which can receive from only a single source at a time. The indication prevents the data being lost; the transmitter will retry that mini-packet. This corresponds to explicit support for blast protocols in the hardware. The Cambridge Fast Ring [**Temple84**] supported a similar selection and channel mode where bandwidth was reserved for the duration of the burst.

5.1.4.5 Discussion

In general, attempts to date to make an efficient implementation of Application Data Units have failed. This has been mostly due to either network hardware / protocols that made rapid decisions of recipient difficult, or due to operating systems which added substantial complexity in the path from device to application. The combination of ATM networking (where the VCI can be used to demultiplex at the lowest level) and a micro-kernel (where applications may communicate directly with device drivers) leads to a possibility of reversing this failure. This is addressed in the Fawn buffer system described below.

5.2 Requirements

Apart from the attempt to support ADUs, the requirements for an I/O buffering system in Fawn were slightly different from all of the above systems. In the Pegasus philosophy, applications negotiate for real rather than virtual resources. This means that an application has a certain amount of buffer memory which will not be paged. If the system is short of memory then it will require the application to free a certain amount,[7] the application is free to choose the optimal memory to retain from its perspective. Like the Fbuf system there was no need for highly dynamic reallocation of buffers between different I/O data paths. Also it would be preferable if multi-recipient data need not be copied.

5.2.1 Considering Device Hardware

As mentioned in section 5.1.1 network hardware is frequently one of two types. An interface is either high bandwidth (e.g. ATM) in which case DMA is usually supported and the device is reasonably intelligent in using the VCI (or similar) in the header of arriving data to access the correct buffer control information; such interfaces are *self-selecting*. Otherwise the interface is usually of low bandwidth and requiring software copying of data (e.g. Ethernet); such interfaces are *non self-selecting*. Of course there are exceptions (e.g. [**Greaves94**]) but it is reasonable to optimise for the common cases.

[7]If an application does not free memory when required within a reasonable length of time then the resource manager may simply kill it.

Examples of self-selecting interfaces include the Aurora TURBOchannel interface [**Druschel94**] and the Jetstream / Afterburner combination [**Edwards94**]. In Jetstream the arriving packets enter a special buffer memory based on the arriving VCI. The device driver then reads the headers and instructs a special DMA engine to copy the data to the final location. Knowledgeable applications may make special use of the buffer pools in the special memory.

It has been recent practice in operating systems to support a protocol independent scheme for determining the process for which packets arriving at an interface are destined. This is known as packet filtering [**Mogul90**] and this technology is now highly advanced [**McCanne93**, **Yuhara94**]. For non-self-selecting interfaces packet filtering can determine which I/O path the data will travel along as easily as it can determine which process will be the receiver. This property is assumed in the Fawn buffer mechanism derived below.

On older hardware many devices which used DMA required a single non-interrupted access to a contiguous buffer. On more recent platforms such as the TURBOchannel [**DEC93**] the bus architecture requires that a device burst for some maximum period before relinquishing the bus. This is to prevent the cache and write buffer being starved of memory bandwidth and halting the CPU. Devices are expected to have enough internal buffering to weather such gaps. Also, the high bandwidth that is available from DMA on typical workstations depends on accessing the DRAMs using page mode. Such accesses mean that the DMA cycle must be re-initiated on crossing a DRAM page boundary. Furthermore most workstations are designed for running Unix with its non-contiguous Mbuf chains. The result of this is that most high performance DMA hardware is capable of (at least limited) scatter-gather capability.

5.2.2 Protocol Software

Most commonly used protocols wish to operate on a data stream in three ways. These are:

1. To add a header (e.g. Ethernet, IP, TCP, UDP, XTP)
2. To add a trailer (e.g. XTP, AAL5)
3. To break up a request into smaller sizes.

5.2.2.1 Headers

Headers are usually used to ensure that data gets to the right place, or to signify that it came from a particular place. We can consider how such operations affect high performance stream I/O, particularly in respect of security. In the Internet much of the (little) security which does exist relies on secure port numbers. These are port numbers which are only available to the highest authority on that machine, and receivers may assume that any such packet bears the full authority of the administrators of the source machine rather than a particular user. It is similarly important that machines accurately report their own addresses. For this reason the transmission of arbitrary packets must be prohibited - transmission must include the correct headers as authorised by the system. This has been the traditional reason for having such networking protocols in the kernel or, in a micro-kernel, implemented in a single "networking daemon". However this is not a foregone conclusion.

It is possible instead to have protocol implementations within the user process, and still retain the required security. The device driver must then perform the security control. There is a broad spectrum of the possible ways of engineering such a solution. In one extreme the device drivers actually include code (via a trusted library) which "understands" the protocol and checks the headers; which is close to implementing the protocol itself in each device driver.

Alternatively, the device driver could include a mechanism similar in concept to packet filtering code which determines if the packet is valid for transmission (rather than reception). This process can be highly optimised. Most of the headers of such protocols are well defined fixed length fields[8] which can be easily checked under a mask and compared within the device driver. The mask and compare values would be initialised by the privileged centralised server when the I/O channel was initiated.

For any implementation the volatility of the buffer memory must be taken into consideration; the driver must protect against the process corrupting the headers after they have been checked. This may entail copying the security related fields of the header before checking them. Another solution may rely on caching the secure part of the header in the device driver's private memory and updating the per-packet fields. Some common headers are shown in figure 5.4; the IP and TCP headers can potentially include a variable number of additional 32 bit words

[8]IP options are extremely rare and difficult to use from any operating system.

specified using the **hlen** fields.

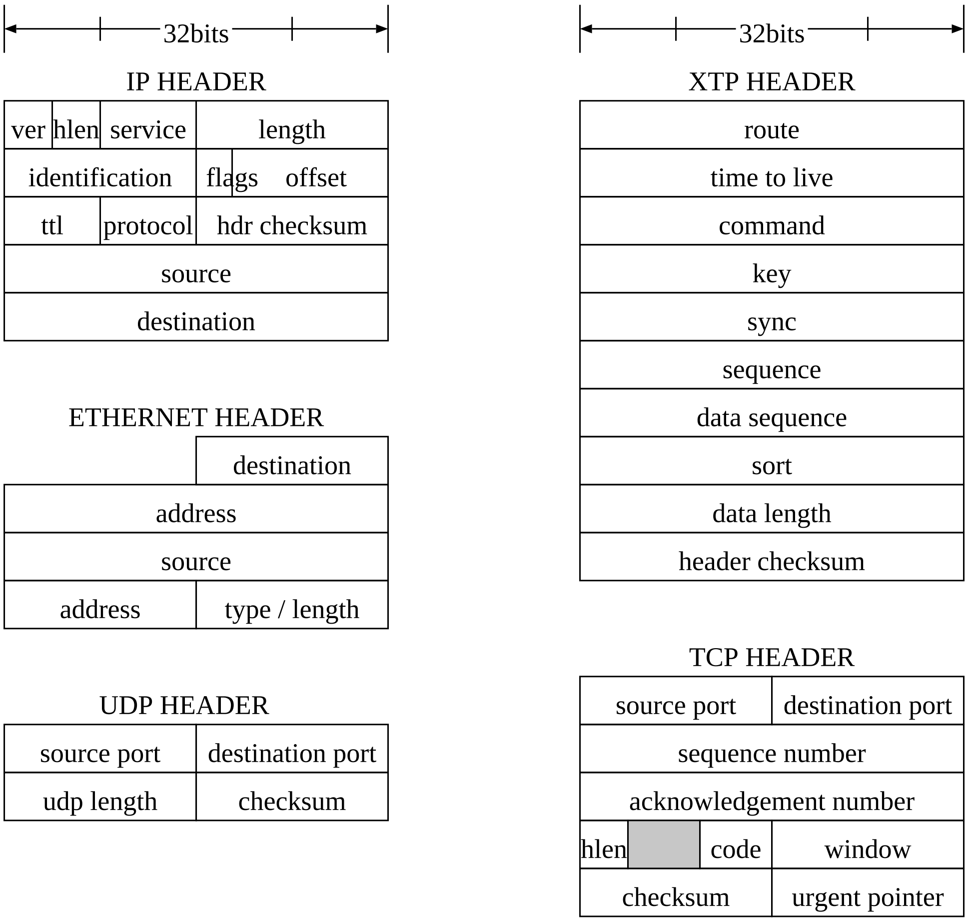

Figure 5.4: Headers for various protocols

For many other fields such as checksums, the user process is the only one to suffer if they are not initialised correctly. Considering the protocols shown in 5.4 then, we see that for UDP and TCP only the port values need to be secured. For IP all but the *length* and *checksum* fields must be secured, and for Ethernet all the fields must be secured. For XTP the fields which must be secured are *route*, *time to live*, *command* and *key*. The *sort* field may need to be checked although its exact semantics are not clearly defined.

One final possible concern would be with respect to flow control or congestion avoidance; conceivably a user process could have private code which disobeyed the standards on TCP congestion control. There are various answers to this. First, a malevolent user process could simply use UDP, which has no congestion control, instead if it wished. Second since the operating system is designed with quality of service support, the system could easily limit the rate at which a process is permitted to transmit. Third, the application may in fact be able to make better use of the resources in the network due to application specific knowledge, or by using advanced experimental code. Recent research [**Brackmo94**] shows that improving one's own responsiveness to network behaviour in order to improve ones own performance may in fact have a beneficial effect on others using the network.

5.2.2.2 Trailers

Unlike headers, trailers do not usually contain any security information. They usually contain checksum information. Trailers for common protocols may be found in figure 5.5. For AAL5 the pad is of variable length up to a maximum of 47 bytes.

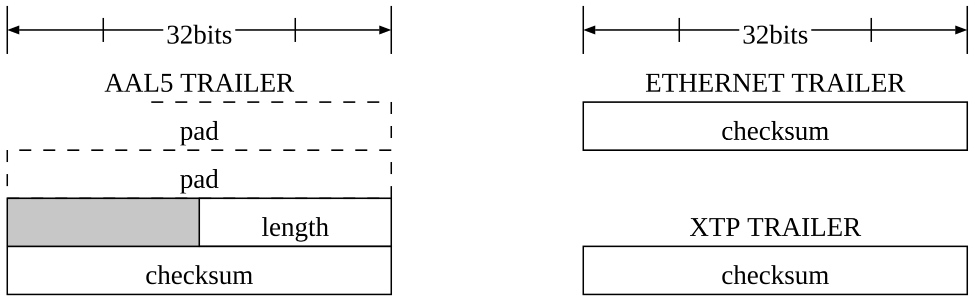

Figure 5.5: Trailers for various protocols

Trailers are most easily dealt with by requiring the user process to provide enough space (or the correct padding) for the packet on both receive and transmit. If there is not enough, the packet will simply be discarded - a loss to the user processes. Providing this space is not difficult for a process once it is known how much is necessary; this value can be computed by a shared library or discovered using an IPC call.

5.2.2.3 Fragmentation

Like trailers there is no security consideration for local or remote systems as a result of incorrect length transfers on an I/O channel. If a transmit request is too large the data can simply be discarded. If a receive buffer is too small the data may be discarded or truncated. Truncation may be used deliberately in the case of network monitoring applications which are frequently only interested in the headers of the passing packets.

5.2.3 Application Software

Many applications have application specific basic data units which may be too large for individual network packets. For example, NFS blocks over Ethernet are usually fragmented at the IP level. Ideally a system should permit the application to specify receive buffers in such a way that the actual data of interest to the application ends up in contiguous virtual addresses.

On the other hand for some applications, the application's basic data unit (i.e. the unit over which the application considers loss of any sub part to be loss of the total) may be very small. This may be found in multimedia streams such as audio over ATM, and compressed tiled video [**Pratt94c**]. For such streams, the application should not have to suffer very large numbers of interactions with the device driver; it should be able to handle the data stream only when an aggregate of many small data units is available.

5.2.4 Scheduling

Within Fawn, the use of jubilees means that I/O channels must have a substantial amount of buffering to cope with the delays until the receiver of the data is next scheduled. Due to the heuristic used for dispatching interrupts described in section 3.4.4 the delay can be up to twice the length of the jubilee minus twice the guaranteed CPU allocation for the process in each jubilee. For example, with a jubilee of 25 ms an application which is guaranteed 25% of the CPU to handle a 10 Mbit/sec stream will require approximately 48 Kbytes of buffering. Of course delays within the process itself due to thread contention or remote invocation need also be considered.

A device driver process may have specific scheduling requirements in order to meet its quality of service contracts. In particular it is likely to require a non-blocking access method to I/O channels.

5.2.5 Streaming Memory

In [**Hayter93**] the concept of a stream cache was defined. A stream cache is a special area of the cache on a system which is used directly for I/O without the data actually being represented in the underlying memory. It is particularly suited to the processing of multimedia streams. The MMU on the system prevents applications accessing data in the stream which is "stale" (i.e. has been overwritten in the cache by newer data). If the application accesses data which has yet to arrive then either the processor is halted, just like a conventional cache miss, or if the accessed data is sufficiently far into the future, the MMU traps the access and allows the operating system to schedule some other process.

5.3 The Adopted Solution

The design for I/O buffering adopted in this system, called **Rbufs** will now be presented, together with a discussion on the operation of I/O channels within the system.

5.3.1 Operation

The Rbuf design separates the three issues of I/O buffering, namely:

- The actual data.
- The offset/length aggregation mechanisms.
- The memory allocation and freeing concerns.

An I/O channel is comprised of a data area (for the actual data) and some control areas (for the aggregation information). The memory allocation is managed independently of the I/O channel by the owner of the memory.

5.3.1.1 Data Area

The Rbuf Data Area consists of a small number of large contiguous regions of the virtual address space. These areas are allocated by the system and are always backed by physical memory. Revocation of this memory is subject to out of band discussion with the memory system. To as large an extent as possible the memory allocator will keep these contiguous regions of virtual addresses backed by contiguous regions of physical addresses (this is clearly a platform dependent factor).

The system provides a fast mechanism for converting Rbuf Data Area virtual addresses into physical addresses for use by drivers that perform DMA. On many platforms a page table mapping indexed by virtual page number exists for use by the TLB miss handler; on such platforms this table can be made accessible to device driver processes with read only status.

Protection of the data area is determined by the use of the I/O channel. It must be at least writable in the process generating the data and at least readable in the process receiving the data. Other processes may also have access to the data area especially when an I/O channel spanning multiple processes (see section 5.3.3) is in use.

One of the processes is logically the owner in the sense that it allocates the addresses within the data area which are to be used.

The Rbuf data area is considered volatile and is always updateable by the process generating the data. This was justified in section 5.1.3.

5.3.1.2 Data Aggregation

A collection of regions in the data area may be grouped together (e.g. to form a packet) using a data structure known as an I/O Record or **iorec**. An iorec is closest in form to the Unix concept of an **iovec**. It consists of a header followed by a sequence of base and length pairs. The header indicates the number of such pairs which follow it and is padded to make it the same size as a pair.

This padding could be used on some channels where appropriate to carry additional information. For example the exact time at which the packet arrived or partial checksum information if this is computed by the hardware. [**Sreenan93**] points out that sometimes it is more important to know exactly when something happened than actually getting to process it immediately.

5.3.1.3 Control Areas

A control area is a circular buffer used in a producer / consumer arrangement. A pair of event channels is provided between the processes to control access to this circular buffer. One of these events channels (going from writer to reader) indicates the *head* position and the other (going from reader to writer) indicates the *tail*.

A circular buffer is given memory protection so that it is writable by the writing process and read-only to the reading process. A control area is used to transfer **iorec** information in a simplex direction in an I/O channel. Two of these control areas are thus required to form an I/O channel and their sizes are chosen at the time that the I/O channel is established.

Figure 5.6 shows a control area with two **iorec**s in it. The first **iorec** describes two regions within the Rbuf data area whereas the second describes a single contiguous region.

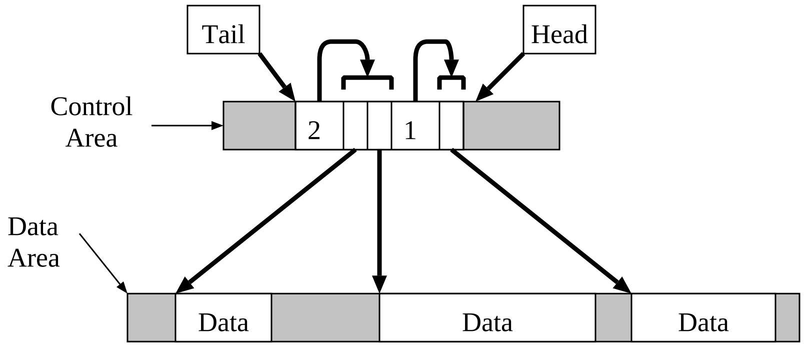

Figure 5.6: Rbuf memory arrangement

5.3.2 Usage

Figure 5.7 shows two processes A and B using control areas to send **iorecs** between them. Each control area, as described above, basically provide a fifo queue of **iorec**s between the two ends of an I/O channel. Equivalently, an I/O channel is composed of two simplex control area fifos to form a duplex management channel. The control areas are used indistinguishably no matter how the I/O channel is being used.

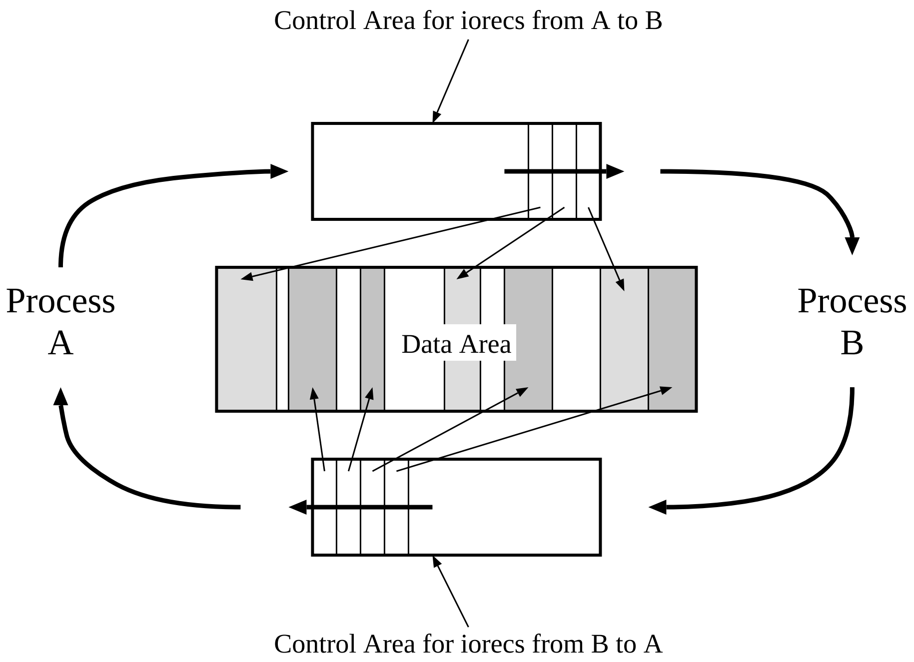

Figure 5.7: Control Areas for an I/O channel between A and B

A typical I/O channel is in fact a simplex data channel operating in one of two modes. The purpose of these two modes is to allow for the support of ADUs in various contexts. Note that there is no requirement for either end of the I/O channel to process the data in a FIFO manner, that is merely how the buffering between the two ends is implemented.

In Transmit Master Mode (TMM), the originator of the data chooses the addresses in the Rbuf data area, places the data into the Rbufs, and places the records into the control area. It then updates the *head* event for that control buffer indicating to the receiver that there is at least one record present. As soon as the downstream side has read these records from the control buffer it updates the other (*tail*) event, freeing the control buffer space for more records. When the downstream side is finished with the data it places the records into the control area for the queue in the other direction and signals its *head* event on that control buffer. The originator likewise signals when it has read the returned acknowledgement from the control buffer. The originator is then free to reuse the data indicated in the returning control buffer.

In Receive Master Mode (RMM), the operation of the control areas is indistinguishable from TMM; the difference is that the Rbuf data area is mapped with the permissions reversed and the data is placed in the allocated areas by the downstream side. It is the receiver of the data which chooses the addresses in the Rbuf data area and passes **iorec**s which indicate where it wishes the data to be placed to the downstream side. The downstream side uses the other control area to indicate when it has filled these areas with data.

The Master end, which is choosing the addresses, is responsible for managing the data area and keeping track of which parts of it are "free" and which are "busy". This can be done in whatever way is deemed appropriate. For some applications, where FIFO processing occurs at both ends, it may be sufficient to partition the data area into **iorecs** at the initiation of an I/O channel, performing no subsequent allocation management.

Table 5.1 presents a summary of the differences between TMM and RMM for the diagram shown in figure 5.7; without loss of generality **A** is the master - it chooses the addresses within the data area.

	TMM	RMM
Chooses the Addresses	A	A
Manages data area	A	A
Write access to data	A	B
Read access to data	B	A

Table 5.1: TMM and RMM properties

Since the event counts for both control areas are available to a user of an I/O channel it is possible to operate in a non-blocking manner. By reading the event counts associated with the circular buffers, instead of blocking on them, a process can ensure both that there is an Rbuf ready for collection and also that there will be space to dispose of it in the other buffer. This functions reliably because event counts never lose events. Routines for both blocking and non-blocking access are standard parts of the Rbuf library.

5.3.3 Longer channels

Sometimes an I/O channel is needed which spans more than two processes. An example may be a file serving application where data arrives from a network device driver process, passes to the fileserver process, and then passes to the disk driver process.

When such an I/O channel is set up it is possible to share certain areas of Rbuf data memory which are already allocated to that process for another I/O channel. A process may wish to have some private Rbufs for each direction of the connection (i.e. ones which are not accessible to processes in the other direction) for passing privileged information. In the fileserver example, the fileserver may have Rbufs which are used for **inode** information which are not accessible by the network device driver.

The management of the channel may either be at one end or it may be in the middle. In the example of the fileserver, it is likely to be in TMM for communicating with the disk driver, and RMM for communicating with the network driver. The important point is that the data need not be copied in a longer chain provided trust holds.

Figure 5.8 shows the I/O channels for a fileserver. For simplicity, this only shows the control paths for writes. The **iorec**s used in the channel between the fileserver and the disk driver will contain references to both the network buffer data area and the private inode data area. Only the network data buffer area is used for receiving packets. The fileserver (operating in RMM) will endeavour to arrange the **iorec**s so that the disk blocks arriving (probably fragmented across multiple packets) will end up contiguous in the single address space and hence in a suitable manner for writing to disk.

5.3.4 Complex channels

In some cases the flow of data may not be along a simple I/O channel. This is the case for multicast traffic which is being received by multiple processes on the same machine. For such cases the Rbuf memory is mapped readable by all the recipients using TMM I/O channels to each recipient. The device driver places the records in the control areas of all the processes which should receive the packet and reference counts the Rbuf areas so that the memory is not reused until all of the receivers have indicated they are finished with it via their control areas.

Apart from the lack of copying, both processes benefit from the buffering memory provided by the other compared with a scheme using copying.

A problem potentially arises if one of the receivers of such multicast data is slower at processing it than the other and falls behind. Ideally it would not be able to

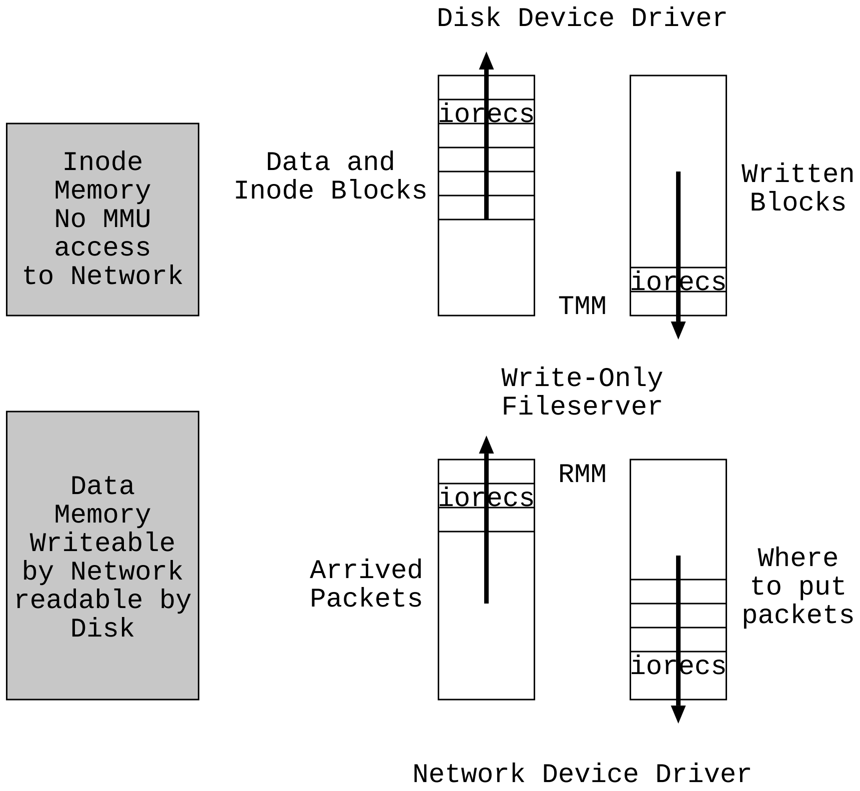

Figure 5.8: A longer Rbuf channel: Control path for Fileserver Writes

have an adverse affect on the other receiver. This can be done by limiting the amount of memory in use by each I/O channel. When the limit is reached, the **iorec**s are not placed in that channel and the reference count used is one less. The buffers are hence selectively dropped from channels where the receiver is unable to keep up. An appropriate margin may be configured based on the fan-out of the connection.

One approximate but very efficient way of implementing this margin is to limit the size of the circular control buffer. **Iorec**s are then dropped automatically when they cannot be inserted in the buffer in a non-blocking manner. Even if a more accurate implementation of the margin is required, the Rbuf scheme ensures that the cost is only paid for I/O channels where it is required, rather than in general.

5.3.5 Out of band control

For I/O channels in other operating systems there exist mechanisms for out of band modifications or information requests. In Unix this is done via the **ioctl** mechanism.[9] In Wanda this is done by marshalling into an **IOBuf** and using **WandaIPCControl**. In Rbufs an I/O channel is normally established with a parallel IPC channel which is used for out of band RPCs to the other end of the I/O channel. IPC is discussed in section 4.4.

5.4 Summary

	Mbufs	IOBufs	Fbufs	Rbufs
Page Faults Possible	No	No	Yes	No
Alignment OK	No	Yes	??	Yes
Copy to user process	Yes	No	No	No
Copy to clever device	Yes	No	No	No
Copy for multicast	Yes	Yes	Yes?	No
Copy for retransmission	Yes	Yes	No	No
Support for ADUs	No	No	No	Yes
Limit on Resource Usage	Yes[10]	No	No	Yes
Must be cleared for security.	No[11]	Yes	No[12]	No[12]

Table 5.2: Comparison of Buffering Properties

This chapter has considered the (de-)merits of various buffering schemes. Many of these schemes are what they are for historical reasons (for example mbufs were designed when maximum memory efficiency was pre-eminent). The requirements for a buffering scheme in a high performance micro-kernel were presented in detail.

The Rbuf I/O channelling and buffering mechanisms have been described. The principal feature is the separation of the three issues of data transmission, struc-

[9] **Setsockopt** and **Getsockopt** are cleaner interfaces to the intra-kernel identical **ioctl**.

[10] This limit is actually as a result of socket buffering.

[11] This is because of the copy to user process memory. However some networking code rounds up the sizes of certain buffers without clearing the padding bytes thus included; this can cause an information leak of up to three bytes.

[12] Buffers must be cleared when the memory is first allocated. This allocation is not for every buffer usage in Fbufs but is still more frequent in Fbufs than in Rbufs.

ture / aggregation control, and memory allocation. This separation allows for great flexibility in the management of communications within the machine. Rbufs are designed to minimise copying in order to support high bandwidth applications and also the explicit scheduling of network device drivers. Table 5.2 shows a comparison of the various schemes considered.

Chapter 6

Experimental Work

There are three primary goals of the experimental programme described in this chapter. First, to evaluate the effectiveness and performance of the inter-process scheduling, intra-process scheduling and process communication mechanisms described in chapter 3. Second, to quantify the operational benefit of using this scheme to tackle device driver scheduling in general by confronting the particular problem of the Fairisle Port Controller identified in section 3.1.5. Third, to consider the suitability of Rbufs for I/O channel communication in an event based micro-kernel system.

6.1 Experimental Platform

The experimental platform chosen was the ARM processor [**ARM91**, **ARM92**] and the Fairisle Port Controller version 3 [**Hayter94a**, **Hayter94b**]. This was due to the author's detailed knowledge of this hardware gained during the porting of the Wanda micro-kernel to this platform and the writing of the code for the Fairisle ATM switch.

The ARM610 processor on this platform has a 25 MHz clock with a 4Kbyte internal shared instruction / data write-through read-allocate cache which is 64 way set associative with 16 byte cache lines. There is a two address and eight data write buffer between the processor core and the bus. The bus on the FPC3 runs at $8\frac{1}{3}$ MHz and a cache line can be loaded in 5 cycles. This gives a peak memory to cache bandwidth of approximately 210 Mbits/sec. The ARM does not have separate instruction and data paths to the cache so all data accesses stall the pipeline. Also there are no branch delay or load delay slots so these also stall the pipeline.

The Fairisle network interface is optimised for controlling data passing through the port controller rather than for data sourced or sunk at that device. Accesses to the cell buffer memory take three cycles per word assuming *no* contention or, using cache line sized bursts, a peak of 82 Mbits/sec. This is particularly relevant to the experiments of section 6.5.

The port controller provides 4 Mbytes of DRAM and two 16 bit programmable counters which decrement every 480 ns tick. These counters are only accessible 8 bits at a time via costly I/O accesses and require a special latch command to be written before they can be read.

It can be seen that this is not a fast machine by current standards, in particular the cache is very small. It will be seen below that the slow speed and particularly the small cache of this machine show the Fawn design under somewhat unfavourable conditions. For a more modern machine the Fawn system would perform even better.

6.1.1 System Configuration

The system is configured with a jubilee size of 2^{16} ticks. This is using the maximum size of one of the timers on this hardware and equates to about 31.5ms. Clearly this was fairly arbitrarily chosen rather than giving any particular special effect for these experiments. Other jubilee sizes are possible by simply modifying the system configuration. A discussion of the costs associated with each jubilee are found below in section 6.2.2.

The configuration of the kernel normally includes the console daemon, the Basic Resource Allocator, and the bootstrap trader, plus whatever test programs are relevant. The number of ticks of CPU allocated to each process is test specific.

The other timer is reprogrammed on each occasion that a process is given the CPU. This will interrupt after a number of ticks which represents the remaining allocation of the selected process. Unfortunately due to a clock synchronisation problem the previous interrupt must be cleared no less than four times. This programming operation takes an average of 4μs. When a process loses or gives up the CPU, reading the timer to measure how long the process had takes roughly either 2μs or 1μs (depending on whether the timer expired or not).

6.1.2 Measurement Details

The timing measurements presented in this chapter were made by reading the timer which is used for generating the jubilee interrupt. This timer counts in ticks of 480ns. The overhead of reading the time has been measured as 4 ticks, or 1.92μs. This overhead is included in all the times reported.

Measurements are written to a table in memory (usually of 1024 elements) with no attempt being made to process the data as it is being logged by generating reference counts or otherwise. Since the Arm's cache is read-allocate only, this logging has no effect on cache behaviour. The data is post processed before being output on the console.

6.2 Inter-process scheduling

6.2.1 Interrupt Latency

One of the potential concerns with moving device drivers out of the kernel and into user space processes is that the latency for interrupts may be increased to an unacceptable amount. The purpose of this experiment is to measure interrupt latency in this system.

As well as the obvious difference in latency between the idle and busy cases, there is also an expected difference depending on the length of time between the process relinquishing the CPU and the interrupt occurring. This is because of the heuristic described in section 3.4.4. When an interrupt occurs for a process other than the currently active one, the kernel compares the length of time that the current process has been running against a configured threshold. If that time is less than the threshold then the kernel considers it too inefficient to take the CPU away - this is to prevent a large amount of time being wasted in the system switching too frequently between processes. If the process has been running for more than the threshold then the kernel suspends it and initiates a reschedule. Figure 6.1 shows the expected sequence of events for the idle case and for the busy cases when the threshold is exceeded and when it is not exceeded. In the experimental system this threshold is 100μs.[1]

[1]i.e. 208 ticks.

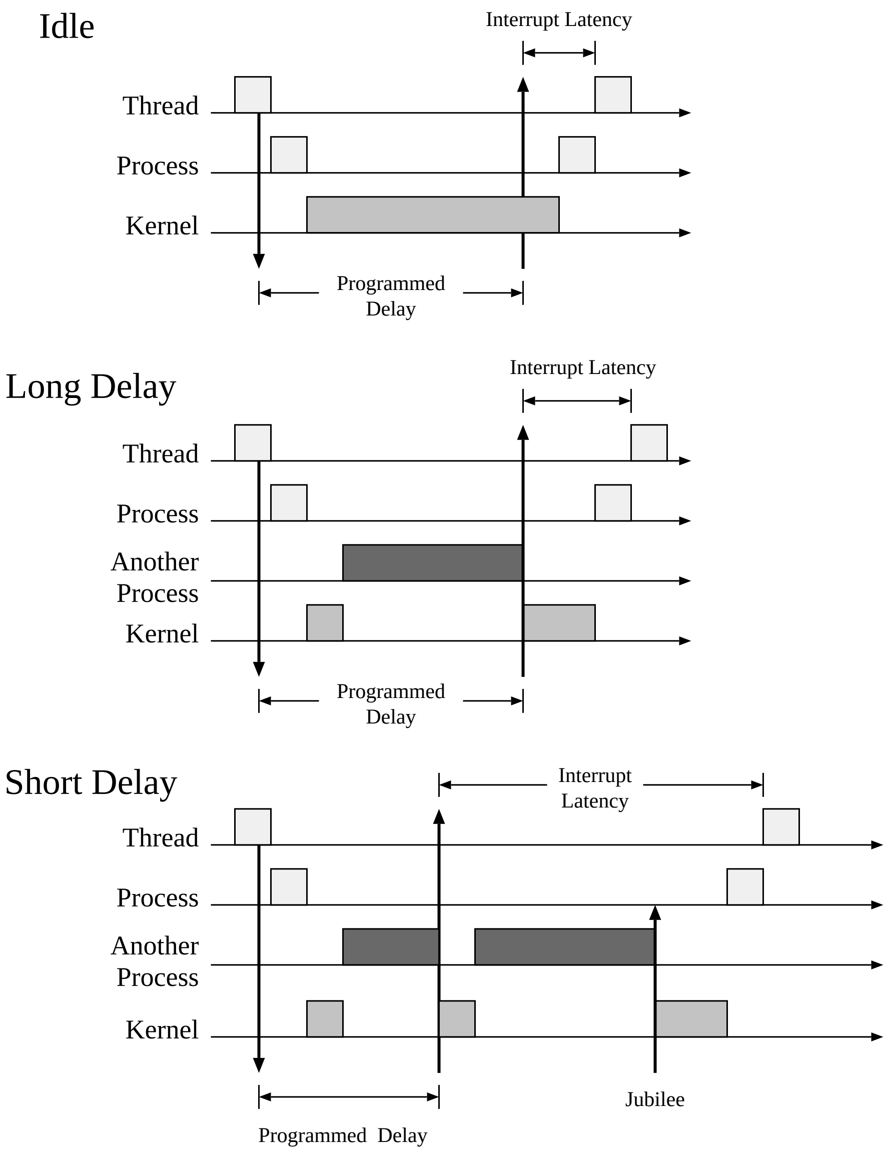

Figure 6.1: Interrupt progress for three scenarios

Since both timers available in the system are already in use by the low level scheduler another source of timed interrupts was required for this experiment. The Fairisle Xilinx chip contains some telemetry hardware amenable to this task. It provides a 16 bit free running counter (FRC) and a 16 bit down counter (TC). The latter raises an interrupt when it reaches zero, which remains asserted until it is reinitialised by software. The clock used for these counters is actually the Fairisle cell synchronisation pulse. This occurs every 64 periods of the fabric 18MHz clock or once every 3.555μs.

In this experiment the FRC is first read, then the TC is programmed with a value and the thread blocks on the interrupt event. When the interrupt occurs the event is triggered; when the thread awakes, it reads the FRC again. The difference in the FRC values minus the programmed TC sleeping time is considered to be the interrupt latency.

The system was measured both when idle and when under overload conditions. The system is overloaded by creating a daemon process which loops, being allocated a full jubilee period of time in the highest non-guaranteed scheduling level (see section 3.2.3 for a discussion of scheduling levels).

	Programmed Delay	
System	Small	Large
Idle	25μs	25μs
Busy	Jubilee	34μs

Table 6.1: Interrupt Latency

The results of this experiment can be shown in table 6.1. It can be seen that when the system is idle then the interrupt dispatch latency is small. In particular it is not excessively worse than might be expected on a traditional system (e.g. Unix). When the system is busy however we see the expected effect of the heuristic described above.

If the interrupt went off before the threshold was up for the competing process, then the measurement process would have to wait until the next reschedule before gaining the CPU. Since in this configuration the competing process is looping this will not be until the start of the next jubilee.[2]

What can be observed from this experiment is that even when the system is in overload the interrupt dispatching latency is only about 10μs worse than when idle in the typical case, and the worst case is bounded by the length of the Jubilee.

In comparison, for a more powerful DECstation 5000/200,[3] [**Shand92**] measures the interrupt latency to a kernel device driver on Ultrix 4.2A to be usually between 14μs and 21μs. In the worst case they report delays of over 1ms on a machine

[2]It could be argued that the system ought to force a reschedule because the current process is operating in a lower scheduling level than the process which has received the interrupt; this is disabled to allow this test. See section 3.2.3 for a discussion on scheduling levels.

[3]Bus clock frequency 25MHz, twin 64Kbyte caches.

performing no other I/O, and higher (instance specific) delays for a machines with networking hardware.

Fawn is structured so that it is always able to handle an interrupt, but it chooses (using the heuristic) to do so at an appropriate time. The decision is made as soon as the interrupt occurs; obviously the heuristic could, if necessary, be easily altered for particular circumstances.

6.2.2 Jubilee Startup Costs

One of the costs of the system proposed is the time taken to reinitialise the various data structures at the beginning of each Jubilee. The purpose of this experiment is to quantify the effects of this mechanism.

A test process was written which intercepted its own application startup vector and read the value of the jubilee timer. This was run on a system with the usual daemon processes and a variable number of other dummy processes. The system daemons were configured without any guaranteed time, the test process and dummy processes were given guaranteed time. As a result the test process would be the first process to run in each jubilee. A diagram of the expected order of execution during each jubilee is shown in figure 6.2.

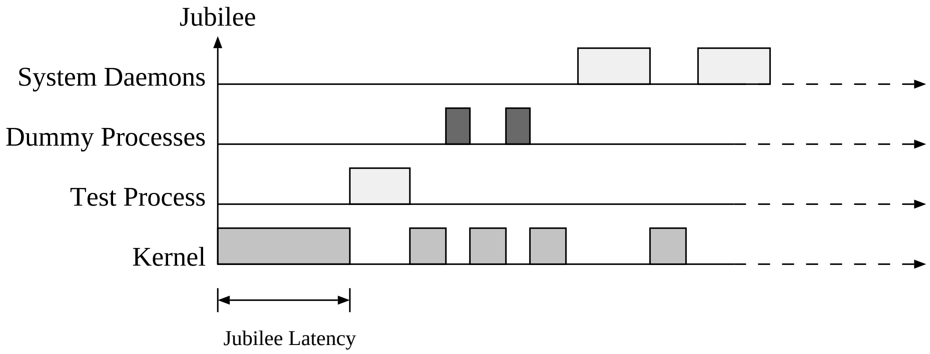

Figure 6.2: CPU activity during a Jubilee

The additional dummy processes consist of a simple program whose initial thread exits in a way that means that the process does not exit (thus the process is still activated each jubilee but immediately surrenders the CPU).

The system was run with the two normal daemons (the system trader having been eliminated) and some number of these dummy programs. The Jubilee initialisation latency was recorded. The latency was found to be very variable with the exact locations of various processes in the memory map. One machine with an Arm 710 processor was available late during experimentation. Figure 6.3 shows the results for both the Arm 610 and the Arm 710 running identical system images.

Figure 6.3: Latency from jubilee start to first activation

The difference between these two lines is entirely due to cache artifacts. The 710 has a larger 8K cache with larger 32byte cache lines but the associativity is less – 4 way. For the 610 the cost appears to be 8.8μs per process with an intercept of 1.5μs. For the 710 the cost is about 3.4μs per process with an intercept of 23.8μs. With the cache disabled the cost is 12.5μs per process with an intercept of 68.4μs. Interestingly the cold cache case (derived by flushing the cache as soon as a new jubilee is detected) has a per process cost of 16.6μs with an intercept of 17.9μs, indicating that the cache is highly effective for the code but may be thrashing on the data structures.

Various other configurations of processes were tried, many of which showed distinct "knees" related to caching behaviour.

Clearly the fact that all the processes' data segments (and hence VPI information) are page aligned on this system is causing substantial cache conflict. Although this, unfortunately, makes the direct jubilee cost difficult to measure, the salient point is that the jubilee cost appears to be mostly dwarfed by the indirect cache costs of the fine grain sharing provided.

A few researchers have observed performance problems due to cache conflicts in client-server environments although this is not a standard benchmark and therefore not greatly studied. For example, [**Chen94**] presents studies of X11 clients and server for the DECstation 5000/200 running Ultrix 4.2. This problem has tended to be regarded as soluble using shared libraries (which Ultrix lacks) and greater cache associativity. This study, having both shared libraries and a highly associative cache, shows that further considerations may be necessary.

One possibility is to relax the alignment requirements. Another, more interesting approach, of providing Quality of Service support in the cache implementation is proposed in [**Pratt94a**]. Further consideration of this issue is beyond the scope of this dissertation.

6.2.3 Same machine RPC

This experiment attempts to measure the cost of invoking a service in another process on the same machine. A testing interface includes an operation called **ping** which provides a null-RPC service. The invocation uses the IPC channels described in section 4.4.

In order to exclude the jubilee effects mentioned in section 6.2.2 the test performs 64 pings per jubilee and then blocks until the next, ensuring that no calls are outstanding from one jubilee to the next. Figure 6.4 shows the distribution of the round trip times - a mean of $114\mu s$ with 90% sample interval between $103\mu s$ and $147\mu s$. The long tail is a result of cache misses; the points above the gap between $190\mu s$ and $210\mu s$ represent the first invocations after a jubilee which suffer poorer cache performance. An approximate breakdown of the time taken was obtained by embedding LED changing code throughout the system, and watching with a logic analyser. This is presented in table 6.2.

Of the time in the stubs a **TRY FINALLY** clause in the server stubs takes $8\mu s$. This clause is superfluous in this context (the procedure has no arguments which require memory allocation) but the stub compiler is not currently capable of optimising it out.

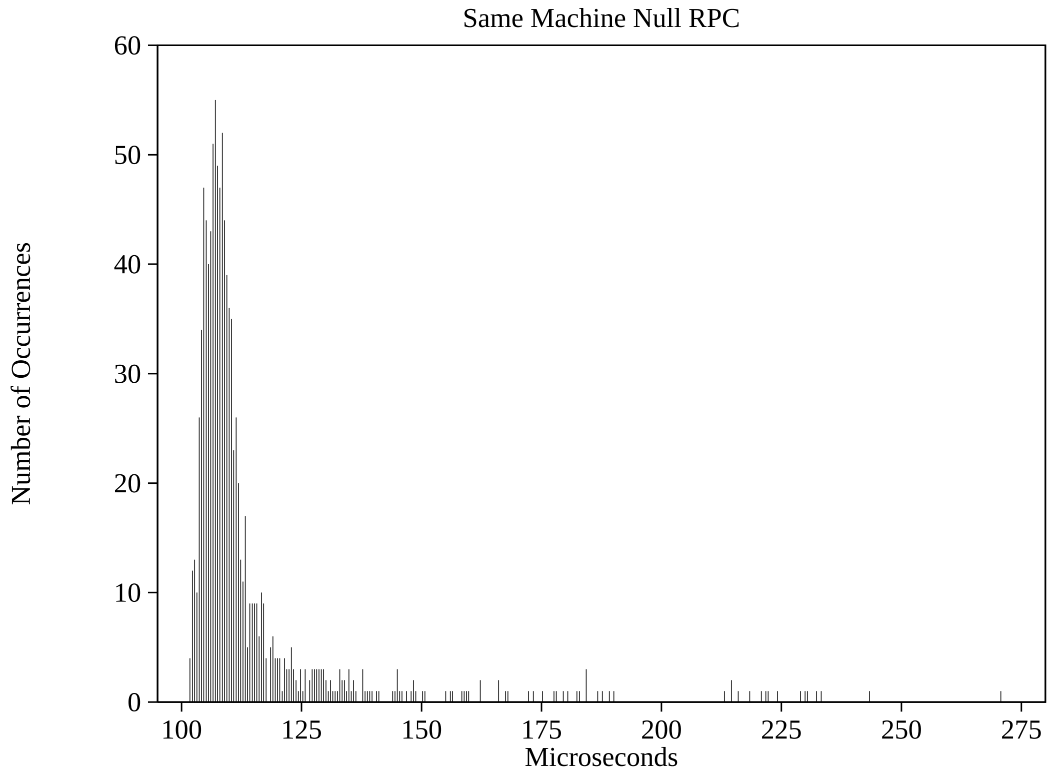

Figure 6.4: Same machine Null RPC times

System Component	Percentage
Kernel Scheduler	29
Kernel event propagation	12
User process schedulers	40
User process in stubs	19

Table 6.2: Approximate null RPC latency breakdown

As a related experiment the time taken to advance an event which must be propagated to another process was also measured. The 90% sample interval for this was 15 to 17 ticks, averaging 7.8μs. This is an important measurement since it is crucial to general system performance; in general (e.g. when using Rbufs) there will be many such events sent between inter-process scheduling decisions.

6.3 Intra-process scheduling

The design of Intra-process scheduling and synchronisation primitives was discussed in section 3.5. During the development of the prototype implementation no fewer than four distinct instances of intra-process schedulers meeting this specification were developed. The potential for many more, application-specific, schedulers is a benefit of the Fawn design.

The simplest scheduler (known as **triv**) is one which supports a single user thread. It runs with the *disable* event count always higher than *enable* at all times so it is always resumed and not activated. When the single thread blocks (either on an event or for a time or both) the process loops checking the condition(s) and yielding the CPU to the kernel. The process will be resumed and check the condition if any events (including a time change) arrive.

The first scheduler to be implemented (known as **cor**) was one which deals with multiple threads but is strictly coroutine based. It does not support time[4] (except by busy-waiting around a call to **yield**). For each event it has a linked list of threads waiting on that event ordered by the value that they are waiting for. When an event is changed locally (by **advance**) or remotely (in which case it will be indicated to the domain as described in section 3.4.2) this list is examined and any runnable threads are then moved to the run queue. This scheduler was used for the console daemon while the rest of the system (including the other schedulers) was being debugged.

Another scheduler (known as **cort**) is the same as the one just described except that it supports time fully. This entails each blocked thread being queued on up to two different lists. These lists are now doubly linked to ease removal since threads may now be awoken due to activity on the other list and need removing from the middle of the list.

Finally, due to the observation of the costs associated with a thread context switch in the doubly linked double list implementation, a scheduler (known as **corn**) was implemented which keeps a fixed sized array of threads and, when idle, scans the array and re-evaluates the condition on which each thread is waiting. This is clearly of linear cost in the number of threads, but has a small constant.

[4] In fact it was written before the kernel supported time at all!

6.3.1 Comparative performance

This section presents the result of measuring the time for a same process thread context switch using each of the above schedulers. Each process has just the two threads involved in the experiment. The context switch was performed 1024 times each of which was measured by reading the jubilee timer. Measurements which cross jubilees are excluded from the results. This is presented in figure 6.5.

The time in ticks is measured for a switch from one thread to the other, and the switch back, and the time to read the time counter (which is known to be 4 ticks). Removing the time to read the counter (which is a non-trivial fraction for these measurements) a single context switch for these three schedulers takes approximately 10, $12\frac{1}{4}$ and 15 microseconds respectively. This compares with about 30μsec for Wanda (with all checks disabled) on the same platform.

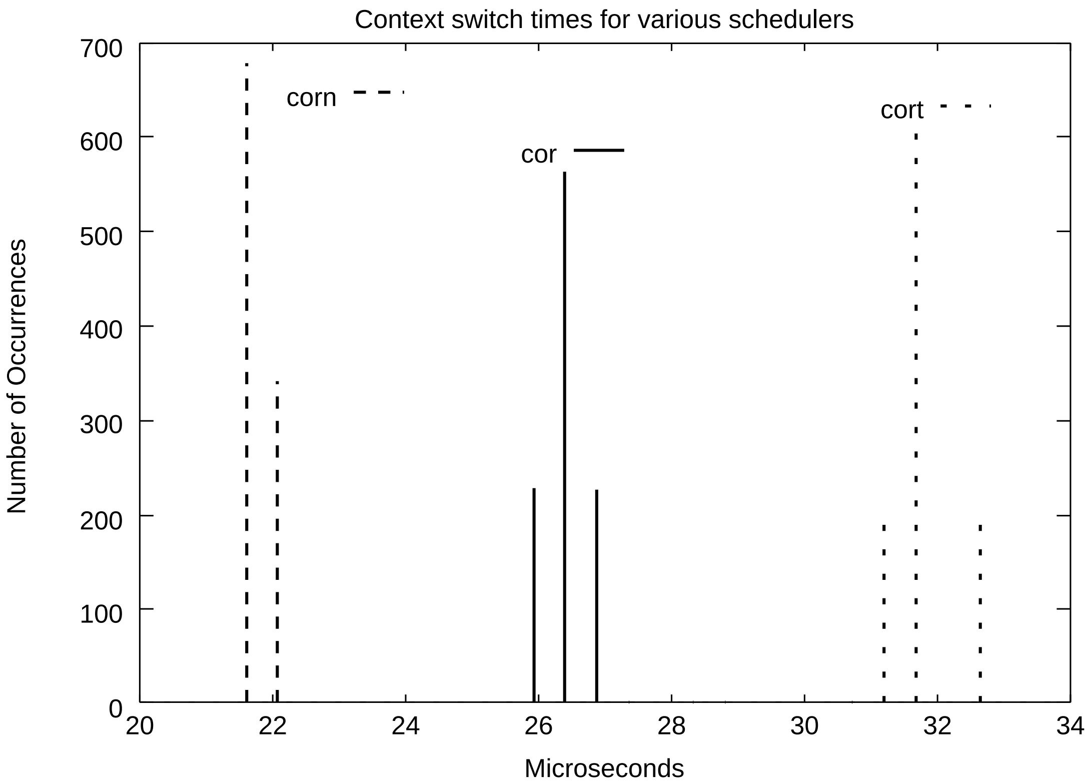

Figure 6.5: Context switch times for various schedulers

For the platform used in this investigation, the crossover point for the **corn** scheduler compared to the **cort** scheduler turns out to be 12 threads. Thus for any process which expects to have a small number of threads, the **corn** scheduler provides a performance benefit. This is an example of the sort of benefit that is gained by allowing application specific scheduling rather than a system wide generic (and therefore complex) scheduler.

6.3.2 Effects of Sharing

The purpose of this experiment was to quantify the effect of sharing scheduling code by placing it in the shared library, against each process having a private copy of the code of its intra-process scheduler.

Three identical processes are given a small amount of guaranteed time in the system and the others are not. This ensures that they will be run in succession at the beginning of each jubilee. Each process contains a thread which notes the elapsed time from the start of the jubilee at which it is woken, does a small amount of housekeeping, and then sleeps until the next jubilee. This is repeated 256 times. The experiment is run with each process having its intra-process scheduler in the static library, and compared with the same experiment where the scheduler is in the shared library. The distributions of elapsed time from the jubilee may be seen in figure 6.6.

It can be seen that the difference for the first process is marginal, a few μs at best. However for successive processes the difference is quite substantial. This is due to the increased locality of reference outweighing the cost of having the code in the shared library. Note that systems which do not use a single virtual address space may be unable to make use of this benefit (especially if the hardware provides a virtually addressed cache).

For each pair, the spread of the distributions is similar indicating that the offset is solely due to caching effects. For the same experiment with caching disabled on the processor the numbers for the three processes are deterministically 174, 402, and 630μs respectively, the location of the scheduling code being immaterial.

Clearly the exact benefit of sharing is dependent on the typical run-time of processes within the system, and particularly their own cache behaviour.

Although the experimental system does not perform memory protection the inclusion of MMU domain switching code does not alter the effect shown here. The addition of protection code adds the cost of changing the MMU domain status and, in the worst case, a TLB flush but does not affect the cache behaviour.

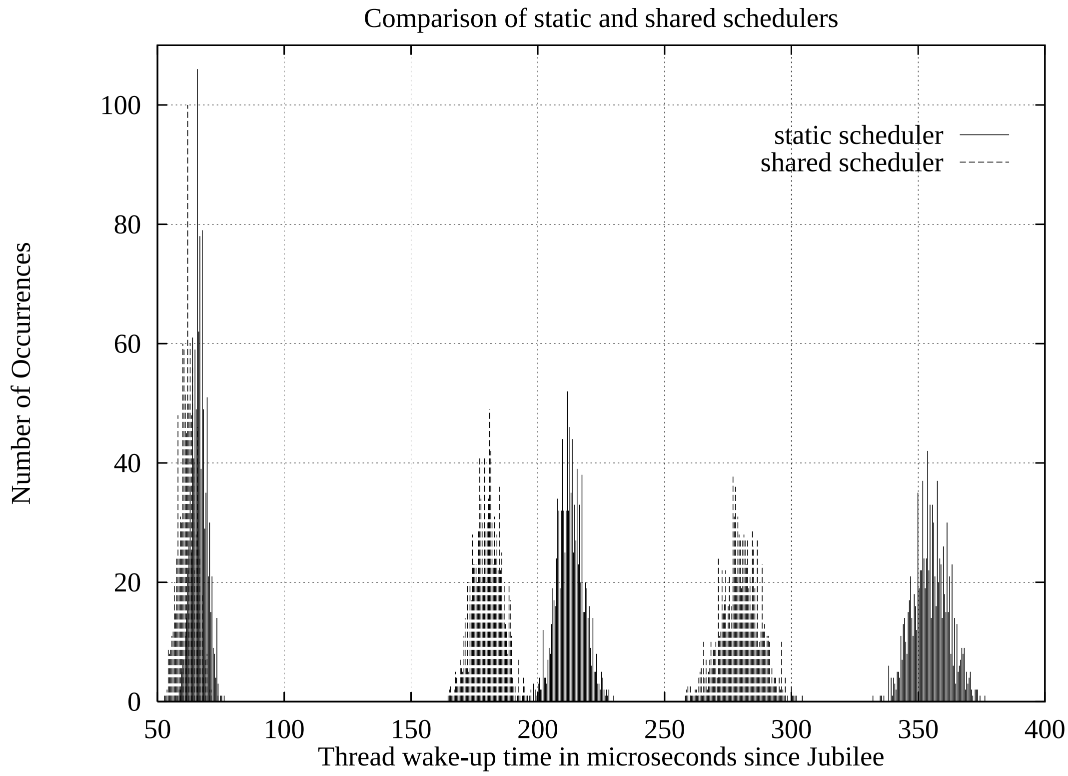

Figure 6.6: Comparison of static and shared schedulers

6.4 Fairisle Throughput

In this experiment the aim is to compare the throughput performance of the Fawn system against the Wanda system when performing cell forwarding duties as a port in a Fairisle switch.

The Fairisle device driver is written as a normal process in the system which happens to have access to the Fairisle hardware (see section 2.1.2). It performs an RPC to the basic resource allocator to initialise a hardware interrupt event for the Fairisle device hardware interrupts.

The scheduler chosen for this device driver is a coroutine scheduler in which the threads **yield** at optimum points. This permits various locks to be held for long periods (reducing the number of times they are obtained and released) while the threads are known to execute only non-blocking operations. This reduces the concurrency overhead.

To make comparison with the Wanda performance as realistic as possible the device driver is as close as possible to the Wanda form whose performance is reported in [**Black94d**] (in particular the cell scheduling policy is the same; the FIFO queue for forwarded cells has priority over locally generated cells which in turn has priority over cells to be freed) with the following differences:

- The special purpose bank of FIQ registers available to the low level Wanda interrupt code is not available to this user process.
- The code is restricted to using the C calling convention.
- When a cell arrives, a queue pointer is read from a table indexed by the incoming VCI. This may indicate that this cell is for local consumption (this includes cells with unknown VCIs and free cells). In this case an upcall takes place via the VCI specific closure using the C calling convention. On Wanda such cells are put on a queue indistinguishable from any other and are processed later at a lower interrupt priority.
- The free cell buffer pool is managed using locks and condition variables. In Wanda concurrency is controlled by changing dynamic interrupt priority level.

Throughput performance is measured in user payload bits only. For an 100 Mbits/sec line this corresponds to 87.3 Mbits/sec.[5]

The amount of CPU time per jubilee for this process is configured in the image build file. This was varied over a large number of values for the contracted time value and the process was always allocated zero additional (i.e. non-guaranteed) time. During the time that the Fairisle device driver process is not running the queues in the Xilinx buffer memory build up and during the time that the process is running it must catch up.

There are two hard limits on the maximum traffic that can be processed by the device driver. These are shown in figure 6.7 where x is the bandwidth of the arriving stream and y is the amount of CPU not dedicated to the Fairisle device driver. First, the Xilinx buffer memory of 2048 cells must not overflow during the period that the process is descheduled. In this environment there is a slightly tighter bound than the one of section 5.2.4, the device driver is always run as the first process of any jubilee. The limit is that the total arrivals over the descheduled period must fit within the 2048 cell buffers.

[5]Two TAXI command symbols and 53 bytes gives 48 bytes of user payload; this gives a factor of $\frac{48}{55}$.

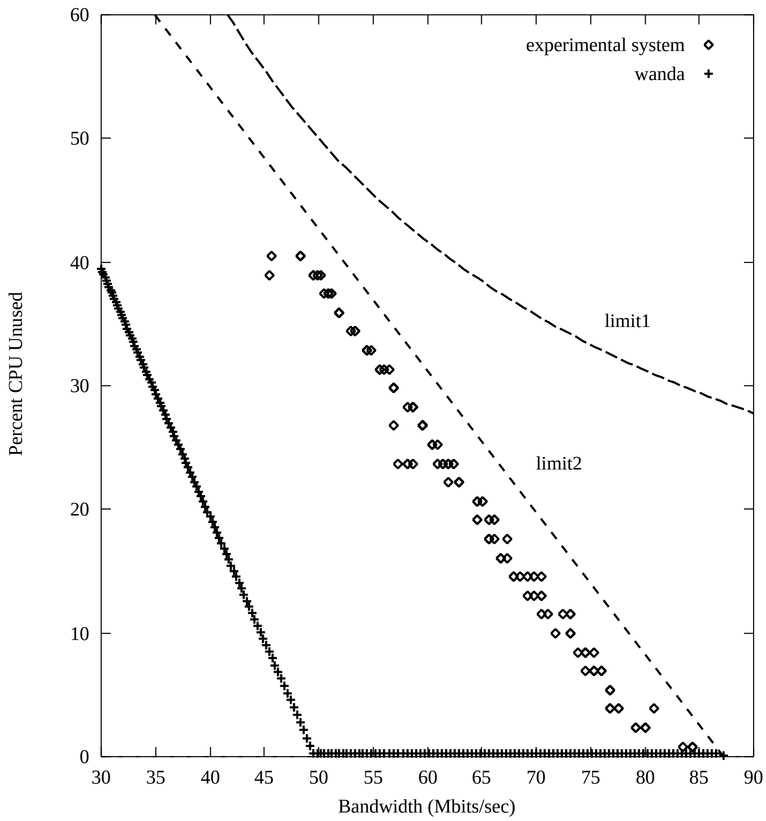

Figure 6.7: CPU Usage vs Fairisle Throughput

Second, since both the output fifo on the switch and the transmit decoupling queue on the Xilinx chip are very small, the software is essentially limited to a maximum of the outgoing line rate over the duration that it is running. In other words cells cannot be processed at faster than the line rate when the device driver has the CPU so the overall throughput is directly limited by the percentage of the CPU times the line rate.

Because Wanda always gives highest priority to the interrupts from the Fairisle hardware it spends a great deal of time in interrupt dispatching code. This leads to the entire CPU being dedicated to the task of cell forwarding for throughput values as low as 50 Mbit/sec. It is able to sustain the maximum line rate.

Fawn is massively superior because, although the optimum line is not quite reached anywhere, the CPU required for a particular throughput is much lower allowing the system to be configured to leave a reasonable amount of CPU to other processes (such as routeing updates) which would be starved in the Wanda system.

In addition to the tradeoff between CPU allocation and raw throughput performance, the jitter experienced by the traffic will be directly affected by the duration of the periods when the Fairisle process is descheduled. For this reason a service system may wish to allocate more CPU to the Fairisle process than is strictly necessary for the expected throughput alone. This is a policy decision which is likely to vary depending on the other tasks required of the node and its position in the network. The choice of the length of the jubilee is also directly relevant, this cost was discussed in section 6.2.2.

6.5 Fairisle Host Interface Performance

The experiments described in this section test both the effectiveness of the Rbuf mechanism and also the appropriateness of constructing the Fairisle device driver as a process from the point of view of data receiving or transmitting. As described in section 6.1 the theoretical transfer rate is limited by the bus bandwidth rather than the line rate, so the best comparison is with the performance achieved for the identical MSSAR/FDL [**Black94b**, **Black94a**] protocol on Wanda using identical hardware.

Two of the simple test programs on Wanda are a data sink and a data source. The former receives IOBufs from a socket and frees them as fast as possible (without referencing the data), the latter allocates an IOBuf of the default maximum size of 1920 bytes and sends it without initialising the data. Obviously on such tests the interaction with the cache is important. For the receiver, there is no per-byte cache penalty since the Arm cache only allocates lines on a read miss and the data in the IOBuf is never read. For the transmitter, since Wanda uses a MRU policy for allocating IOBufs the same IOBuf will be used over and over again so the data is likely to be in the cache.[6] During these experiments there was only one process on the Wanda machine.

For the same tests on Fawn, the Fairisle device driver process (which is where all the work is done) was configured with a guaranteed time of 62000 ticks (about 95% of each jubilee). The application was configured with 2500 ticks. Apart from 500 ticks for the console driver, no other activity of the system was given

[6] Note that the buffer size is small compared to the cache, and that Wanda does not clear buffers on allocation.

guaranteed time.

The test programs were configured to use a size identical to the Wanda number of 1920 bytes. Further, the same Rbuf was used for every packet to rule out any caching differences from the Wanda code. The Rbuf control areas used had a size of 512 bytes which represents a maximum pipeline of 32 packets – enough to last from one jubilee to the next.

6.5.1 Transmit

Using the Rbuf mechanism and the configuration described above transmission is possible at 13.0 Mbits/sec. This compares favourably with Wanda, which can achieve 11.0 Mbits/sec.

6.5.2 Receive

The Rbuf receiver can sink data in excess of 13.0 Mbits/sec. Again, this is considerably higher than the value attained by the Wanda system (it can sink data at approximately 10.5 Mbits/sec).

Of particular interest is the comparison between the Rbuf and Wanda code when receive overload occurs. On Wanda the system continues to take interrupts from the Fairisle hardware, all IOBufs are filled up with data queued for the overloaded connection, no other network activity can take place and the machine runs no processes (either other ones or the sink). Eventually the Fairisle cell buffer overflows; a fatal system error.

On the Rbuf scheme, if the receiving process is configured to block occasionally for a period of a few Jubilees (such that the pipeline goes dry) the system remains stable with the cells being efficiently discarded in the device driver without being copied to main memory and with minimum penalty to the system.

The Fawn/Rbuf combination is obviously better both in performance and in stability under load, both important considerations for a multi-media system.

6.6 Summary

This chapter has presented the experimental evaluation of the design proposed in this dissertation. This was considered in three main sections.

First the cost of using a micro-kernel with all operations including device drivers scheduled as normal processes was considered. It was shown that the interrupt latency was comparable with a conventional system in most cases and bounded above in the worst case by the (configurable) length of a Jubilee. The cost of each Jubilee was assessed and found to be mostly attributable to the effect of the fine grain sharing of the CPU on the cache. Null RPC time was also measured and, although this experimental platform does not include virtual address protection, was encouraging.

Second the effects of using events for intra-process scheduling were investigated. For local thread context switches the performance was found to be much superior to that of Wanda. Additionally it was shown that extra benefits were available due to the potential for application specific schedulers.

Third, the performance of the Fairisle device driver, both in the cell forwarding case and for received and transmitted data was measured. For cell forwarding, the explicit scheduling of the device driver by the system was shown to reduce the CPU requirements. For local I/O the device driver was shown to outperform Wanda, and distinctively, to remain stable in the presence of overload.

Chapter 7

Further Work

The Fawn system described in this dissertation is of necessity only a prototype. The measurements made in chapter 6 suggest that this is a powerful way to structure such an operating system. As further work, there are many issues involved in turning the prototype into a more robust and interesting system. There are also a number of further research topics which have been raised as a result of this work.

7.1 Operating System Development

A large amount of the design presented has already been adopted by the programming team working on the Operating System work package of the Pegasus project [**Evers94a**]. It is hoped that within that context the following can be resolved.

7.1.1 Event value overflow

The event system as described relies on monotonically increasing values in order to operate. On a 64-bit architecture (such as the Alpha) this can be easily implemented directly in a single machine word. For 32-bit architectures (such as the Arm used for the practical work) the possibility of overflow in a long-lived system is non-trivial. This can be solved by the usual mechanisms e.g. that used for transport sequence number rollover.[1]

7.1.2 Virtual Address System

The system as presented does not provide any form of virtual address management or protection. Instead the requirements and expectations of such a system have been taken into consideration where appropriate. In addition to the features described in section 5.3 the observations of section 6.2.2 should be taken into consideration.

7.1.3 Resource Recovery and Higher Level Issues

This experimental implementation lacks some of the support for recovering resources allocated to processes. Since this system does not include a file system or shell there is no dynamic demand for this activity. The system does provide for full recovery for resources which are allocated while establishing a communication channel (either Rbuf or IPC) which subsequently fails. The way to present recovery of resources to the application is strongly dependent on higher level models of longevity and re-negotiation procedures which are beyond the scope of this work.

Section 3.2.3 calls for a higher level process to administer the share of the CPU over a longer term interval. This process, known as the QoS manager, is a topic for further investigation. This system deliberately excludes a prototype for such a process since a static system is easier to analyse.

One system for re-negotiation is to include deliberate *inflation* in the value of CPU resources [**Larmouth75**]. In this system the allocations of processes are regularly downgraded by small amounts. Processes would need to re-negotiate regularly based on observed loss of resource to maintain their allocation. This method has a tendency to defragment the spare cycles in the system, returning them to the allocating authority.

[1] Values of events are always used in pairs so they can be compared by looking at the sign following a subtraction. This gives 31 bits of significance.

7.1.4 Heuristic Event Hints

There are various possible heuristics for inter-process scheduling within a particular level which could be investigated. One example would be to give the CPU to the process which was most recently the recipient of an event. Another would be the process which has the largest number of outstanding event notifications. Evaluation of the potential benefits and costs of such heuristics is futile unless carried out in the context of a much more general implementation and so has been left for further study.

7.1.5 Protocol Support

The experimental evaluation of the Rbuf architecture presented in chapter 6 was limited to considering the very basic operations. A fuller implementation, including support for many heterogeneous protocols and hardware interfaces is bound to include much scope for researching exact system structure for optimal performance and jitter.

7.1.6 Exponential jubilees

When a much more complete system is available for experimentation it is possible that one of the observations may be that non-guaranteed processes could benefit by being scheduled at a coarser granularity. If much of the CPU resource is committed at the guaranteed scheduling level then to share the remaining CPU fairly amongst processes whose allocations are solely at lower levels, the QoS manager may have to allocate very small slices. For some processes (such as compilers) the fine grain sharing may provide no benefit and merely consume resource due to context switching.

One possible alteration which could be researched is the use of exponential jubilees for lower scheduling levels. In such a scheme the reinitialisation of scheduling parameters for levels below the highest (guaranteed) level would occur at exactly half of the frequency of the immediately higher level. This would reduce the costs of fair sharing amongst non-guaranteed processes without destroying the fundamental principles of the jubilee concept such as the deterministic fairness over a bounded period and the ease of transfer of resources between processes.

Clearly such a subtle alteration could only be evaluated in the presence of genuine real-life loading.

7.2 Specialist Platforms

Apart from porting to a more general workstation with frame buffer and disk systems there is much to be investigated with respect to the operation of this system on shared and non shared multiprocessor hardware.

7.2.1 Shared memory multiprocessors

One of the key design aims of using events for communication between processes was that they indicated explicitly when synchronisation was being performed and that on a multiprocessor this could be used for causing synchronisation at the memory system level. One of the areas that will need investigation on such a platform is the virtual processor interface between the kernel and the intra-process scheduler. Negotiation of the maximum amount of true parallelism within a process is also a matter for further research.

7.2.2 Non-shared memory multiprocessors

Two particular instances of such a machine are the Fairisle Switch (considered as a whole) and the Desk Area Network. One of the particularly interesting possibilities is when a single process is running on an individual node (for example the X server on the frame buffer node). The similarity between the virtual processor interface of section 3.4 and the operation of a physical processor suggests that it may be possible to operate such a node *without any kernel whatsoever*. Clearly the intra-process scheduler would contain some specialist code, however the feasibility of such an implementation is much higher than for systems such as Wanda or Unix. In those systems, integrating applications into the kernel has been rarely attempted and when performed[2] requires substantial changes to the application code and / or adaptation layers.

[2] Such as MSRPC2 in Wanda and NFS in Unix.

7.2.3 Micro-threaded processors

The anaconda processor [**Moore94**] is a micro-threaded data-flow / control-flow hybrid. It uses interleaved execution of blocks of code known as micro-threads from different threads. A separate unit known as a matching store performs executability analysis based on availability of previous results. Multiple loads and stores may be posted from a micro-thread and the next micro-thread of the same thread will be scheduled when these have completed.

In this system the fundamental unit of concurrency control is a spin-lock comprised of two micro-threads. The first issues a load from the lock and a store of the *busy* indication to the lock. The second checks the value of the load that took place in the previous micro-thread and determines if it was successful in gaining access to the spin-lock, looping back if it was not.

In such a system atomic update of an event count can also be performed in two micro-threads by protecting each event with a spin-lock. The first micro-thread loads both the value of the event count as well as initiating the two transfers of the spin-lock. If the second determines that the lock was acquired then the new value can be written to the event count and the lock can be freed, otherwise a branch back must occur in the same manner as for a spin-lock.

Whilst this architecture provides many esoteric methods of communicating, once events have been implemented, code which uses generic concurrency control primitives may be ported easily building on the mechanisms discussed in chapter 3.

7.3 Inter-process Communication

The stub compiler used in the prototype generates poor quality stubs. This included redundant exception handling code, failure to optimise repeated tests, and separate (identity function) marshalling for fields of fixed length structures. The interaction between the clients and servers and the stubs has also been shown to be quite inefficient. One of the reasons for this is the interface definition language used. Although it is very strictly typed (sometimes irritatingly so) it provides little expression for the semantics of operations (i.e. what a particular operation actually *does* with the arguments and results) leading to large descriptions in comments. This is, of course, an issue for research in its own right. It is also very difficult to use it to interface with other interface definition languages or

RPC schemes. A particular problem with MIDDL which was experienced during the implementation of Fawn was the fact that MIDDL types are all designed for use in an external context; every type is specified in a particular number of bits. As a result use of types whose size is dependent on the natural sizes of various constructs on a particular platform can be difficult. One possible alternative is a generic stub compiler such as USC [**O'Malley94**] which can interconvert easily in a heterogeneous environment.

7.4 Desk Area Network

Within the Desk Area Network operating system project, one of the concerns has been to consider operating system scheduling of the ATM interconnect resources. One possible way of achieving this would be to run Fawn on all the nodes, where the fabric clock was used to generate jubilees and control process scheduling. The kernel scheduling algorithm could possibly be altered to run processes guaranteed time at particular well-defined offsets from the jubilee, and possibly include heuristics about when to run the DAN device driver code.

A further consideration is related to the use of the stream cache (see section 5.2.5) where a special area of the virtual address range is used for direct access to a multimedia stream which is routed to the processor cache rather than to main memory. This concept is taken further and generalised in Pratt's cache system [**Pratt94a**] where cache allocation may be process specific. In such an environment, the Rbuf data area may in fact represent an area of cache rather than an area of memory. The Rbuf control areas could be used to indicate when a process is finished with a particular part of the stream in the cache, where to put newly arriving material within the cache memory, and also to indicate in a homogeneous way the arrival (deduced from control information) of particular network packets within the cache for processing. This is a particularly interesting further research topic.

7.5 Other Observations

Whilst not related to the subject under consideration, a number of language issues have been observed during the experimental phase of this work which are recorded here for the benefit of future researchers.

As well as the problem noted in section 4.4.2 where availability of garbage collection could greatly increase the performance of many IPC calls, some surprising insights were gained from examination of code generated by the C compiler.[3] Code generation was close to optimal for this simple RISC processor i.e. it was difficult in general for a human to find optimisations which could be made to the resulting code except in one respect. The exception was pointer aliasing, where the semantics of C mean that the compiler cannot tell when an update at the target of one pointer variable may alter the value stored at another. In some pointer intense data structures the additional knowledge that the human has about which pointers can possibly be aliases for others could make a surprising difference. As an example of this the kernel level scheduler was considered in detail and instruction costs were calculated which determine that knowledge of pointer aliasing could reduce processor time by approximately 40%.

Together these indicate that the semantics of the C language may now be the principal barrier to better code rather than compiler technology. It is interesting to note that it is these same semantics which make formal verification of C so difficult.

The use of closures to pass necessary state into procedures entails some cost over a static linking method, due to the extra arguments, but it was observed that in many cases this cost is compensated for by lower cost access to that state. This is because on RISC processors it is more efficient to generate a memory access to an address which is a small delta from an address in a register than to access a full address which is bound into the code at link time and necessitates multiple instructions to build up as it is too large for a single immediate value. Of course, for stateless functions, static linking (even when it requires stubs) is still superior.

7.6 Summary

This chapter has identified some further potential work in this field. Some of the issues identified are to be addressed within the Pegasus project. Of particular interest is examining this system for use as the operating system of the Desk Area Network.

[3] gcc version 2.5.6.

Chapter 8

Conclusion

This dissertation has considered various problems associated with the scheduling and network I/O organisation found in conventional operating systems for effective support for multimedia applications which require Quality of Service within a general purpose operating system. A solution for these problems has been proposed in a micro-kernel structure. The proposed design is known as Fawn.

Chapter 2 describes the background to this work. Asynchronous Transfer Mode technology was discussed, in particular the Fairisle switch with which the author is particularly acquainted. The particular features of ATM switching which are relevant to this work are its high bandwidth (which is necessary for multimedia data), the provision of fine grain multiplexing to reduce jitter even in the presence of bursty sources, and that multiplexing is performed at the lowest possible level. Previous operating system and scheduling research is presented showing the increasing adoption of split-level scheduling and support for Quality of Service. Multimedia systems were typically used in a distributed environment and were considered to be soft real-time. Particular attention was drawn to the Nemo system, which was the inspiration for much of the work presented in this dissertation.

Chapter 3 began with a consideration of the use of priority for scheduling. By considering many typical examples from the literature and the author's experience it was shown that the use of priority was entirely inappropriate for a multimedia system. In particular the Rate Monotonic and Earliest Deadline First algorithms were shown to have high costs, to have delusive assumptions for a general purpose system, and to be inappropriate for a distributed system.

The Fawn design presents a system where processes are guaranteed a share of the CPU over a system wide period called a Jubilee. Access to the CPU at a finer granularity is probabilistic rather than certain. This is exactly the sort of guarantee required by soft real-time applications. This jubilee mechanism makes scheduling decisions extremely simple and facilitates contracts between clients and servers.

The virtual processor interface of the Fawn system is described, concentrating on those areas where it differs from or extends the Nemo system. A mechanism for interprocess communication based on event channels is devised. It is shown to be also highly suitable for the virtualisation of device interrupts, and the delivery of information about system time to the application – an important concern of any multimedia application.

Intra-process scheduling is then considered. For homogeneity with the inter-process event mechanism, event-counts and sequences are chosen for intra-process synchronisation primitives. Implementations of common synchronisation schemes using these primitives is shown to be simple and efficient.

Chapter 4 described some mechanisms used for same machine RPC in other systems. The migrating model was considered inappropriate because it led to problems in accounting for CPU, poor cache locality, and an unacceptable interface for device drivers (or other processes which require non-blocking communication primitives). The mechanisms used for same-machine RPC in Fawn, based on the switching model, were discussed. This comprises pairwise shared memory and use of event channels for synchronisation. The interface definition language used was adopted from the Pegasus project.

The chapter also considered the naming and binding schemes used in Fawn; and additionally shared libraries, exceptions, bootstrapping and other general infrastructure issues were discussed.

Chapter 5 examined bulk stream I/O. Three previous schemes are considered in detail. The Rbufs scheme is proposed for use in Fawn. The principal feature is the separation of the three issues of data transmission, structure / aggregation control, and memory allocation. This separation allows for great flexibility in the management of communications within the machine. Rbufs are designed to minimise copying in order to support high bandwidth applications and also the explicit scheduling of network device drivers.

Various situations were considered in the context of typical workstation hardware.

Rbufs were shown to be highly effective particularly in the support of application data units, and multicast; both areas where previous work has been deficient.

Chapter 6 described a platform on which an instance of the Fawn design has been constructed; the Fairisle port controller. Various features of the design were evaluated experimentally. Interrupt latency to the device driver thread was shown to be approximately the same as what would be expected for the entry of the device driver handler for a conventional Unix system. In the worst case it was bounded by the length of the jubilee.

The costs of adopting the jubilee mechanism for CPU allocation were measured and shown to be small and primarily due to the caching effects of the fine grain sharing that it provides. The performance of same machine RPC was also measured.

Three different implementations of the intra-process scheduler were then considered. It was shown that applications could increase their performance by choosing a scheduler which met their particular requirements. One of the benefits of a single address space system, the sharing of code, was also considered. It was shown that placing the intra-process scheduler within the shared library reduced scheduling latency.

This chapter then moved on to consider the impact of the Fawn system on I/O. The performance of the system when used as a cell forwarding engine in a Fairisle switch was measured. The Fawn system was shown to require much less CPU to forward at almost any rate than the dedicated Wanda device driver due to *explicit network scheduling* – the abolition of unnecessary interrupt dispatching costs found in other systems. The suitability of Rbufs was considered by measuring the I/O throughput of the system in comparison with the same protocols in the Wanda micro-kernel, and shown to sustain higher rates. In particular the Rbuf / Fawn combination was observed to be stable under periods of overload.

Finally chapter 7 considered many of the future areas of research in which this work may prove beneficial, and other issues which have come to light as a result.

At least one measure of the success of this work is the extent to which it has been adopted by the Pegasus project. All of the work of chapter 3 has been adopted except the use of jubilees. The IPC transport mechanisms of chapter 4, and the Rbuf design in its entirety, have already been included in the current design for the Nemesis system.

The pivotal features of the Fawn design are that the processing of device interrupts is performed by user-space processes which are scheduled by the system like any other, that events are used for both inter- and intra- process synchronisation, and Rbufs, an especially developed high performance I/O buffer management system.

It is the thesis of this dissertation that the fundamental job of an operating system kernel is to implement fine grain sharing of the CPU between processes and hence synchronisation between those processes. System performance, both with respect to throughput and soft real-time dependability, has been shown to be enhanced. This is due to the empowering of processes to perform task specific optimisations.

Bibliography

[**Anderson90**] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* Technical Report 90-04-02, Department of Computer Science, University of Washington, April 1990. Revised October 1990. (pp 28, 48)

[**ANSA89**] Architecture Projects Management Ltd. *ANSA Reference Manual release 01.01*, March 1989. (pp 9, 43, 49)

[**ARM91**] Advanced RISC Machines. *ARM6 Macrocell Datasheet*, 0.5 edition, November 1991. (p 83)

[**ARM92**] Advanced RISC Machines. *ARM610 Datasheet*, 1.0 edition, 1992. (p 83)

[**Barham95**] P. Barham, M. Hayter, D. McAuley, and I. Pratt. *Devices on the Deak Area Network.* IEEE Journal on Selected Areas in Communication, 13(1), January 1995. To Appear in the special issue on ATM LANs. (p 2)

[**Bershad89**] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. *Lightweight Remote Procedure Call.* Technical Report 89-04-02, Department of Computer Science, University of Washington, April 1989. (pp 46, 47, 63)

[**Bershad91**] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. *User-Level Interprocess Communication for Shared Memory Multiprocessors.* ACM Transactions on Commputer Systems, 9(2):175–198, May 1991. (p 48)

[**Birrell87**] A.D. Birrell and J.V. Guttag. *Synchronization Primitives for a Multiprocessor: A formal specification.* Technical Report 20, Digital Equipment Corporation Systems Research Center, 1987. (pp 37, 41)

[**Birrell93**] A. Birrell. *Taming the Windows NT (TM) Thread Primitives.* A presentation before the Systems Research Group of the University of Cambridge Computer Laboratory, October 1993. (pp 37, 41)

[**Birrell94**] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. *Network Objects.* Technical Report 115, Digital Equipment Corporation Systems Research Center, February 1994. (p 50)

[**Black94a**] R. Black. *FDL Cell formats and Meta-Signalling.* In ATM Document Collection 3 (The *Blue* Book), chapter 5. University of Cambridge Computer Laboratory, March 1994. (p 98)

[**Black94b**] R. Black. *Segmentation and Reassembly.* In ATM Document Collection 3 (The *Blue* Book), chapter 3. University of Cambridge Computer Laboratory, March 1994. (p 98)

[**Black94c**] R. Black and S. Crosby. *Experience and Results from the Implementation of an ATM Socket Family.* In USENIX Winter 1994 Conference, pages 143–152, January 1994. (pp 6, 19)

[**Black94d**] R.J. Black, I.M. Leslie, and D.R. McAuley. *Experiences of building an ATM switch for the Local Area.* In Computer Communication Review, volume 24, pages 158–167. ACM SIGCOMM, September 1994. (pp 6, 19, 96)

[**Bosch94**] P. Bosch. *A Cache Odyssey.* Master's thesis, University of Twente, Faculty of Computer Science, June 1994. Also available as Pegasus report number 94–6. (p 18)

[**Brackmo94**] L. Brackmo, S. O'Malley, and L. Peterson. *TCP Vegas: New Techniques for Congestion Detection and Avoidance.* In Computer Communication Review, volume 24, pages 24–35. ACM SIGCOMM, September 1994. (p 72)

[**Burrows88**] M. Burrows. *Efficient Data Sharing.* Technical Report 153, University of Cambridge Computer Laboratory, December 1988. pages 30-31. Ph.D. Dissertation. (p 19)

[**Burrows94**] M. Burrows. *The software of the OTTO ATM network interface.* Personal Communication, October 1994. (p 19)

[**Cardelli89**] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. *Modula-3 Report (revised).* Technical Report 52, Digital Equipment Corporation Systems Research Center, November 1989. (p 44)

[**Carter89**] J. Carter and W. Zwaenepoel. *Optimistic bulk data transfer protocols.* Sigmetrics and Performance, 17(1):61–69, May 1989. (p 66)

[**Chen94**] J. Chen. *Memory Behaviour of an X11 Window System.* In USENIX Winter 1994 Conference, pages 189–199, January 1994. (p 90)

[**Coulson93**] G. Coulson, G. Blair, P. Robin, and D. Shepherd. *Extending the Chorus Micro-kernel to support Continuous Media Applications.* In Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video, pages 49–60, November 1993. (p 12)

[**Crosby94**] S. Crosby, R. Hayton, and T. Roscoe. *MSRPC2 User Manual.* In ATM Document Collection 3 (The *Blue* Book), chapter 16. University of Cambridge Computer Laboratory, March 1994. (p 49)

[**DEC91**] Digital Equipment Corporation Workstation Systems Engineering. *MAXine System Module Functional Specification Revision 1.2*, February 1991. (p 61)

[**DEC92**] Digital Equipment Corporation Workstation Systems Engineering. *Flamingo Macrocoder's Manual*, November 1992. (p 61)

[**DEC93**] Digital Equipment Corporation TURBOchannel Industry Group. *TURBOchannel Specifications Version 3.0*, 1993. (p 69)

[**Dixon91**] M.J. Dixon. *System Support for Multi-Service Traffic.* Technical Report 245, University of Cambridge Computer Laboratory, September 1991. Ph.D. Dissertation. (pp 9, 16, 17, 18, 62)

[**Druschel93**] P. Druschel and L. Peterson. *Fbufs: A High-Bandwidth Cross-Domain Transfer Facility.* In Proceedings of the fourteenth ACM Symposium on Operating Systems Principles, pages 189–202, December 1993. (p 64)

[**Druschel94**] P. Druschel, L. Peterson, and B. Davie. *Experiences with a High-Speed Network Adaptor: A Software Perspective.* In Computer Communication Review, volume 24, pages 2–13. ACM SIGCOMM, September 1994. (pp 65, 69)

[**Edwards94**] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton. *User-space protocols deliver high performance to applications on a low-cost Gb/s LAN.* In Computer Communication Review, volume 24, pages 14–23. ACM SIGCOMM, September 1994. (p 69)

[**Evers94a**] D. Evers. *Nemesis Structure and Interfaces.* Pegasus Project Internal Memorandum, August 1994. (p 101)

[**Evers94b**] D.M. Evers. *Distributed Computing with Objects.* Technical Report 332, University of Cambridge Computer Laboratory, March 1994. Ph.D. Dissertation. (pp 49, 53)

[**Fairbairns93**] R. Fairbairns. *Experience of implementing POSIX threads on Wanda.* Personal communication, March 1993. (pp 37, 41)

[**Ford94**] B. Ford and J. Lepreau. *Evolving Mach 3.0 to a Migrating Thread Model.* In USENIX Winter 1994 Conference, pages 97–114, January 1994. (p 47)

[**Forum93**] The ATM Forum. *ATM user-network interface specification Version 3.0.* Prentice Hall, 1993. (p 64)

[**Fraser92**] A. Fraser, C. Kalmanek, A. Kaplan, E. Marshall, and R. Restrick. *Xunet 2: A Nationwide Testbed in High-Speed Networking.* In IEEE Infocomm, pages 582–589, May 1992. (p 4)

[**Fraser93**] S. Fraser. *Early Experiments with Asynchronous Time Division Networks.* IEEE Network, 7(1):12–26, January 1993. (p 4)

[**Greaves94**] D. Greaves. *The Olivetti Research "Yes V2" option module.* In ATM Document Collection 3 (The *Blue* Book), chapter 33.

University of Cambridge Computer Laboratory, March 1994. (p 68)

[**Hamilton93**] G. Hamilton and P. Kougiouris. *The Spring nucleus: a microkernel for objects.* Technical Report SMLI TR-93-14, Sun Microsystems Laboratories, April 1993. Also in USENIX 93. (pp 47, 56)

[**Hayter91**] M. Hayter and D. McAuley. *The Desk Area Network.* ACM Operating Systems Review, 25(4):14–21, May 1991. Also available as University of Cambridge Computer Laboratory Technical Report number 228. (p 2)

[**Hayter93**] M. Hayter. *A Workstation Architecture to Support Multimedia.* Technical Report 319, University of Cambridge Computer Laboratory, September 1993. Ph.D. Dissertation. (pp 2, 74)

[**Hayter94a**] M. Hayter and R. Black. *Fairisle Port Controller Design and Ideas.* In ATM Document Collection 3 (The *Blue* Book), chapter 23. University of Cambridge Computer Laboratory, March 1994. (pp 8, 83)

[**Hayter94b**] M. Hayter and R. Black. *FPC3 Xilinx Xi5 Design and Notes.* In ATM Document Collection 3 (The *Blue* Book), chapter 25. University of Cambridge Computer Laboratory, March 1994. (pp 8, 83)

[**Hopper78**] A. Hopper. *Local Area Computer Communication Networks.* Technical Report 7, University of Cambridge Computer Laboratory, 1978. Ph.D. Dissertation. (pp 4, 67)

[**Hopper90**] A. Hopper. *Pandora – an experimental system for multimedia applications.* ACM Operating Systems Review, 24(2):19–34, April 1990. (p 2)

[**Hutchinson91**] N. Hutchinson and L. Peterson. *The* x-*Kernel: An Architecture for Implementing Network Protocols.* IEEE Transactions on Software Engineering, 17(1):64–75, January 1991. (p 9)

[**Hyden94**] E. Hyden. *Operating System Support for Quality of Service.* Technical Report 340, University of Cambridge Computer Laboratory, February 1994. Ph.D. Dissertation. (pp 11, 20, 22)

[**ISO90**] International Organisation for Standardization. *Programming languages - C*, 1990. Draft international standard ISO/IEC DIS 9899 UDC 681.3.06 : 519.682 : 800.92. (p 44)

[**Jacobson93**] V. Jacobson. *The Synchronisation of Periodic Routing Messages.* In Computer Communication Review. ACM SIGCOMM, September 1993. (p 16)

[**Jardetzky92**] P.W. Jardetzky. *Network File Server Design for Continuous Media.* Technical Report 268, University of Cambridge Computer Laboratory, October 1992. Ph.D. Dissertation. (pp 18, 63)

[**Johnson81**] M. Johnson. *Exception handling in domain based systems.* Technical Report 27, University of Cambridge Computer Laboratory, September 1981. Ph.D. Dissertation. (p 48)

[**Jones93**] A. Jones and A. Hopper. *Handling Audio and Video Streams in a Distributed Environment.* Proceedings of the fourteenth ACM Symposium on Operating Systems Principles, pages 231–243, December 1993. Also available as Olivetti Research Ltd. Technical Report number 93–4. (p 18)

[**Larmouth75**] J. Larmouth. *Scheduling for a Share of the Machine.* Software – Practice and Experience, 5:29–49, January 1975. Also available as University of Cambridge Computer Laboratory Technical Report number 2, October 1974. (p 102)

[**Leffler84**] S. Leffler and M. Karels. *Trailer Encapsulations.* Internet Request for Comment Number 893, April 1984. (p 66)

[**Leffler89**] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989. (pp 17, 59)

[**Leslie83**] I. Leslie. *Extending the Local Area Network.* Technical Report 43, University of Cambridge Computer Laboratory, 1983. Ph.D. Dissertation. (p 5)

[**Leslie91**] I. Leslie and D. McAuley. *Fairisle: An ATM Network for the Local Area.* In Computer Communication Review, volume 21(4), pages 327–336. ACM SIGCOMM, September 1991. (pp 2, 6)

[**Liu73**] C.L. Liu and J. Layland. *Scheduling Algorithms for Multiprogramming in a hard Real-Time Environment.* Journal of the Association for Computing Machinery, 20(1):46–61, January 1973. (p 20)

[**McAuley90**] D.R. McAuley. *Protocol Design for High Speed Networks.* Technical Report 186, University of Cambridge Computer Laboratory, January 1990. Ph.D. Dissertation. (pp 6, 9, 64)

[**McAuley94**] D. McAuley. *The design of the ARX operating system.* Personal communication, September 1994. (p 48)

[**McCanne93**] S. McCanne and V. Jacobson. *The BSD Packet Filter: A New Architecture for User-level Packet Capture.* In USENIX Winter 1993 Conference, pages 259–269, January 1993. (p 69)

[**Mills89**] D. Mills. *Internet Time Synchronisation: The Network Time Protocol.* Internet Request for Comment Number 1129, October 1989. (p 33)

[**Mogul**] J. Mogul. *Livelock in Ultrix 4.2.* Internal Technical Report, Digital Equipment Corporation Western Research Laboratory. (p 19)

[**Mogul90**] J. Mogul. *Efficient Use of Workstations for Passive Monitoring of Local Area Networks.* In Computer Communication Review, volume 20. ACM SIGCOMM, September 1990. (p 69)

[**Moore94**] S. Moore. *Multithreaded Processor Design.* PhD thesis, University of Cambridge Computer Laboratory, October 1994. (p 105)

[**Mullender92**] S. Mullender, I. Leslie, and D. McAuley. *Pegasus Project Description.* Technical Report 92-1, Pegasus Esprit Project, September 1992. Also available as University of Cambridge Computer Laboratory Technical Report number 281. (pp 2, 10)

[**Mullender94**] S. Mullender, I. Leslie, and D. McAuley. *Operating-System Support for Distributed Multimedia.* In USENIX Summer 1994 Conference, July 1994. (p 10)

[**Nakamura93**] A. Nakamura. *An investigation of real-time synchronisation.* PhD thesis, University of Cambridge Computer Laboratory, 1993. (p 15)

[**Needham82**] R. Needham and A. Herbert. *The Cambridge distributed computing system.* International computer science series. Addison-Wesley (London), 1982. (p 57)

[**Newman89**] P. Newman. *Fast Packet Switching for Integrated Services.* Technical Report 165, University of Cambridge Computer Laboratory, 1989. Ph.D. Dissertation. (p 6)

[**Nicolaou90**] C. Nicolaou. *A Distributed Architecture for Multimedia Communication Systems.* Technical Report 220, University of Cambridge Computer Laboratory, December 1990. Ph.D. Dissertation. (p 2)

[**Nieh93**] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. *SVR4 UNIX Scheduler Unacceptable for Multimedia Applications.* In Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video, pages 35–47, November 1993. (p 17)

[**Oikawa93**] S. Oikawa and H. Tokuda. *User-Level Real-Time Threads: An Approach Towards High Performance Multimedia Threads.* In Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video, pages 61–71, November 1993. (p 13)

[**O'Malley90**] S. O'Malley, M. Abbott, N. Hutchinson, and L. Peterson. *A Transparent Blast Facility.* Internetworking: Research and Experience, 1(2):57–75, December 1990. (p 67)

[**O'Malley94**] S. O'Malley, T. Proebstring, and A. Montz. *USC: A Universal Stub Compiler.* In Computer Communication Review, volume 24, pages 295–306. ACM SIGCOMM, September 1994. (p 106)

[**Owicki89**] S. Owicki. *Experience with the Firefly Multiprocessor Workstation.* Technical Report 51, Digital Equipment Corporation Systems Research Center, September 1989. (p 49)

[**Pike91**] R. Pike, D. Presotto, K. Thomson, H. Trickey, T. Duff, and G. Holzmann. *Plan 9: The Early Papers.* Computing Science Technical Report 158, AT&T Laboratories, July 1991. (p 55)

[**Pratt94a**] I. Pratt. *Hardware Support for Operating System Support for Continuous Media.* University of Cambridge Computer Laboratory Ph.D. Research Proposal, July 1994. (pp 90, 106)

[**Pratt94b**] I. Pratt. *Internet Connectivity on the Desk Area Network.* Personal communication, April 1994. (p 66)

[**Pratt94c**] I. Pratt and P. Barham. *The ATM Camera V2: AVA200.* In ATM Document Collection 3 (The *Blue* Book), chapter 36. University of Cambridge Computer Laboratory, March 1994. (p 73)

[**Reed77**] D. Reed and R. Kanodia. *Synchronization with eventcounts and sequencers.* Technical Report, MIT Laboratory for Computer Science, 1977. (pp 27, 35)

[**Rodeheffer94**] T. Rodeheffer. *The hardware of the OTTO ATM network interface.* Personal Communication, January 1994. (p 19)

[**Roscoe94a**] T. Roscoe. *Linkage in the Nemesis Single Address Space Operating System.* ACM Operating Systems Review, 28(4):48–55, October 1994. (p 45)

[**Roscoe94b**] T. Roscoe. *The MIDDL Manual, 3rd Edition.* available by anonymous ftp from ftp.cl.cam.ac.uk in pegasus/Middl.ps.gz, January 1994. now superseded by the 4th edition. (p 49)

[**Saltzer78**] J.H. Saltzer. *Naming and Binding of Objects*, chapter 3A, pages 99–208. Number 60 in Lecture Notes in Computer Science. Springer-Verlag, 1978. (pp 15, 43, 45)

[**Schroeder89**] M. Schroeder and M. Burrows. *Performance of Firefly RPC.* Technical Report 43, Digital Equipment Corporation Systems Research Center, April 1989. (p 62)

[**Sha87**] L. Sha, R. Rajkumar, and J.P. Lehoczky. *Priority Inheritance Protocols.* Technical Report CMU-CS-87-181, Carnegie Mellon Computer Science Department, December 1987. (p 15)

[Shand92] M. Shand. *Measuring System Performance with Reprogrammable Hardware.* Technical Report 19, Digital Equipment Corporation Paris Research Laboratory, August 1992. (p 87)

[Sreenan93] C.J. Sreenan. *Synchronisation services for digital continuous media.* Technical Report 292, University of Cambridge Computer Laboratory, March 1993. Ph.D. Dissertation. (pp 17, 75)

[Stroustrup91] B. Stroustrup. *The C++ programming language.* Addison-Wesley, second edition, 1991. (p 44)

[Sun94] *Re: Sun's Sparc technology business discloses next-generation processor.* USENET News article in `comp.arch` `Message-ID: <35ppf8$bf2@engnews2.Eng.Sun.COM>` from M. Tremblay (tremblay@flayout.Eng.Sun.COM), September 1994. (p 47)

[Tanenbaum81] A. Tanenbaum and S. Mullender. *An Overview of the Amoeba Distributed Operating System.* ACM Operating Systems Review, 15(3):51–64, July 1981. (p 8)

[Temple84] S. Temple. *The Design of a Ring Communication Network.* Technical Report 52, University of Cambridge Computer Laboratory, January 1984. Ph.D. Dissertation. (pp 2, 67)

[Thacker87] C. Thacker, L. Stewart, and E. Satterthwaite. *Firefly: A Multiprocessor Workstation.* Technical Report 23, Digital Equipment Corporation Systems Research Center, December 1987. (p 62)

[Trickey93] H. Trickey. *Internals of Plan 9 Naming.* Personal Communication, September 1993. (p 55)

[Yuhara94] M. Yuhara, C. Maeda, B. Bershad, and J. Moss. *The MACH Packet Filter: Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages.* In USENIX Winter 1994 Conference, pages 153–165, January 1994. (p 69)