Devices in a Multi-Service Operating System

Paul Ronald Barham

Churchill College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

July 1996

Summary

Increases in processor speed and network and device bandwidth have led to general purpose workstations being called upon to process continuous media data in real time. Conventional operating systems are unable to cope with the high loads and strict timing constraints introduced when such applications form part of a multi-tasking workload. There is a need for the operating system to provide fine-gained reservation of processor, memory and I/O resources and the ability to redistribute these resources dynamically. A small group of operating systems researchers have recently proposed a *vertically-structured* architecture where the operating system kernel provides minimal functionality and the majority of operating system code executes within the application itself. This structure greatly simplifies the task of accounting for *processor* usage by applications. The prototype Nemesis operating system embodies these principles and is used as the platform for this work.

This dissertation extends the provision of Quality of Service guarantees to the I/O system by presenting an architecture for device drivers which minimises crosstalk between applications. This is achieved by clearly separating the data-path operations which require careful accounting and scheduling, and the infrequent control-path operations which require protection and concurrency control. The approach taken is to abstract and multiplex the I/O data-path at the lowest level possible so as to simplify accounting, policing and scheduling of I/O resources and enable application-specific use of I/O devices.

The architecture is applied to several representative classes of device including network interfaces, network connected peripherals, disk drives and framestores. Of these, disks and framestores are of particular interest since they must be shared at a very fine granularity but have traditionally been presented to the application via a window system or file-system with a high-level and coarse-grained interface.

A device driver for the framestore is presented which abstracts the device at a low level and is therefore able to provide each client with guaranteed bandwidth to the framebuffer. The design and implementation of a novel client-rendering window system is then presented which uses this driver to enable rendering code to be safely migrated into a shared library within the client.

A low-level abstraction of a standard disk drive is also described which efficiently supports a wide variety of file systems, and other applications requiring persistent storage, whilst providing guaranteed rates of I/O to individual clients. An extent based file system is presented which can provide guaranteed rate file access and enables clients to optimise for application-specific access patterns.

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 my dissertation has already been, or is being currently submitted for any such degree, diploma or other qualification.

Acknowledgements

I would like to thank my supervisor Derek McAuley for providing invaluable encouragement most evenings throughout the duration of this work, and the entire Systems Research Group for providing both the infrastructure and atmosphere necessary for research of this kind. Particular thanks are due to Richard Black, Simon Crosby, David Evers, Robin Fairbairns, Mark Hayter, Eoin Hyden, Ian Leslie, Roger Needham and Ian Pratt.

For reading and commenting on drafts of this dissertation, I am indebted to Richard Black, Simon Crosby, Mark Hayter, Ian Leslie, Derek McAuley, Ian Pratt and Timothy Roscoe. Robin Fairbairns deserves a special mention for invaluable assistance in mastering the mystic runes of LaTeX, and Martyn Johnson for such smooth and efficient system management.

I would also like to thank Lance Berc, Mike Burrows, Mark Hayter, Mike Schroeder, Bob Taylor and numerous other members of staff at Digital Equipment Corporation's Systems Research Centre, for an extremely enjoyable and rewarding internship during the Summer of 1994, and the continuous encouragement since.

This work was supported by a grant from the EPSRC (formerly SERC), and a CASE studentship funded by Olivetti Research Ltd. and Nemesys Research Ltd.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This dissertation presents the design and implementation of an I/O architecture for the Nemesis multi-service operating system. The architecture supports the execution of both conventional and multimedia applications by the provision of guaranteed Qualities of Service (QoS) to individual applications at the lowest possible level.

1.1 Motivation

The continuing advances in communications network and processor speeds are enabling interesting new areas of computation hitherto considered infeasible. General-purpose processing of continuous media (CM) data is perhaps the most challenging of these areas. CM data differs from conventional data in two important respects:

- A CM stream is often tolerant to some degree of information loss. This property can be exploited by applications when computational and I/O resources are scarce.
- The usefulness of CM data is dependent on the timeliness with which it is delivered.

Applications processing CM data can often produce acceptable results at a number of different quality levels, however it is usually the case that they will be able to perform more effectively if they know in advance what level of system resources they will have access to. For this reason, the notion of Quality of Service (QoS), commonly used in the field of networking research, is being extended up into the operating system.

This dissertation addresses a specific problem which may be characterised as follows:

- Supporting applications which perform general-purpose computation with CM data types and high-bandwidth I/O.
- Simultaneous execution of a number of tasks on the same machine, some of which are highly time-sensitive, some of which require only *best-effort* service.
- Applications may be both computationally intensive, and I/O intensive, and may change their behaviour *dynamically*, for example in response to external events or as a result of processing a CM stream.
- Demand for operating system provided resources is far in excess of that which is available. The machine is invariably running at 100% load.
- Adequate information must be provided to enable application-specific degradation when insufficient resources are available.

The effective support of a number of simultaneous activities on a single machine relies upon the ability to place resource firewalls between applications. The aim is to remove QoS *crosstalk* whereby use of a resource by one client has an adverse effect on the QoS received by other clients.

General purpose workstations are readily equipped with multimedia peripheral devices to provide capture and playback of CM streams. In such a system it should be possible to build interesting new applications which both store and process CM data. User interfaces may be extended to include face recognition, eyeball-tracking and gesture or voice input. Video recording applications may index features automatically and support searching for particular scenes. These new applications must peacefully coexist with more conventional workloads such as interactive text editing and batch computation.

1.2 Background

Much operating system research has been devoted to supporting the execution of multimedia applications. Many researchers have taken existing operating systems as a starting point and attempted to retrofit support for "real-time" activities, often by extending the CPU scheduler. Resource allocation has traditionally been performed using a notion of "priority" to determine which task should receive a given resource. It has been demonstrated that priority based allocation is unacceptable in a multimedia system, and that schemes are necessary which determine both the *quantity* of resource and the *time* at which it should allocated.

When using a conventional operating system for multimedia, particularly in the presence of high-bandwidth I/O, the majority of resources are consumed by device drivers, shared server processes and the operating system, rather than by the application process. A small group of operating system researchers have recently proposed a radical restructuring of the operating system to migrate functionality into the application itself where resource usage is more easily accounted - so called *vertically-structured* operating systems.

To date however, all research in this area has focussed solely on the allocation of CPU resource. Multimedia applications, by definition, perform large amounts of I/O and though guarantees of CPU resource are necessary, they are not sufficient.

A number of extensions have been made to conventional operating systems to support end-to-end QoS for network I/O. These have typically involved using real-time threads within the kernel to service individual network connections at guaranteed rates. These ad-hoc solutions do not fully solve the problem described above, and are inapplicable in the context of a vertically-structured operating system.

Until the advent of the personal workstation, it was common practice to implement resource firewall mechanisms simply by providing separate hardware. For example, the I/O *channels* of IBM mainframes effectively isolate the performance of distinct I/O activities by having all I/O data-path operations performed by separate channel controllers. Out-of-band control operations are performed on the main processor by the operating system. The inflexibility and increased expense of replicating I/O hardware makes it an impractical approach for building a multimedia workstation.

Contemporary workstation designs do not provide any hardware support for per-connection control of I/O resources by the operating system. Whilst prototype workstations have been constructed which address this problem, it may be some time before they are commonplace. A software solution is therefore required.

1.3 Contribution

The author has built upon the work described in [Hyden94], [Roscoe95] and [Black94], investigating the extension of a QoS-based operating system to provide guarantees of I/O performance which are useful at the application level.

The thesis of this work is that a vertically-structured operating system requires a device driver architecture where I/O resources are:

- Multiplexed only once,
- Multiplexed at the lowest possible level,
- Protected at the lowest possible level, and
- Scheduled at a fine granularity in order to provide QoS guarantees.

— and that using these design principles it is possible to implement high-level functions within the application itself without sacrificing the protection afforded by traditional server-based approaches.

This dissertation both describes the device driver architecture and demonstrates its application to a number of representative devices. The effectiveness of the device abstraction is evaluated by implementation of higher level services which allow applications to take full advantage of the QoS guarantees provided by the operating system. In addition, this dissertation also presents:

- Mechanisms for low-level protection of problematic devices which have traditionally been accessed via servers with high-level interfaces.
- An operating system mechanism for providing applications with protected and accounted access to devices which do not support DMA.

- A new disk head scheduling algorithm which delivers guaranteed rate I/O and supports best-effort clients without excessively sacrificing disk performance.

The above issues are illustrated by describing a prototype implementation for the Nemesis multi-service operating system, running on the DEC 3000/400 Sandpiper workstation. The Nemesis system was developed in Cambridge over a 2 year period by Timothy Roscoe, David Evers and the author, with significant influence from the Fawn system written by Richard Black.

1.4 Outline

Chapter 2 provides background material relevant to this work and describes the research environment in which it was carried out.

Chapter 3 describes the structure of Nemesis, an operating system structured so as to support application level Quality of Service. The architectural principles of the design are presented, and a brief overview of the prototype implementation is given. The remainder of the chapter considers the areas of Nemesis where Quality of Service provision has not been addressed, focussing on the problems presented by multimedia I/O.

Chapter 4 discusses the scheduling of I/O resources. It provides an overview of the architecture of a contemporary workstation and highlights some of the design features which present particular problems for a multi-service operating system.

Chapter 5 presents a generic device driver architecture designed to provide individual applications with secure, direct access to the underlying hardware resources with Quality of Service guarantees.

The following two chapters provide an evaluation of the architecture by its application to two particularly troublesome devices. The abstraction of the framebuffer device and a complete implementation of a novel window system are presented in chapter 6. Chapter 7 describes the abstraction of disk devices and a new file-system which is able to provide guaranteed qualities of service to its clients.

Finally, chapter 8 summarises the main arguments of the dissertation and makes some suggestions for further work.

Chapter 2

Background

2.1 Environment

There has been much previous work in Cambridge in the areas of high-speed networks, distributed computing and multimedia systems, with a strong tradition of building and using these systems. Early work on Asynchronous Transfer Mode (ATM) networks led to deployment of the Cambridge Fast Ring (CFR) [Temple84] enabling much of the subsequent continuous media research. The CFR was later supplemented by the Fairisle ATM network [Leslie91]. An important focus of the ATM research in Cambridge has been the elimination of layered-multiplexing [Tennenhouse89], especially in the network protocol stack [McAuley89].

These ATM networks were used to transport digital audio and video for the Pandora's Box [Hopper90], a continuous media peripheral providing support for capture and display of audio and video in a workstation environment. The Pandora system made use of distributed computing technology [Nicolaou90] to provide applications such as multi-party video conferencing and mail. A continuous media file-server was also provided [Jardetzky92].

Although highly successful in supporting "first generation" multimedia applications concerned primarily with presentation and orchestration issues, Pandora included no support for *processing* of continuous media data - digital audio and video could not be accessed by the workstation itself.

With continuing increases in processor and workstation speeds, it quickly became possible to provide all of the functionality of a Pandora system in software on an ATM-equipped general-purpose workstation [Barham95], although it rapidly became apparent that the bus-based design of contemporary workstations forces continuous media data to traverse the bus several times en-route to its eventual destination. The Desk Area Network project [Hayter91] led to a prototype second generation multimedia workstation which addresses this problem [Barham95].

Second generation multimedia applications, where continuous media data is processed as well as merely being transferred, present enormous problems for conventional operating systems. The Pegasus project [Leslie93] proposed a completely new operating system architecture which could support these demanding applications, and resulted in the prototype Nemesis operating system [Leslie96].

The implementation work in support of the ideas presented in this dissertation has been performed entirely over Nemesis on a conventional workstation equipped with a number of multimedia peripherals, many of which were designed and constructed in the Computer Laboratory.

2.2 Properties of Continuous Media

The desire to perform computation on continuous media data types has dramatically changed the demands placed on the operating system and I/O system of workstations. Properties which differentiate these data types from conventional data types include time-sensitivity and susceptibility to quality/correctness tradeoffs. These are referred to by [Hyden94] as the *temporal* and *informational* properties.

The temporal property of CM data means that the usefulness of the result of a computation depends on the timeliness with which it is delivered. The informational property means that is often possible to perform a calculation on CM data in a number of ways producing results of varying accuracy [Liu91] and consuming different levels of operating system provided resources. If an application is provided with a Quality of Service contract by the operating system, it may maximise the quality of the result obtained for the level of resources available.

The two most common classes of CM traffic are still digital audio and video, although streams of sensor readings or location information [Want92] exhibit some of the same properties. This section briefly discusses some of the issues relevant to computation with these media types.

2.2.1 Digital Video

Until recently the bandwidth requirements of high quality digital video has made it fairly uncommon. An *uncompressed* digital video stream of comparable quality to a PAL encoded analogue TV broadcast requires about 160Mbps. Consumers expect video of at least this quality, but this amount of network bandwidth is still prohibitively expensive.

In order to provide more interactive services, cable television distributors are becoming increasingly interested in digital video distribution. The intended large scale deployment of this technology is already having a beneficial effect on the cost and availability of multimedia peripherals for workstations and home computers.

It is important, however, to bear in mind that the constraints on an interactive video distribution system and a general purpose multimedia workstation are significantly different and much of the available hardware is not completely suited to the purposes for which it is required. Section 2.2.3 considers the example of video compression.

The majority of video standards are designed for eventual presentation to a human observer - digital video is usually transmitted as a number of sequential frames, where the frame rate is related to the persistence of human vision. Most video sources are conventional analogue video cameras and recorders using one of the broadcast video encoding standards. As computation with CM data types becomes more common, video may often be "observed" only by computers working on a time-scale much faster than human perception and so this will not necessarily remain the case.

2.2.2 Digital Audio

Although audio bandwidths are usually much lower than video, its sensitivity to loss and jitter is much greater - a 10ms gap in an audio stream is readily discernible. In these respects, digital audio presents greater challenges to current multimedia systems than does video.

Digital audio transmission has been in widespread use over the Synchronous Transfer Mode (STM) networks of most telephone operators. STM networks provide reliable transmission, constant bit-rate and effectively zero jitter, and these properties have strongly influenced the standards used for audio transmission. For example, compression of audio is usually limited to *companding* and silence suppression.

The transmission of audio across wide-area Packet Transfer Mode (PTM) networks, in particular the Internet, has caused renewed interest in audio coding for increased loss-tolerance and to minimise bandwidth and latency. Extremely low bandwidth, variable-bit-rate compression schemes are becoming more common and can require significant software processing in the end-points. This processing introduces latency and synchronisation issues which can be critical to the usability of interactive digital audio systems. Section 2.3 discusses some of these problems.

2.2.3 Video Compression

Digital video is often compressed in an attempt to reduce the bandwidth, particularly for transmission across wide-area networks. Whilst compression is a useful mechanism for reducing the bandwidth of individual streams, it is invariably used to increase the quality or number of streams which may be simultaneously handled by a system. Experience has shown that multimedia systems are almost always run at 100% load. This section examines the characteristics of several popular video compression schemes including so-called motion-JPEG, MPEG and the MPEG2 standards.

JPEG [Wallace91] is a "lossy" compression scheme exploiting the responsiveness of the human eye to various spatial frequencies by transforming 8×8 pixel tiles of video data into the frequency domain (using a Discrete Cosine Transform) and then quantising selectively. The amount of information discarded by this quantisation process can be controlled by a parameter referred to as the Q-factor. The resulting data is run-length and then Huffman coded. The intermediate frequency domain information is often useful for image-processing computations.

Using JPEG, a typical 24-bit video image can be encoded using approximately one bit-per-pixel without significant subjective loss. JPEG compression and decompression may easily be performed in real-time using relatively inexpensive hardware [CCube94]. Software compression and decompression is computationally expensive, but becoming more feasible. Processor designers have even begun to extend the instruction set of general purpose CPUs to facilitate software decompression of JPEG, for example the HP PA-RISC architecture [Hew94].

The JPEG standard was initially designed for compression of still images. Independent compression of each frame of a video stream using JPEG is usually referred to as "motion-JPEG" compression. Advantages of this technique over more sophisticated video compression schemes include comparatively low latency and higher tolerance to data loss when a stream is carried across an unreliable network.

The MPEG video compression scheme [LeGall91, ISO93] achieves greater compression than JPEG by using motion-estimation techniques to take advantage of inter-frame redundancy in motion video. The compression technique requires cross-correlation of each frame with adjacent frames and is thus fairly computationally intensive. Although hardware is available which can perform real-time MPEG compression, it is expensive. Decompression is roughly comparable in cost to JPEG. The asymmetry between compression and decompression makes MPEG more amenable to Video on Demand (VOD) applications where the most important consideration is to minimise the cost of the decoder in the set-top-box and the additional latency is not a problem.

MPEG2 [ISO95] is even more asymmetric in this respect than its predecessor. Real-time MPEG2 encoding is exceptionally expensive even in hardware, whilst decoders are relatively simple and inexpensive. The standard is primarily aimed at applications where off-line compression may be used.

The inter-frame dependencies inherent in MPEG and MPEG2 mean that decompression is less tolerant to data loss. The extra latency and increased buffering requirements are also disadvantageous. For these reasons, motion-JPEG is still the preferred technology for live video.

Although compressed video occupies significantly less bandwidth, most schemes result in a Variable Bit-Rate (VBR) stream encoded in a manner which also makes it much more difficult to process. Figure 2.1 shows a 3 minute trace of the bandwidth requirements of a motion-JPEG compressed video stream.[1] It can be seen

[1] This trace was obtained by connecting a VCR, receiving BBC1 TV broadcasts, to an AVA-200 and capturing a JPEG compressed 384x288 pixel video stream at Q-factor 32, with the AVA's cell spacing feature disabled. The resulting ATM stream was passed through a Fairisle port controller to log cell inter-arrival times.

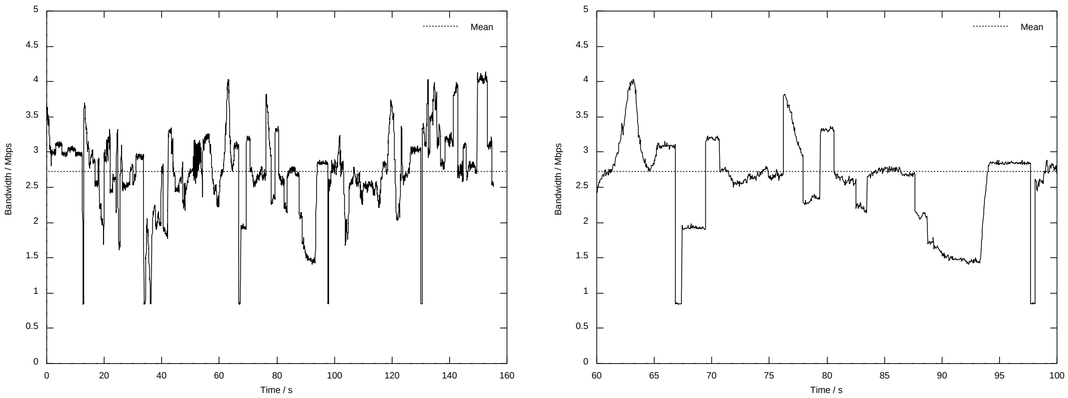


Figure 2.1: Trace of a motion-JPEG Compressed Video Stream

from the main plot that the peak bandwidth requirement of the stream is substantially larger than the mean. The discontinuities visible in the expanded plot are caused by scene changes in the original video. The bandwidth requirements of each scene are often fairly constant with time.

An interesting possibility is the application of motion-compression techniques to 3D video streams such as those used by the Cambridge Autostereo Display [Castle95]. Such streams contain a number of 2D views of the same scene, but from slightly different viewing angles. A slice taken across all of the views corresponds closely to panning the camera. MPEG compression works particularly well when the images are strongly correlated, although distortions introduced by the change in viewing angle may cause problems. The high latency and loss sensitivity properties of motion-based compression schemes do not present a problem when it is used in this way.

2.3 Synchronisation and Latency

Most non-trivial multimedia applications are distributed in nature comprised of a number of communicating processes on different machines [Nicolaou90]. A common problem for such multimedia applications is to process a number of CM streams at various points in the system in a time-synchronised manner.

One of the simplest instances of this problem is that of synchronising the playout of an audio stream to the display of frames of video in order to achieve "lip-sync". This form of synchronisation has relatively loose timing constraints, requiring an accuracy of only around $20ms$, but is often impossible to achieve over a conventional operating system where processes may not receive the CPU for periods far in excess of $20ms$.

In situations where *latency* is not an issue (e.g. playback of pre-recorded video and audio), this particular problem can be alleviated by use of substantial playout buffers. Synchronising multicast streams for VOD applications could easily tolerate latencies of a number of seconds. Indeed, most of the distribution problems in VOD systems would disappear immediately if each "set-top-box" contained a small hard disk capable of buffering 15 minutes worth of data. Video and audio conferencing applications are of a more interactive nature and round-trip times must be kept to a minimum.

Synchronisation of multiple audio streams for the purposes of digital recording or distributed performance potentially requires much greater accuracy. For these applications, whilst it is desirable to keep latency to an absolute minimum, the synchronisation accuracy is of primary importance. For example, musicians who play in an orchestra are used to audio latencies of $30ms$ or more, but are not used to the various sound sources appearing to move relative to each other. The visual cues of the conductor are used to keep time.

2.4 Systems for Handling Continuous Media

Until quite recently, multimedia systems have been unconcerned with the processing of CM data, and have focussed mainly on transport quality of service and orchestration issues such as synchronisation of a number of continuous media streams and presentation on suitable output devices [Campbell92]. Much work has been devoted to architectural support for end-to-end QoS negotiation and translation of high-level QoS specifications, for example the IMAC [Nicolaou90] and QOS-A [Campbell93] architectures. This following sections briefly describe the evolution of multimedia systems and discuss the new demands which these system place upon the workstation operating system.

2.4.1 Pandora: A First Generation Multimedia System

The Pandora multimedia system [Hopper90] consisted of a custom multimedia peripheral, the Pandora's Box, controlled by a conventional workstation. The

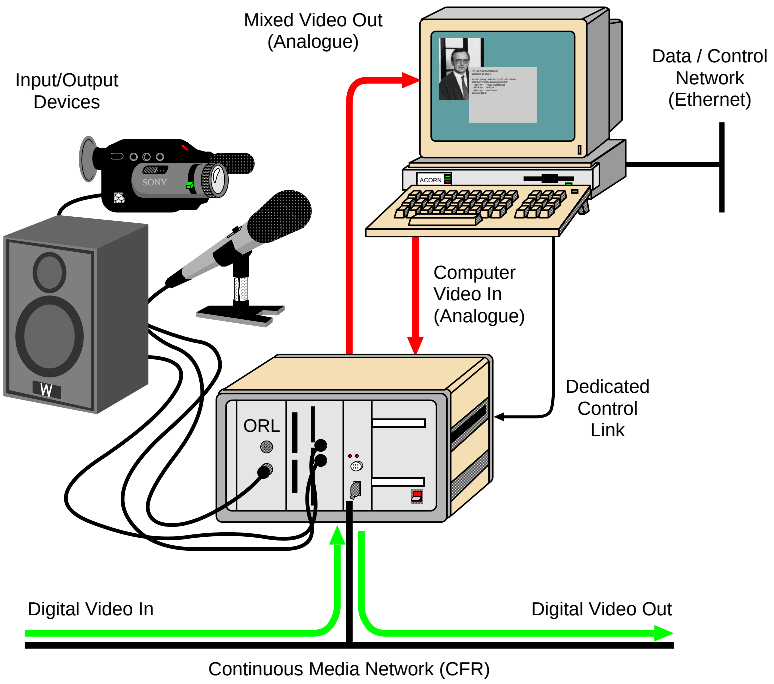


Figure 2.2: The Pandora Multimedia System

box contained, amongst other things, video and audio hardware, a framebuffer, an ATM network interface and a number of Transputers. The Transputers were mainly used to move video and audio data between the various devices and the ATM interface but were also used for resizing video streams. The workstation directed the actions of the Pandora's Box via a dedicated Transputer-link serial interface.

The video output of the workstation passed through the box en-route to the monitor and an analogue video mixer was used to merge the outputs of the framebuffer in the box and that in the workstation. The X server running on the workstation contained a video extension which informed the Pandora's Box of the size and positions of video windows, and which regions of the screen were currently unobscured. The host workstations used a conventional 10Mbps Ethernet for all low-bandwidth out-of-band communications such as connection setup and teardown.

Pandora's Boxes were connected via a 100Mbps CFR which was used to transport the audio and video streams. Only simple fixed-ratio compression schemes were used[2] to simulate the network bandwidth requirements of colour

[2]Pandora used a form of Differential Pulse Code Modulation (DPCM) which achieved a

video streams using more sophisticated compression techniques. Although an MPEG-2 compressed colour video stream would require comparable bandwidth and be approximately Constant Bit-Rate (CBR), the scheme used by Pandora does not exhibit the high latency and poor loss-tolerance properties.

Pandora's design was highly successful in separating in-band and out-of-band I/O operations - by implementing the in-band operations in entirely separate hardware, and leaving control operations to be performed by the host workstation. The drawback of this rigid hardware separation was that general purpose processing of video and audio was impossible since this data could not be passed through the workstation. This limitation was unimportant at the time since processor speeds were such that only very rudimentary calculations could have been performed.

2.4.2 Second-Generation Multimedia Systems

Modern workstations are now fast enough to implement similar functionality to Pandora entirely in software and have sufficient resources spare to be able to perform processing of the continuous media streams. A software emulation of a Pandora's Box implemented on a DECStation 5000/25 is described in [Barham95]. Systems which support computation with CM data types have been referred to as "Second Generation" multimedia systems [Hayter93].

Pandora's successor, Medusa [Wray94] aims to build a highly distributed multimedia system composed of ATM network-connected multimedia devices such as the ORL Disk-Brick [Chaney95] and the AVA-200 [Barham94] which are used as sources and sinks of CM data. In systems such as this, video and audio may easily be processed by any ATM-equipped workstation.

Several researchers have proposed designs for second generation multimedia workstations which replace the traditional bus interconnect with a connection-oriented space-division switch. The Desk Area Network (DAN) [Hayter91] and the MIT VuNet [Houh95] are the two most prominent examples of this approach. These systems are equipped with connection-oriented multimedia devices which such as the DAN Framestore (DFS) [Pratt95] and the MIT Vidboard [Adam93] supporting peer-to-peer communication of continuous media data. Specialised processing nodes such as the DAN DSP node [Atkinson93] may also be used to perform calculations which are not suited to a conventional CPU.

---
fixed 2:1 compression ratio.

Second generation multimedia applications place new demands upon the operating system. Typically the application will be composed of a number of time sensitive tasks, whose total demand for resources vastly exceeds that which is available. Due to the properties of CM data, it is often possible for an application to provide acceptable results at a lower quality when insufficient resources are available.

The relative importance of the various multimedia activities on a machine will potentially change dramatically in response to external events. For example, it should be possible to write a simple application which, using a small fraction of the resources of the machine, monitors a low-quality video or audio stream in the background watching for "interesting" events.[3] In response to such an event, the application may immediately request additional resources to be able to present a high-quality form of the stream to the user.

The operating system must therefore be able to control the distribution of resources ensuring that all parts of the system receive that which is currently deemed necessary.

2.4.3 Discussion

On poorly designed hardware, the majority of the available CPU time is consumed purely by copying CM data between the various I/O devices. In this situation, guarantees of CPU QoS may initially appear to be of use in controlling the distribution of I/O resources within the system. Device drivers may be provided with appropriate levels of CPU resource to support all of their clients, but in order to effectively support more than one multimedia client they must still *internally* account and schedule use of these resources.

As workstation technology improves, the burden of high bandwidth I/O must be removed from the CPU. In order to build larger and faster machines, scalable architectures will become more important, supporting peer-to-peer transfers and minimising the amount of processor interaction required to perform I/O. Given such well-designed hardware, first generation multimedia applications tend to be I/O-bound with small amounts of CPU time being used to orchestrate the times at which events occur. The CPU QoS guarantees provided to such applications have limited benefit since the rate of progress is determined almost entirely by the degree of competition for I/O resources.

[3] Such as motion in a video stream from a security camera.

Second generation multimedia systems are designed to allow computations to be performed on the CM data itself. In this environment, applications may either be CPU-bound or I/O-bound, perhaps changing their behaviour in response to data they receive or an external stimulus. In order for such applications to perform predictably and be able to produce useful results on systems which are usually running at 100% load, it is essential that guaranteed levels of I/O resources can be provided by the underlying operating system and hardware.

2.5 Operating System Structure

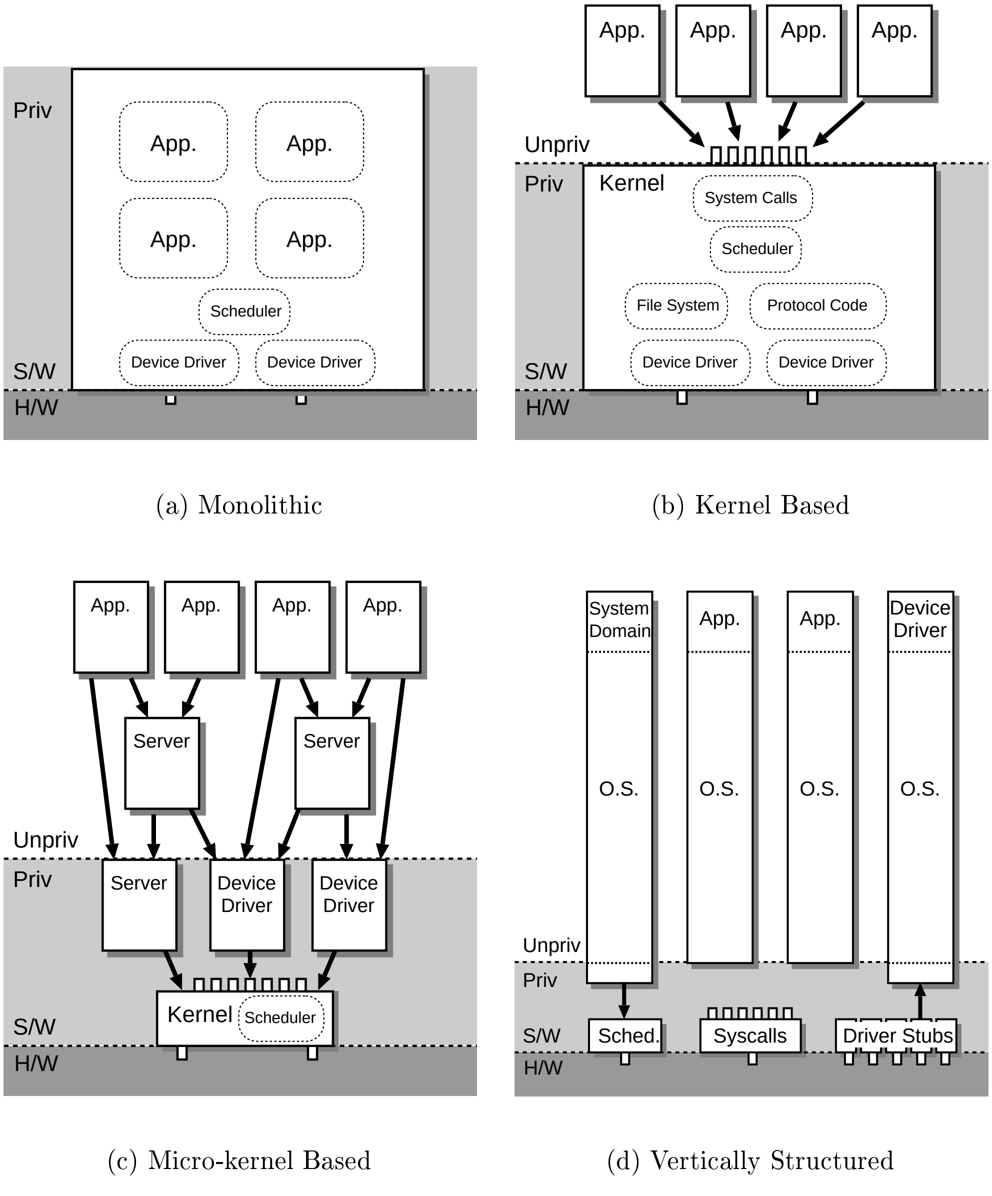
The structure of an operating system invariably reflects the workload which it was designed to handle. As the workload has evolved from off-line batch computation, to include interactive processing and now time-sensitive continuous media applications, operating systems have also evolved to meet the significantly differing resource control requirements.

The majority of operating systems currently running on personal workstations fall into one of three architectural categories: monolithic, kernel based or micro-kernel based. Recently a vertically-structured operating system architecture has been proposed as a means of simplifying resource accounting. The following sections briefly describe these architectures and discuss their respective suitabilities for processing continuous media.

2.5.1 Monolithic Operating Systems

Several early personal workstations ran monolithic operating systems where applications and system code executed in exactly the same environment or *protection domain* (figure 2.3a). Cedar [Swinehart86], the Apple Macintosh [Apple85], MS-DOS and *Windows* fall into this category.

Monolithic systems are usually assumed to be under the control of a single user and so a cooperative multi-tasking model is often applied. Invocation of system services is often performed using simple indirect procedure calls via vector tables at well-known locations or by processor trap instructions. Good programming discipline and sophisticated compilers must be relied upon to minimise the probability of one application interfering with another or even crashing the entire system.



(a) Monolithic

(b) Kernel Based

(c) Micro-kernel Based

(d) Vertically Structured

Figure 2.3: Operating System Structure

By avoiding potentially costly changes of protection domain, monolithic operating systems achieve high performance, but with the disadvantages of no protection from misbehaving processes, and no effective resource firewall mechanisms to prevent one application from monopolising system resources.

2.5.2 Kernel-Based Operating Systems

Later workstation operating systems grouped all services and functions into a single large *kernel* running at a privileged level, with applications running as unprivileged processes over the top (figure 2.3b). System calls provide a mechanism for applications to invoke services within the kernel. Examples include early varieties of UNIX [Ritchie74], VMS [Goldenberg92] and more recently *Windows-NT* [Custer93].

This structure allows numerous performance improvements to be obtained by taking advantage of the tightly coupled nature of the system. Distinct internal functions of the operating system usually interact with each other using a set of locking disciplines based around a notion of Interrupt Priority Level (IPL) [Leffler89] which are difficult to enforce and often result in unexpected deadlock or even livelock [Mogul95].

Such operating systems were originally designed to allow multiple users to simultaneously execute computationally intensive tasks. Processes with which users interacted, usually via an attached terminal, required special treatment in order to keep down the interactive response time. In a number of varieties of UNIX, the scheduler identifies interactive processes by the fact that they block performing I/O and, in order to minimise response times, they are preferentially given the CPU with the assumption that they will soon block again. This heuristic has disastrous results in the multimedia environment where an application requires only a small amount of CPU time to cause high volumes of I/O to take place.

In kernel-based systems, the resources consumed by the operating system kernel whilst performing this high-volume I/O are typically unaccounted, with the result that in a multimedia system implemented over UNIX, the majority of CPU time will be spent within the kernel and only a small amount will be accounted to user applications. For example, in an experiment in which a single stream of ATM video from an AVA-200 was displayed on a UNIX workstation, 30% of the processor time was accounted to the application, 30% to the X server displaying the video, and the remaining time was consumed by the operating system fielding interrupts from the ATM interface.

2.5.3 Microkernel-Based Operating Systems

Recent operating systems research has caused functionality to be moved out of monolithic kernels and into server processes, usually for reasons of modularity and maintainability (figure 2.3c).[4] Operating systems of this kind are generally referred to as *microkernel-based*. Well-known examples include Amoeba [Tanenbaum81], Mach [Accetta86], Chorus [Rozier90], Plan9 [Pike90] and Spring [Hamilton93].

The operating system kernel usually provides little more than support for "kernel-threads" and an Inter-Process Communication (IPC) mechanism such as the message-passing system of Mach. Microkernel based operating systems necessarily incur performance penalties due to the increased amount of communication which must take place. For this reason, several researchers have investigated mechanisms for migrating functionality back into the kernel [Bricker91]. The *SPIN* microkernel [Bershad94] provides mechanisms for servers to move specially verified code sections known as *SPIN*DLEs into client address-spaces and even into the kernel. A portable hardware abstraction layer provides access to processor features such as the translation lookaside buffer (TLB) and operating system datastructures allowing applications to effectively implement their own communications primitives.

In an operating system designed to support multimedia, moving system provided services out of the kernel and into servers introduces a complicated accounting problem. Resources consumed by an application must be transferred to and accounted within each server, and each resource must now be multiplexed more than once. Not only is this an inelegant approach, but in the field of networking, layered multiplexing has been shown to be harmful to the provision of Quality of Service [Tennenhouse89].

[4] Although some researchers seem to view the minimisation of kernel size as its own justification — hence the proliferation of "nano-kernels" and "pico-kernels".

2.5.4 Vertically Structured Operating Systems

A large proportion of the code executed on behalf of an application in a traditional operating system requires no additional privilege and does not therefore need to execute in a different protection domain. Typically code which must atomically and securely update important datastructures is rarely executed and usually associated with out-of-band operations such as opening or closing a file. It is only this code which must necessarily execute in a different protection domain.

The Nemesis operating system [Leslie96] makes use of these observations, together with a platform and language independent interface definition language (IDL), to transparently move the majority of operating system services into the application itself, leading to a *vertically structured* operating system (figure 2.3d). The kernel is still responsible for implementing scheduling and protection of hardware resources, but this functionality is provided at a much lower level of abstraction. In a multimedia system this also has the advantage of allowing applications to make their own resource management policy decisions. Nemesis is described in greater detail in chapter 3.

The design principles used in Nemesis have been parallelled closely by the MIT Exokernel project [Engler95], although for different reasons. The motivation for this work, as with *SPIN* was to improve efficiency, rather than accountability, by providing the minimal primitives to support per-process customisation of the operating system. This led to a design where the Aegis kernel does little more than virtualise processor features to support multiple "library-operating systems". Aegis provides minimalist interrupt and exception dispatching and a software TLB abstraction. Capability mechanisms are used to implement *secure-bindings* for the update of the software TLB. The Ethernet device is abstracted using low-level packet-filtering [Mogul87] and although discussions of framebuffer and disk I/O mechanisms are presented, these have not been fully realised.[5]

2.6 Quality of Service in Operating Systems

Operating systems QoS research has focussed almost exclusively on scheduling the processor resource so as to provide support for multimedia applications – so called "soft-real-time" technology. This has often involved adding a real-time

[5]For example, Aegis currently runs on DEC5000/125 platforms, yet [Engler95] describes a Silicon Graphics framebuffer not available for TURBOchannel based machines.

thread scheduling class to the kernel thread scheduler of an existing microkernel based operating system [Tokuda93]. Such an extension does not alleviate the fundamental problem that the majority of system resources consumed by an application are not accounted to that application and so any QoS guarantee given to that application is therefore meaningless.

Efforts have been made to provide high-level interfaces to allow applications to specify their QoS requirements in more meaningful terms [Coulson93]. Whilst these QoS translation schemes are often useful, defining any high-level QoS architecture *within* the operating system only serves to restrict the classes of application which can be supported.

The extension of QoS to I/O within an operating system is discussed in [Coulson95]. High priority threads within the driver of an ATM network interface are used to demultiplex I/O streams at a low level and perform protocol processing operations for each connection with some form of timeliness guarantee. Three classes of threads are provided by the system, ranging from best-effort to guaranteed threads with absolute priority over other classes. Threads of the latter class are used per-connection to shepherd packets from the device driver up to applications. The lack of hardware demultiplexing functions in the ATM interface means that a significant fraction of the processing overheads are incurred by a single "interrupt handler" thread.

The inappropriate structure of microkernel-based operating systems, where the majority of system services are provided by servers, introduces the need for complicated resource transfer mechanisms. The Processor Capacity Reserves scheme [Mercer93] allows processor time originally allocated to an application, in the form of a *reserve* to be used by a server when performing an Remote Procedure Call (RPC).

There is no way to ensure, however, that the processor time is used for the purposes for which it was originally intended, and for any timeliness guarantees to be preserved, the server must internally schedule clients' requests. [Mercer93] describes the need to modify the X Window System Server to service its clients in a prioritised fashion to maintain the QoS guarantees of video display applications.

Lottery scheduling [Waldspurger94] shares the processor between a number of kernel threads using statistical mechanisms. A cheap pseudo-random number generator selects one of a number of "tickets" and the application currently holding the ticket is allowed to execute for a fixed quantum of time. Mechanisms for transferring tickets between clients and servers are also provided. The statistical nature of this scheduling technique is only suitable to provision of guarantees over large time-scales in comparison with the reschedule rate, and unless the properties of the random-number generator are well understood, it is possible for processes not to receive their allocated resources.

The later refinement, Stride scheduling [Waldspurger95] uses a deterministic method to apportion resources with the same ticket-based abstraction. This technique has also been used to control the rate of transmission of UDP packets across and Ethernet with some success.

Thread tunnelling/migration has been used to allow a client thread to enter the protection domain of a server for the duration of an RPC, effectively transferring the *exact* amount of resources to allow the call to be performed. Examples of this include lightweight-RPC [Bershad90], *doors* in Spring [Hamilton93] and *portals* in the Opal system [Chase93]. In general, all of these mechanisms imply the use of kernel threads, and mean that all scheduling decisions must be taken by the kernel and/or the server. This both removes thread scheduling policy from the application, and requires that the application describe its scheduling requirements to the kernel using a fixed and necessarily restrictive interface.

The approach taken by Nemesis is described in chapter 3.

Chapter 3

Nemesis

Nemesis is the prototype operating system developed at the University of Cambridge Computer Laboratory as part of the Pegasus Project [Leslie93]. Nemesis currently runs on a number of platforms including the DEC 3000/AXP series of workstations [DEC94b], DECchip EB64 Alpha evaluation board [DEC93], DEC 5000/25 (Maxine) [Voth91] and the Fairisle FPC3 Port Controller [Hayter94].[1] Several ports to other platforms and architectures are also underway, including the DECchip EB164 evaluation board [DEC95] and the Intel Pentium processor. This chapter describes the structure of Nemesis.

3.1 Introduction

The purpose of an operating system is to multiplex shared resources between applications. Traditional operating systems have presented physical resources to applications by virtualisation. e.g. UNIX applications run in virtual time on a virtual processor – most are unaware of the passage of real time and that they often do not receive the CPU for prolonged periods. The operating system proffers the illusion that they are exclusive users of the machine.

Multimedia applications tend to be sensitive to the passage of real time. They need to know when they will have access to a shared resource and for how long. In the past it has been considered sufficient to implement little more than access-

[1] Nemesis was written from scratch over a 2 year period by Timothy Roscoe, David Evers and the author, with significant influence from the Fawn system described in [Black94].

control on physical resources. It is becoming increasingly important to account, schedule and police shared resources so as to provide some form of QoS guarantee.

Whilst it is necessary to provide the *mechanisms* for multiplexing resources, it is important that the amount of *policy* hard-wired into the operating system kernel is kept to an absolute minimum. That is, applications should be free to make use of system provided resources in the manner which is most appropriate. At the highest level, a user may wish to impose a globally consistent policy, but in the Nemesis model this is the job of a QoS-manager agent acting on the user's behalf and under the user's direction. This is analogous to the use of a "window-manager" process to allow the user to control the decoration, size and layout of windows on the screen, but which does not otherwise constrain the behaviour of each application.

3.2 Previous Work

Previous work in the Pegasus project has restricted attention to the CPU resource. [Hyden94] discusses scheduling mechanisms for soft-real time problems and demonstrates a system which provides QoS guarantees to applications and mechanisms which allow applications to degrade gracefully in conditions of high load.

[Black94] describes the advantages of QoS guarantees and resource accounting mechanisms for controlling the behaviour of device-drivers in an high-speed network environment.

[Roscoe95] addresses the problem of QoS crosstalk in microkernel systems. By migrating operating system functionality into applications themselves, the use of servers is minimised. Servers are only required for out-of-band operations and do not reside on the data path for most operations. This architecture minimises the problem of QoS crosstalk without having to resort to complicated resource transfer mechanisms.

3.3 The Structure of Nemesis

Nemesis was designed to provide QoS guarantees to applications. In a microkernel environment, an application is typically implemented by a number of processes,

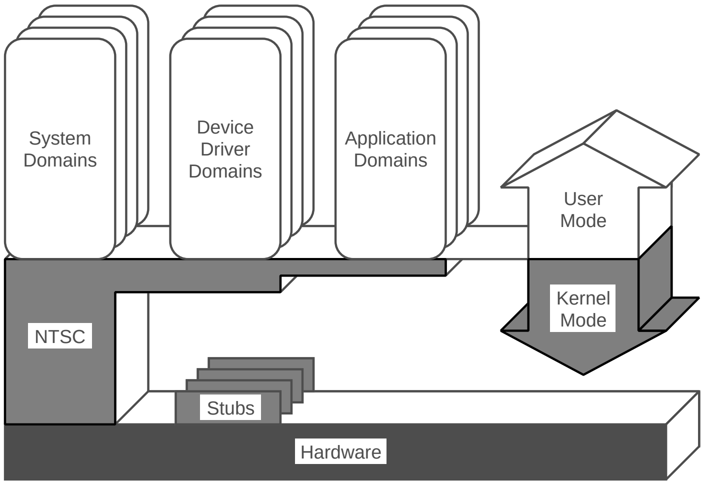


Figure 3.1: The Structure of a Nemesis System

most of which are servers performing work on behalf of more than one client. This leads to enormous difficulty in accounting resource usage to the application. The guiding principle in the design of Nemesis was to structure the system in such a way that the vast majority of functionality comprising an application could execute in a single process, or *domain*. As mentioned previously, this leads to a *vertically-structured* operating system (figure 3.1).[2]

The Nemesis kernel consists of a scheduler (less than 250 instructions) and a small amount of code known as the Nemesis Trusted Supervisor Code (NTSC), used for Inter-Domain Communication (IDC) and to interact with the scheduler. The kernel also includes the minimum code necessary to initialise the processor immediately after booting and handle processor exceptions, memory faults, unaligned accesses, TLB misses and all other low-level processor features. The Nemesis kernel bears a striking resemblance to the original concept of an operating system kernel or *nucleus* expressed in [Brinch-Hansen70].

The kernel demultiplexes hardware interrupts to the stage where a device specific first-level interrupt handler may be invoked. First level interrupt handlers consist of small stubs which may be registered by device-drivers. These stubs are entered with all interrupts disabled and with the minimal number of registers saved and usually do little more than send notification to the appropriate device driver.

[2]This diagram is derived from a diagram in [Hyden94].

3.3.1 Domains

The term *domain* is used within Nemesis to refer to an executing program and can be thought of as analogous to a UNIX process – i.e. a domain encapsulates the execution state of a Nemesis application. Each domain has an associated *scheduling domain* (determining CPU time allocation) and *protection domain* (determining access rights to regions of the virtual address space).

Nemesis is a Single Address Space (SAS) operating system i.e. any accessible region of physical memory appears at the same virtual address in each protection domain. Access rights to a region of memory, however need not be the same. The use of a single address space allows the use of pointers in shared data-structures and facilitates rich sharing of both program text and data leading to significant reduction in overall system size.

All operating system interfaces are written using an IDL known as MIDDL which provides a platform and language independent specification. MIDDL interfaces,[3] modules and code-structuring tools are used to support the single address space and allow operating system code to be migrated into the application. These techniques are discussed in detail in [Roscoe95].

3.3.2 Nemesis Trusted Supervisor Code (NTSC)

The NTSC is the low level operating system code within the Nemesis kernel which may be invoked by user domains. Its implementation is potentially architecture and platform specific, for example, the NTSC is implemented almost entirely in PALcode on Alpha platforms [Sites92] whilst on MIPS [Kane88] and ARM [ARM91] platforms, the NTSC is invoked using the standard system call mechanisms.

The NTSC interface may be divided into two sections - those calls which may be invoked by any domain, and those which may only be invoked by a privileged domain. Unprivileged calls are used for interaction with the kernel scheduler and to send events to other domains. Privileged calls are used to affect the processor mode and interrupt priority level and to register first level interrupt stubs. As an example, table 3.1 lists the major NTSC calls for the Alpha architecture.

The NTSC interacts with domains and the kernel scheduler via a per-domain

[3] MIDDL interfaces in Nemesis are written in files with a `.if` suffix, e.g. `Activation.if`

Unprivileged Domains	
Name	Purpose
ntsc_rfa	Return from activation.
ntsc_rfa_resume	Return from activation, restoring a context.
ntsc_rfa_block	Return from activation and block.
ntsc_block	Block awaiting an event.
ntsc_yield	Relinquish CPU allocation for this period.
ntsc_send	Send an event.
Privileged Domains	
Name	Purpose
ntsc_swpipl	Change interrupt priority level.
ntsc_entkern[a]	Enter kernel mode.
ntsc_leavekern	Leave kernel mode.
ntsc_regstub	Register an interrupt stub.
ntsc_kevent	Send an event from an interrupt stub.
ntsc_rti	Return from an interrupt stub.

[a] ntsc_entkern is necessarily implemented as an unprivileged PAL-code call, but which results in an illegal instruction fault for unprivileged domains.

Table 3.1: Alpha NTSC Call Interface.

area of shared memory known as the Domain Control Block (DCB). Portions of the DCB are mapped read-write into the domain's address-space, whilst others are mapped read-only to prevent modification of privileged state. The read-only DCB contains scheduling and accounting information used by the kernel, the domain's privilege level, read-only datastructures used for implementing IDC channels and miscellaneous other information. The read-write section of the DCB contains an array of processor-context save slots and user-writable portions of the IDC channel datastructures.

3.4 Virtual Processor Interface

Nemesis presents the processor to domains via the Virtual Processor Interface (`VP.if`). This interface specifies a platform independent abstraction for managing the saving and restoring of CPU context, losing and regaining the real processor and communicating with other domains. It does *not* however attempt to hide the multiplexing of the underlying processor(s). The virtual processor interface is implemented over the NTSC calls described in section 3.3.2.

3.4.1 Activation

Whenever a domain is given the CPU, it is upcalled via a vector in the DCB known as the *activation handler* in a similar manner to Scheduler Activations [Anderson92]. A flag is set disabling further upcalls until the domain leaves activation mode allowing code on the activation vector to perform atomic operations with little or no overheads. Information is made available to the activation handler including an indication of the reason why the domain has been activated, the time when it last lost the real processor and the current system time. The purpose of this upcall is to afford QoS-aware applications an opportunity to assess their progress and make application-specific policy decisions so as to make most efficient usage of the available resources.

When a domain is handed the processor it is informed whether it is currently running on guaranteed time, or merely being offered use of some of the slack-time in the system. QoS-aware applications must take account of this before deciding to adapt to apparent changes in system load. This may be used to prevent QoS feedback mechanisms from reacting to transient improvements in resource availability.

3.4.2 Processor Context

When a virtual processor loses the CPU, its context must be saved. The DCB contains an array of context-save slots for this purpose. Two indices into this array specify the slots to use when in activation mode and when in normal mode, based on the current state of the activation flag.

When a domain is preempted it will usually be executing a user-level thread. The context of this thread is stored in the *save slot* of the DCB and may be reloaded by the activation handler of the domain when it is next upcalled. If a domain is preempted whilst in activation mode, the processor context is saved in the *resume slot* and restored transparently when the domain regains the CPU rather than the usual upcall.

3.4.3 Events

The only means of communication directly provided by the Nemesis kernel is the *event*. Each domain has a number of *channel-endpoints* which may be used either to transmit or to receive events. A pair of endpoints may be connected by a third party known as the *Binder*, to provide an asynchronous simplex communications channel.

This channel may be used to transmit a single 64-bit value between two domains. The event mechanism is intended to be used purely as a synchronisation mechanism for shared memory communication, although several simple protocols have been implemented which require nothing more than the event channel itself, e.g the `TAP` protocol described in [Black94] used for start-of-day communication with the binder. Unlike message-passing systems such as Mach [Accetta86] or Chorus [Rozier90], the kernel is not involved in the transfer of bulk data between two domains.

Nemesis also separates the act of sending an event and that of losing the processor. Domains may exploit this feature to send a number of events before being preempted or voluntarily relinquishing the CPU. For bulk data transports such as the *Rbufs* mechanism described in section 3.5.3, pipelined execution is usually desirable and the overheads of repeatedly blocking and unblocking a domain may be avoided. For more latency-sensitive client-server style communication a domain may choose to cause a reschedule immediately in order to give the server domain a chance to execute.

3.5 Inter-Domain Communication (IDC)

Various forms of IDC have been implemented on top of the Nemesis event mechanism. Some of the most commonly used are described below.

3.5.1 Shared Datastructures

Since Nemesis domains share a single address space, the use of shared memory for communication is relatively straightforward. Datastructures containing pointers are globally valid and the only further requirement is to provide some synchronisation mechanism to allow the datastructure to be updated atomically and to prevent readers from seeing an inconsistent state. Very lightweight locking primitives may easily be built on top of the kernel-provided event mechanism.

3.5.2 Remote Procedure Call (RPC)

Same-machine RPC [Birrell84] is widely used within Nemesis. Although the majority of operating system functionality is implemented within the application, there are many out-of-band operations which require interaction with a server in order to update some shared state.

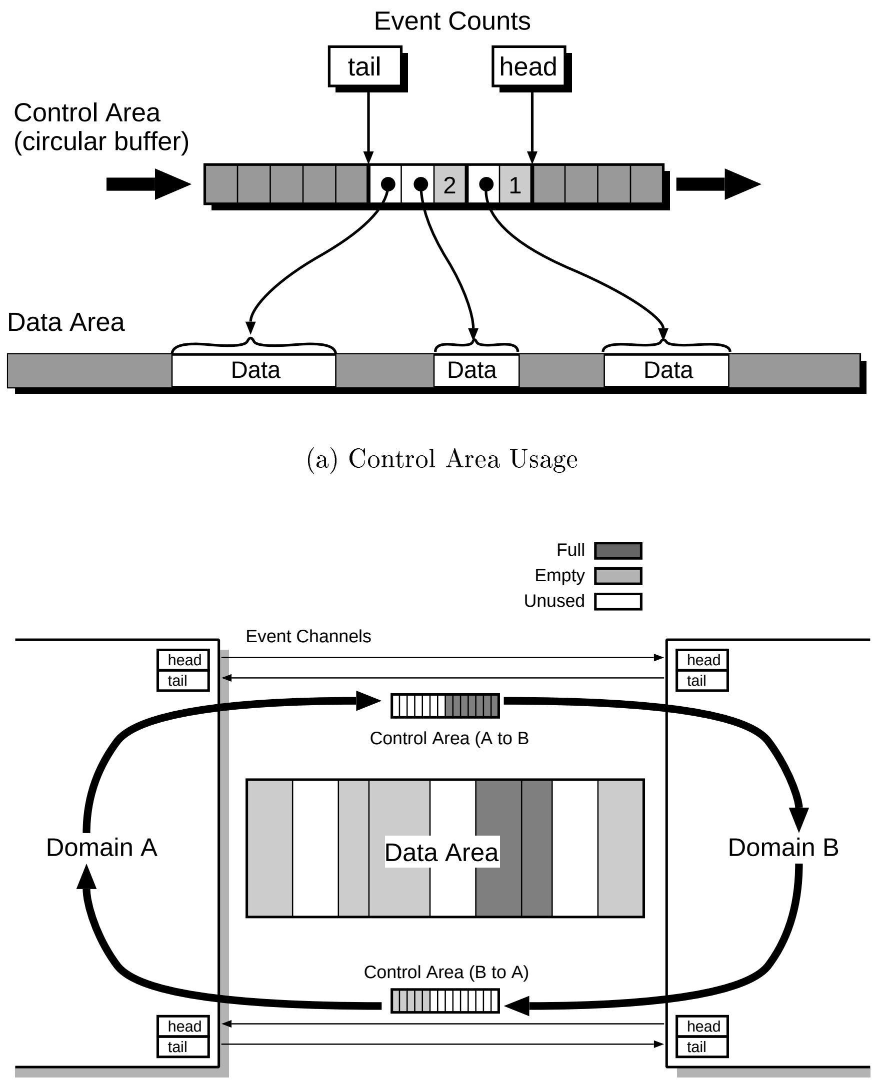
The default RPC transport is based on an area of shared memory and a pair of event channels between the client and server domains. To make an invocation, the client marshalls an identifier for the call and the invocation arguments into the shared memory and sends an event to the server domain. The server domain receives the event, unmarshalls the arguments and performs the required operation. The results of the call, or any exception raised are then marshalled into the shared memory and an event sent back to the client. Marshalling code and the client and server *stubs* are generated automatically from the MIDDL interface definition and loaded as shared libraries.

The average cost of a user-thread to user-thread null-RPC between two Nemesis domains using the default machine-generated stubs and the standard user-level threads package, was measured at just over $30\mu s$ on the Sandpiper platform [Roscoe95].

3.5.3 *Rbufs*

Whilst RPC provides a natural abstraction for out-of-band control operations and transaction style interactions, it is unsuited to transfer of bulk data. The mechanism adopted by Nemesis for transfer of high volume packet-based data is the *Rbufs* scheme detailed in [Black94]. The transport mechanism is once again implemented using Nemesis event-channels and shared memory. Three areas of shared memory are required as shown in figure 3.2. One contains the data to be transferred and the other two are used as FIFOs to transmit packet descriptors between the source and sink. The head and tail pointers of these FIFOs are communicated by Nemesis event-channels.

Packets comprised of one or more fragments in a large pool of shared memory



(a) Control Area Usage

(b) Overview

Figure 3.2: High Volume I/O Using *Rbufs*

are described by a sequence of $(base, length)$ pairs known as `iorec`s. Figure 3.2a shows `iorec`s describing two packets, one with two fragments and the other with only a single fragment. *Rbufs* are highly suited to protocol processing operations since they allow simple addition or removal of both headers and trailers and facilitate segmentation and reassembly operations.

In receive mode, the sink sends `iorec`s describing empty buffer space to the source, which fills the buffers and updates the `iorec`s accordingly before returning them to the sink. In transmit mode, the situation is the converse. The closed loop nature of communication provides back-pressure and feedback to both ends of the connection when there is a disparity between the rates of progress of the source and sink.

The intended mode of operation relies on the ability to pipeline the processing of data in order to amortise the context-switch overheads across a large number of packets. Sending a packet on an *Rbufs* connection does not usually cause a domain to lose the CPU. Figure 3.3 shows the MIDDL interface type for the *Rbufs* transport.

3.6 Scheduling

Scheduling can be viewed as the process of multiplexing the CPU resource between computational tasks. The schedulable entity of an operating system often places constraints both on the scheduling algorithms which may be employed and the functionality provided to the application.

The recent gain in popularity of multi-threaded programming due to languages such as Modula-3 [Nelson91] has led many operating system designers to provide kernel-level thread support mechanisms [Accetta86, Rozier90]. The kernel therefore schedules threads rather than processes. Whilst this reduces the functionality required in applications and usually results in more efficient processor context-switches, the necessary thread scheduling *policy* decisions must also be migrated into the kernel. As pointed out in section 2.6, this is highly undesirable.

Attempts to allow applications to communicate thread scheduling policy to the kernel scheduler [Coulson93, Coulson95] lead to increased complexity of the kernel and the possibility for uncooperative applications to misrepresent their needs to the operating system and thereby gain an unfair share of system re-

```
IO : LOCAL INTERFACE =
  NEEDS IDC;
BEGIN

  -- An "IO" channel has to be one of two kinds, a "Read"er or
  -- "Write"r. Readers remove valid packets from the channel with
  -- "GetPkt" and send back empty "IO_Rec"s with "PutPkt"; Writers
  -- send valid packets with "PutPkt" and acquire empty "IO_Rec"s by
  -- calling "GetPkt".
  Kind : TYPE = { Read, Write };

  -- The values passed through "IO" channels are "IO_Recs",
  -- essentially "base" + "length" pairs describing the data.
  Rec : TYPE = RECORD [
    base : ADDRESS,
    len  : WORD
  ];

  -- "PutPkt" sends a vector of "IO_Rec"s down the channel. The
  -- operation sends "nrecs" records in a vector starting at "recs" in
  -- memory. Of these, the first "valid_recs" are declared as holding
  -- useful data.
  PutPkt : PROC    [ nrecs      : CARDINAL;
                     recs       : REF Rec;
                     valid_recs : CARDINAL  ]
           RETURNS [];
    -- Send a vector of I/O records down the channel, or release them
    -- at the receiving end.

  -- "GetPkt" acquires a maximum of "max_recs" "IO_Rec"s, which are
  -- copied into memory at address "recs". At the receive end these
  -- typically constitute a packet, which uses the first "valid_recs"
  -- for pointing to its data. The total number of records read is
  -- returned in "nrecs".
  GetPkt : PROC    [ max_recs   : CARDINAL;
                     recs       : REF Rec;
                 OUT valid_recs : CARDINAL  ]
           RETURNS [ nrecs      : CARDINAL  ];
    -- Pull a vector of I/O records out of the channel.

  -- "PutPktNoBlock" sends a packet assuming that the client has
  -- already determined that "PutPkt" would not block.
  PutPktNoBlock : PROC    [ nrecs      : CARDINAL;
                            recs       : REF Rec;
                            valid_recs : CARDINAL  ]
                  RETURNS [];
    -- Guaranteed non-blocking "PutPkt".

  -- "GetPktNoBlock" checks whether it would block, and returns
  -- "False" if this is the case.
  GetPktNoBlock : PROC    [ max_recs   : CARDINAL;
                            recs       : REF Rec;
                        OUT nrecs      : CARDINAL;
                        OUT valid_recs : CARDINAL  ]
                  RETURNS [ avail      : BOOLEAN   ];
    -- As "GetPkt", but fails rather than block.

  -- "GetPoolInfo" returns information about the pool used to send
  -- data.
  GetPoolInfo : PROC [ OUT buf: IDC.Buffer ] RETURNS [];
    -- Return the main pool buffer.

  Slots : PROC [ ] RETURNS [ ns: CARDINAL ];
    -- Return the number of slots of the tx fifo.

  Dispose  : PROC [] RETURNS [];

END.
```

Figure 3.3: MIDDL interface for *Rbufs* (`IO.if`)

sources. For example, in the above systems user processes are required to communicate the earliest deadline of any of their threads to the kernel thread scheduler.

Nemesis allows domains to employ a split-level scheduling regime with the multiplexing mechanisms being implemented at a low level by the kernel, and the application-specific policy decisions being taken at user-level within the application itself. Note that the operating system only multiplexes the CPU resource *once*. Most application domains make use of a threads package to control the internal distribution of CPU resource between a number of cooperating threads of execution.

3.6.1 Inter-Process Scheduling

Inter-process scheduling in Nemesis is performed by the kernel scheduler. This scheduler is responsible for controlling the exact proportions of bulk processor bandwidth allocated to each domain according to a set of QoS parameters in the DCB. Processor bandwidth requirements are specified using a tuple of the form (p, s, x, l) with the following meaning:

p	The *period* of the domain in ns.
s	The *slice* of CPU time allocated to the domain every period in ns.
x	A flag indicating willingness to accept *extra* CPU time.
l	A *latency hint* to the kernel scheduler in ns.

The p and s parameters may be used both to control the amount of processor bandwidth and the smoothness with which is is provided. The latency hint parameter is used to provide the scheduler with an idea as to how soon the domain should be rescheduled after unblocking.

The kernel scheduler interacts with the event mechanism allowing domains to block until they next receive an event, possibly with a timeout. When a domain blocks it loses any remaining CPU allocation for its current period - it is therefore in the best interest of a domain to complete as much work as possible before giving up the processor.

The current kernel scheduler employs a variant of the Earliest Deadline First (EDF) algorithm [Liu73] where the deadlines are derived from the QoS parameters of the domain and are purely internal to the scheduler. The scheduler is capable of ensuring that all guarantees are respected provided that

$$\sum_i \frac{s_i}{p_i} \leqslant 1$$

and is described in detail in [Roscoe95]. Despite the internal use of deadlines, this scheduler avoids the inherent problems of priority or deadline based scheduling mechanisms which focus on determining who to allocate the *entire* processor resource to, and provide no means to control the quantity of resource handed out.

In order to provide fine-grained timeliness guarantees to applications which are latency sensitive, higher rates of context-switching are unavoidable. The effects of context-switches on cache and memory-system performance are analysed in [Mogul91]. It is shown that a high rate of context switching leads to excessive numbers of cache and TLB misses reducing the performance of the entire system. The use of a single address space in Nemesis removes the need to flush a virtually addressed cache on a context switch, and the process-ID fields present in most TLBs can be used to reduce the number of TLB entries which need to be invalidated. The increased sharing of both code and data in a SAS environment also helps to reduce the cache-related penalties of context-switches.

3.6.2 Intra-Process Scheduling

Intra-process scheduling in a multimedia environment is an entirely application-specific area. Nemesis does not have a concept of kernel threads for this reason. A domain may use a user-level scheduler to internally distribute the CPU time provided by the kernel scheduler using its own policies. The application specific code for determining which context to reload is implemented in the domain itself.

The activation mechanism described in section 3.4.1 provides a convenient method for implementing a preemptive user-level threads package. The current Nemesis distribution provides both preemptive and non-preemptive threads packages as shared library code.

The default thread schedulers provide lightweight user-level synchronisation primitives such as event counts and sequencers [Reed79], and the mutexes and condition variables of SRC threads [Birrell87]. The implementation of various sets of synchronisation primitives over the top of event counts and sequencers is discussed in [Black94].

It is perfectly possible for a domain to use an application specific threads package, or even to run without a user-level scheduler. A user-level threads package based on the ANSAware/RT [ANSA95] concepts of *Tasks* and *Entries* has been developed as part of the DCAN project at the Computer Laboratory.[4] The ANSAware/RT model maps naturally onto the Nemesis Virtual Processor interface.

3.7 Device Driver Support

In order to present shared I/O resources to multiple clients safely, device-drivers are necessary. The driver is responsible for ensuring that clients are protected from each other and that the hardware is not programmed incorrectly. This often involves context-switching the hardware between multiple concurrent activities. The exact nature of the hardware dictates the methods employed and therefore the level of abstraction at which a device may be presented to applications.

Device drivers typically require access to hardware registers which can not safely be made accessible directly to user-level code. This can be achieved by only mapping the registers into the address space of the device driver domain.

Some hardware registers are inherently shared between multiple device drivers, e.g. interrupt masks and bus control registers. The operating system must provide a mechanism for atomic updates to these registers. In kernel-based operating systems this has traditionally been performed by use of a system of interrupt-priority levels within the kernel. On most platforms, Nemesis provides similar functionality via privileged NTSC calls.

In the design of Nemesis it was considered essential that it was possible to limit the use of system resources by device driver code so that the behaviour of the system under overload could be controlled. For this reason, Nemesis device drivers are implemented as privileged domains which are scheduled in exactly the same way as other domains, but have access to additional NTSC calls.

[4] This work was performed by Timothy Roscoe and David Evers.

3.7.1 Hardware Interrupts

The majority of I/O devices have been designed with the implicit assumption that they can asynchronously send an interrupt to the operating system which will cause appropriate device-driver code to be scheduled immediately with absolute priority over all other tasks. Indeed failure to promptly service interrupt requests from many devices can result in serious data loss. It is ironic that serial lines, the lowest bit-rate I/O devices on most workstations, often require the most timely processing of interrupts due to the minimal amounts of buffering and lack of flow-control mechanisms in the hardware. Section 4.3.1 describes how this phenomenon influences DMA arbitration logic on the Sandpiper.

More recently designed devices, particularly those intended for multimedia activities, are more tolerant to late servicing of interrupts since they usually have more internal buffering and are expected to cope with transient overload situations.

In order to effectively deal with both types of device, Nemesis allows drivers to register small sections of code known as *interrupt-stubs* to be executed immediately when a hardware interrupt is raised. These sections of code are entered with a minimal amount of saved context and with all interrupts disabled. They thus execute atomically. In the common case, an interrupt-stub will do little more than send an event to the associated driver causing it to be scheduled at a later date, but for devices which are highly latency sensitive it is possible to include enough code to prevent error conditions arising. The unblocking latency hint to the kernel scheduler is also useful for this purpose.

This technique of decoupling interrupt notification from interrupt servicing is similar to the scheme used in Multics which is described in [Reed76], but the motivation in Nemesis is to allow effective control of the quantity of resources consumed by interrupt processing code, rather than for reasons of system structure. [Dixon92] describes a situation where careful adjustment of the relative priorities of interrupt processing threads led to increased throughput under high loads when drivers were effectively polling the hardware and avoiding unnecessary interrupt overhead. The Nemesis mechanisms are more generic and have been shown to provide better throughput on the same hardware platform [Black94].

3.7.2 Kernel Critical Sections

The majority of device-driver code requires no privilege, but small regions of device driver code often need to execute in kernel mode. For example, performing I/O on a number of processors requires the use of instructions only accessible within a privileged processor mode. Nemesis provides a lightweight mechanism for duly authorised domains to switch between kernel and user mode.[5]

Although the current implementation requires explicit calls to enter and exit kernel mode, an alternative would be to register these code sections (ranges of PC values) in advance and perform the switch to kernel mode when the processor detects a privilege violation. The PC range tables generated by many compilers to enable efficient language-level exception mechanisms may be used for this purpose. Although this support is expected soon, the version of `gcc` currently in use does not yet include these features.[6]

3.8 Nemesis and I/O

Although Nemesis is intended as an operating system for a personal multimedia workstation, much of the previous experimental work has been evaluated using workloads which are unrepresentative of those found in multimedia systems – i.e. processes which are entirely CPU-bound. The Nemesis approach to QoS provision has been proven to be highly successful in environments where the bottleneck resource for all applications is the CPU.

The work described in [Roscoe95] approaches the QoS-crosstalk problem by migrating operating system code into user domains. Whilst this solution works well for code which does not require to run with elevated privilege, such as protocol-processing code, it cannot be used in situations which requires write access to shared state or access to hardware registers. A multimedia system by definition deals with a large volume of I/O which invariably involves privileged operations at the lowest levels. Since these operations must be therefore be performed by a privileged domain, and Nemesis provides no low-level mechanisms for resource transfer, some degree of QoS-crosstalk is unavoidable.

The problem of effective control over I/O resources is tackled more convinc-

[5] The implementation takes approximately 16 PALcode instructions on the Alpha.
[6] Version 2.7.2.

ingly in [Black94]. Although a number of useful mechanisms for streamlining I/O in a connection-oriented environment are presented, the prototype system known as *Fawn* was designed for the port-controller of an ATM switch, and so multimedia activities were restricted. The only high-bandwidth I/O device available was an ATM interface which required use of the CPU on a per-cell basis.

Explicitly scheduling the activities of the ATM device driver as a user-level domain, rather than performing the cell-forwarding function in an interrupt handler with no resource accounting, was demonstrated both to improve overall throughput and to allow QoS firewalls to be introduced protecting various other activities such as connection management. This scheduling, however, was only effective due to the the lack of DMA support causing the CPU resource to be the system bottleneck. Provision of QoS guarantees during concurrent use of the ATM device by multiple clients was not addressed, but would require high-level QoS management functions and more sophisticated intra-process scheduling mechanisms within the device driver.

3.9 Summary

Nemesis as described in [Roscoe95] provides highly effective mechanisms for multiplexing the CPU resource between a number of concurrent activities according to QoS contracts negotiated out-of-band with a QoS manager.

For these guarantees to be meaningful, the majority of in-band operations traditionally performed by the operating system are performed by unprivileged code in shared libraries forming part of the application. Only infrequent out-of-band operations are performed by trusted servers required to maintain shared state in a consistent manner.

The CPU, however, is only one of a number of resources required by a second-generation multimedia application. Effective partitioning of other system resources, particularly those involved in I/O, has not been previously addressed.

The remainder of this dissertation will address this issue.

Chapter 4

Scheduling I/O Resources

If an operating system is to be able to control the distribution of I/O resources between competing clients, mechanisms are necessary for scheduling access to those resources. It must be possible for the operating system to determine both when each client is handed the resource, and for how long. The effectiveness with which this can be achieved is determined by the hardware architecture of the workstation.

4.1 Traditional Workstation Architecture

Most modern workstations are designed around a bus architecture. All I/O devices are accessed across a time-division multiplexed bus where I/O transactions may either be performed using the processor (often called Programmed I/O) or using some form of bus-mastering or Direct Memory Access (DMA).

In order to achieve reasonable performance and avoid deadlocks, it is essential that the bus implement some form of scheduling and policing. It is extremely rare, however, that these functions are under the control of the processor. For example, DMA controllers often give the programmer the illusion that long transfers from a device to main memory may be performed uninterrupted, but it is normal for a bus to provide mechanisms for aborting DMA transfers in order for the CPU to gain access. It has also been found necessary for buses to provide some sort of arbitration scheme whereby more important transactions can be given preferential access to the bus. The policies for these arbitration mechanisms are usually implemented in hardware and are rarely more sophisticated than simple static-priority mechanisms.

Although the bus often has sufficient aggregate bandwidth to support all of the connected I/O devices, data usually has to travel across the bus more than once in order to reach its eventual destination. This situation is unavoidable on buses which support only a single master device, effectively forcing the use of main memory as one endpoint of each transfer. Despite the fact that recent bus architectures such as PCI [PCI95] support peer-to-peer transfers, it is rare for peripheral devices to take advantage of this feature. This is perhaps the main reason why the I/O bus becomes a bottleneck when a traditional workstation is used for multimedia activities.

The TURBOchannel and SCSI specifications are considered in slightly more detail since they are the interconnect technologies found in the experimental platforms used for this work.

4.1.1 TURBOchannel

The TURBOchannel [DEC90] is a 32-bit wide synchronous, asymmetric I/O channel which can be operated at any fixed frequency between 12.5MHz and 25MHz. TURBOchannel is asymmetric in that it supports one *system* module and a number of *option* modules. The system module usually contains the processor and memory system and the option modules are used for connection of peripheral I/O devices.

A variety of transactions may be performed on the TURBOchannel falling into two broad categories:

- **Programmed I/O (PIO) Transactions** The system module can perform a read or write to a single option module.
- **DMA Transactions** An option module can read or write to the system module.

It is impossible for an option module to address another option module on the TURBOchannel - i.e. peer-to-peer DMA is impossible.

DMA transactions may be of arbitrary length up to a system defined maximum which must be at least 64 words. After an initial overhead of 5 cycles to transfer the first word, an additional word of data may be transferred per clock cycle. DMA transactions may not cross 2048-byte page boundaries. The exact scheme used for DMA request arbitration is implementation specific allowing elaborate fair-service mechanisms - but most implementations currently use static priority or round-robin arbitration.

Programmed I/O transactions transfer only a single word of data and take 5 clock cycles.[1] Back-to-back transactions to the same option must be separated by a special inter-transaction cycle. Some implementations may even require additional idle cycles between PIO transactions. These factors conspire to make PIO an excessively costly operation.

4.1.2 Small Computer System Interface (SCSI)

SCSI is a vendor and architecture independent ANSI standard for the connection of peripheral I/O devices [ANSI86]. The standard covers all layers from the physical and electrical connection of the devices up to defining several generic device classes and protocols for communicating with these devices. Non-generic functionality may still be exploited via vendor-unique fields and commands.

The original SCSI-I standard specified an 8-bit wide bus clocked at 5MHz, but the newer SCSI-II standard allows 8, 16 or even 32-bit wide buses clocked at up to 10MHz providing data transfer rates of up to 40MBps. In all cases, the maximum number of devices which may be connected is eight. These are assigned unique addresses from 0 to 7 which statically determine their priority during bus arbitration phases (the higher the numeric value of the address, the higher the priority).

SCSI transactions take place between an *initiator* and a *target* device. Although the standard allows multiple initiators and multiple targets connected to the same bus it is normal to have a single initiating device (the host computer) and multiple target devices. It is also possible for a device to function both as an initiator and as a target, though not simultaneously. In addition to the normal transfers between the host computer and a single peripheral device, limited peer-to-peer transfer support is provided.

The SCSI bus allows logical connections to persist between pairs of devices

[1] TURBOchannel also supports a *Block I/O* mode for writes only which allows additional words of data to be transferred as for DMA transactions.

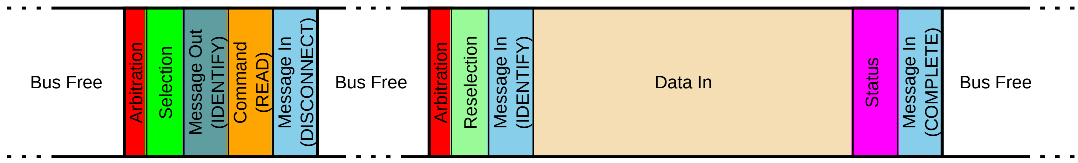


Figure 4.1: SCSI Bus Phase Sequences.

even while they are physically disconnected from the bus. This allows targets to free the bus for use by other devices if for some reason the transaction cannot be completed immediately. This feature is commonly used by disk drives whilst seeking the head.

A typical SCSI transaction is composed of a number of bus phases, the timing of which are not always under control of the initiator. Figure 4.1 shows the sequence of bus phases for a very simple read transaction. For longer transactions, the DATA IN phase shown in the diagram would often be fragmented using a number of additional disconnections and reselections. The meanings of the various phases are as follows:

- The ARBITRATION phase determines which of potentially several devices requiring access to the bus has the highest priority and gives that device control of the bus.
- The initiator uses a SELECTION phase to select the target of the operation. This phase establishes a connection between the two devices.
- A RESELECTION phase may be used by a target to re-establish a connection to an initiator in order to complete an interrupted transaction.
- The data transfer phases (COMMAND, DATA, STATUS and MESSAGE) are used to move bytes across the bus between the initiator and target. Two methods of transfer are supported - *synchronous* and *asynchronous*. The faster synchronous mode is only supported during the DATA phase.

4.2 The Experimental Platform

Figure 4.2 shows a block-diagram of the DEC 3000/400 AXP (Sandpiper) workstation [DEC94b] upon which most of the work described in this dissertation was

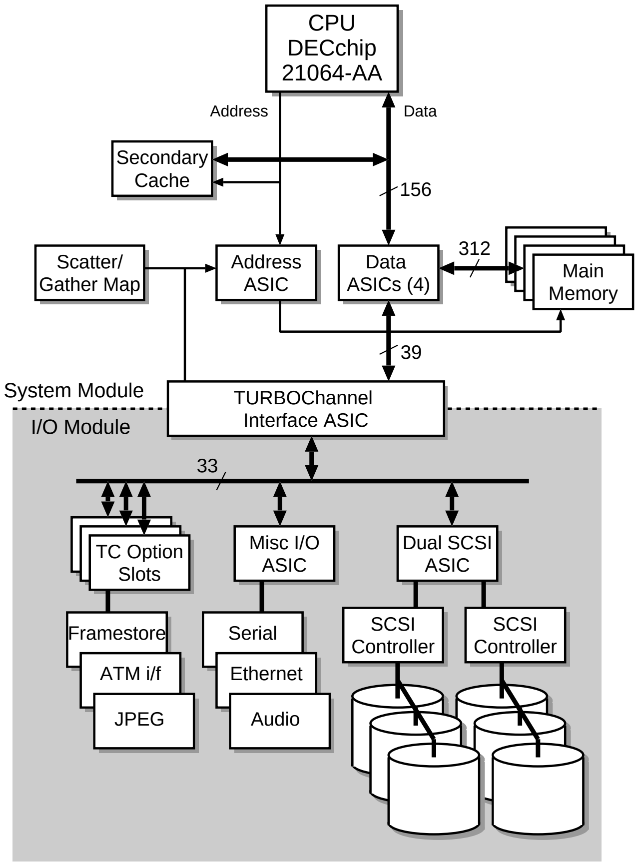


Figure 4.2: Block Diagram of DEC 3000/400 AXP (Sandpiper)

Device	Bandwidth
Dual SCSI ASIC	80 Mbps
OTTO Network Interface	300 Mbps
J300 Video Capture Card[1]	720 Mbps
PMAG-BA Framestore[2]	240 Mbps
Ethernet[3]	10 Mbps
ISDN[3]	128 Kbps
Serial, Mouse and Keyboard[3]	30 Kbps
Real Time Clock[3]	n/a
Flash EPROM[3]	n/a

[1] Bandwidth limited by TURBOchannel DMA.
[2] Bandwidth limited by TURBOchannel PIO.
[3] These devices are all connected via the IOCTL ASIC and thus share a single TURBOchannel slot.

Table 4.1: Peak Bandwidth Requirements of TURBOchannel Devices

performed. The workstation is built around a 22.5MHz TURBOchannel giving an aggregate bandwidth of 720 Mbps. DMA arbitration is performed in hardware using a round-robin mechanism and the maximum contiguous DMA transfer length supported is 128 words.

Although the TURBOchannel control ASIC on the DEC 3000 AXP series workstations decodes 8 option slots, only 5 are used on the Sandpiper. Three of these slots are used for plugging in expansion cards: the experimental platforms were equipped with an OTTO ATM interface, a PMAG-BA framestore and a J300 video capture and JPEG compression card. The remaining two slots are multiplexed further using ASICs to allow connection of two SCSI controllers and a variety of miscellaneous low-bandwidth devices including an ISDN device, two dual UARTs and a Lance Ethernet.

The SCSI buses are attached via NCR53C94 controllers and are clocked at 5MHz giving an aggregate bandwidth of 40Mbps each. The experimental machines were supplied with a CDROM drive and a Digital RZ26 hard disk supporting a maximum transfer rate of around 26Mbps.

4.3 Scheduling the Interconnect

One possible approach to providing QoS guarantees for I/O transactions would be to schedule the workstation interconnect since this would effectively schedule the activity of all devices connected to the interconnect. The feasibility of this technique clearly depends to a large extent on the type of interconnect around which the workstation is constructed, and the degree of control which the processor can exert over access to the interconnect by other devices.

The following sections consider the possibility of scheduling the various interconnects found in the experimental platforms available in the Computer Laboratory, and also the possibilities where the workstation interconnect is provided by an ATM switch as in the case of the DAN.

4.3.1 TURBOchannel

The possibility of scheduling the TURBOchannel depends entirely on the exact implementation of the DMA arbitration mechanism and the ability of the processor to control the background activities of the option modules. This section examines the possibility of controlling I/O activities by scheduling the TURBOchannel on the Sandpiper workstation.

A number of option modules, for example network interfaces, perform unsolicited DMA transactions as a result of external events. As packets arrive on the network, they must be DMAed into buffers in main memory. Some devices have only a small amount of internal buffering and, if prompt access to the TURBOchannel is not available, are forced to discard data. Most devices provide a means for the operating system to disable DMA, but DMA request signals are not available to the processor.

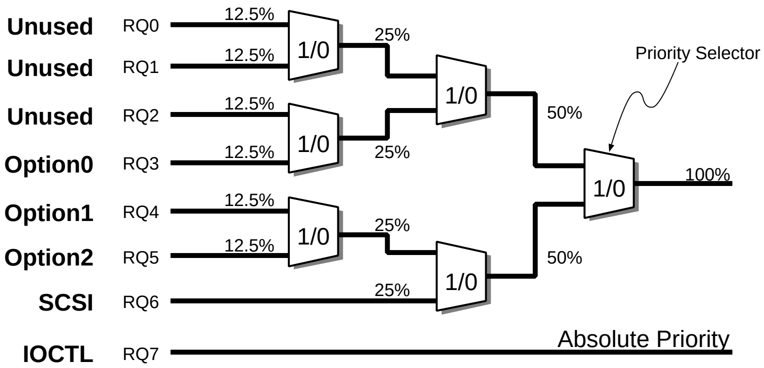


Figure 4.3: DEC 3000 AXP TURBOchannel DMA Arbitration Logic

The TURBOchannel control ASIC used in the Sandpiper provides a number of registers for configuring the operational parameters of the bus. Whilst the ASIC provides control over the types and maximum length of I/O transactions, and raises interrupts when error conditions such as I/O timeouts or parity errors arise, it does *not* provide any mechanisms for affecting the DMA arbitration or policing processes. The DMA arbitration mechanism is implemented entirely in hardware and consists of a binary tree of priority selectors as shown in figure 4.3. Each priority selector maintains state causing it to give priority to the input which lost the previous arbitration contest.

Despite the fact that the IOCTL ASIC has very low total bandwidth requirements, it is connected to input 7 of the arbitration logic which has absolute priority over all other DMA request lines. This is due to the fact that a number of devices connected to the IOCTL ASIC, notably the Lance Ethernet chip, require almost immediate access to the TURBOchannel.

It should also be noted that dual SCSI ASIC, connected to input 6 has double the normal probability of gaining access to the bus. If request lines 0-6 were asserted simultaneously, there would be a 25% probability of device 6 obtaining control of the bus, and 12.5% probability of each other device gaining the bus. These probabilities are obviously seriously affected by the fact that slots 0-2 are unused on the Sandpiper and slot 3 contains the PMAG-BA framestore which is incapable of DMA.

Without a means for the processor, and hence the operating system to determine when a device requires access to the bus, and without mechanisms either to prevent all but one device from performing DMA, or to explicitly grant DMA requests from software, it is impossible to implement any form of scheduling of the TURBOchannel. This situation is exacerbated on the Sandpiper by the hierarchical nature of the I/O system, in particular the two SCSI buses connected to a single option slot. Section 4.3.2 discusses the additional problems presented by SCSI.

Even if it were possible for the processor to control the amount of interconnect bandwidth available to each TURBOchannel option, it would do little to assist the provision of QoS guarantees to individual applications. Such guarantees would still require all devices to implement scheduling of requests from distinct clients.

4.3.2 SCSI Transactions

The classes of device usually connected to a SCSI bus (e.g. Disks, CDROMs, tape drives, etc.) have the common property that the timing of their I/O transactions is often dictated by mechanical constraints, such as the rotation speed of a disk. The controller firmware running on the device is usually designed to minimise this effect, and techniques such as read-ahead and cacheing are common. Despite these attempts, performance and I/O latency often suffer badly if the device is not serviced soon after it signals its readiness. The SCSI reselection mechanism mentioned in section 4.1.2 is indicative of this problem.

Although devices are required to ask the host controller for a reselection when they are prepared to complete an I/O transfer, the hardwired bus arbitration mechanism means that the processor only knows about the highest priority device requesting reselection, and is unable to preferentially service requests from devices with a lower numeric device ID. The allocation of SCSI IDs can have serious effects on performance and is usually performed by moving hardware jumpers on the devices. It is normal for the host controller to be allocated SCSI ID 7 to allow it to gain access to the bus with the minimum latency.

The lack of useful information by which the processor could order I/O transactions, and the relatively high latency of sending small messages across the SCSI bus mean that scheduling of transactions is often better performed by the device itself. A number of high-performance SCSI-2 devices are prepared to accept multiple outstanding transactions and service them out of order using information only available internally to make the best scheduling decisions. These features can improve performance in a number of cases, but they usually result on policy being built into the hardware, which can have a catastrophic effect for non-conforming applications.

4.3.3 Hierarchical Interconnects

For reasons of cost-effectiveness, current workstations are usually constructed using a hierarchical bus topology. The processor and memory system are connected using proprietary bus interface designs, and the main I/O bus will usually be a high-performance industry standard bus, but mass-produced components and peripherals are attractive due to the low price and usually require connections to legacy bus architectures, such as the ISA, EISA and VESA buses in the PC market. For this reason, most workstations provide support for legacy buses using bridge chips on the main I/O bus.

Performing I/O to a devices may therefore require simultaneous use of all buses en-route. Scheduling such an I/O interconnect in the generic case becomes a simultaneous resource scheduling problem akin to the scheduling of data transfers in parallel computers and communications systems. [Jain92] provides a good analysis of the issues involved, showing that the general problem is NP complete. In certain restricted cases an algorithm is presented which leads to an optimal interconnect schedule for a single bus in $O(n^4)$ time. An extension to hierarchical buses is also described.

In the area of parallel computing, however, the aim of scheduling the interconnect is to maximise its utilisation so as to minimise the elapsed execution time of a large number of communicating parallel computations. The solutions presented assume a fixed task set with well known communication patterns, and rely on the off-line computation of a static schedule. These aims and techniques are at odds with the provision of dynamic Quality of Service guarantees.

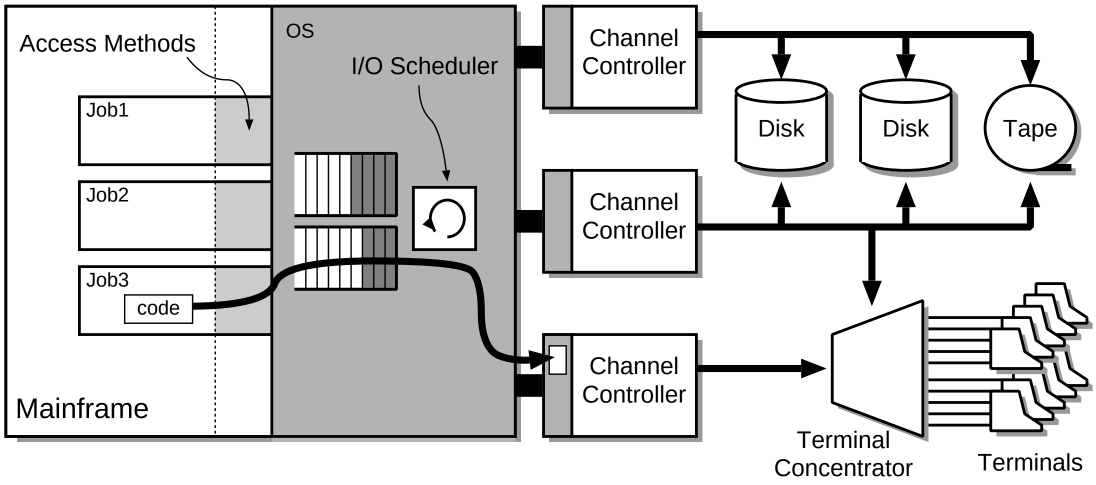


Figure 4.4: Use of Channel Controllers Under MVS.

4.4 Channel Controllers

Mainframes are usually designed to make the most efficient use of the main processors in a batch computing environment. I/O operations are slow in comparison and are often performed in the background using *channel controllers* to allow the main CPU to be used more productively. Channel controllers serve much the same rôle as DMA controllers in a modern workstation, but since the classes of I/O device involved generally require less direct interaction they are often almost entirely autonomous.

A typical mainframe would have several channel controllers providing multiple concurrent paths to I/O devices such as disks, tape and terminal concentrators (figure 4.4). The channel controller is responsible for scheduling transactions, enforcing the necessary protection and allows each I/O resource to be treated effectively independently.

Under MVS [IBM80], code written dynamically by user processes is downloaded into a free channel controller by an I/O scheduler in the operating system which maintains queues of pending I/O transactions. Simple verification of the code if performed before it is executed asynchronously by the channel controller. The channel controller interrupts the operating system when the I/O transaction has completed and in the case of synchronous I/O the job is restarted. Shared libraries known as *access methods* provide standard I/O functionality hiding the complexity from the programmer.

This model of I/O is highly appropriate both in situations where I/O is a time-consuming operation, and in situations where high volumes of data must be transferred without intervention by the main CPU. Multimedia workstations have similar I/O requirements, yet it is rare for hardware to provide this degree of unsupervised operation.

4.5 The Desk Area Network (DAN)

Conventional bus-based architectures are already proving inadequate for high-end workstation designs. For example, in high-end PCI systems [PCI95], the electrical constraints of running a 64-bit wide bus at 66MHz or more mean that the number of devices which may be connected to the bus is limited to 2 or 3 - clearly inadequate for building a sensibly configured workstation. Many manufacturers are therefore moving towards switch-based interconnects.

The DAN architecture [Hayter91] is an attempt to make explicit the inherent multiplexing issues of a workstation interconnect. At the same time it addresses the scalability problems of contemporary workstation architecture by simplifying the use of peer-to-peer transfers. The DAN architecture differs from a traditional workstation in a number of important aspects.

In a DAN-based workstation, all communication across the interconnect is connection-oriented and is performed using a small fixed-size transfer unit. Although [Hayter91] describes the use of a space-division ATM interconnect, it should be noted that this does not in any way imply the full complexity of the emerging ATM networking standards [ATMForum93]. The 100% reliability of a workstation interconnect allows the use of much simpler protocols. The DAN approach has a number of important benefits:

- Connection-oriented communication and the fixed-size transfer unit aid accounting for resource usage.
- The implementation of hardware protection mechanisms is much simpler allowing the design of devices which provide a *user-safe* interface.
- By supporting peer-to-peer transfers the usual memory bottleneck is removed.

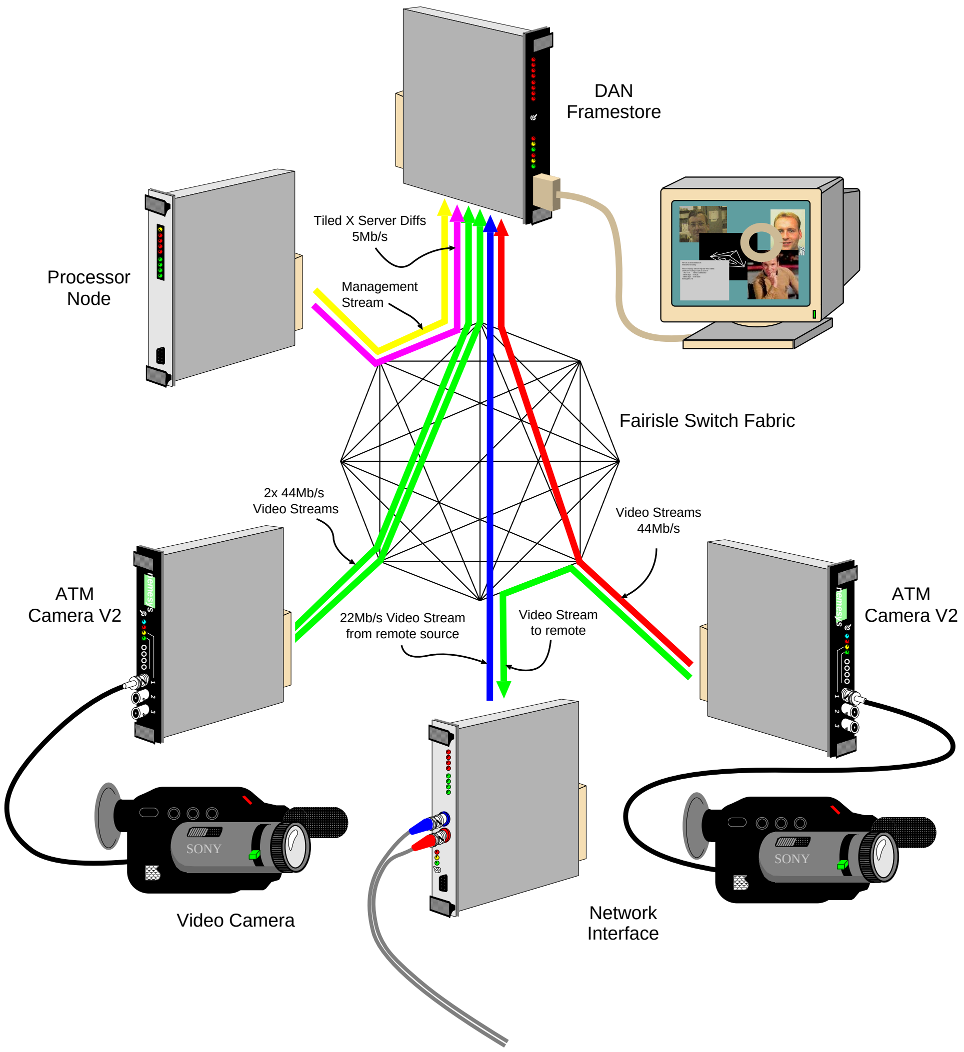


Figure 4.5: DAN Based Workstation

- The small fixed-size transfer unit permits frequent reassessment of priorities and provides regular preemption points.

The use of an ATM-based interconnect for user-class I/O devices is also being investigated by the SUN DeskNet project [Lee95]. DeskNet proposes connection of a number of devices including the framestore, keyboard, mouse, video and audio hardware to a conventional workstation using a 1Gbps ATM ring. Other examples of research in this area include the MIT ViewStation project [Houh95] built around the VuNet switch fabric, and the Symphony architecture [Bovopoulos93].

4.5.1 The Prototype DAN Workstation

The prototype DAN multimedia workstation described in [Barham95] uses the 8×8 space-division ATM switch fabric developed by the Fairisle project [Leslie91] as its interconnect. A typical configuration is shown in figure 4.5. One important result of enabling simple and efficient peer-to-peer communication is that the bandwidth required for cache-memory traffic is greatly reduced due to the lack of data-copying.

The possibility of using this same interconnect for carrying cache-memory traffic was investigated in [Hayter93]. Despite the relatively dated technology[2] used in the implementation of the prototype, the results achieved demonstrated the feasibility of this approach. Also described was the concept of a *stream-cache* whereby continuous media data may be piped through a reserved region of the CPU cache without having to pass through memory, allowing streamlined processing of data with spatial but not temporal locality-of-reference.

4.5.2 User-Safe Devices

Although interconnects such as PCI support peer-to-peer transfers, it is rare for I/O cards to make use of this capability. The traditional model for hardware devices is one of a relatively simple state machine which is guided through each transition by a single privileged device driver inside the operating system kernel. The driver must often execute rapidly in response to interrupts raised by the device if performance is not to suffer.

This device model relies on the fact that I/O has always been performed by the operating system, rather than the application. Quality of Service considerations within the operating system are beginning to make this model unacceptable - it is no longer desirable, particularly in a multi-service operating system, that unrestricted amounts of system resources be consumed by device-drivers performing work on behalf of applications.

For peer-to-peer transfers to be used effectively, it is essential that some form of connection exists between the communicating devices. When an I/O transaction is performed, the connection identifier (e.g. the Virtual Circuit Identifier

[2] The Fairisle switch fabric has an 8-bit wide data path and is clocked at up to 20MHz compared with the external cache interface of the Sandpiper which is 128 bits wide and can perform a read or write in 5 cycles at 133MHz giving a cache bandwidth of 427MBps.

(VCI) in the case of an ATM interconnect) may be used to index into a table of per-connection state to rapidly determine whether the transaction should be allowed. Without per-connection state in the device, third party intervention is required to check the validity of each operation.

With this additional functionality present in device hardware, it becomes possible to remove the device driver completely from the I/O data-path. The device driver is still necessary to maintain the per-connection state in a consistent and secure manner, but data-path operations may easily be made available to unprivileged processes. This may be achieved by mapping I/O registers into the virtual address space in multiple places, once for each client, or by extending the processor context-switch code to update client-ID registers in I/O hardware. A more sophisticated technique would be for the CPU node to provide a pool of DMA engines for use by user-level processes. These ideas are more fully treated in [Pratt96].

A number of devices conforming to this model have been constructed as part of the DAN Devices project [Barham95], and are referred to as *User-Safe Devices*. Sections 5.4.1 and 6.6.5 describe two such devices which have been used for part of this experimental work.

4.6 Scheduling an ATM Interconnect

Although the use of an ATM interconnect within a workstation is still rare, switch-based solutions are likely to become more common in the near future. The use of an ATM switch-fabric enables the interconnect to be scheduled in terms of individual I/O connections discriminated via the VCI, rather than just the destination device. Scheduling mechanisms designed for use within Local Area Network (LAN) switches are potentially applicable.

The Parallel Iterative Matching (PIM) algorithm of the AN2 switch described in [Anderson93] is designed to approach maximum utilisation of the space-division interconnect. Rate control for each virtual circuit is imposed in the source host-interface and cell-by-cell credit-based flow-control is used to prevent cell-loss.

Support for CBR traffic has also been added to the Fairisle network by computing a schedule for the switch fabric. The technique works by reserving an appropriate number of slots for each CBR connection in a switch-wide framing structure using a distributed algorithm described in [Khan94]. The short duration of individual connections in a DAN, and the extreme burstiness of traffic present problems for this sort of long-term peak reservation strategy.

The number of concurrent streams within a DAN, however, is likely to be much smaller than in the case of a LAN switch with each stream consuming a larger proportion of the aggregate bandwidth. The degree of correlation between streams is also likely to be much higher. When combined, these two factors mean that scheduling on an ATM interconnect is almost essential. Unfortunately, the resulting decrease in potential for "statistical multiplexing" of traffic tends to make scheduling algorithms designed for LAN switches inappropriate.

DeskNet uses a novel MAC protocol which provides a guaranteed minimum QoS to each connection in situations of high load but which allows transmission at a second, higher rate if the interconnect is observed to be idle. The interconnect scheduling mechanims are designed to be readily implemented in hardware, but the parameters controlling their behaviour are intended to be under software control.

The tightly coupled nature of the DAN, and of a workstation interconnect in general, means that global information is more easily available to the operating system with which to make scheduling decisions. The time scales over which connections may endure, and the burstiness of traffic require hardware mechanisms similar to those proposed for DeskNet. It is also likely that close integration of the CPU scheduler and the interconnect scheduler would be required in such a machine where the CPU is used more like a stream processing device.

4.7 Summary

Effective control of the distribution of I/O resources requires scheduling of *both* the interconnect and the devices, since both are multiplexed. Scheduling the interconnect alone is not sufficient to provide Quality of Service guarantees to *individual* clients of devices.

In conventional workstation designs however, the interconnect is multiplexed in a fashion which is not amenable to control by the operating system. Many devices such as disk drives have natural timing constraints which lead to poor performance if they are not respected. Network devices assume that they can gain access to the bus rapidly in response to events which are not under the control of the operating system. In order to support devices of this nature, hardware arbitration mechanisms often embody policy decisions which should be under software control.

Hardware solutions such as channel controllers have been used in the past to provide effective resource firewalls between competing I/O activities. Although the abstraction provided is ideally suited to the multimedia environment, these solutions rely on replication of hardware to eliminate crosstalk and are both expensive and inflexible.

Bus-based designs are already becoming unsuitable for high-end workstations and switch-based solutions are starting to appear. It is possible that future interconnects will provide connection-oriented I/O and support peer-to-peer transfers by use of user-safe devices. This will greatly simplify the integration of device and interconnect scheduling.

In conventional workstations, scheduling of the interconnect by the operating system is not a feasible approach. Admission-control techniques can be used to minimise the probability of the interconnect being overloaded, but the hardware arbitration mechanisms must be relied upon to minimise jitter and latency. Since the multimedia environment is one where applications can make effective use of soft-guarantees, uncoordinated software scheduling of each device is sufficient to provide satisfactory QoS guarantees to individual clients.

Chapter 5

Device Driver Architecture

The ability of a multimedia application to make efficient use of non-CPU related resources provided by an operating system is clearly influenced by the manner in which that operating system abstracts device hardware and I/O in general. This chapter presents an architecture for device drivers, designed specifically for the Nemesis operating system, which is able to deliver guaranteed performance I/O to *individual* applications.

5.1 Introduction

The most important reasons for the existence of device drivers in an operating system are:

- **Safe programming** (of hardware). Incorrect programming of hardware can often result in failure of the system. It invariably results in loss of service. The driver provides code which is trusted to program the hardware in a safe manner and cooperate with the operating system as a whole.
- **Abstraction** (of device interface). Clients of a device do not wish to be concerned with variations between hardware supplied by different manufacturers. The interface provided by the driver should hide these variations.
- **Protection**. The device driver is responsible for protection of a number of varieties. Clients should be protected from the actions of each other,

malicious or otherwise. Suitable protection mechanisms should also be provided to prevent unprivileged clients from performing privileged operations or denying the use of the resource completely.

- **Multiplexing**. The device driver allows simultaneous use of the hardware by a number of clients. It must maintain a consistent view of the state of the hardware and perform the necessary context-switches between clients. It is the device driver which dictates when each client will gain access to the physical resource.

The above functions are necessary within *any* environment where a number of clients share the same resources. The Nemesis environment however places additional demands on device drivers:

- Drivers must not hide the shared nature of the underlying physical resource but instead provide explicit control over the multiplexing of that resource.
- Applications need to be aware of the current level of resources to which they have access. Negotiated QoS-guarantees should be provided to each client of the driver.
- There should be simple and effective feedback mechanisms to allow an application to monitor its progress and adapt its behaviour in light of the rate at which I/O requests are being serviced.

5.2 Nemesis Device Driver Architecture

This dissertation proposes a new approach to device abstraction which separates the control- and data-path functions required for I/O in a multi-service operating system such as Nemesis. The structure of a generic Nemesis device driver is shown diagrammatically in figure 5.1. As can be seen from the diagram, the functionality is implemented by two main *modules*:

- The Device Abstraction Module (DAM)
- The Device Management Module (DMM)

Whilst these two modules may often execute in the same Nemesis *domain*, for certain devices the DMM is more logically implemented in a separate domain.

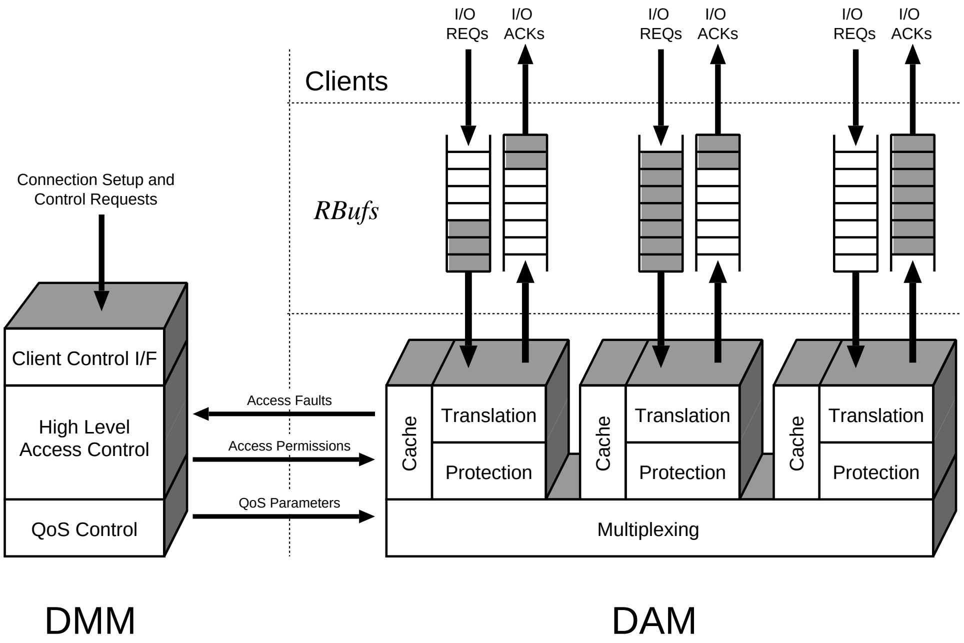


Figure 5.1: Nemesis Device Driver Architecture

5.2.1 Device Abstraction Module (DAM)

The DAM resides on the I/O data path and is intended to contain the **minimal** functionality to provide secure user-level access to the hardware and support QoS guarantees to clients. The DAM serves three main purposes:

- **Translation** (of *stream* addresses). It is highly desirable that the addresses present in an I/O stream should be independent of the destination of the stream. This allows a single stream to be multicast efficiently to a number of destinations, and also to be processed entirely in hardware should the workstation provide this facility.
- **Protection** (between clients). The addresses to which each client may perform I/O are usually different. The DAM should ensure that a client cannot perform I/O to addresses for which it does not have suitable access permissions.
- **Multiplexing** (of physical resources). The DAM is the *single* multiplexing point for a hardware resource. It is responsible not only for providing shared access to an I/O resource, but also for controlling both the amount of resource which each client receives and the time at which the client

receives it. The DAM must provide a *scheduler* for this purpose. This is the most fundamental difference between a Nemesis device driver and those of traditional operating systems, microkernel or otherwise.

Address translation and protection functions are performed individually for each I/O connection, since all I/O requests on a particular connection inevitably have the same source.

Explicit and visible multiplexing is performed at a single point within each driver. It is at this point where hardware resources are scheduled. With each connection are associated QoS parameters which are used to control the multiplexing. These parameters are themselves determined by out-of band negotiation between the client and a QoS-manager domain. The exact scheduling algorithms applied are likely to be device specific, but in general any algorithm which supports QoS-guarantees is potentially applicable. Examples include *stride-scheduling* [Waldspurger95], Jubilee-scheduling [Black94] and the `RSCAN` algorithm for disk-head scheduling presented in section 7.6.1.

The low-latency feedback requirement is provided by use of the asynchronous *Rbufs* mechanism described in section 3.5.3. Clients are able to observe both the length of the queue and the rate at which requests are being serviced and may use this information to adapt their behaviour. Each connection to the driver has an independent queue and so the QoS-crosstalk prevalent in first-come, first-served (FCFS) queueing systems is avoided.

5.2.2 Device Management Module (DMM)

Out-of-band control of the translation, protection and multiplexing functions of the DAM is performed by a separate management entity known as the Device Management Module (DMM). The DMM communicates with the multiplexing layer of the DAM in order to set up new connections and adjust the QoS-parameters associated with existing connections. The DMM is **never** involved with the in-band operations of the device driver.

The DMM uses high-level descriptions of access-permissions (e.g. file-system meta-data or window arrangements), together with access-control policies to generate the low-level protection and translation information required by the DAM. These low-level permissions are often cached within the DAM's per-connection state records to reduce the number of interactions with the device manager. This cache provides conceptually similar functionality to the TLB of modern processors.

5.3 Network Interfaces

Network interfaces are one of the few areas where some attention has recently been paid to QoS considerations at the hardware level. For this reason, it is to be expected that the above device model should map most naturally onto the network interface device.

[Black94] proposes a binary classification of network interfaces based upon their ability to identify the eventual destination of data, often by some pre-established connection identifier, and arrange to place data in the correct place as it is received. Devices which have this property are described as *Self-Selecting* interfaces.

The remainder of this section considers the application of the Nemesis I/O model to network interfaces of both classes.

5.3.1 Non-Self-Selecting Interfaces

Perhaps the most common non-self-selecting interface is the Ethernet. The networks to which these devices attach are typically connectionless and therefore a good deal of protocol processing is required to determine the eventual endpoint of the data. In the majority of operating systems, all protocol processing is performed in the kernel and not accounted to the application for whom the data was destined.

Packet-filtering techniques [Mogul87] have recently been used in conventional operating systems to allow protocol processing code to be removed from the kernel for ease of debugging and to minimise the complexity and size of the kernel. A suitable privileged process may download a description of "interesting packets" to the device driver, often in the concise form of a set of masks and comparisons applied to the protocol headers. Any packets matching this *filter* are then passed up to the process in question for further processing. Usually however the protocol processing is still performed using a single privileged server such as the x-Kernel [Hutchinson91], rather than by the application.

In order to remove QoS-Crosstalk completely, it is necessary for the protocol processing operations of each application to be effectively isolated. For reasons of security, it is highly undesirable to allow applications to transmit or receive arbitrary packets across a network. [Black94] describes a mechanism which uses packet-filtering techniques on both the receive and transmit sides of a network interface to enforce these security issues. An application may only receive or transmit packets which match the filters set in place by some trusted server at connection setup time.

The filters installed in the network interface device driver also allow packets to be correctly accounted at the lowest level allowing QoS guarantees to be supported. Similar use of packet-filters is made by the Aegis Exokernel [Engler95] which provides secure multiplexing of an Ethernet interface by dynamic packet filter code generation within the kernel, but the motivation in this case was to allow multiple "library operating-systems" to transparently share the same hardware.

Until recently, non-self-selecting interfaces have been typically low bandwidth and with modern workstation speeds and efficiently implemented protocol stacks, their detrimental effect on QoS is minimal. The x-Kernel was ported to Nemesis in order to provide interim support for interfaces of this kind, and in particular to provide TCP/IP connectivity over Ethernet enabling inter-operation with UNIX platforms in the experimental environment.

A disturbing trend for high bandwidth non-self-selecting interfaces is emerging. Network interfaces for FDDI and the various 100Mbps Ethernet standards are often designed to appear to the programmer to be exactly like their low-bandwidth counterparts.[1] With interfaces such as these, only highly efficient packet filtering can prevent QoS crosstalk in protocol code, and the resources expended in data copying operations must be carefully controlled. Since the underlying networks invariably provide little in the way of QoS support, any attention to QoS in the endpoint is likely to be of limited use.

5.3.2 Self-Selecting Interfaces

More recent high-speed networks necessarily use protocols which allow significant fractions of the protocol processing to be performed in hardware. Most

[1] Largely due to the widespread use of operating systems such as *Windows* where third-party development is difficult and seamless inter-operability is therefore essential.

notable of these are ATM networks, where all communication is performed over pre-established virtual circuits allowing the network interface to demultiplex conversations at the lowest level. It is common for these networks to support QoS provision on a per-connection basis. These QoS guarantees are useless however if the operating system of the destination machine does not deliver the data to its eventual endpoint in a timely manner [Saltzer84].

Considering the high throughput of these host-interfaces, it is important to ensure that data is not forced to traverse the workstation bus more often than absolutely necessary. Protocol stacks in operating systems such as UNIX often require data to be copied a number of times in addition to the unavoidable copies between user paged virtual memory and kernel locked-down physical memory. This can be highly detrimental to performance. Nemesis' single virtual address space and *Rbufs* I/O transport mechanism address these problems.

5.3.3 The OTTO ATM Interface

The OTTO is a 155Mbps ATM host interface adaptor for either TURBOchannel or PCI, originally designed for use with the AN2 network [Anderson93].[2] The OTTO provides extensive hardware support for the ATM Forum standard AAL5 ATM adaptation layer used by most commercially available equipment.

The OTTO reassembles complete AAL5 PDUs in its internal cell memory. When a complete PDU has been received and the CRC32 checksum is correct, the PDU is enqueued for DMA into main memory. A record associated with each receive VCI contains a pointer to a *buffer-ring*. Each buffer in the ring consists of a number of *fragments* which describe contiguous regions of physical memory. When a packet DMA has been completed, a timestamped *report* is written into a circular buffer, and an interrupt is raised.

The driver removes full buffers from the ring and replaces them with fresh buffer descriptors. Hardware mechanisms prevent the buffer ring from overflowing if not serviced promptly.

To transmit an AAL5 PDU, a free transmit buffer is obtained and its fragments made to point to the user data in physical memory. The buffer descriptor is placed on a per-VCI transmit queue. The packet is eventually DMAed into

[2] Product versions of the OTTO are now marketed by Digital Equipment Corporation as the *ATMWorks 750* (TURBOchannel) and *ATMWorks 350* (PCI) [DEC94a]

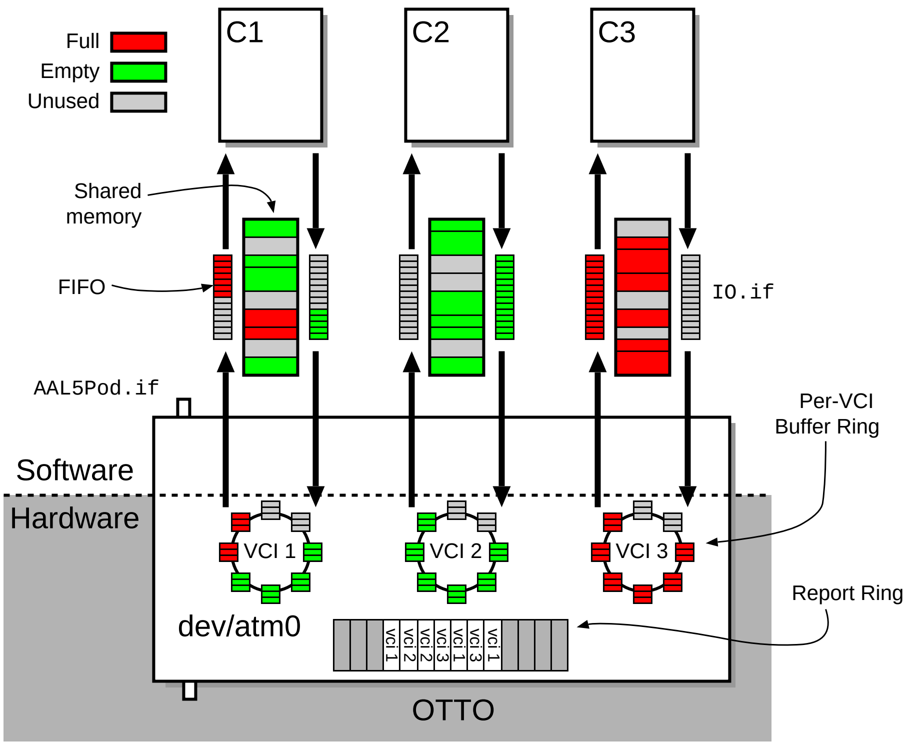


Figure 5.2: OTTO Device Driver Structure

the OTTO and fragmented into cells, computing the CRC32 checksum in the process. At this point, a report is generated indicating that the buffer in main memory may safely be freed.

Each transmit VCI may be marked as either scheduled or best-effort. The OTTO driver computes a cell-by-cell transmit schedule, based on QoS parameters, which is downloaded into the hardware. Each position in the schedule contains a VCI. For each slot in the schedule, the hardware first checks to see if the named VCI has cells to transmit. If it has, then a cell is sent, otherwise a cell from a best-effort VCI is sent in its place.

The OTTO also supports a proprietary flow-control mechanism known as FlowMaster, which is based on per-VCI cell-by-cell credits. This feature is only useful when talking to an AN2 switch or another OTTO.

5.3.4 DAM: The OTTO Device Driver

From the above description, it is clear that the OTTO hardware implements almost the entire functionality of the DAM in hardware. The majority of code in the driver is concerned with out-of-band operations such as initialising the device and reconfiguring when connections are set up or torn down. Since the hardware is relatively complicated, the device driver is certainly non-trivial[3] but despite this, very little software intervention is required on the data-path.

Figure 5.2 shows three clients of the OTTO driver receiving packets on different VCIs. Each client has obtained an *Rbufs* channel via the RPC interface exported by the DMM (`AAL5Pod.if`). Each client sends `iorecs` describing empty buffers to the OTTO driver which loads them onto the appropriate receive ring for the particular VCI. When complete packets are received, the OTTO driver returns the full buffer descriptors to the client.

In the expected mode of operation, the pipelining available when using *Rbufs* allows multiple packets to be transferred at a time without the requirement to reschedule (e.g. client `C1`). Clients which process their data too quickly, such as `C2` will block on an empty receive FIFO. In the situation where a client is unable to keep up (`C3`), the OTTO driver will not remove full buffers from the receive ring and the hardware will automatically stop receiving on that VCI until an empty buffer is available. This mechanism prevents the situation known as *livelock* where so much CPU resource must be expended during an overload condition that it is impossible to rectify [Mogul95].

Per-packet protocol processing in the OTTO driver is so cheap in comparison to CPU speeds that, when the system is lightly loaded, the client and driver enter a state where only one packet is available for processing at a time. The operating system must therefore context switch between the driver and the client on a per-packet basis. Although the CPU resource expended per packet is much higher than the case when the *Rbufs* pipelining is working efficiently, this situation is *caused* by the system being lightly loaded and so neither the total throughput obtainable nor the latency suffer significantly. The commonly used mechanism of spinning for a while before leaving the interrupt handler waiting for the next packet merely reduces the amount of CPU available to other domains. This is analogous to optimising the behaviour of the whole system during normal working operation, rather than attempting to optimise unimportant statistics

[3] Approximately 3000 lines of C.

such as “null-RPC times”.

At high loads, the number of packets received per interrupt increases until a situation is reached where the driver is effectively polling the hardware. Provided that the driver is scheduled with sufficient regularity, the hardware-maintained buffer rings will not overflow and no data will be lost.

5.4 Network Attached Peripherals

As device bandwidth requirements increase, and scalability problems of current interconnects start to impact performance, it is becoming increasingly common for devices to be designed for connection to a high speed local area network. Multimedia file servers and peripherals are highly amenable to this approach. A number of recent proposals for the connection of disk drives, for example Serial Storage Architecture (SSA) [Deming95], bear a strong resemblance to LAN technology.

Devices connected in this way must be designed to cope with the higher latency control path. This usually necessitates the clear separation of in-band and out-of-band functionality of the Nemesis I/O model. Network attached peripherals often provide most of the functionality of the DAM in the remote hardware, whilst the DMM is intended to be provided by a “manager” process running on a workstation.

The AVA-200 described in the following section is an example of such a network-attached peripheral which was designed in the Computer Laboratory. The firmware resident in the device was written to conform to the Nemesis I/O model and can effectively support a number of sophisticated higher-level applications. When video or audio clients are executing on a Nemesis workstation equipped with an OTTO, QoS guarantees from the AVA-200 and the OTTO driver may be used in conjunction to ensure that end-to-end quality of service is provided for each stream.

5.4.1 AVA-200 Hardware

The AVA-200 is a network-connected audio and video capture device derived from the much simpler ATM Camera originally built for the DAN [Pratt92]. The first

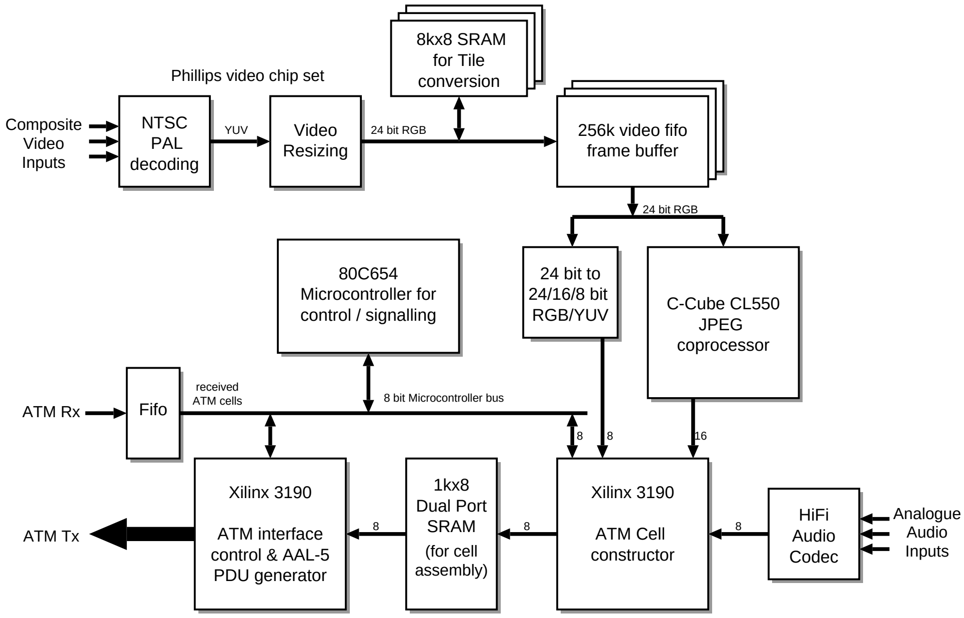


Figure 5.3: AVA-200 Major Data-paths

ATM camera was a fairly “dumb” device which needed close supervision by a processor node on the DAN. The AVA-200, however, was designed for connection to an ATM LAN and therefore needed to be largely autonomous. For this reason, a limited amount of processing power was included and the device is able to communicate with a remote *manager* process using a simple RPC protocol. The major data-paths within the AVA-200 are shown diagrammatically in figure 5.3.[4]

5.4.2 DAM: AVA-200 Firmware

The firmware executed by the AVA-200's microcontroller was designed and written by the author to support concurrent use of the device by a number of clients. Access control and out-of-band QoS negotiation is performed by a more intelligent and trusted *manager* process running on a workstation connected to the ATM network. Due to the obvious timing constraints, all multiplexing of the hardware must be performed by the device itself. The manager communicates with the AVA-200 on a notionally secure management VCI and is able to download simple descriptions of the required video and audio streams.

Since the hardware only contains a single digital video chipset, and a single

[4]This diagram is derived from a schematic drawn by Ian Pratt.

audio codec, there are inevitable hardware constraints on the number of audio and video streams which may be generated. When using *gen-locked*[5] video sources, it is possible to multiplex the video digitising hardware on a per-frame basis. For audio this is clearly not sensible. The firmware therefore supports multiple concurrent video streams, but only a single audio stream.

For each video stream it is necessary to download a number of parameters to the unit. These parameters specify picture sizes and scaling, pixel formats, data rates, the physical video channel to use and the VPI/VCI for the outgoing video stream. This information is loaded into a *video bucket*. Similarly, parameters for the audio streams including sample rate, format and physical audio input must be loaded into an *audio bucket*.

A *schedule* is then loaded into the AVA-200 which specifies a sequence of video buckets. This schedule is executed in a round-robin fashion, grabbing a frame using the parameters specified in the relevant video bucket. The reserved video bucket index of zero indicates that the unit should idle for a frame. The schedule also identifies which audio bucket should be used.

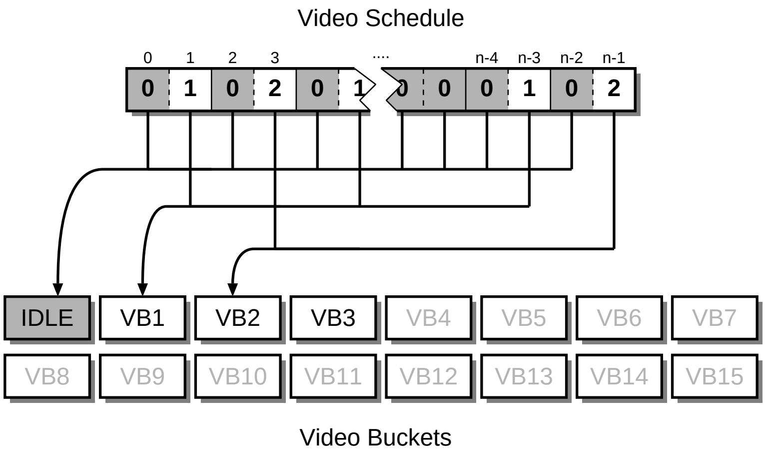


Figure 5.4: AVA-200 Video Programming Paradigm.

These datastructures are shown diagrammatically in figure 5.4 which illustrates two concurrent streams whose parameters are specified in video buckets `VB1` and `VB2`. The first stream has been allocated 3 frames out of every n, and the second 2 frames out of every n. The remaining frames are unallocated and

[5] PAL encoding follows an 8 frame sequence so resynchronisation between signals which are not at the same point within the sequence can be expensive.

are therefore spent idling. Bucket `VB3` has been loaded with the parameters for a third stream in preparation for an atomic change of schedule.

All audio and video streams are transmitted directly to the client across the network, without any further intervention by the manager process. In addition, a single synchronisation stream is sent to the manager containing the current sequence numbers of all audio and video streams. The manager is then able to forward appropriate synchronisation streams to each client. It is also possible to enable a credit-based flow control mechanism on a per-stream basis which can be used to cause the AVA-200 to cease transmission on a particular VCI if the sink is unable to keep up.

5.5 Summary

A software device-driver architecture has been presented which provides QoS guarantees to individual clients and minimises QoS-crosstalk between applications by clearly separating the control-path and data-path operations necessary for I/O.

The Device Abstraction Module resides on the data-path and is responsible for providing the minimum functionality required to multiplex the hardware between a number of clients in a safe and effective manner. This involves providing mechanisms for translation, protection and scheduling of I/O requests. In order to simplify the accounting and scheduling of I/O resources, devices are abstracted at a low-level using small, fixed-length operations. Visible multiplexing and effective feedback mechanisms are provided by use of the *Rbufs* mechanism.

The Device Management Module resides on the control-path and is responsible for directing the operation of the DAM. Clients communicate with the DMM using a synchronous RPC interface to create new connections and perform out-of-band control requests.

Network interfaces and network-connected peripherals often contain hardware which provides most of the functionality of the DAM. Application of the Nemesis device driver architecture to these devices has been demonstrated to be straightforward and highly effective.

Chapter 6

Window System

This chapter considers the application of the Nemesis device-driver architecture presented in the previous chapter to the framebuffer device. This device is of particular concern in a multimedia system where it must often deal with simultaneous high bandwidth I/O from a number of applications. It is also a device which is particularly difficult to abstract securely at a low level and for this reason the majority of operating systems make do without a device driver at all and rely on a window system to abstract the device at a much higher level.

6.1 Introduction

The purpose of a window system is to mediate access to shared resources including the pixels of the framebuffer and a number of input devices. The abstraction presented to clients is usually that of a virtual framebuffer supporting both read and write operations. It is the job of the window system to protect clients against each other. This usually involves preventing one client from updating the pixels logically owned by another client.

The window system is also responsible for demultiplexing input events from the various hardware devices to interested clients according to some policy usually based on the position of a pointer.

In a multi-service environment, it should also be the job of the window system to support the provision of end-to-end QoS guarantees. Consider a client of a window system whose purpose is to render an incoming network video stream in a window. Any operating system and network provided QoS guarantees become meaningless if it is impossible for the application to deliver the video data to the framebuffer in a timely manner.

Window systems can be divided into two broad categories based on the location of the actual rendering code:

1. Server Rendering window systems.
2. Client Rendering window systems.

Current window systems fall almost without exception into the first category.

6.2 Server Rendering

In such a system, protection is enforced by having all rendering operations performed by a single centralised server which maintains the state describing which pixels are owned by which window. Perhaps the best known example of a server rendering window system is the X Window System [Scheifler86]. Microsoft *Windows* [Hyman88] is unusual in that the operating system itself is also the window system server. Under *Windows-NT* [Custer93], a separate server process is now responsible for providing this same interface.

The client communicates rendering requests to the server using some form of IDC. Due to the relatively large overheads of communication (mainly processor context-switch overheads) compared with the cost of updating pixels in a framebuffer most systems use a variant of pipelined RPC or message passing.

The necessity for the window system to support all rendering primitives which may be required leads to a vast increase of complexity in the server. The client libraries must also provide RPC marshalling code for all of these operations. The X Window System supports the *X Extension* mechanism which allows new proprietary operations to be added to the server. However, since for the majority of platforms extensions may only be added at server compile time, this does not help to reduce the size and complexity of the server.

The set of rendering primitives supported are often numerous and relatively high level in order to amortise the cost of communication. High level primitives also reduce the cost of checking access permissions to regions of the framebuffer before performing updates. Large primitives are recursively subdivided into subtasks which are either wholly permissible or disallowed, enabling the low level rendering code to bypass additional checks.

Typically there is a single communications channel between each client and the server which is used for both rendering requests and configuration/control requests. Since the IDC bandwidth required for communication of high-level rendering primitives is small, and the channel is also used to carry in-band control requests, a reliable transport mechanism is often employed (e.g. X uses UNIX domain sockets for local connections and TCP/IP sockets for remote connections).

Unfortunately, this combination of large-grained rendering primitives and a single multiplexed connection between each client and server presents a number of problems for QoS provision:

- A client can cheaply generate rendering requests at a rate much faster than it is possible for the server to complete them.
- The IDC transport mechanism cannot distinguish between control messages requiring reliable semantics and rendering requests which potentially do not.
- The service time for requests can vary by several orders of magnitude between a simple request like "set pixel" and a complicated one like "fill arbitrarily shaped non-convex polygon".
- Requests on a connection may not be reordered or discarded since applications often make use of the reliability and ordering semantics when rendering complicated graphics.
- The server itself is often a single threaded application which processes client requests to completion in some arbitrary order.[1] This introduces large amounts of jitter.

In a multimedia environment, support for high bandwidth updates to the framebuffer and associated QoS guarantees is essential. In server rendering systems, support for high bandwidth updates has often been added by using a shared-memory transport for the video data and the standard transport for sending the update request.

[1] X uses FIFO queueing on each connection and services clients in a round-robin fashion.

It is obviously impossible to provide QoS guarantees to an application unless the server itself has enough aggregate resources to fulfil the guarantees of all of its clients. It is also necessary that the server respects the QoS guarantees of each client when performing work on their behalf.

Several mechanisms have been investigated for the transfer of resources and even threads between clients and servers (processor capacity reserves [Mercer93], lottery scheduling [Waldspurger94], thread migration [Hamilton93], etc.). In all of these mechanisms, resources consumed by the server on behalf of each client must be accounted and the servicing of client requests must be scheduled in some way. The differences lie solely in the level at which the accounting is performed and scheduling policy applied.

All of the above methods however require that the server is trusted to use the client's resources for performing work on behalf of the client.

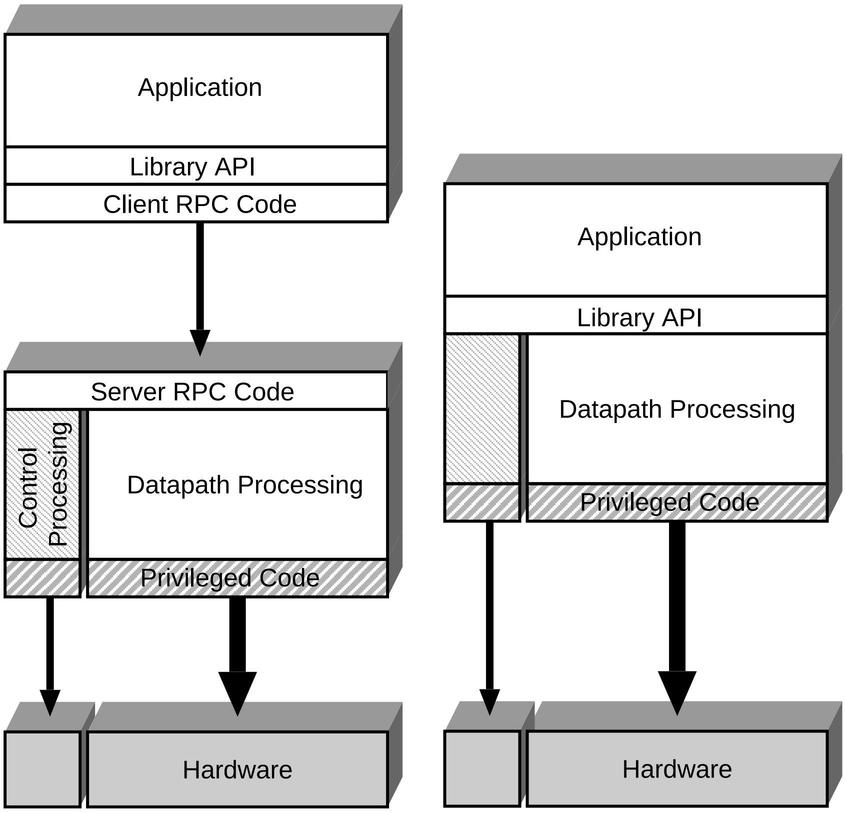
6.3 Client Rendering

In a client rendering system, an application typically has direct access to the framebuffer and is able to perform updates using whatever rendering algorithms are most suitable. It may choose to render a complicated graphic into cached main memory and only make updates to the real framebuffer when all rendering operations have been completed. Alternatively, an incoming video stream may be written directly into the framebuffer avoiding unnecessary copying operations.

This situation has a number of important advantages:

- An application is not restricted to a standard set of rendering primitives.
- No client-server communications overheads are incurred.
- Resources consumed by the client are naturally accounted to the client and thus the problem of QoS crosstalk is largely avoided.

Client rendering has been used in the past by the Cedar system [Swinehart86], early versions of SunView [Sun88], the Amiga [CBM91] and the Apple Macintosh [Apple85]. The majority of these systems were intended as single user machines and so the lack of protection between clients was not seen as a serious problem.



(a) Server Rendering (b) Client Rendering

Figure 6.1: Comparison of Window System Structure

Providing consistent behaviour with respect to window visibility and "decoration" relies on applications using the same library code or following a set of conventions which can not be rigidly enforced. In some systems, a server is retained for performing out-of-band control operations such as window creation/deletion and for demultiplexing of input events. This server can also keep clients informed about the visibility of their windows. However, unless the framebuffer provides some sort of hardware protection there is usually no way to prevent an application from updating pixels it does not own.

Malevolent applications, often executing without the knowledge of the user can easily disregard these conventions, corrupting the output or stealing the input of other clients. An application which deliberately disrupts the user interface can be particularly difficult for a user to kill.

Such window systems are therefore becoming increasingly rare.

6.4 Framestores

Despite the fact that the framebuffer is the eventual destination of the majority of incoming video streams, most multimedia systems fail to recognise the importance of QoS issues in this part of the system. Partly responsible for this fact is the prominence of server-based window systems such as X where the window system server performs most of the functions traditionally provided by a device driver, but for reasons of efficiency and network transparency must present a much higher level interface. When a device driver *is* provided for the framestore, it usually does little more than provide a means to map the framebuffer and video control registers into the address space of the window system server.[2]

6.4.1 PMAG-BA Hardware

The Sandpiper workstations are equipped with Digital PMAG-BA framestores, which plug into one of the three available TURBOchannel option slots. The device provides a memory-mapped 8-bit pseudo-colour framebuffer. Each pixel may be one of 256 colours selected from a single palette. The device does not

	`ldl`	`ldq`	`stl`	`stq`
TURBOchannel	142	143	18	36
Memory System	5	7	1	3

Table 6.1: Average Times for Loads and Stores (in 133MHz cycles)

support DMA of any description and therefore the only method of updating pixels is via a 32-bit TURBOchannel PIO cycle, although the TURBOchannel ASIC on the Sandpiper will automatically convert a 64-bit read or write operation into two consecutive 32-bit PIO cycles. These cycles are expensive in comparison with the equivalent operations when performed to the memory system. Table 6.1 shows the average costs of various relevant operations on the Sandpiper workstation when reading or writing consecutive addresses in the framebuffer and in the memory system.

Although the theoretical peak bandwidth of the TURBOchannel is 720Mbps, the maximum sustained PIO write performance to the framebuffer is only around

[2] On DECStations running Ultrix, this functionality is embarrassingly provided by `/dev/mouse`.

x11perf Test	*SRC*	*DST*	*Diamond Stealth 64*	*Digital PMAG-BA*
copypixwin500	MEM	FB	127 reps/s	78 reps/s
copywinpix500	FB	MEM	26 reps/s	26 reps/s
copypixpix500	MEM	MEM	597 reps/s	288 reps/s

Table 6.2: Performance of TURBOchannel and PCI Framebuffers.

240Mbps.[3] Reading from the framebuffer is extremely expensive since cacheing is disabled in regions of the physical address space used for I/O. This feature is common to a number of interconnects. Table 6.2 shows relevant results from the X benchmark utility `x11perf` for a typical PCI framebuffer and for the PMAG-BA. In both cases the discrepancy between memory speeds and I/O performance is clearly visible.

Devices which do not support any form of DMA present a serious problem for the Nemesis QoS architecture. They introduce a multiple resource scheduling problem since performing I/O to the framebuffer device necessarily consumes CPU time,[4] as well as framebuffer bandwidth. For DMA devices, the I/O scheduler may initiate a transaction for a client process which is not currently executing, decoupling the two resources, but any transaction scheduler for a PIO-only device would have to be closely integrated with the CPU scheduler.

Performing I/O with these devices not only requires privileged code to be executed on the control-path, but also on the data path. Ideally, we would like to be able to execute this code within the protection domain of the appropriate device driver, but within the scheduling and accounting domain of the client. A solution to this problem is presented in section 6.5.

6.5 CallPriv Sections

A simple extension of the Nemesis PALcode provides a mechanism for privileged domains to register small sections of code with the NTSC.[5] An additional unpriv-

[3]This figure corresponds to the `x11perf -shmput500` performance of the DEC Xserver of 76 reps/sec. (4.75×10^6 PIO cycles/sec. = 28 cycles/write)

[4]240Mbps consumes the entire CPU resource.

[5]Currently the **privileged** PALcode call `ntsc_regstub` is used for this purpose.

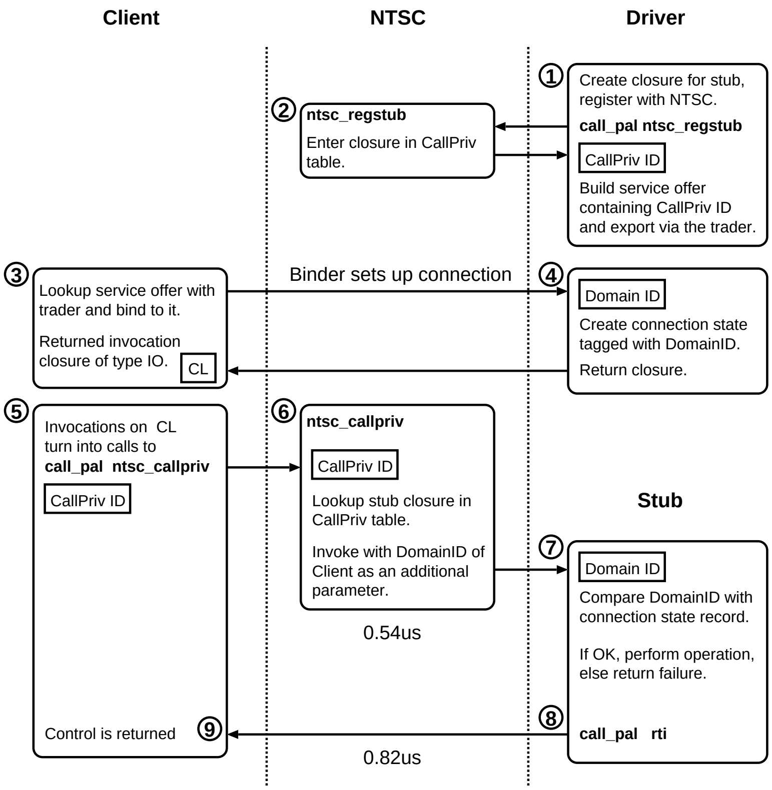


Figure 6.2: Use of CallPriv Sections.

ileged NTSC call[6] allows clients to invoke these sections of code, with arbitrary arguments, within the protection domain of the device driver. The intended use is for performing small, fixed-size PIO operations to dumb devices in situations where the overheads of using *Rbufs* would be excessive in comparison to the cost of the operation.

When control is transferred to the privileged code section, an additional argument is provided by the kernel indicating the ID of the domain which invoked the call (figure 6.2). Given the explicit binding model used in Nemesis, it is impossible to invoke a service, or perform I/O without there being an associated connection. If the connection state, or a closure is passed as one of the user arguments, then a device driver may use the domain ID to immediately verify the arguments of the call.

The code sections are invoked with all interrupts except the lowest level timer interrupt disabled, in a similar manner to interrupt stubs. The reason for this

[6] The **unprivileged** PALcode call `ntsc_callpriv` was added for this purpose

is to prevent a reschedule from occurring during the call and thus simplify the implementation and reduce the overheads of the mechanism. The lowest-level timer continues to run throughout the call and may post a reschedule interrupt, but the kernel scheduler itself will not be entered until the CallPriv completes.

6.5.1 Discussion

It is important to contrast the operation of CallPrivs with thread migration techniques. CallPriv stubs execute within an environment which is very different from user code. They are entered with the minimal number of registers saved and with all interrupts disabled. It is not possible for code within a CallPriv section to block, and it is not possible to invoke another CallPriv section. The device driver is expected to ensure that the duration of the CallPriv is kept to a minimum. The mechanism should rather be thought of as a software version of a channel-controller.

Device drivers are privileged domains and as such must be trusted by the operating system. Incorrect or malicious code within a driver is clearly capable of compromising the system. The execution of this code within CallPriv sections invoked by an unprivileged domain therefore introduces no additional danger.

6.5.2 Performance

As can be seen from figure 6.2, the cost of invoking a CallPriv section has been measured at $0.54\mu s$ and returning to the calling process takes $0.82\mu s$. These overheads are small in comparison with the $100\mu s$ clock granularity of the Sandpiper. For the purposes of comparison, the cost of using *Rbufs* for the same purpose have been measured at $5.2\mu s$ for the `IO$PutPkt` operation and $5.4\mu s$ for the `IO$GetPkt` operation.[7]

Time spent within a CallPriv section therefore introduces some amount of jitter into the kernel scheduler. Device drivers, as always, are expected to minimise the time spent with interrupts disabled. For all devices considered so far, it has been possible to devise a simple and useful atomic I/O operation which executes in a time comparable with the granularity of the system clock.

[7] These are the costs when pipelining is working correctly and no reschedules occur.

The driver for the PMAG-BA framestore described in section 6.6 uses the update of a small rectangular tile of pixels as its I/O primitive for this reason.

In order to amortise the overheads of the protection domain switch still further, some drivers may allow clients to request fairly long transactions which may be broken into smaller units if they would take too long to complete. The remainder of the transaction may be completed using a second CallPriv.

6.6 DAM: The Framebuffer Driver

The Nemesis framebuffer driver (`dev/fb`) implements the DAM for the PMAG-BA device. As such it is responsible for providing protected access to the pixels of the framebuffer with guaranteed rates of I/O to each of its clients. As with all other devices, the operation of these protection and scheduling mechanisms is directed by the DMM and the driver exports the `FB.if` interface for this purpose (figure 6.3). This control interface is used exclusively by the DMM to:

- Create, destroy, move and resize *windows*.
- Create *update streams* for a window (using the `IO.if` interface).
- Change the update permissions of areas of the framebuffer.

Since the framebuffer is the primary video output device of a multimedia workstation, the driver is optimised for the display of high bandwidth digital video streams. The driver is stream oriented in that the only way to update the pixels in the framebuffer is to send a packet on a pre-established connection to the driver.

Clients communicate window updates directly to the driver using one of a number of simple packet based protocols. The driver however supports a **single** primitive: the update of a small fixed-size rectangular region know as a *tile*.[8] Packets contain an (x, y) coordinate and a rectangular region of pixels to replace the pixels in the framebuffer. Even traditional graphics rendering is performed by sending a video stream to the framestore. In order to minimise bandwidth, this stream may consist only of the changes since the last time the window was repainted.

[8] A recent experimental version of $8\frac{1}{2}$, the Plan 9 Window System [Pike91] has also adopted tiles as the single update primitive [Pike94].

```
FB : LOCAL INTERFACE =
  NEEDS IDCOffer;
  NEEDS Time;
BEGIN

  BadWindow : EXCEPTION [];
  Failure   : EXCEPTION [];
  Unsupported : EXCEPTION [];
  NoResources : EXCEPTION [];

  WindowID : TYPE = LONG CARDINAL;
  StreamID : TYPE = LONG CARDINAL;
  Protocol : TYPE = { Bitmap, AVA, DFS };
  QoS      : TYPE = RECORD [ tiles : CARDINAL,
                            period : Time.ns ];

  CreateWindow : PROC [          x, y : INTEGER,
                        width, height : CARDINAL,
                                 clip : BOOLEAN ]
                 RETURNS [ wid : WindowID ]
                 RAISES Failure;

  -- "CreateWindow" creates a window with the specified
  -- position and size.  If "clip" is "False" then updates
  -- to the window do not take account of the clip mask.
  -- A "WindowID" is returned.

  DestroyWindow : PROC    [ wid : WindowID ]
                  RETURNS [ ]
                  RAISES BadWindow;

  -- "DestroyWindow" frees resources allocated to window "wid".

  UpdateStream : PROC [ wid : WindowID,
                          p : Protocol,
                          q : QoS,
                       clip : BOOLEAN ]
                 RETURNS [ s : StreamID,
                       offer : IREF IDCOffer ]
                 RAISES BadWindow, Failure, Unsupported;

  -- "UpdateStream" returns an IDCOffer for a video
  -- updates stream using protocol "p".  If "clip" is
  -- False then updates to the window do not take account
  -- of the clip mask.

  MapWindow : PROC [ wid : WindowID ]
              RETURNS [ ]
              RAISES BadWindow;

  -- "MapWindow" causes the window "wid" to become mapped
  -- on the framestore device.  Updates to the window
  -- become possible.

  UnMapWindow : PROC [ wid : WindowID ]
                RETURNS [ ]
                RAISES BadWindow;

  -- "UnmapWindow" causes the window "wid" to become
  -- unmapped on the framestore device.  Updates to the
  -- window will be silently discarded.

  ExposeWindow : PROC [         wid : WindowID,
                               x, y : CARDINAL,
                      width, height : CARDINAL ]
                 RETURNS [ ]
                 RAISES BadWindow;

  -- "ExposeWindow" causes window "wid" to become visible
  -- in the rectangle described.  (Coordinates are frame
  -- buffer coordinates.)

  MoveWindow : PROC [  wid : WindowID,
                      x, y : INTEGER ]
               RETURNS [ ]
               RAISES BadWindow;

  -- "MoveWindow" ssks for the window "wid" to be moved to
  -- position $("x", "y")$.

  ResizeWindow : PROC [          wid : WindowID,
                       width, height : CARDINAL ]
                 RETURNS [ ]
                 RAISES BadWindow, Failure;

  -- "Resize" asks for the window "wid" to be resized to
  -- $"width" \cross "height"$.

  AdjustQoS : PROC [ sid : StreamID,
                       q : QoS ]
              RETURNS [ ]
              RAISES NoResources;

  -- Attempts to set the QoS for stream "sid" to "q".

END.
```

Figure 6.3: MIDDL for `dev/fb` management interface (`FB.if`).

Update streams provide a direct data path between the owner of a window and the framebuffer driver. Each update stream is associated with a single window - the per-stream state record identifies the window and also contains QoS parameters and protocol specific information. In most cases update streams use *Rbufs* [Black94] as a highly efficient packet based shared-memory transport, although other experimental transports have been used including the CALLPRIV mechanism described in section 6.5.

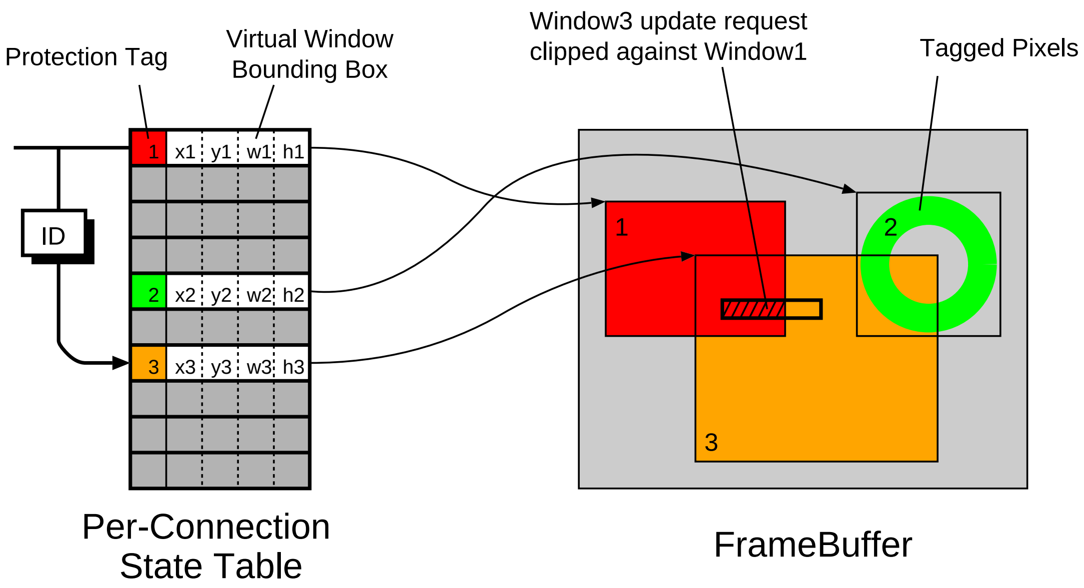


Figure 6.4: Virtual Windows and Key Based Protection

6.6.1 Virtual Windows

The framebuffer driver will usually have a large number of update streams connected to it. When I/O requests arrive at the driver, the connection identifier is used to index into a table of per-stream state records. The entry in this table describes a rectangular region of the screen known as a *virtual window*. The (x, y) coordinates of each update request are translated by the on-screen coordinates of the virtual window allowing rendering code and hardware video capture devices to operate independently of the position of the destination window. Any update requests which lie outside the virtual window are silently discarded. This *translation* mechanism also allows a single video stream to be multicast to a number of destinations without CPU intervention on the data-path.

6.6.2 Key Based Pixel Protection

For each pixel in the framebuffer, an additional *tag* field is stored in a separate bank of memory. A similar tag is held in the per-stream state record. Reads to or writes from the framebuffer are only permissible for those pixels whose tags match the tag of the current stream.

As above, writes to pixels whose tag value does not match are silently discarded. This situation commonly occurs when a client is rendering to a window which is partially obscured. Reads from pixels whose tag value does not match return undefined data.

6.6.3 Protection Overheads

The overhead of this software protection is not as large as one would expect since the cost of a TURBOchannel programmed I/O read or write operation is so large in comparison with the cost of main memory reads and writes and the necessary arithmetic operations for comparing tags and masking individual pixels.

Since the 21064 CPU of the Sandpiper has a 4-entry write buffer, it is possible to execute a large number of arithmetic operations "in the shadow" of writes to the framebuffer. These may be used to perform the per-pixel clipping operations effectively with zero cost. Also, the majority of rendering operations supported by an X server will frequently need to update only a single 8-bit pixel, but since the TURBOchannel only supports 32 bit reads and writes this will result in two expensive TURBOchannel PIO operations. Rendering in the cached DRAM of the main memory system will often prove significantly faster.

6.6.4 Quality of Service

QoS support is provided at the lowest levels in the device driver i.e. at the level where concrete resources are being consumed. In the case of the PMAG-BA driver these resources consist of both TURBOchannel I/O bandwidth and CPU cycles since the card has no DMA support. A scheduler in the driver determines the order in which to service transactions on the various connections according to the current QoS parameters. When using the *Rbufs* transport, CPU resource expended servicing requests is unavoidably accounted to the driver rather than to the clients. Although the scheduler ensures that the QoS guarantees of each client are respected, it is still necessary for the driver to be provided with enough bulk CPU resource to service all of its clients. This function must be performed by the QoS-manager.

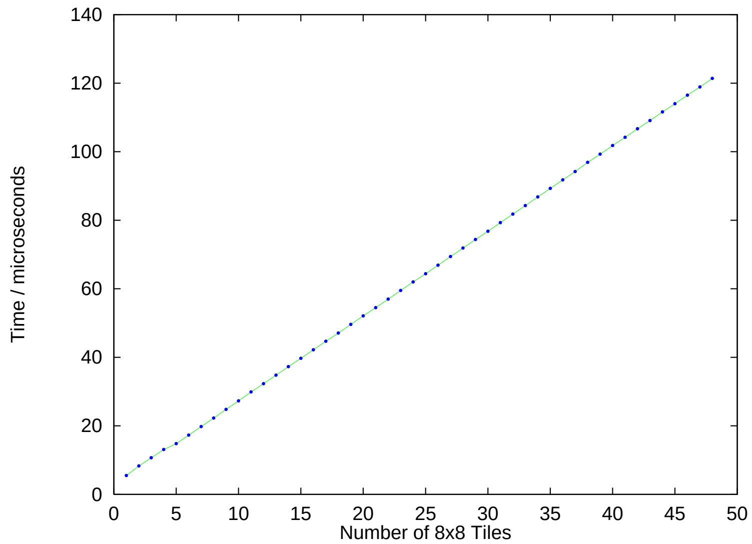


Figure 6.5: Timings of `dev/fb` CALLPRIV Tile Blitting Stub.

When using the CALLPRIV mechanism, this resource transfer problem does not arise. The tile update primitive provided by the driver is ideally suited to execution within a CALLPRIV section - it is simple to check access permissions and the primitive is small enough to avoid significant jitter problems. Figure 6.5 shows timings of the tile blitting CALLPRIV for various numbers of 8×8 tiles. The gradient of the graph corresponds to an achieved bandwidth of 202Mbps to the framebuffer, and the y-intercept shows a total overhead of $3.03\mu s$ per call.[9] For transactions smaller than 48 tiles, the total execution time is less than the granularity of the system clock.

6.6.5 The DAN Framestore (DFS)

The abstraction provided by `dev/fb`, even when using a dumb framestore, is very similar to that provided directly in hardware by the DFS [Pratt95]. In fact, a large fraction of the code in the driver was used as a software emulation of

[9] $1.67\mu s$ is taken to load information from the per-stream state record, the remainder of this figure corresponds to the CALLPRIV overheads measured above.

the DFS written for the DECStation 5000/25 whilst the hardware was being developed.

A framebuffer driver has also been implemented for the DFS connected via the OTTO ATM interface. The level of hardware support provided by the PMAG-BA and the DFS differ substantially but the drivers export the same interface, namely `FB.if`. Naturally, the `dev/fb` driver for the DFS is much simpler and does little more than perform connection setup and provide a higher level mechanism for updating the protection tag RAM.

When using the DFS, `dev/FB` communicates with the framestore using a simple single-cell ATM protocol to create virtual windows and update streams on particular VCIs. The framebuffer driver invokes the OTTO driver to obtain an IDC offer for an AAL5 connection on the appropriate VCI and with the necessary QoS parameters.[10] This offer is handed back to the client which then binds directly to the OTTO driver.

6.7 DMM: The $\mathcal{WS}$ Window System

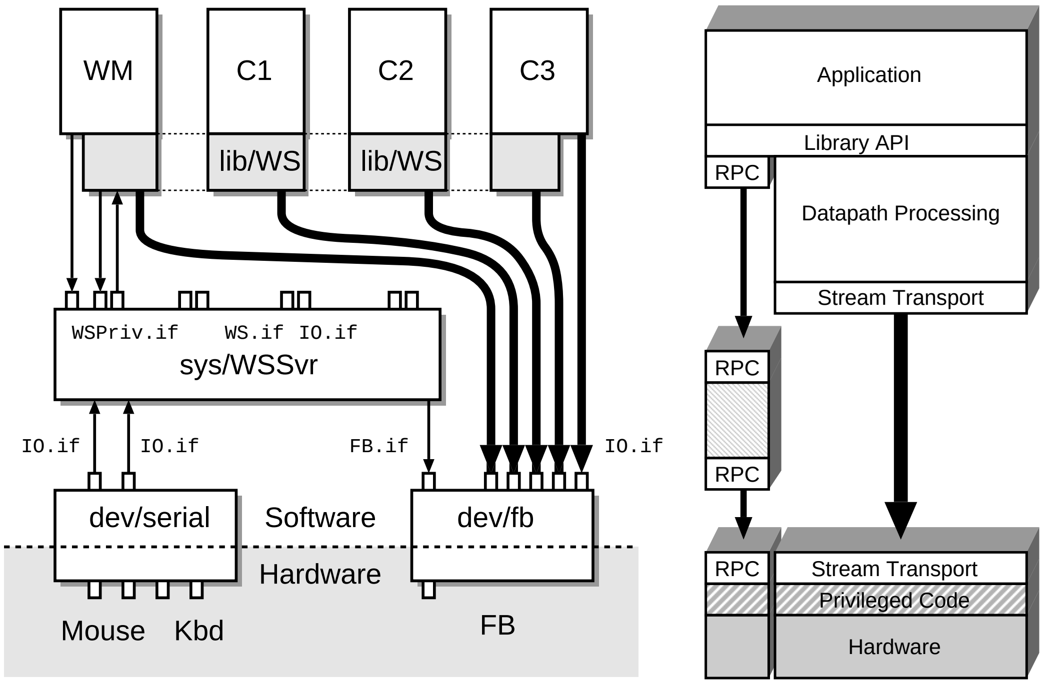
This section describes a prototype Nemesis window system ($\mathcal{WS}$) and a number of simple example applications. The $\mathcal{WS}$ window system is an example of a client-rendered window system. An architectural overview of the system is shown in figure 6.6(a) together with a description of the various Nemesis domains involved.

6.7.1 The $\mathcal{WS}$ Server

The $\mathcal{WS}$ server is a Nemesis domain which manages the mouse, keyboard and framebuffer devices. In the prototype system, the mouse and keyboard drivers are part of `dev/serial` and both export the `IO.if` interface which is used to transport streams of timestamped mouse and keyboard events. The $\mathcal{WS}$ server demultiplexes these events to the appropriate client event stream (another `IO.if` interface) depending on the position of the mouse pointer.

The $\mathcal{WS}$ server is also responsible for out of band control operations such

[10] ATM signalling software has not yet been ported to Nemesis, so for the moment permanent virtual circuits are assumed.



(a) System Overview

(b) Protected Client Rendering

`dev/serial:`	DAM: The serial device driver. This privileged domain provides access to the keyboard and mouse devices.
`dev/fb:`	DAM: The framebuffer device driver described in section 6.6.
`sys/WSSvr:`	DMM: The $\mathcal{WS}$ server.
`WM:`	Window manager domain using an extended interface to the $\mathcal{WS}$ server.
`C1, C2:`	Client domains using the *shared* library `lib/WS` to access the framebuffer.
`C3:`	A client domain using custom rendering code.

Figure 6.6: The $\mathcal{WS}$ Window System

as window creation, deletion and reconfiguration. It exports an RPC control interface of type `WS.if` for this purpose (figure 6.7). After applying suitable argument checking and access control mechanisms, the $\mathcal{WS}$ server translates these requests into invocations on the framebuffer control interface (`FB.if`). Clients are unable to bind to the `FB.if` interface directly.

Note that the $\mathcal{WS}$ server is responsible only for out-of-band control operations which require atomic write access to shared state. It is **not** involved with the

```
WS : LOCAL INTERFACE =
  NEEDS IDCOffer;
  NEEDS Time;
  NEEDS FB;
BEGIN

  Failure : EXCEPTION [];
  BadWindow : EXCEPTION [];

  WindowID  : TYPE = LONG CARDINAL;

  -- A window is an opaque identifier

  Button  : TYPE = { Left, Middle, Right };
  Buttons : TYPE = SET OF Button;
  MouseData : TYPE =  RECORD [ buttons : Buttons,
                                  x, y : INTEGER ];

  KeySym : TYPE = CARDINAL;
  NoData : TYPE = CARDINAL;
  Rectangle : TYPE = RECORD [ x1, y1, x2, y2 : CARDINAL ];

  ExposeData : TYPE = Rectangle;

  EventType : TYPE = {
    Mouse, KeyPress, KeyRelease,
    EnterNotify, LeaveNotify, Expose, Obscure
  };

  EventData : TYPE = CHOICE EventType OF {
    Mouse       => MouseData,
    KeyPress    => KeySym,
    KeyRelease  => KeySym,
    EnterNotify => NoData,
    LeaveNotify => NoData,
    Expose      => Rectangle,
    Obscure     => Rectangle
  };

  Event : TYPE = RECORD [ t : Time.ns,
                          w : WindowID,
                          d : EventData ];

  -- Events are a discriminated union of all possible event types.

  EventStream : PROC [ ]
               RETURNS [ evoffer : IREF IDCOffer ]
               RAISES Failure;

  -- Ask for the WS server to provide an event stream

  CreateWindow : PROC [         x, y : INTEGER,
                       width, height : CARDINAL ]
               RETURNS [ w : WindowID ]
               RAISES Failure;

  -- Create a window with the given position and size.  Returns a
  -- window identifier "w".

  DestroyWindow : PROC [ w : WindowID ]
                RETURNS [ ]
                RAISES BadWindow, Failure;

  UpdateStream : PROC [    w : WindowID,
                           p : FB.Protocol,
                           q : FB.QoS,
                        clip : BOOLEAN ]
               RETURNS [  fbid : FB.StreamID,
                         offer : IREF IDCOffer ]
               RAISES BadWindow, Failure;

  -- Returns an IDC Offer for an update stream for window "w"
  -- using the protocol "p".

  MapWindow : PROC [ w : WindowID ] RETURNS [ ]
            RAISES BadWindow, Failure;

  -- Causes the window "w" to become mapped on the framestore
  -- device. Updates to the window become possible.

  UnMapWindow : PROC [ w : WindowID ] RETURNS [ ]
              RAISES BadWindow, Failure;

  -- Causes the window "w" to become unmapped on the framestore
  -- device.  Updates to the window will be ignored.

  MoveWindow : PROC [    w : WindowID,
                      x, y : INTEGER ]
             RETURNS [ ]
             RAISES BadWindow, Failure;

  ResizeWindow : PROC [             w : WindowID,
                       width, height : CARDINAL ]
               RETURNS [ ]
               RAISES BadWindow, Failure;

END.
```

Figure 6.7: MIDDL for $\mathcal{WS}$ interface (WS.if).

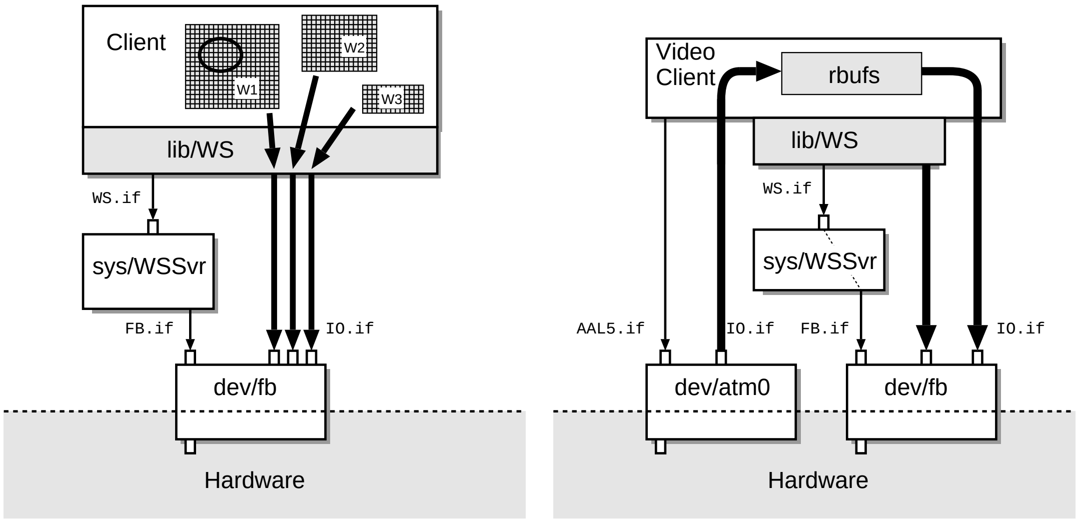
Figure 6.8: $\mathcal{WS}$ Screendump

data path.

6.7.2 Non-Multimedia Applications

In the $\mathcal{WS}$ system, non-multimedia applications *usually* render their graphics into a private copy of the window in main memory and when finished flush their updates to the framebuffer as a stream of tile differences. For many applications which render complicated user interfaces using "painter's algorithm" or similar, the benefits of drawing in fast cached DRAM and copying only the resulting differences to the slower framebuffer can be significant. [Stratford96] describes a client-rendering port of `libX11` which in most cases performs faster than using the server.

Although most applications link against a shared library containing a set of default rendering operations similar to those provided by the X server (figure 6.9a), it is perfectly possible for an application to supply its own customised rendering code.



(a) Conventional Application (b) Video Display Application

Figure 6.9: Example Window System Clients

6.7.3 Multimedia Applications

Multimedia applications may often send tile streams directly to the framebuffer without maintaining a copy of the window in its entirety. Figure 6.9b shows a multimedia application receiving a live video stream and sending it to the framebuffer driver via a second update stream on the same window. This update stream does not necessarily need to use the same protocol as the conventional update stream. In particular, it may choose to use an unreliable transport mechanism (allowing updates to be discarded if insufficient resources are available) and a payload format which closely matches the incoming video stream (e.g. the AVA-200 tiled video format).

6.8 Evaluation of the $\mathcal{WS}$ System

In order to evaluate the benefits of the DAM/DMM approach it was considered necessary to compare the $\mathcal{WS}$ system with a traditional server-rendered window system. Therefore, for the following experiments a server-rendering version of the $\mathcal{WS}$ system was also implemented ($\mathcal{WS}_R$) supporting the same programming interface. As is the case with X, this server implements no internal accounting or

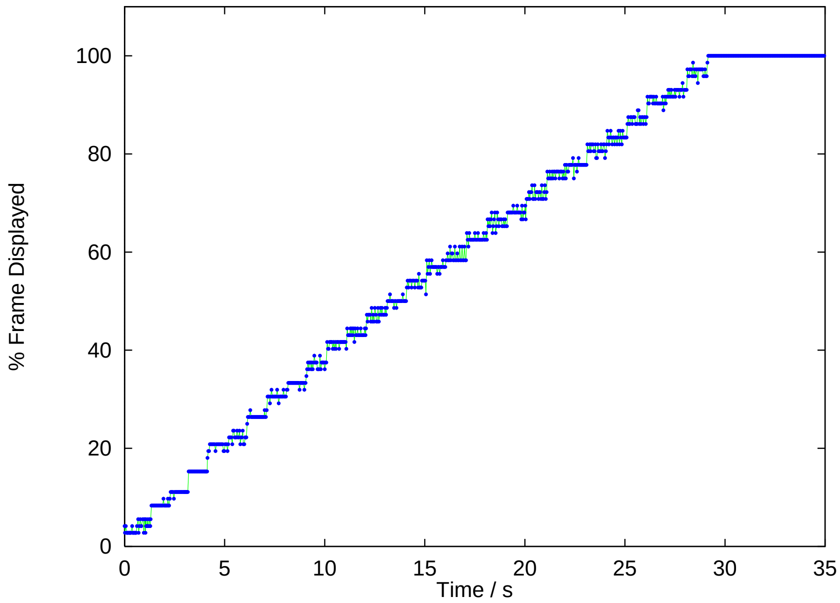


Figure 6.10: Effect of Varying /dev/fb QoS Guarantees

scheduling of client requests and no resource transfer mechanisms. It therefore relies solely on the QoS guarantees of its clients.

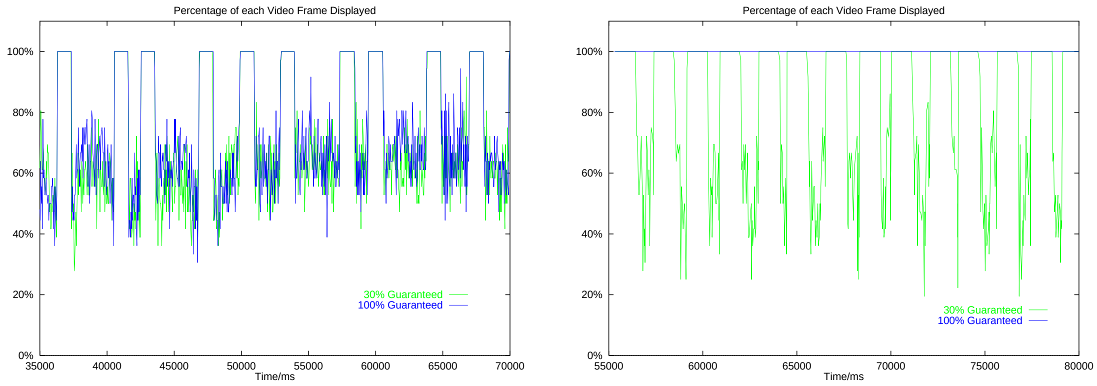
Two important comparisons will now be made.

6.8.1 QoS Guarantees

The ATM video application described in section 6.7.3 was instrumented to record the percentage of each video frame which it was able to render to the framebuffer via the dev/fb driver. The application was provided with an initial QoS guarantee of 1% of the framebuffer bandwidth in each 20ms period. The guarantee was gradually increased by an additional 1% each second as the application progressed. Figure 6.10 shows the QoS observed by the application.

The graph demonstrates that the QoS guarantee provided by /dev/fb allows the application to render an increasing proportion of each video frame until, when provided with 28% of the available bandwidth, the entire frame may comfortably be rendered.

Whilst this simple video application is able to produce results whose quality is in some sense proportional to its bandwidth guarantee, many applications will



(a) Server Rendering ($\mathcal{WS}_R$) (b) Client Rendering ($\mathcal{WS}$)

Figure 6.11: QoS Crosstalk Between Window System Clients

only have a small number of "sensible" modes of operation. These applications may not be able to produce acceptable results *at all* unless they know in advance the QoS which they will receive.

6.8.2 QoS Crosstalk

In this experiment three clients of the window system compete for resources. Two of the clients are video display applications receiving almost identical (but deliberately non-synchronised) video streams from separate AVA-200s at around 40Mbps. One of these video applications (`V1`) has a QoS guarantee sufficient to display its entire video stream. The other (`V2`) has been guaranteed only 30% as much as `V1`. A third application (`B`) uses the CPU to repeatedly generate bursts of rendered graphics and then sleep for a second. This application has been guaranteed the remaining resources.

Figure 6.11 shows 30 second traces of the percentage of each frame displayed by the two video applications `V1` and `V2` for both the client- and server-rendering versions of the window system.

6.8.2.1 Server Rendering ($\mathcal{WS}_R$)

In this configuration, the $\mathcal{WS}_R$ server is responsible for performing all rendering of video and graphics. When the competing process (`B`) is not running, both video applications are able to display their entire video streams since the $\mathcal{WS}_R$ server has access to the spare resources in the system. When the competing process `B` becomes active, however, the additional workload of the server prevents it from keeping up with the requests from either video application (figure 6.11a).

This QoS crosstalk prevents *either* video application from performing acceptably, irrespective of their individual QoS guarantees.

6.8.2.2 Client Rendering ($\mathcal{WS}$)

When using the client rendering version of the $\mathcal{WS}$ server, all framebuffer updates are sent directly to `dev/fb` where they are accounted to the appropriate client application at the lowest level. Requests from each client are scheduled according to their respective QoS guarantees.

The QoS observed by the video application `V1` (with sufficient guaranteed resources) is unaffected by the activities of competing application `B`. Video application `V2`, however, is forced to discard a large proportion of its data whenever the competing process is running since `dev/fb` no longer has enough spare bandwidth to service its requests (figure 6.11b).

QoS crosstalk between the various clients of the window system has largely been eliminated.

6.9 Summary

It has been common practice for a workstation operating system to leave the job of multiplexing the framebuffer device entirely to the window system. Window systems have traditionally provided a high-level abstraction to the framebuffer device both to reduce the number of interactions required between client and server, and to support remote clients. These high-level primitives are difficult to account and schedule and are therefore undesirable in an environment where Quality of Service is an issue.

In a server-based window system, it is necessary for the server to contain code for every rendering primitive which could conceivably be required by a client. This both constrains the range of applications which may be constructed and leads to "code bloat" in the server. Previous client-rendering window systems do not suffer from this problem, but have included no protection at the device level and are thus prone to abuse by uncooperative clients.

The framebuffer device demands sharing and protection at a fine granularity; protection which is not provided by conventional graphics hardware. A low-level software protection mechanism has been presented which has a negligible performance overhead and is readily implemented in hardware at a minimal cost.

Most framebuffer devices support no form of DMA, instead requiring the processor to be used to move data. This code must be executed within the protection domain of the device driver but the processor resource consumed should ideally be accounted to the client. The CallPriv mechanism effectively supports devices of this nature.

The above two techniques have been combined to produce a Device Abstraction Module for the framebuffer device which both provides fine-grained protected access to the framebuffer and accounts for resource usage at the lowest possible levels. A new client-rendering window system has been presented which makes use of this low-level device driver to allow migration of rendering code into the application where it gains the benefit of QoS guarantees provided by the framebuffer driver and minimises application QoS crosstalk.

Chapter 7

File System

The storage capacities of modern disk drives are starting to approach levels comparable to the typical size of an MPEG compressed feature-film. The read and write performance of drives has also increased to a level in excess of that needed to handle real-time video streams. For these reasons, disks are increasingly being required to act as sources or sinks of multimedia data.

This chapter considers the application of the Nemesis device driver architecture to a standard SCSI disk drive and presents mechanisms for abstracting the device which effectively support the implementation of a wide variety of file systems and potentially an application-specific virtual memory system.

7.1 Introduction

A number of operating systems provide an environment where the file system is used for inter-process communication. For example, the UNIX environment encourages the composition of a number of simple programs to perform more complicated tasks, and although it is often possible to use "pipes" between processes, many applications require the use of temporary files. In order to achieve acceptable performance, write-buffering and cacheing in the file system code attempts to prevent this data being written to disk. This additional code is a significant disadvantage when dealing with high volume CM data.

The majority of information used to direct the course of file system research is derived from one or two low-level traces obtained from instrumented UNIX file systems [Ousterhout85][Baker91].[1] As expected, these traces show a disproportionate number of short-lived files written sequentially in their entirety, read once in their entirety, and then deleted. This observation has led file system designers to optimise their designs for such behaviour. It is argued that in an operating system such as Nemesis, this form of inter-process communication should not be necessary.

7.2 General Purpose File Systems

General purpose workstation file systems can be categorised into 3 broad categories based on the manner in which they use the underlying physical storage devices:

- Block-Structured File Systems
- Log-Structured File Systems
- Extent-Based File Systems

Although the functionality provided by all 3 storage schemes is similar, they are optimised for different usage patterns.

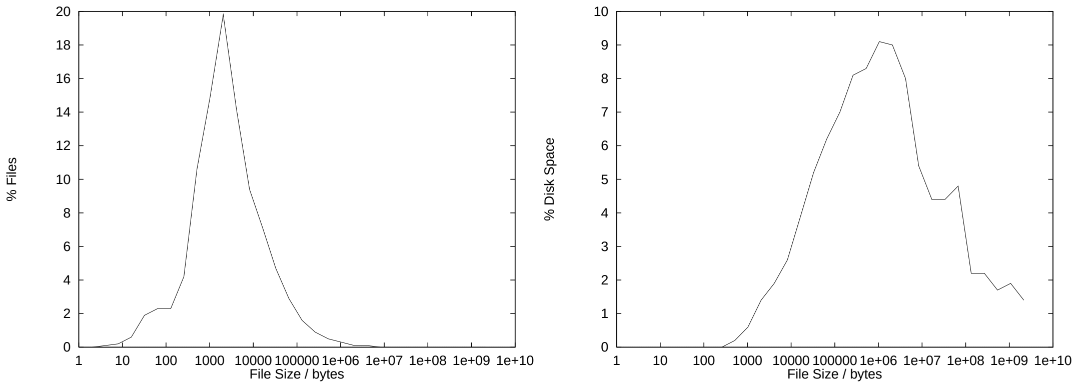


Figure 7.1: UNIX File Size Statistics

[1] The traces described in Mary Baker et al.'s 1991 SOSP paper are available on-line as `http://now.CS.Berkeley.EDU/Xfs/SpriteTraces/`

7.2.1 Block-Structured File Systems

Block-structured file systems divide the surface of the disk into a large number of equal sized *blocks*. This is the unit of disk storage allocation and also the unit of disk I/O. The size of a block is a compromise between the amount of disk space wasted per-file and the I/O transaction overheads. Block structured file systems are the most common variety of file system and until recently have been used by most varieties of UNIX.

Block structured file systems are designed primarily to achieve high space utilisation; hence the small unit of disk allocation. Since half a disk block, on average, is wasted per file there is a strong motivation to keep the disk block size fairly small[2] if the file system is expected to contain a large number of small files. The underlying assumptions seems to be that disk space is an expensive resource and this is currently not the case, although the cost of managing large volumes of disk space should not be ignored.

Figure 7.1 shows the distribution of file sizes, both in terms of number of files of a particular size and in terms of the proportion of disk space consumed by files of various sizes. The graphs are based on a survey of UNIX file size data for 12 million files, residing on 1000 separate file systems, with a total size of 250 gigabytes.[3]. The first graph shows that the majority of files are small - 90% of files are less than 16KB in length - whilst the second shows that the majority of disk space is occupied by files over 1MB in length.

Another reason for the popularity of block structured file systems is that they are easy to integrate with virtual memory systems and buffer caching schemes due to the fixed-sized unit of I/O. It is often arranged that the block-size supported by the file system is the same as the page-size supported by the virtual-memory system.

Block allocation is necessarily a frequent operation and requires update of shared state. In a microkernel environment this incurs the cost of communication with the file-system server. When writing high volume data to disk, as is the case with CMfiles, it would be preferable to be able to perform disk space allocation in larger units.

[2]Most UNIX file systems use a block size of 4096 or 8192 bytes.

[3]These results come from a number of traces which were co-ordinated and analysed by Gordon Irlam <gordoni@home.base.com> Further information may be obtained on the World Wide Web at http://www.base.com/gordoni/ufs93.html

Another drawback of block-structured file-systems is that, since all disk I/O requests are single blocks, the disk seek overheads are extremely large in comparison with the time taken to transfer the actual data.[4] This leads to very poor performance if the pattern of disk I/O requests is essentially random. Even when using disk-head scheduling techniques in an attempt to reduce the seek overheads, disk utilization figures of around 7% are reported in [Gopal86]. By using very large write buffers capable of dealing with I/O queue lengths of around 1000 transactions it is possible to increase the write utilisation of the disk to around 25% [Seltzer90]. Scheduling algorithms which take into account rotational latency such as the `ASATF`[5] algorithm described in [Jacobson91] achieve utilisations of up to 32%[6] but suffer from response times of about a second at that load. The `ASATF` algorithm even required an explicit special case to prevent infinite service times!

The vast majority of work in the field of disk and file-system performance is devoted to increasing total throughput and/or decreasing average response times. The large variance in service times caused by several head scheduling algorithms is usually considered as of secondary importance. The multimedia environment often requires that total throughput of a system is sacrificed in order to maintain predictable levels of performance on each individual connection.

7.2.2 Log-Structured File Systems

As a result of analysing large traces of UNIX file system activity, the observations were made that the majority of files are written exactly once in their entirety, and that most files are fairly short-lived. By implementing a file-system where the only disk write operation permitted is to append to a log, a large fraction of the seek overheads of a block-structured file-system may be eliminated [Ousterhout89].

The addition of a cache allows most reads to be serviced from memory and in addition means than most short-lived temporary files never need to hit the disk. The log is periodically compacted to remove old and deleted file data by a background process similar to a garbage-collector.

[4]The time taken to transfer an 8KB block across a 5MHz SCSI bus is 1.6ms, whilst the typical access time (comprised of seek time and rotational latency) is around 15ms.

[5]Aged Shortest Access Time First

[6]These results were acheived using a software simulator and an unrealistic workload generated using exponential inter-arrival times and a uniform random distribution of requests across the disk surface.

Although log-structured file systems allow disk write throughput to approach the maximum transfer rate of the device, they require that all file updates are performed by a single server. This potentially introduces large amounts of QoS crosstalk. Read performance depends on the amount of fragmentation in the log and the number of concurrent read requests. Attempting to read a number of continuous-media files simultaneously will still incur seek penalties.

Examples of log-structured file systems include Sprite LFS [Rosenblum92], BSD-LFS [Seltzer92] and the Huygens File Server developed as part of the Pegasus project [Bosch93].

7.2.3 Extent-Based File Systems

Extent-based file systems are a compromise between the predictability of block-structured file systems and the throughput achievable using a log-structured file system. Disk space is allocated as contiguous ranges of blocks called *extents*. The number of blocks in an extent is variable and typically dependent on the expected size of the file.

Allocating disk space in this way cuts down the frequency of operations requiring access to shared state and allows disk throughput to be increased by use of larger transactions. It also results in less disk fragmentation and therefore an increased likelihood of consecutive reads and writes. In the case where all extents are of length one, the data-path performance can be expected to converge with that of a block-structured file system. In the majority of cases, however, a significant performance increase should be observed.

File creation and deletion is potentially more expensive than in a block-structured file system, but this can be amortised using schemes similar to the Partitioned Datasets of MVS [IBM80] where a number of files owned by the same user are grouped together into a single dataset stored in preallocated extents on disk.

Extent-based file systems are becoming popular in situations where time constraints are important and the unpredictability of a log-structured file system is unacceptable. Examples include the QNX file system [Hildebrand92] and CMFS [Jardetzky92].

7.3 Multimedia File Systems

In order to more effectively support multimedia file types, a number of researchers have abandoned conventional file system technology altogether in favour of a dedicated multimedia file system optimised for large files. Indeed such file servers are typically constructed as an embedded hard-real time system running on stand-alone machines connected to a high speed network and so do not experience contention from other activities on the workstation.

A multimedia file server may make the assumptions that there will only be a small number of files open at any particular time and that clients will perform predominantly sequential accesses. The abstraction supported is often closer to that of a VCR with `play`, `pause` and `rewind` operations [Jardetzky92] implying that demands for I/O bandwidth are constant for prolonged periods of time. With these simplified file access patterns it is possible to provide stream-oriented I/O at a predetermined rate purely by use of large amounts of read-ahead [Reddy94]. Unfortunately this abstraction is completely unsuited to conventional general-purpose file access. Random access to files is usually very slow and often not allowed at all. Where it is allowed, it often causes transient QoS problems for other streams.

In a multimedia file system, disk layout is optimised for very large files. It is not uncommon to allocate disk space, and indeed perform I/O in units related to the physical geometry of the drive, e.g. a cylinder at a time. Often the file system is assumed to be essentially read-only and its layout optimised off-line. For example, in a VOD system serving a number of streams from a single disk it is possible to achieve more predictable average-cost seek times by deliberately striping file data across the disk.

The Huygens file server [Bosch93] and CMSS [Lougher93] use a log-structured file system as the underlying storage service. Although a log-structured file system is ideally suited to simultaneous recording of a number of continuous media streams with differing QoS guarantees, simultaneous playback of these streams suffers unavoidable QoS crosstalk due to the interleaving of file data on the disk surface and the impossibility of caching large continuous media files. In both of the above systems, prefetching and request scheduling are used to provide playback of a small number of streams at predetermined rates. Knowledge of the internal structure of files is built into the file server to allow files to be "played back" at the "correct rate", and it is not uncommon to support only a predefined set of file *types*. The log-structure of file system provides no significant advantage over extent-based layouts for this application.

The Multi-Service Storage Architecture (MSSA) [Lo93] separates the functions of data storage and the provision of higher-level file abstractions in a two-level architecture. At the lowest level, Byte-Segment Custodes (BSC) provide rate-based access to persistent data, stored in an extent-based fashion, using a notion of *sessions* with QoS guarantees. A number of higher level services are provided including directory services and support for continuous media and structured typed data. The architecture is primarily intended for a network file server, but many of its features are equally applicable to this work.

7.4 Custom Storage Systems

There are a number of applications such as persistent programming languages such as PS-ALGOL [Atkinson83] or Napier88 [Morrison89], and Database Management Systems (DBMS) [Date90] whose performance is highly dependent on disk I/O, and which have disk access patterns which differ significantly from conventional or multimedia file access patterns. A true multi-service operating system should equally well be able to support applications of this form.

It is common for DBMS software running over a conventional general-purpose operating system to use a "raw" disk interface, bypassing the file system layer altogether. The DBMS is allowed to use an entire disk partition in whatever manner it desires, and using application specific knowledge is able to do a much better job of scheduling its disk accesses. It would be highly desirable to be able to make these application specific optimisations without having to bypass the file system completely.

7.5 Disk Drives

The technology used in modern disk drives is fairly uniform across vendors. Data is stored magnetically in concentric *tracks* on a number of rotating *surfaces*, each with its own read/write *head*. The heads are mounted on a movable arm which may be used to place them over the selected tracks (the group of tracks which are simultaneously accessible are usually referred to as a *cylinder*). Although drives

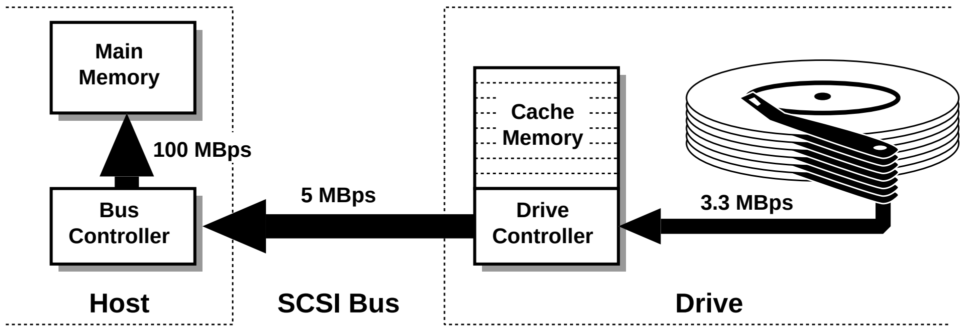


Figure 7.2: Major Disk I/O Data-path Components

with multiple arms are available, the majority only have a single arm.

After moving a head to a new track it is necessary to make fine adjustments to the position to ensure that the head is *settled* over the centre of the track and data is transferred reliably. More expensive drives use error correction logic to enable reads to be serviced before the head is completely settled, and a drive controller will often begin reading from a track before the required data has come around assuming that the entire track will be of interest.

Each track is divided into a number of sectors, but since the length of a track depends on the distance from the centre of the disk, it is common for large capacity drives to divide each surface into a number of *zones* with different numbers of sectors per track. The data transfer rate for a transaction may vary by up to a factor of two from the centre of the disk to the outside.

Due to imperfections in the surfaces of the disk it is also common to manufacture a drive with more tracks than necessary. Track numbers are then remapped by the drive controller to avoid damaged regions - a procedure known as *track sparing*. Individual bad sectors may also be remapped, either to a different sector on the same cylinder, or to a reserved cylinder elsewhere on the disk surface. Disk I/O is performed in units of *blocks* which appear to be arranged as a contiguous array, but since the physical location of a block may be impossible to determine, the penalties incurred when accessing these sectors can not always be predicted.

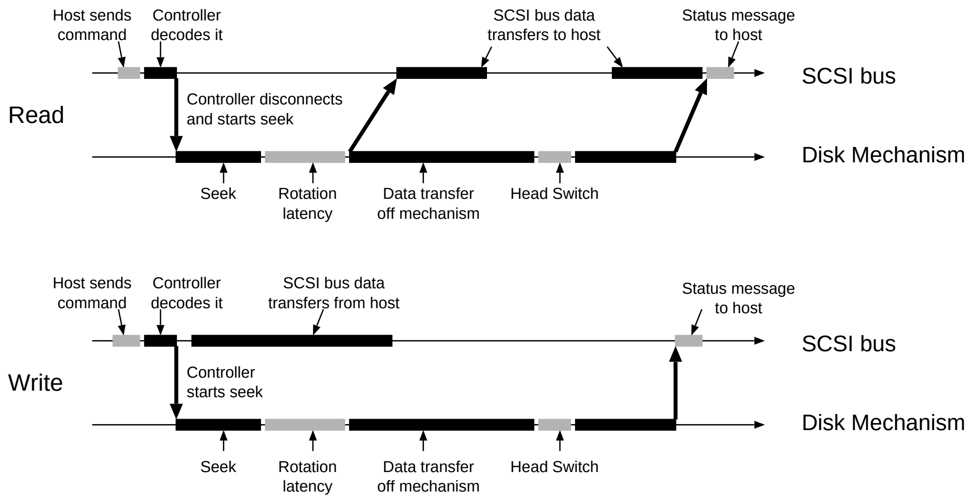


Figure 7.3: Typical Activity During SCSI Transactions

7.5.1 Disk I/O Data-path

The drive is connected to the host using one of the standard peripheral buses such as SCSI. Figure 7.2 shows the major components on the disk I/O data-path. Both the host and the drive typically contain a standard SCSI controller chip which deals with the low-level protocols required to access the bus. Although data rates from the disk heads are usually less than the bus bandwidth, cache memory inside the drive makes it possible for the device to use the bus at full speed for short periods. Given the relative data rates, it is rare to connect more than two disk drives to the same SCSI bus. The host controller is usually attached to the workstation I/O interconnect which provides more than enough bandwidth into main memory of the workstation.

Since the peak media transfer rate is typically substantially lower than the bus transfer rate, the drive controller will often disconnect from the SCSI bus during a long read until sufficient data has been read into internal buffer memory. The controller will also usually perform some amount of read-ahead and write-behind. Figure 7.3 shows typical activity on the SCSI bus and in the drive controller during read and write transactions.[7]

[7] This figure is taken directly from [Ruemmler94].

Average Seek Time:	9.5 ms	A weighted average of the time taken to move the disk arm from one part of the surface to another.
Average Access Time:	15.1 ms	A weighted average of the combined time taken to move the disk arm to the correct position, the time taken for the head to settle plus the rotational latency.
Rotation Speed:	5400 rpm	The rate of rotation of the disk platter.
Media Transfer Rate:	3.3 MBps	The speed at which data may be read from the surface of the disk into the controller's buffer memory. This is usually proportional to the rotation speed of the disk.
Bus Transfer Rate:	10 MBps	The speed at which data may be transferred from the disk controller's buffer memory to the host machine. This usually depends on the type of bus to which the device is connected.
Buffer Size:	512 KB	The amount of buffer memory in the disk controller.

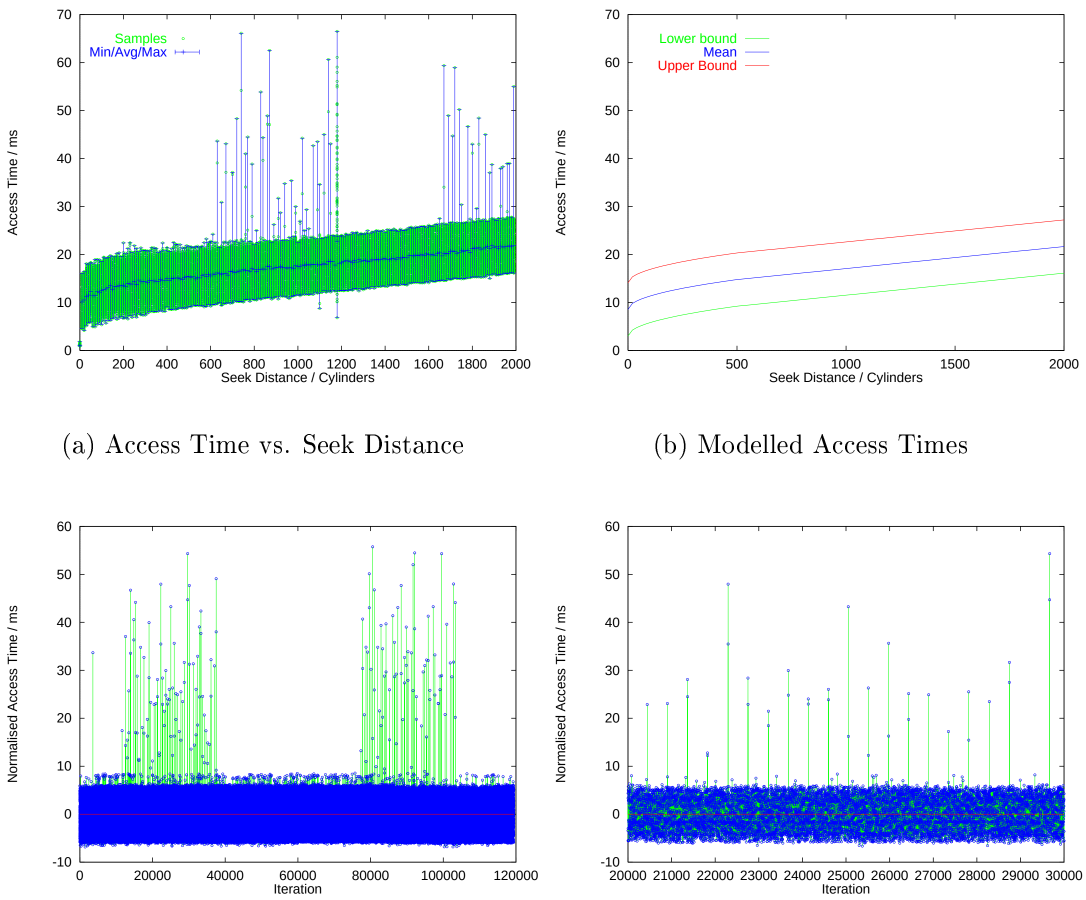
Table 7.1: Published Specifications of the Digital RZ26 Disk Drive

7.5.2 Performance Characterisation

The performance of disk drives is usually characterised by a handful of parameters related to the mechanical timings of the device. The published specifications of the Digital RZ26 drive used in the Sandpiper are shown in table 7.1. Although these figures are typically the only information provided with a drive, they are a dramatic over-simplification of the actual behaviour of the device. The average seek and access times given in the drive specifications are sometimes of little use when trying to estimate the cost of a transaction since drive controllers attempt to optimise performance for certain common access patterns and when these optimisations fail the results are costly.

7.5.3 Access Time Variations

Figure 7.4(a) shows a graph of the measured times to access a single block at various distances across the surface of the disk. Measurements were obtained by first reading a single reference block at the start position on the disk surface, then waiting a random about of time (to remove any correlation with disk rotation) and measuring the time taken to read the destination block. Results were averaged over 100 measurements for each seek distance starting at different positions on



(a) Access Time vs. Seek Distance

(b) Modelled Access Times

(c) Variation of 100 Cyl. Access Times

(d) Detail of 100 Cyl. Access Times

Figure 7.4: Access Times for Digital RZ26 Disk Drive

the disk surface.

The graph has several interesting features. Firstly, the average access times are very stable and easily modelled using a constant transfer overhead T_t, a $\sqrt{d}$ component for short seeks (accelerating the head), an additional linear component for longer seeks ("coasting") and the rotation time T_r as shown in equation 7.1. Lower and upper bounds for access times can be obtained by adding or subtracting $\frac{1}{2}T_r$ as in figure 7.4(b).

$$\overline{A}(d) = \begin{cases} T_t & \text{if } d = 0, \\ K_a\sqrt{d} + \frac{1}{2}T_r + T_t & \text{if } d \leqslant 500, \\ K_a\sqrt{500} + K_c(d-500) + \frac{1}{2}T_r + T_t & \text{if } d > 500. \end{cases} \tag{7.1}$$

Secondly, the effects of disk rotational latency are clearly visible as a wide band and are of comparable magnitude to the seek time. Unless disk rotation is taken into account, the variation of access times will clearly present a problem. In addition, a number of unavoidable limitations of the hardware, deliberately hidden by the device abstraction, can introduce essentially "random" variables. For example, if the disk's forward-error-correction (FEC) circuitry (intended to allow reading of data before the head is completely settled) does not succeed, it is often preferable to try a different transaction rather than wait an entire rotation time for the desired block to come around again.

Thirdly, during the course of the measurements, the drive regularly entered phases where approximately 0.5% of the accesses took a disproportionately long time.[8]

Figure 7.4(c) shows the results of another experiment designed to investigate the cause of these anomalous measurements. A single-block access requiring a seek of a fixed distance of 100 cylinders was performed repeatedly over a 3 hour period. The graphs show the experimentally measured access times normalised by subtracting the mean access time of 12.8ms for ease of plotting. The results show that the drive is changing its behaviour over quite a coarse time-scale. The anomalous measurements appear to be regularly spaced with a period of around 30 seconds, presumably due to some internal background processing within the drive. Many papers on disk scheduling and modelling mention discrepancies of this magnitude but usually attribute the results to the vagueries of the UNIX scheduler. Whilst measurements of this kind are often difficult in the UNIX environment, it is readily apparent that these are actually features of the drive's behaviour which are impossible to take into account in a model.

Another complicating factor is that the internal buffer memory in the drive is not simply used as a FIFO. Drive controllers will typically partition the buffer space so as to cache around 8 regions of the disk surface. Models of disk

[8]The concentration of anomalous measurements at around 1200 cylinders are believed to be due to drive thermal recalibration.

drives are usually highly inaccurate unless they take into account these factors [Ruemmler94]. A detailed model of one particular disk drive is described in [Kotz94] which manages to estimate the cost of *most* disk transactions to within 1%. The model is based around an event-driven simulator, required intimate knowledge of the internals of the particular drive and takes over 12,000 lines of code. Techniques for on-line extraction of the important parameters governing disk performance are described in [Worthington94b], although the resulting models still require cache contents to be continuously tracked and the prefetching behaviour of the drive to be simulated.

Due to the features described above, computing an estimate of the cost of a disk transaction is a complicated process. The total cost is derived from a large number of factors, the most significant of which depend on the pattern of previous accesses and invisible state within the drive [Worthington94a]. Although it would be highly desirable for the microcode inside the drive to schedule transactions according to QoS parameters, it is not feasible to apply hard-real-time scheduling techniques outside the drive itself.

A technique for abstracting a conventional disk drive which supports application specific I/O scheduling policies and QoS guarantees will now be presented.

7.6 DAM: The User-Safe Disk Driver

The User-Safe Disk (USD) device driver provides provides the device abstraction module (DAM) for the disk. The driver (`dev/USD`) is therefore the single multiplexing point for disk I/O and is responsible for implementing all necessary protection between clients. The granularity of protection provided is the *extent* - a contiguous range of blocks on the disk.

The driver exports a privileged control interface (`USDCtl.if`) for each partition of each disk to which a single DMM may bind. The DMM must register a callback interface (`USDCallback.if`) which, amongst other things, provides a *fault-handler* called whenever a client attempts to access a new area of the disk.

The control interface also provides operations for creation and deletion of I/O streams. With each stream is associated a QoS which may be updated via the control interface. In addition, each stream keeps a cache of disk extents which the client may access.

```
USDCtl : LOCAL INTERFACE =
  NEEDS USD;
  NEEDS IDCOffer;
BEGIN

  Failure     : EXCEPTION [];
  NoResources : EXCEPTION [];

  RegisterHandler : PROC [ handler : IREF IDCOffer ]
                         RETURNS [ ];

  -- Used to register an interface of type "USDCallback".

  CreateStream : PROC [ cid : USD.ClientID,
                        m : USD.Mode,
                        q : USD.QoS ]
                 RETURNS [   sid : USD.StreamID,
                           offer : IREF IDCOffer ]
                 RAISES Failure, NoResources;

  -- "cid" is an opaque value which is associated with this stream
  -- and is passed as an argument to the USD fault handler.
  -- "CreateStream" returns an offer for an "IO" channel.

  DestroyStream : PROC [ sid : USD.StreamID ]
                  RETURNS [ ];

  AdjustQoS : PROC [ sid : USD.StreamID,
                     q : USD.QoS ]
              RETURNS [ ]
              RAISES NoResources;

  AddExtent : PROC [ sid : USD.StreamID,
                     e : USD.Extent ]
              RETURNS [ ];

  DeleteExtent : PROC [ sid : USD.StreamID,
                        e : USD.Extent ]
                 RETURNS [ ];

END.
```

(a) USDCtl.if

```
USDCallback : LOCAL INTERFACE =
  NEEDS USD;
BEGIN

  Fault : PROC [ sid     : USD.StreamID,
                 cid     : USD.ClientID,
                 blockno : CARDINAL,
                 nblocks : CARDINAL,
                 OUT e   : USD.Extent ]
          RETURNS [ ok : BOOLEAN ];

END.
```

(b) USDCallback.if

Figure 7.5: MIDDL for dev/USD interfaces (USDCtl.if and USDCallback.if).

Clients perform I/O directly to the disk by sending read or write requests down the *Rbufs* channel. Requests consist of a small record containing the block number and the number of blocks to read or write. In the case of a write, this record is followed by `iorecs` pointing to the data itself. For a read, `iorecs` describing an empty buffer are sent. The USD driver will first check the extent cache for that stream to see if the permissions for that area of the disk are already known. If an entry is not found in the cache, then the DMM is upcalled via the `USDCallback.if` interface. The management interface also allows the extent cache to be explicitly loaded in advance and flushed if necessary.

If the client has suitable permissions for that area of the disk, then the transaction is enqueued with the disk I/O scheduler. If the transaction is not permitted, then the buffer will not be updated in the case of a read and data will simply be discarded in the case of a write. In either case, an acknowledgement is sent to the client containing the length of data successfully read or written.

7.6.1 The `RSCAN` Algorithm

In order to support QoS guarantees on client connections, it is vital that the USD driver schedule disk transactions. Although the exact scheduling algorithm used is largely unimportant, provided that it is capable of delivering the necessary QoS guarantees, excessive disk head seeking can dramatically reduce the aggregate throughput of the device. The `RSCAN` algorithm was a simplistic first attempt at QoS directed disk head scheduling and is a compromise solution which also attempts to minimise the cost of "context-switches".

Research into disk head scheduling algorithms has invariably focussed on increasing the utilisation of the drive, at the expense of predictability. As mentioned in section 7.2.1, the resulting high variance of service times for individual transactions has proved to be a source of problems. The `RSCAN` algorithm employed by the USD scheduler differs from previous disk head scheduling algorithms in that it aims to provide per-client rate guarantees at the possible expense of disk utilisation. The scheduler may even split a long contiguous transfer at a block boundary in order to meet the QoS guarantee of another client.

The disk I/O scheduler maintains a list of pending transactions for all streams sorted by block number. The scheduler attempts to minimise seek overheads by servicing transactions in a manner similar to the `SCAN` algorithm described in [Coffman72]. Due to the impracticality of computing the cost of a disk transaction

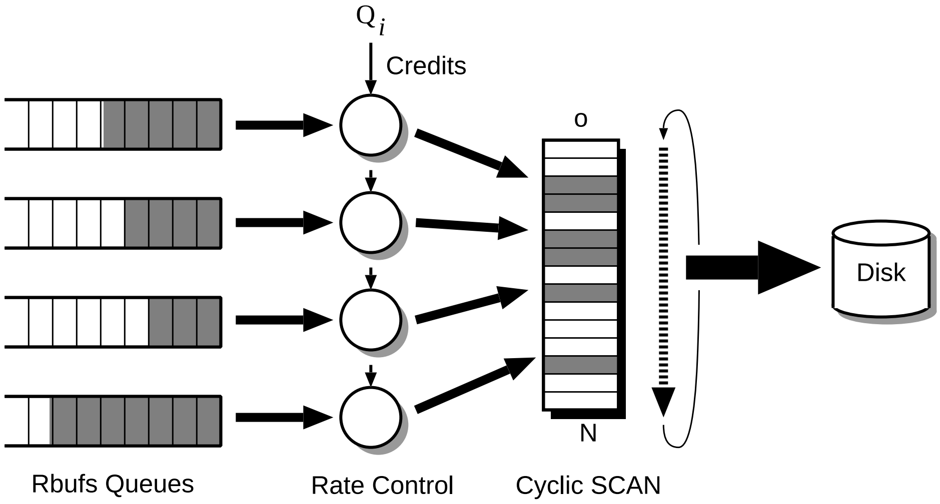


Figure 7.6: RSCAN Scheduling Algorithm

in advance and the disproportionate cost of "context-switches", the scheduler accounts the *actual* cost of each transaction in arrears and uses a credit scheme based on "leaky-buckets" to rate-control each stream. To support best-effort I/O, the scheduler also maintains an estimate of the remaining slack-time in the system and distributes this in the form of additional credits between all clients with pending I/O transactions.

In order to ensure that it is possible to deliver the QoS guarantees of each stream, it is necessary to modify the admission-control policy to take account of the likely seek overheads incurred by multiplexing between the various streams. Although this requires that worst-case seek overheads be assumed,[9] useful guarantees of minimum bandwidth may still be made, and this does not prevent the entire bandwidth of the disk being available in situations where worst-case overheads are not realised. A single client may read or write directly to the disk at the maximum rate supported by the physical drive.

A more sophisticated approach to QoS directed disk-head scheduling could potentially provide each client with a *seek budget* allowing an additional number of seek operations in each period as part of the QoS contract. Any seeks over this budget would necessarily be performed using *best-effort* resources. This would be useful for DBMS style applications with non-sequential access patterns, or for supporting paging in a virtual memory system.

[9] An estimate may be cheaply calculated using the bandwidth guarantee periods and extent sizes of each client.

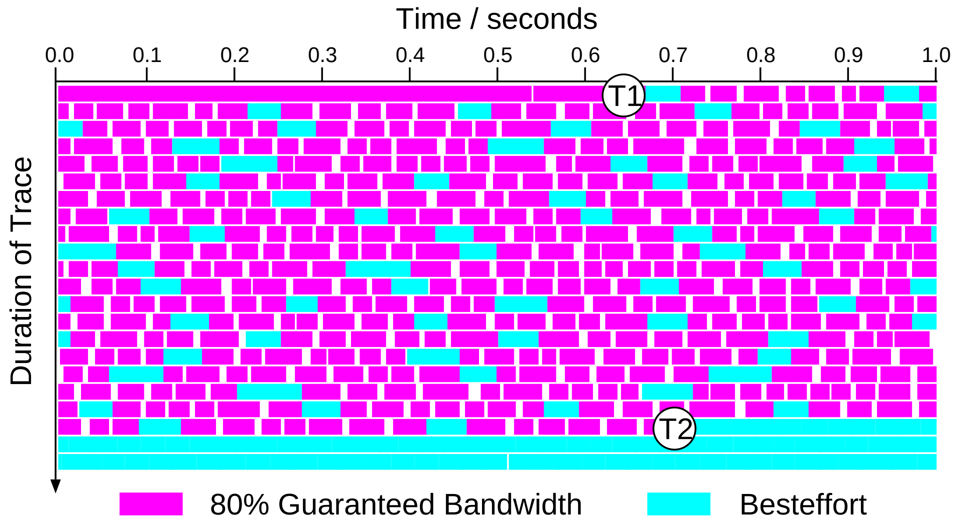
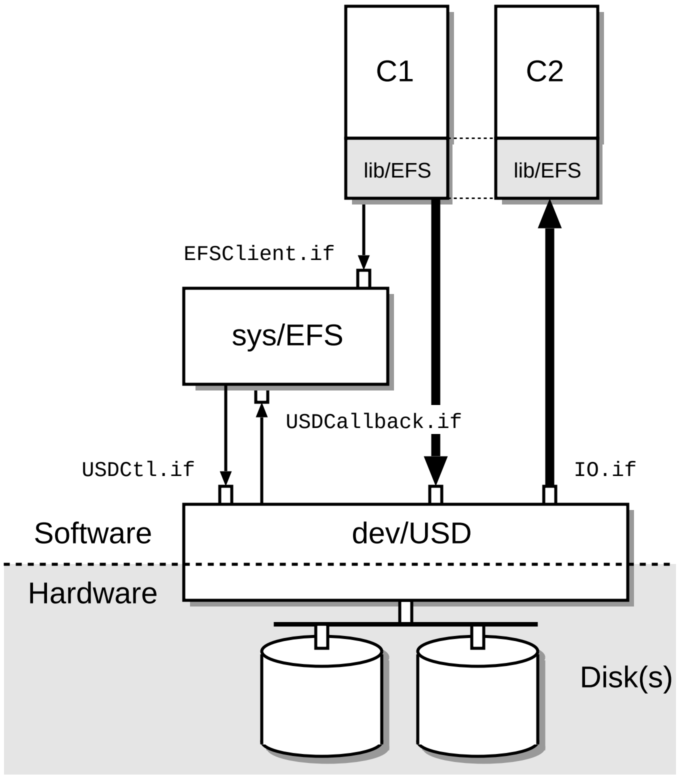


Figure 7.7: RSCAN Scheduler Trace

7.6.2 Evaluation

Figure 7.6.1 shows a 20 second trace obtained from the RSCAN scheduler. The trace shows two clients, C1 and C2, attempting to use the device simultaneously. Client C1 has been guaranteed 80% of the device bandwidth, whilst C2 has no guarantee and is relying entirely on best-effort bandwidth. Initially C1 obtains almost 100% of the disk bandwidth. At time T1, C2 begins making I/O requests and the non-guaranteed bandwidth is shared approximately equally between C1 and C2. At time T2, C1 completes its I/O and the entire bandwidth is available for best-effort clients. The small gaps in the trace are caused by conservative behaviour of the RSCAN algorithm in the presence of both guaranteed bandwidth and best-effort traffic.

Despite the simplistic approach, the USD is able to provide useful QoS guarantees between clients without discarding the protection traditionally provided by file-system code. This benefit is achieved at the expense of some performance. If it were possible to embed a scheduler in the drive itself, or if the SCSI bus were replaced with a more tightly-coupled disk interface (e.g. SSA [Deming95]) then it would perhaps be possible to achieve the same goals whilst sacrificing significantly less in the way of performance.



`dev/USD:`	DAM: The user-safe disk device driver described in section 7.6.
`sys/EFS:`	DMM: The $\mathcal{EFS}$ server.
`C1, C2:`	Two client domains liked against the `lib/EFS` shared library to provide a traditional higher level file system abstraction.

Figure 7.8: The $\mathcal{EFS}$ File System

7.7 DMM: The $\mathcal{EFS}$ File System

In order to make effective use of the USD device it is also necessary to manage both disk space usage and allocation of disk bandwidth. The DMM functions for the disk device are traditionally part of a file system. As a demonstration of the flexibility of the Nemesis I/O architecture is was decided to implement a prototype file system which could equally well support conventional, multimedia and DBMS applications. The $\mathcal{EFS}$ file system was written for this purpose.

$\mathcal{EFS}$ is a simple extent-based file system, but implements only the out-of-band file system functions. Files are comprised of a number of extents which are used to control the extent-based protection provided by the USD. $\mathcal{EFS}$ uses the same disk layout and meta-data format as CMFS [Jardetzky92], but the in-band data-path operations are provided entirely by the USD.

An architectural overview of the system is shown in figure 7.8 together with a brief description of the various Nemesis domains involved. The $\mathcal{EFS}$ server is responsible for maintaining the meta-data associated with each file. This is performed in exactly the same manner as for a traditional file system. The file-system server ensures that it is the only domain permitted to update file meta-data, which is cached in memory in order to reduce the latency of out-of-band operations.[10]

Clients interact with the $\mathcal{EFS}$ using an unprivileged RPC interface which supports all of the out-of-band control operations of a traditional file-system (`EFSClient.if`). This interface is used to create, destroy, open and close files and to read the file meta-data. Files are named using unique identifiers from a single flat name-space, although a directory service may easily be built on top of this name space [Lo93].

Opening a file for read or write causes the $\mathcal{EFS}$ server to invoke the USD control interface and obtain an IDC offer for an *Rbufs* channel. This offer is returned to the client who may bind to the offer creating an I/O stream connected directly to the USD device. Updating the data contained in a file is achieved by sending read or write requests on the *Rbufs* connection. These I/O transactions are serviced sequentially at a rate determined by the QoS associated with the connection.

An attempt to access an area of the disk for which the permissions are not in the USD driver's per-stream cache causes the file-system to be upcalled. The file-system uses information supplied by the USD to determine the client's permissions for that area of the disk. If the attempted access lies within an extent belonging to the file then that extent is recorded in the USD cache and the access is permitted.

Files are extended by adding a new extent. The size of this extent may vary depending on the type of data the file contains. Multimedia files typically use large extents to minimise the overheads of checking access rights during I/O. As a rough guide, the extent size chosen should be comparable to the granularity at

[10] In UNIX file system traces, `stat` operations account for a large proportion of requests.

```
EFSClient : LOCAL INTERFACE =
  NEEDS USD;
  NEEDS IDCOffer;
BEGIN

  NoSuchFile : EXCEPTION [];
  Failure : EXCEPTION[];
  PermissionDenied : EXCEPTION [];

  FileID : TYPE = LONG CARDINAL;

  -- Files are named by a 64-bit identifier.

  Create : PROC [      perms : CARDINAL,
                  extentsize : CARDINAL ]
           RETURNS [ id : FileID ];
           RAISES Failure;

        -- Creates a new file with permissions "perm" and given extent
        -- size.  The file may be referred to by the identifier "id".

  Delete : PROC [ id : FileID ]
           RETURNS [ ]
           RAISES NoSuchFile, PermissionDenied;

        -- Deletes the file with the identifier "id".

  Stat  : PROC [ id : FileID ]
          RETURNS [ owner : CARDINAL,
                     mode : CARDINAL,
                     type : CARDINAL,
                     size : CARDINAL,
                     time : CARDINAL ]
          RAISES NoSuchFile;

        -- "Stat" returns the owner, mode, type, size and time
        -- associated with the file "id".

  Chmod : PROC [   id : FileID,
                 mode : CARDINAL ]
          RETURNS [ ]
          RAISES NoSuchFile, PermissionDenied;

  Open : PROC [   id : FileID,
                mode : USD.Mode,
                 qos : USD.QoS ]
         RETURNS [ usdio : IREF IDCOffer ]
         RAISES NoSuchFile, PermissionDenied;

        -- Opens the file specified by "id" for either read or write.
        -- Returns an offer for an "IO" connection to the USD.

  Grow : PROC [ id : FileID ]
         RETURNS [ e : USD.Extent ]
         RAISES NoSuchFile;

        -- Adds another extent to the specified file.

  Close : PROC [ id : FileID ] RETURNS [ ];

        -- Close the specified file.

END.
```

Figure 7.9: MIDDL for $\mathcal{EFS}$ interface (EFSClient.if).

which I/O is expected to be performed.[11]

7.7.1 Discussion

The $\mathcal{EFS}$ design allows files to be used for a number of purposes. The asynchronous nature of disk I/O allows applications to perform appropriate amounts of read-ahead and avoids the problem of the file system being required to reverse-engineer the behaviour of the application.

Should an application desire, it would be perfectly possible to run a log-structured file system in a file over $\mathcal{EFS}$ with a guaranteed proportion of the disk I/O bandwidth.

7.8 Summary

File system designs have been heavily influenced by the programming environment of operating systems such as UNIX. In such operating systems, where the file system is often used for inter-process communication, a log-structured file system can provide significant advantages over the more conventional block-structured approach. This is both due to the reduced frequency of storage allocation and the sequential disk access patterns.

Due to their ability to achieve 100% utilisation for disk writes, it has become common to use such a file system for recording of CM data. Simultaneous playback of a number of CM streams still requires the disk head to be moved and the only benefit of the file system layout is to increase the likelihood of contiguous placement of file data. An extent-based file system provides exactly the same benefits.

The majority of multimedia file systems have been implemented as dedicated CM servers and support a stream abstraction rather than the more traditional file abstractions of general purpose file systems. This amounts to communicating file access patterns to the file system which is then able to schedule I/O using global knowledge of the future behaviour of all clients.

A low-level disk abstraction has been presented can provide the same QoS

[11] The CMFS file system performed low level disk I/O in units of an extent.

guarantees as stream based CM file servers, but without the oversimplified and restrictive interface. The device abstraction module implements protection and translation at a low level using a per-connection cache to minimise the number of interactions with the device management module.

An extent-based file system is described which uses the protection afforded by the device driver to determine the areas of the disk to which each client has access. Clients of the file system may use application specific storage management policies and access patterns whilst receiving the benefits of guaranteed I/O performance.

Chapter 8

Conclusion

This dissertation has presented an architecture for device drivers in a multi-service operating system which is designed to meet the demands of applications handling time-sensitive media. This chapter summarises the work and its conclusions, and makes suggestions for further areas of study.

8.1 Summary

Chapter 2 presented the background to this work, including a brief description of the Cambridge environment, and the local influences on this work. The chapter then discussed two properties of continuous media which must influence the design of a multi-service operating system. The *temporal* property requires the ability to cope with high volumes of time-sensitive data. The operating system must therefore be able to provide applications with fine-grained guarantees on the availability of processing and I/O resources. The *informational* property means that applications can often adapt to lower levels of resources and that *soft-guarantees* are therefore sufficient.

A discussion of operating system architectures was presented which concentrated on their effectiveness for providing Quality of Service guarantees. The complexity of resource accounting, and the inevitability of QoS-crosstalk in traditional operating systems makes them unsuited to the general-purpose processing of CM data types. Attempts to extend such operating systems to support multimedia applications have been both inelegant and narrow in focus. Vertically-structured operating systems do not suffer from the above problems and provide a framework where QoS-guarantees can potentially be provided for *all* operating system resources.

Chapter 3 presented the design and prototype implementation of the Nemesis operating system. In Nemesis, the majority of operating system services are implemented within the protection and accounting domain of the client application, with only the minimum functionality in servers. This approach enables accurate and direct accounting for resource usage and enables meaningful QoS guarantees to be provided at the lowest levels.

The prototype provides a number of mechanisms which are intended to support the operation of device drivers. Of particular importance are the decoupling of hardware interrupts to enable device drivers to be scheduled as conventional processes, and the *Rbufs* data transport mechanism which efficiently supports asynchronous inter-process communication of high-bandwidth data. Nemesis, as presented however, provides no guarantees other than processor bandwidth.

Chapter 4 discusses the problem of scheduling hardware I/O resources. Conventional workstations are constructed using a hierarchical bus architecture. The majority of bus implementations do not provide any software means of influencing the arbitration mechanism, or controlling background DMA activity by bus-mastering devices. These problems are exacerbated by the use of hierarchical topologies. Even if such facilities existed, it is argued that device I/O must be scheduled on a per-connection basis rather than a per-device basis.

Alternative workstation architectures are discussed which address the problem of scheduling I/O resources to minimise QoS-crosstalk. The use of channel controllers in mainframes provides much of the required functionality, but at the expense and inflexibility of replicated I/O hardware. DAN-based workstations use a connection-oriented interconnect and support peer-to-peer transfers. Devices for such a workstation may use the connection identifier to implement protection and scheduling mechanisms in hardware — these are referred to as *User-Safe Devices*. In such a system, scheduling of devices, the processor and the interconnect may be closely integrated.

Chapter 5 presented the Nemesis Device Driver Architecture. The architecture provides a clear separation of control- and data-path operations and requires devices to be abstracted at a low level where accounting and scheduling of resource usage is most effective. Network interfaces and network-connected peripherals often provide a hardware abstraction which is highly appropriate for this architecture.

Chapter 6 considered the example of the framebuffer device. Framebuffers requires fine-grained sharing mechanisms, and are usually abstracted at a high level using a window system. This approach makes the provision of QoS-guarantees extremely difficult. Techniques are presented for abstracting the device at a much lower level than in a conventional system. These include a fine-grained protection mechanism and an extension to Nemesis which allows device driver code to be invoked by clients in a restricted environment, thereby aiding resource accounting for devices which do not support DMA.

The framebuffer driver is used to construct a window system where all rendering is performed by the client, and updates to the framebuffer are performed in a protected fashion at a rate determined by per-connection QoS parameters. The system is demonstrated to provide useful QoS guarantees and remove application QoS-crosstalk.

Chapter 7 presented techniques for restructuring the file system so as to provide QoS-guarantees. Disk drives have a number of features which make their abstraction difficult. As with the framebuffer, fine-grained sharing is necessary. In addition, mechanical timing constraints of the device mean that scheduling is necessarily a compromise between predictability and performance. A disk head scheduling algorithm was presented which, unlike conventional algorithms, does not solely attempt to maximise throughput, but instead respects QoS-guarantees of individual clients.

The disk device driver is used to implement the data-path operations of a prototype extent-based file system with per-connection QoS guarantees. The file system permits application specific optimisation of storage allocation and file access patterns.

It is the thesis of this dissertation that, given an operating system which supports Quality of Service, such as Nemesis, it it possible to construct a software architecture for converting conventional devices into User-Safe Devices providing QoS guarantees to applications. This dissertation supports the above thesis by exhibiting such an architecture and presenting implementations for a number of particularly troublesome devices — the `fb`/$\mathcal{WS}$ and `usd`/$\mathcal{EFS}$ combinations.

8.2 Further Work

The implementation of Nemesis described in this dissertation is a prototype written to enable investigation of operating system research ideas. It currently does not have a Virtual Memory (VM) system, and the window system and file system presented in this dissertation are intended mainly as proofs of concept. The Pegasus II project in the Computer Laboratory plans to produce a more robust Nemesis implementation on new platforms (including Pentium PCs), including a native Nemesis-style protocol stack, window system and file system.

Using the User-Safe Disk abstraction of chapter 7, and a driver providing pages of physical memory, it should be possible to construct a VM system for Nemesis where applications are responsible for performing their own paging to disk. Such a VM system would allow fine control over the amount of physical memory and disk bandwidth dedicated to each application, effectively being able to guarantee a maximum page-fault rate. Domains would also be free to implement whatever paging strategies best suited their requirements.

So far, Nemesis provides only the low-level mechanisms required for Quality of Service. High level system-wide resource allocation and admission control has yet to be implemented. In Nemesis, this is the task of the QoS Manager domain. The operation of the QoS Manager and the interface it presents to the user are still an open issue.

An unexpected benefit of the Nemesis approach to resource management is that the operating system has potentially become amenable to mathematical analysis. The unpredictability introduced by features such as the priority-feedback scheduler used in many UNIX implementations is no longer present. The Measure project [Measure95] is investigating call-admission control for ATM networks based on estimation of traffic *entropy* by on-line measurement. A more speculative aspect of this project is the investigation of QoS admission control in Nemesis using the same techniques.

Bibliography

[Accetta86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. *Mach: A New Foundation for* UNIX *Development.* In USENIX, pages 93–112, Summer 1986. (pp 19, 29, 32)

[Adam93] Joel Adam and David Tennenhouse. *The Vidboard: A Video Capture and Processing Peripheral for a Distributed Multimedia System.* In Procedings SIGGRAPH, August 1993. (p 14)

[Anderson92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, 10(1):53–79, February 1992. (p 28)

[Anderson93] Thomas Anderson, Susan Owicki, James Saxe, and Charles Thacker. *High Speed Switch Scheduling for Local Area Networks.* Technical Report 99, DEC Systems Research Center, April 1993. (pp 53, 62)

[ANSA95] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK. *ANSAware/RT 1.0 Manual*, March 1995. (p 36)

[ANSI86] *American National Standard for Information Systems – Small Computer System Interface (SCSI)*, 1986. ANSI X3.131-1986. (p 42)

[Apple85] Apple Computer. *Inside Macintosh*, volume 1. Addison-Wesley, 1st edition, 1985. (pp 16, 72)

[ARM91] Advanced RISC Machines. *ARM6 Macrocell Datasheet*, 0.5 edition, November 1991. (p 26)

[Atkinson83] M.P. Atkinson, P.J. Bailey, K.J. Chishold, W.P. Cockshott, and R. Morrison. *PS-algol: A Language for Persistent Programming.* In 10th Australian National Computer Conference, pages 70–79, 1983. (p 98)

[Atkinson93] Ian Atkinson. *A Digital Signal Processor and Audio Node for the Desk Area Network.* Dissertation for Part II of the Computer Science Tripos, University of Cambridge Computer Laboratory, July 1993. (p 14)

[ATMForum93] The ATM Forum. *ATM User-Network Interface Specification Version 3.0.* Prentice Hall, 1993. (p 50)

[Baker91] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. *Measurements of a Distributed File System.* In Proceedings of the 13th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, pages 198–212, October 1991. (p 93)

[Barham94] Paul Barham and Ian Pratt. *The ATM Camera V2 (AVA-200).* In ATM Document Collection 3 (The *Blue* Book), chapter 36. University of Cambridge Computer Laboratory, March 1994. (p 14)

[Barham95] P. Barham, M. Hayter, D. McAuley, and I. Pratt. *Devices on the Desk Area Network.* IEEE Journal on Selected Areas in Communication, 13, March 1995. (pp 7, 14, 52, 53)

[Bershad90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. *Lightweight Remote Procedure Call.* ACM Transactions on Computer Systems, 8(1):37–55, February 1990. (p 22)

[Bershad94] Brian N. Bershad, Craig Chambers, Susan Eggers, Chris Maeda, Dylan McNamee, Przemyslaw Pardyak, Stefan Savage, and Emin Gün Sirer. *SPIN - An Extensible Mircokernel for Application-specific Operating System Services.* Technical Report 94-03-03, University of Washington, February 1994. (p 19)

[Birrell84] Andrew D. Birrell and Bruce Jay Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2(1):39–59, February 1984. (p 30)

[Birrell87] A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levin. *Synchronisation Primitives for a Multiprocessor: A Formal Specification.* Technical Report 20, Digital Equipment Corporation Systems Research Centre, August 1987. (p 35)

[Black94] Richard J. Black. *Explicit Network Scheduling.* PhD thesis, University of Cambridge Computer Laboratory, 1994. (pp 4, 23, 24, 29, 30, 35, 37, 39, 59, 60, 61, 80)

[Bosch93] Peter Bosch, Sape Mullender, and Tage Stabell-Kulø. *Huygens File Service and Storage Architecture.* In BROADCAST Technical Report Series, 1st year report, October 1993. (pp 96, 97)

[Bovopoulos93] Andreas Bovopoulos, R Gopalakrishnan, and Saied Hosseini. *SYMPHONY: A Hardware, Operating System, and Protocol Processing Architecture for Distributed Multimedia Applications.* Technical Report WUCS-93-06, Washington University in St. Louis, March 1993. (p 51)

[Bricker91] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. *A New Look at Microkernel-based* unix *Operating Systems: Lessons in Performance and Compatibility.* Technical Report CS/TR-91-7, Chorus Systemes, February 1991. (p 19)

[Brinch-Hansen70] Per Brinch-Hansen. *The Nucleus of a Multiprogramming System.* Communications of the ACM, 13(4):238–241,250, April 1970. (p 25)

[Campbell92] Andrew Campbell, Geoff Coulson, Francisco Garcia, and David Hutchinson. *A Continuous Media Transport and Orchestration Service.* Technical Report Internal Report ref. MPG-92-05, Distributed Multimedia Research Group, Department of Computing, Lancaster University, 1992. Presented at SIGCOMM '92. (p 12)

[Campbell93] Andrew Campbell, Geoff Coulson, Francisco Garcia, David Hutchison, and Helmut Leopold. *Integrated Quality of Service for Multimedia Communications.* In Proceedings of IEEE INFOCOMM '93, volume 2, pages 732–739, March/April 1993. (p 12)

[Castle95] Oliver M. Castle. *Synthetic Image Generation for a Multiple-view Autostereo Display.* PhD thesis, University of Cambridge Computer Laboratory, 1995. Available as Technical Report No. 382. (p 11)

[CBM91] Commodore Business Machines. *Amiga ROM Kernel Reference Manual: Libraries.* Addison-Wesley, (3rd edition) edition, 1991. ISBN 0-201-56774-1. (p 72)

[CCube94] C-Cube Microsystems. *Product Catalog*, Spring 1994. (p 10)

[Chaney95] Alan J. Chaney, Ian D. Wilson, and Andrew Hopper. *The Design and Implementation of a RAID-3 Multimedia File Server.* In Proceedings of 5th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV), April 1995. Available as ORL Technical Report 95-3. (p 14)

[Chase93] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. *Sharing and Protection in a Single Address Space Operating System.* Technical Report 93-04-02, revised January 1994, Department of Computer Science and Engineering, University of Washington, Seattle, Washington 98195, USA, April 1993. (p 22)

[Coffman72] E. G. Coffman, L. A. Klimko, and B. Ryan. *Analysis of Scanning Policies for Reducing Disk Seek Times.* SIAM Journal of Computing, 1(3), September 1972. (p 106)

[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 System Support for Digital Audio and Video, pages 49–60, November 1993. (pp 21, 32)

[Coulson95] G. Coulson, A. Campbell, P. Robin, M. Papathomas G. Blair, and D. Shepherd. *The Design of a QoS-Controlled ATM-Based Communications System in Chorus*. IEEE Journal on Selected Areas In Communications, 13(4):686–699, May 1995. (p 21)

[Custer93] Helen Custer. *Inside Windows NT*. Microsoft Press, 1993. (pp 18, 70)

[Date90] C.J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley, 5th edition, 1990. (p 98)

[DEC90] Digital Equipment Corporation. *Turbochannel Developers Kit Version 2*, September 1990. (p 41)

[DEC93] Digital Equipment Corporation. *DECchip 21064 Evaluation Board User's Guide*, May 1993. Order Number EC-N0351-72. (p 23)

[DEC94a] Digital Equipment Corporation. *ATMWorks 750 Infosheet*, November 1994. DEC Order code EC-F3925-42, availible by ftp from `gatekeeper.dec.com`. (p 62)

[DEC94b] Digital Equipment Corporation. *DEC3000 300/400/500/600/700/800/900 AXP Models: System Programmer's Manual*, 2nd edition, July 1994. Order Number EK-D3SYS-PM.B01. (pp 23, 43)

[DEC95] Digital Equipment Corporation. *DECchip 21164 Microprocessor Motherboard User's Manual*, August 1995. Order Number EC-QLJLA-TE. (p 23)

[Deming95] David Deming. *Serial Storage Architecture - A Technical Overview*. Technical Report, SSA Industry Association, San Jose, Ca., 1995. (pp 65, 108)

[Dixon92] Michael Joseph Dixon. *System Support for Multi-Service Traffic*. PhD thesis, University of Cambridge Computer Laboratory, January 1992. Available as Technical Report no. 245. (p 37)

[Engler95] D.R. Engler, M.F. Kaashoek, and J. O'Toole, Jr. *Exokernel: An Operating System Architecture for Application-Level Resource Management.* In Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, pages 251–266, December 1995. (pp 20, 61)

[Goldenberg92] Ruth E. Goldenberg and Saro Saravanan. *VMS for Alpha Platforms Internals and Data Structures*, volume 1. Digital Press, preliminary edition, 1992. (p 18)

[Gopal86] I. S. Gopal and Z. Rosberg. *Quasi-optimal disk-arm scheduling.* Technical Report, International Business Machines TJ Watson Research Center, 1986. (p 95)

[Hamilton93] Graham Hamilton and Panos Kougiouris. *The Spring Nucleus: A Microkernel for Objects.* Technical Report 93-14, Sun Microsystems Laboratories, Inc., April 1993. (pp 19, 22, 72)

[Hayter91] Mark Hayter and Derek McAuley. *The Desk Area Network.* ACM Operating Systems Review, 25(4):14–21, October 1991. (pp 7, 14, 50)

[Hayter93] Mark Hayter. *A Workstation Architecture to Support Multimedia.* PhD thesis, University of Cambridge Computer Laboratory, September 1993. Available as Technical Report No. 319. (pp 14, 52)

[Hayter94] 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. (p 23)

[Hew94] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual*, 3rd ed. edition, 1994. (p 10)

[Hildebrand92] Dan Hildebrand. *An Architectural Overview of QNX.* In USENIX Workshop Proceedings : Micro-kernels and Other Kernel Architectures, pages 113–126, April 1992. (p 96)

[Hopper90] Andy Hopper. *Pandora: An Experimental System for Multimedia Applications.* ACM Operating Systems Review, 24(2):19–34, April 1990. Available as ORL Report no. 90-1. (pp 6, 12)

[Houh95] H. H. Houh, J. F. Adam, M. Ismert, C. J. Lindblad, and D. L. Tennenhouse. *The VuNet Desk Area Network: Architecture, Implementation and Experience.* IEEE Journal on Selected Areas In Communications, 13(4):710–721, May 1995. (pp 14, 51)

[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 60)

[Hyden94] Eoin Hyden. *Operating System Support for Quality of Service.* PhD thesis, University of Cambridge Computer Laboratory, February 1994. Available as Technical Report No. 340. (pp 4, 7, 24, 25)

[Hyman88] Michael Hyman. *Microsoft Windows Program Development.* MIS Press, Portland, Oregon, 1988. (p 70)

[IBM80] IBM Systems. *OS/VS2 MVS: Overview*, 2nd edition, 1980. (pp 49, 96)

[ISO93] *Coding of Moving Pictures and Associated Audio for Digital Storage Media at up to About 1.5 Mbit/s*, 1993. ISO/IEC 11172. (p 10)

[ISO95] *Generic Coding of Moving Pictures and Associated Audio Information*, 1995. ISO/IEC 13818. (p 10)

[Jacobson91] David M. Jacobson and John Wilkes. *Disk Scheduling Algorithms Based on Rotational Position.* Technical Report 91-7, Concurrent Systems Project, Hewlett-Packard Laboratories, Palo Alto, CA, February 1991. (p 95)

[Jain92] R. K. Jain. *Scheduling Data Transfers in Parallel Computers and Communications Systems.* PhD thesis, University of Texas at Austin, 1992. (p 48)

[Jardetzky92] Paul Jardetzky. *Network File Server Design for Continuus Media.* PhD thesis, University of Cambridge Computer Laboratory, October 1992. Available as Technical Report No. 268. (pp 6, 96, 97, 110)

[Kane88] Gerry Kane. *MIPS RISC Architecture*. Prentice-Hall, 1988. (p 26)

[Khan94] Naeem Ahmed Khan. *Cell Scheduling to Support CBR Traffic*. In ATM Document Collection 4 (The *Green* Book), chapter 5. University of Cambridge Computer Laboratory, October 1994. (p 53)

[Kotz94] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. *A Detailed Simulation Model of the HP 97560 Disk Drive*. Technical Report PCS-TR94-220, Dartmouth College, July 1994. (p 104)

[Lee95] D.D. Lee, D.R. McAuley, D. Northcutt, and Wilheim. *High Speed Serial Communications for Desktop Computer Peripherals*. US Patent (pending) 16747-8/P973, 1995. (p 51)

[Leffler89] S.J. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD* UNIX *Operating System*. Addison-Wesley, 1989. (p 18)

[LeGall91] Didier LeGall. *MPEG - A Video Compression Standard for Multimedia Applications*. Communications of the ACM, 34(4):46–58, April 1991. (p 10)

[Leslie91] Ian M. Leslie and Derek R. McAuley. *Fairisle: An ATM Network for the Local Area*. In Proceedings of ACM SIGCOMM, September 1991. (pp 6, 52)

[Leslie93] I. M. Leslie, D. R. McAuley, and S. J. Mullender. *Pegasus—Operating System Support for Distributed Multimedia Systems*. ACM Operating Systems Review, 27(1):69–78, January 1993. (pp 7, 23)

[Leslie96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. *The Design and Implementation of an Operating System to Support Distributed Multimedia Applications*. IEEE Journal on Selected Areas in Communication, 1996. (to appear). (pp 7, 20)

[Liu73] C. L. Liu and James W. 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 34)

[Liu91] J. Liu, J. Lin, W. Shih, A. Yu, J. Chung, and Z. Wei. *Algorithms for Scheduling Imprecise Computations.* IEEE Computer, 24(5):58–68, May 1991. (p 7)

[Lo93] Sai-Lai Lo. *A Modular and Extensible Network Storage Architecture.* PhD thesis, University of Cambridge Computer Laboratory, 1993. Available as Technical Report No. 326. (pp 98, 110)

[Lougher93] Philip Lougher and Doug Shepherd. *The Design of a Storage Server for Continuous Media.* The Computer Journal, 36(1):32–42, February 1993. (p 97)

[McAuley89] Derek McAuley. *Protocol Design for High Speed Networks.* PhD thesis, University of Cambridge Computer Laboratory, September 1989. Available as Technical Report no. 186. (p 6)

[Measure95] University of Cambridge Computer Laboratory, Dublin Institute for Advanced Studies and Telia Research. *Measure: Resource Allocation for Multimedia Communication and Processing Based on On-Line Measurement*, May 1995. An ESPRIT Reactive LTR Full Proposal (Proposal 20.113). (p 117)

[Mercer93] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. *Processor Capacity Reserves: An Abstraction for Managing Processor Usage.* In Proc. Fourth Workshop on Workstation Operating Systems (WWOS-IV), October 1993. (pp 21, 72)

[Mogul87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. *The Packet Filter: An Efficient Mechanism for User-level Network Code.* Research Report 87/2, Digital Equipment Corporation, Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, California 94301, November 1987. (pp 20, 60)

[Mogul91] Jeffrey C. Mogul and Anita Borg. *The Effect of Context Switches on Cache Performance.* In Proceedings of the 18th International Symposium on Computer Architecture, 1991. (p 35)

[Mogul95] Jeffrey C. Mogul and K. K. Ramakrishnan. *Eliminating Receive Livelock in an Interrupt-driven Kernel.* Technical Report 95.8, Digital Equipment Corporation Western Research Laboratory, December 1995. (pp 18, 64)

[Morrison89] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. *The Napier88 Reference Manual.* Technical Report PPRR-77-89, Universities of Glasgow and St. Andrews, 1989. (p 98)

[Nelson91] Greg Nelson, editor. *Systems Programming With Modula-3.* Prentice-Hall, Englewood Cliffs, NJ 07632, 1991. (p 32)

[Nicolaou90] Cosmos Nicolaou. *A Distributed Architecture for Multimedia Communication Systems.* PhD thesis, University of Cambridge Computer Laboratory, December 1990. Available as Technical Report no. 220. (pp 6, 11, 12)

[Ousterhout85] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. *A Trace Driven Analysis of the UNIX 4.2 BSD File System.* In Proceedings of the 10th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, pages 15–24, December 1985. (p 93)

[Ousterhout89] John Ousterhout and Fred Douglis. *Beating the I/O bottleneck: A case for log-structured file systems.* ACM Operating Systems Review, 23(1):11–28, January 1989. (p 95)

[PCI95] PCI Special Interest Group. *PCI Local Bus Specification, Revision 2.1*, June 1995. (pp 41, 50)

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. *Plan 9 from Bell Labs.* In Proceedings of the Summer 1990 UKUUG Conference, London, pages 1–9, July 1990. (p 19)

[Pike91] Rob Pike. $8\frac{1}{2}$, *the Plan 9 Window System.* In Proceedings of the Summer 1991 USENIX Conference, Nashville, pages 257–265, June 1991. (p 78)

[Pike94] Rob Pike. $8\frac{1}{2}$ *in Brazil.* Personal communication, December 1994. (p 78)

[Pratt92] Ian Pratt. *An ATM camera for the Desk Area Network.* Dissertation for Part II of the Computer Science Tripos, University of Cambridge Computer Laboratory, May 1992. (p 65)

[Pratt95] Ian Pratt. *A Key Based Framestore for the Desk Area Network.* In ATM Document Collection 3 (The *Green* Book), chapter 1. University of Cambridge Computer Laboratory, October 1995. (pp 14, 82)

[Pratt96] Ian Pratt. *User Safe Devices.* PhD thesis, University of Cambridge Computer Laboratory, 1996. in preparation. (p 53)

[Reddy94] A. L. Narashima Reddy and James C. Wyllie. *I/O Issues in a Multimedia System.* IEEE Computer, pages 69–74, March 1994. (p 97)

[Reed76] D. P. Reed. *Processor Multiplexing in a Layered Operating System.* PhD thesis, Massachusetts Institute of Technology Computer Science Laboratory, June 1976. Available as Technical Report no 164. (p 37)

[Reed79] David P. Reed and Rajendra K. Kanodia. *Synchronization with Eventcounts and Sequencers.* Communications of the ACM, 22(2):115–123, February 1979. (p 35)

[Ritchie74] D. Ritchie and K. Thompson. *The* UNIX *Time-Sharing System.* Communications of the ACM, 17(7):365–375, July 1974. (p 18)

[Roscoe95] Timothy Roscoe. *The Structure of a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, April 1995. Available as Technical Report No. 376. (pp 4, 24, 26, 30, 35, 38, 39)

[Rosenblum92] Mendel Rosenblum. *The design and implementation of a log-structured file system.* PhD thesis, University of California, Berkeley, 1992. (p 96)

[Rozier90] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. *Overview of the CHORUS Distributed Operating Systems.* Technical Report Technical Report CS-TR-90-25, Chorus Systemes, 1990. (pp 19, 29)

[Ruemmler94] Chris Ruemmler and John Wilkes. *An Introduction to Disk Drive Modelling.* IEEE Computer, pages 17–28, March 1994. (pp 100, 104)

[Saltzer84] J. H. Saltzer, D. P. Reed, and D. Clark. *End-to-End Arguments in System Design.* ACM Trans. on Computer Systems, 2(4), November 1984. (p 62)

[Scheifler86] R. W. Scheifler and J. Gettys. *The X Window System.* ACM Transactions on Graphics, 5(2), April 1986. (p 70)

[Seltzer90] Margo I. Seltzer, Peter M. Chen, and John K. Ousterhout. *Disk Scheduling Revisited.* In Proceedings of the Winter 1990 USENIX Technical Conference, January 1990. (p 95)

[Seltzer92] Margo Ilene Seltzer. *File system performance and transaction support.* PhD thesis, University of California at Berkeley, 1992. (p 96)

[Sites92] Richard L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992. (p 26)

[Stratford96] N. Stratford. *A Window System for the DAN Framestore.* Dissertation for Part II of the Computer Science Tripos, University of Cambridge Computer Laboratory, May 1996. (p 86)

[Sun88] SunSoft, Mountain View, Ca. *SunView Programmer's Guid*, 4.1.1 edition edition, 1988. (p 72)

[Swinehart86] D. Swinehart, P. Zellweger, R. Beach, and R. Hagemann. *A Structural View of the Cedar Programming Environment.* Technical Report CSL-86-1, Xerox Corporation, Palo Alto Research Center, June 1986. (published in ACM Transactions on Computing Systems 8(4), October 1986). (pp 16, 72)

[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 19)

[Temple84] S. Temple. *The Design of a Ring Communication Network.* PhD thesis, University of Cambridge Computer Laboratory, January 1984. (p 6)

[Tennenhouse89] David L. Tennenhouse. *Layered Multiplexing Considered Harmful.* In Protocols for High Speed Networks, IBM Zurich Research Lab., May 1989. IFIP WG6.1/6.4 Workshop. (pp 6, 19)

[Tokuda93] Hide Tokuda and Takuro Kityama. *Dynamic QOS Control based on Real-Time Threads.* In Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio Video, pages 113–122, November 1993. (p 21)

[Voth91] David Voth. *MAXine System Module Functional Specification.* Technical Report, Workstation Systems Engineering, Digital Equipment Corporation, 100 Hamilton Avenue, Palo Alto, CA 94301, July 1991. revision 1.2. (p 23)

[Waldspurger94] Carl A. Waldspurger and William E. Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management.* In Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 1–11, Novemeber 1994. (pp 21, 72)

[Waldspurger95] Carl A. Waldspurger and William E. Weihl. *Stride Scheduling: Deterministic Proportional-Share Resource Mangement.* Technical Report, Massachusetts Institute of Technology Computer Science Laboratory, June 1995. Technical Memo MIT/LCS/TM-528. (pp 22, 59)

[Wallace91] G. Wallace. *The JPEG Still Picture Compression Standard.* Communications of the ACM, 34(4), April 1991. (p 9)

[Want92] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. *The active badge location system.* ACM Transactions on Computer Systems, 10(1), 1992. Available as ORL Report no. 92-1. (p 7)

[Worthington94a] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. *Scheduling Algorithms for Modern disk Drives.* In Proceedings of ACM SIGMETRICS '94, Santa Clara, CA. USA, pages 241–251, May 1994. (p 104)

[Worthington94b] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. *On-Line Extraction of SCSI Disk Drive Parameters.* In Proceedings of ACM SIGMETRICS '95, Ottowa, Ontario. Canada, pages 146–156, 1994. (p 104)

[Wray94] Stuart Wray, Tim Glauert, and Andrew Hopper. *The Medusa Applications Environment.* IEEE Multimedia, 1(4):54–63, Winter 1994. Available as ORL Technical Report 95-3. (p 14)