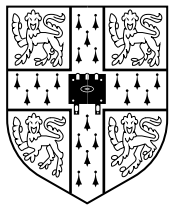
Operating System Support for Quality of Service

Eoin Andrew Hyden

Wolfson College
University of Cambridge



A dissertation submitted for the degree of
Doctor of Philosophy

February, 1994

Abstract

The deployment of high speed, multiservice networks within the local area has meant that it has become possible to deliver continuous media data to a general purpose workstation. This, in conjunction with the increasing speed of modern microprocessors, means that it is now possible to write application programs which manipulate continuous media in real-time. Unfortunately, current operating systems do not provide the resource management facilities which are required to ensure the timely execution of such applications.

This dissertation presents a flexible resource management paradigm, based on the notion of Quality of Service, with which it is possible to provide the scheduling support required by continuous media applications. The mechanisms which are required within an operating system to support this paradigm are described, and the design and implementation of a prototypical kernel which implements them is presented.

It is shown that, by augmenting the interface between an application and the operating system, the application can be informed of varying resource availabilities, and can make use of this information to vary the quality of its results. In particular an example decoder application is presented, which makes use of such information and exploits some of the fundamental properties of continuous media data to trade video image quality for the amount of processor time which it receives.

To my parents

John and Olive

Preface

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

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

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

Trademarks

Acknowledgements

I would like to thank my supervisor Ian Leslie for his guidance and encouragement throughout my time at the Computer Laboratory. I would also like to thank Derek McAuley for many interesting and thought-provoking discussions. Thanks are also due to Roger Needham for his observations and comments which have often provided insight into the problem at hand.

The Systems Research Group has provided a stimulating environment in which to work and my thanks go to those who helped to make it that way. In particular, Paul Barham, Richard Black, Simon Crosby, Mark Hayter, Paul Jardetzky, Guangxing Li, Ian Pratt and Timothy Roscoe were always ready to discuss ideas. The interest expressed by Jean Bacon and Ken Moody in my work is also appreciated.

For reading and commenting on drafts of this dissertation, I am indebted to Shaw Chuang, Robin Fairbairns, Ian Leslie, Derek McAuley, Simon Moore, Cosmos Nicolaou, Timothy Roscoe and Cormac Sreenan.

This work was supported by a studentship from Tadpole Technology plc. I am grateful to George Grey, CEO of Tadpole for his support. Thanks are also due to colleagues past and present at Tadpole, Timothy Bissell, Steve Chamberlain, Jeff Graham, Mark Hancock, Crispin Thomson and Jes Wills for their continued interest and support.

Contents

List of Figures

List of Tables

Glossary

ACME	Abstractions for Continuous Media
ASCII	American Standard Code for Information Interchange
ATM	Asynchronous Transfer Mode
BE	Best Effort
B-ISDN	Broadband Integrated Services Digital Network
CBR	Cambridge Backbone Ring
CBSRP	Capacity Based Session Reservation Protocol
CIF	Common Intermediate Format
CM	Continuous Media
CPU	Central Processing Unit
DAN	Desk Area Network
DCT	Discrete Cosine Transform
DPCM	Differential Pulse Code Modulation
DS	Deferrable Server
EDF	Earliest Deadline First
FCFS	First Come First Served
FIFO	First In First Out
HRT	Hard Real-Time
IDCT	Inverse Discrete Cosine Transform
IMAC	Integrated Multimedia Applications Communication
I/O	Input/Output
IPC	Inter-Process Communication
JPEG	Joint Photographic Experts Group
KLS	Kernel Level Scheduler
LAN	Local Area Network

MAC	Media Access
MAS	Multiple Address Space
MPEG	Motion Picture Experts Group
NPR	Non-preemptive Priority
NRM	Network Resource Manager
NTSC	Nemo Trusted Supervisor Call
PB	Processor Bandwidth
PCB	Process Control Block
pel	Picture Element
PPR	Preemptive Priority
PR	Priority
PTM	Packet Transfer Mode
QCIF	Quarter Common Intermediate Format
QOS	Quality of Service
RISC	Reduced Instruction Set Computer
RM	Rate Monotonic
RPC	Remote Procedure Call
RTRA	Run Time Resource Allocator
SAS	Single Address Space
SLS	Split Level Scheduling
SM	Session Manager
SRM	System Resource Manager
SRT	Soft Real-Time
STM	Synchronous Transfer Mode
ULS	User Level Scheduler
VBR	Variable Bit Rate
VP	Virtual Processor
VPI	Virtual Processor Interface

Chapter 1

Introduction

This dissertation is concerned with the treatment of continuous media by the system and application software within a workstation environment. It commences by describing the factors which motivate the investigation of this topic.

1.1 Motivation

Both the deployment of multi-service networks within the local area and the availability of suitable network interfaces have made it possible to deliver to a user's workstation, new classes of traffic such as digital audio and video, in addition to the types of network traffic previously found in LANs. This has prompted much research into how video and audio are to be treated within a workstation environment. Video and audio are commonly referred to as *continuous media* (CM) because, while they are in fact discrete, their regular, periodic presentation at sufficiently high frequencies makes them appear smooth and continuous to the human perception.

Until recently, the bandwidth and temporal requirements of even moderate video formats have meant that special-purpose, dedicated hardware has been used to handle the video and audio data; the Pandora [Hopper90] project is an example of such a system. In systems like this, the workstation CPU controls the continuous media streams indirectly and typically does not directly manipulate the media data.

The increasing power of microprocessors [Geppert93] has made it possible for general purpose processors to be used to perform tasks which have previously required either dedicated hardware or special purpose, digital signal processors. Equipping a workstation with such a microprocessor and a suitable network interface enables it directly to manipulate continuous media streams in real-time without the need

for any special purpose hardware. Work recently done within the Computer Laboratory has produced a system which has much of the functionality of a Pandora's box, but which runs on an unmodified DECstation 5000/25, containing a 25MHz MIPS R3000 processor and an interface between an ATM network and the DEC TURBOchannel [Greaves92].[1]

While it is anticipated that future workstation hardware will be quite powerful, even moderate video formats are able to consume a considerable amount of system resources. For this reason, even though it is possible to handle video streams with the workstation CPU, it may not necessarily be the case that *all* of the streams in a system ought to be handled by the CPU. [Hayter91] presents a flexible, extensible, workstation architecture which is based on a switch [Leslie91] rather than a bus and in which CM streams can be routed either through a processor if they require processing or around it if they do not. Both bus-based and switch-based architectures have in common the ability to present CM data to applications. Consequently, CM become another data type and the writers of applications can write programs which store, process and present them similarly to other data types. Current workstation operating systems do not provide adequate support for applications which handle such data. This provides the motivation for investigating the treatment of CM data within such an environment.

1.2 Environment

Many CM applications are distributed [Nicolaou91], and these are in many respects more interesting than applications which run on a single machine. So the type of system considered in the following chapters will be based on generic kernel functionality which provides the infrastructure for execution of and communication between active entities (clients and servers). Within the distributed environment, specialised servers run on machines configured for the services they offer. For example, one machine might provide a CM file service [Jardetzky92] and another a synchronisation service [Sreenan92]. To discuss completely and design such a system would require the investigation of many areas such as naming, security and reliability which are beyond the scope of this work. So the work which is described in the following chapters focusses on the development of low-level software which can be used to support the servers and clients with which such a system can be built.

[1] The Pandora framework was ported by Timothy Roscoe, the X server and CM stream software by Paul Barham, and the Base Board Audio driver and AudioFile software by Shaw Chuang.

1.3 Properties of Continuous Media

CM have two important properties. The first property is that their fidelity is often dependent upon the timeliness with which they are presented. This will be referred to as the *temporal property* of CM and creates the requirement that code which manipulates the segments of CM data may need to be executed within suitable windows of time.

The second property is that they are often tolerant of the loss of some of their information content. It is precisely this property which enables compression algorithms such as those described in chapter 3 to obtain such high compression ratios. This property will be referred to as the *informational property* and provides something which might be exploited by systems which handle CM.

1.4 Issues

While there is a considerable body of experience with building real-time systems which are able to provide temporal guarantees for the execution of applications [Stankovic88], the exact relationship between CM and real-time systems is yet to be determined. Correctness in real-time systems requires that the result of a computation is both logically correct and available by a particular time. In many such systems, failure to produce a correct result on time is treated as a fatal system error. Such stringency does not readily accommodate CM applications which might require timely execution most of the time or only some of the time, and it does not allow the informational property of CM to be exploited easily.

Much of the current experience with real-time programming derives from the construction of embedded systems. These may be viewed as "black boxes" whose inputs and corresponding outputs are well defined. In such an environment, the scheduler has a detailed knowledge of system resource availability and application resource requirements. Applications are written with the assumption that they will always receive all of the resources which they require and system overload is an exceptional condition. The design and construction of these systems emphasises the ability to predict system responses to known inputs.

A workstation, particularly one connected to a network, is a dynamic environment in which system inputs, resource availability and requirements are not likely to be known in any detail. The size, complexity and dynamic nature of such systems makes it difficult to predict their behaviour exactly, but in order to support CM applications, they must still be able to maintain the temporal properties of CM

data. The types of applications which are envisioned are distributed and an exact description of their resource requirements is likely to be complex. Some means is needed, whereby an application's resource requirements can be expressed more simply, and the system scheduler needs to be able to interpret such descriptions.

CM applications can consume large amounts of resources and this can impact the performance of the rest of a system. It may be, that even in a lightly loaded system, a user will not want to give an application all of the resources it wants. In an overloaded system, there are not enough resources available to satisfy all requests, and the ability to limit the usage of resources by an application can be used to control the manner in which the system degrades. The operating system mechanisms needed to do this are in many respects similar to those used in traditional mainframe timesharing systems, but they are given a new perspective by requiring that they maintain the temporal properties of CM and enable exploitation of the informational properties.

Giving an application fewer resources than it needs affects its performance so some applications may benefit from being written in a manner which allows them to adjust the quality of their results according to the resources which are available to them. Informing an application of the resources which are available to it would help guide its decisions in such circumstances. Mechanisms suitable for conveying this information to applications need to be investigated.

In situations of extreme overload, the system may have to refuse to run an application on the grounds that running the new application would adversely affect the performance of the system in general. Consideration needs to be given to the development of policies for determining when newly arrived applications are allowed to run, causing the system's performance to deteriorate, and when they ought to be refused.

1.5 Quality of Service

Maintaining the temporal properties of CM data and applications within an operating system is primarily a resource allocation problem. The model of resource allocation which is used throughout this dissertation is based on the notion of Quality of Service (QOS). Within this model, QOS appears in different forms as part of the interfaces between the layers of a system. At the interface between an application and the operating system, the system is viewed as a *service provider* and the application is viewed as a *service user*. The application can then request a certain QOS from the operating system. As a further example, at the interface between

an application and a user, the application might be the service provider, with the user requesting a certain QOS from the application. A QOS specification in this model is a means by which the usual syntactic and semantic definition of an interface such as a system call may be augmented to incorporate extra requirements such as timeliness and accuracy.

1.6 Aims

This dissertation aims to investigate mechanisms and policies which can be implemented in an operating system to provide support for CM applications. In particular, the suitability of QOS as a scheduling paradigm for CM applications is to be examined and the mechanisms required to support it are to be determined. Also to be explored are the effects which this type of resource allocation strategy has on the design of applications, their behaviour, and the interface between applications and the system.

1.7 Synopsis

Chapter 2 presents a summary of some background material covering real-time scheduling and resource allocation within ATM networks, then compares these with QOS-based resource management techniques for scheduling CM applications in a workstation.

Chapter 3 discusses the incorporation of QOS into operating systems as a resource allocation paradigm.

Chapter 4 presents considerations to be taken into account when designing an operating system which is to provide QOS contracts to CM applications.

Chapter 5 describes a small system called Nemo which was implemented to experiment with some of the mechanisms proposed in chapter 4.

Chapter 6 contains an evaluation of Nemo and shows how applications can be implemented to make use of the features provided by it.

Chapter 7 summarises the work contained in the previous chapters and draws a number of conclusions.

Chapter 2

Background

A characteristic of continuous media which distinguishes them from the other types of data which are currently found in workstations is that they have genuine temporal requirements. So it is reasonable to expect that the techniques which have been developed for the construction of real-time systems will be applicable to systems in which applications handle continuous media.

2.1 Real-Time

A *real-time* system is one in which the correct operation of the system depends not only on the logical correctness of any computed result, but also on the time at which the result is delivered. A system in which all computed results must be delivered on time is a *hard real-time* (HRT) system. A system in which the computed results are sometimes allowed to be late is called a *soft real-time* (SRT) system.

A *task* is a single execution of a body of code. The *arrival time* of a task is the time at which the system first becomes aware that the task has to be executed. Tasks can be classified in terms of their arrival time characteristics. A *periodic* task arrives at regular intervals and is characterised by a fixed inter-arrival time or period $T(t)$ and a computation time $C(t)$.

Aperiodic tasks are typically characterised by a stochastic arrival rate and may be *bursty* in nature. Because aperiodic tasks can arrive at any rate, there is the possibility that multiple instances of an aperiodic task could arrive within a short time and overload the system. For this reason, it is commonly assumed that there is a minimum time between the arrivals of different instances of the same task. Tasks with such arrival rates are known as *sporadic* tasks. There is still a chance that in a

system, multiple sporadic tasks will arrive within a short period of time and there will not be sufficient resources available to process them all within the required time. When this happens, a system is said to be experiencing *transient overload*.

A task's *release time* is the time after which it is allowed to execute; its *deadline* is the time by which it must complete execution. These timing parameters can be represented on a timing diagram as shown in figure 2.1. This diagram depicts the execution of a task with arrival time A, release time R, computation time C and deadline D. The arrival and release times are indicated by the symbol $\uparrow$ and the deadline by $\downarrow$.

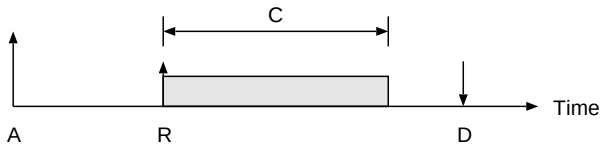


Figure 2.1: Timing diagram for a task.

2.1.1 Real-Time Versus Fast

With these definitions, it is possible to investigate the effect of resource utilisation on real-time problems. Figure 2.2 shows the timing diagram of a periodic process which runs every 100 milliseconds, requires 20 milliseconds of processing time and has a deadline of 20 milliseconds after its arrival. The speed of the processor has been chosen to be just sufficient to complete the task by its deadline.

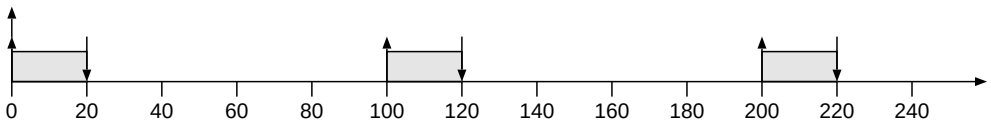


Figure 2.2: A periodic task.

Now suppose that another periodic process is introduced, which requires the same processing time, has the same period, but has a deadline of 40 milliseconds after its arrival. There is then only one order in which the processor can execute both processes so that they complete before their deadlines and this is shown in figure 2.3. The addition of a second task means that the system now has to be able to decide which of the tasks to run first. This is a result of there being some contention between the two tasks for a system resource (the processor) which can only be used by one task at a time. Note that the long term processor utilisation in this example

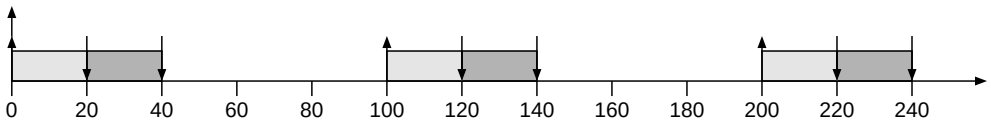


Figure 2.3: Two periodic tasks.

is only 40%, but that during the interval of interest (the first 40 milliseconds of every period) the processor is fully utilised.

Having to make this decision could be avoided by increasing the speed of the processor by *at least* a factor of two. This would halve the amount of time required to execute each task with the result that, regardless of the order in which they would be executed, they would both be completed within 20 milliseconds of being requested. Doubling the processor speed would decrease the overall processor utilisation from 40% to 20%. It is precisely this decrease in processor utilisation which has removed the necessity of having to execute the tasks in a particular order.

Additionally, if the results of these processes were to be delivered at a particular time, for example at their deadlines, then even if the processor is so fast that the order in which the tasks are executed is irrelevant, the system still needs to be able to arrange for the responses to occur at a specified time. Increasing processor speed has solved only part of the problem; extra work is required to ensure that the responses, having been calculated, occur at the appropriate times.

This is a simple example, but it does present some consequences. Firstly, if temporally sensitive data is to be handled within a workstation, then attention to scheduling will increase the amount of useful work which is achievable with a given piece of hardware. Secondly, increasing processor speed alone will neither remove nor solve the scheduling problems presented by CM systems; timely presentation of CM data can mean that, between its arrival and release times, a process can do little or no useful work even if the resources it requires are made available to it. Thirdly, the definition of a deadline within a HRT environment implies that the correct result must be delivered by a task before its deadline and that any result delivered after its deadline is of *no* use. When presenting continuous media, it is sometimes the case that approximate results delivered on time are useful and that correct results have some use if they are delivered only a short time after their deadlines. This suggests that continuous media systems may present some problems which have not yet been completely addressed by the current work on real-time systems.

2.1.2 Scheduling

Correct implementation of a real-time system involves allocating the available resources to the tasks which require them in such a manner that every task completes before its deadline. This is the purpose of the system *scheduler* which, whenever invoked, executes a scheduling algorithm to determine when and how to allocate available resources to the known tasks. Such an allocation is known as a *schedule*. If the allocation is such that all tasks complete before their deadlines, the resulting schedule is called a *feasible* schedule.

In a *static* real time system, all of the tasks in the system, their arrival times and resource requirements are known *a priori*, so the scheduling decisions can be made *off-line*, reducing the run-time scheduling requirements to a triviality. In a *dynamic* real time system, no prior information is known about arriving tasks, so the scheduler has to make all of its decisions *online*. To do this, the scheduler is invoked at startup with the initial task set and at various times thereafter. In general, finding optimal schedules for all but the simplest of scenarios is difficult [Mok83].

2.1.3 Periodic Tasks

Periodic tasks are of interest in scheduling because they can be modelled easily and also because many of the activities which occur in control systems are periodic. [Liu73] presents the Rate Monotonic (RM) algorithm which provides a means of scheduling off-line a set of tasks given the following assumptions:[1]

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

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

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

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

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

[1] These assumptions are quoted directly from [Liu73].

Using RM, a schedule is effected by assigning static priorities to tasks such that tasks with higher arrival rates have higher priorities. The tasks are executed preemptively in a system with a dispatcher which always runs the highest priority, runnable task. The utilisation U of the processor by m tasks can be calculated as $U = \sum_{i=1}^{m}(C_i/T_i)$. It is shown that if $U < \ln 2$ then the schedule is feasible. In practice, the constraint $U < \ln 2$ is conservative, and there are many task sets which can be scheduled using RM whose utilisation exceeds $\ln 2$. [Lehoczky89] extends these results and provides a necessary and sufficient condition for determining whether or not a set of periodic tasks is schedulable using the RM assignment of fixed priorities. [Liu73] also presents a deadline driven algorithm in which the task whose deadline is nearest is assigned the highest priority. It is shown that this algorithm produces a feasible schedule for a given set of m tasks if and only if $\sum_{i=1}^{m}(C_i/T_i) \leq 1$.

Recent developments in the theory of scheduling hard real-time systems have enabled some of the assumptions made in the development of the RM algorithm to be relaxed. [Audsley91] provides a schedulability test for tasks whose deadlines are less than their period, thus removing part of the constraint (A2). Giving a task a tight deadline can be used to control the maximum amount of jitter experienced by the task.

Allowing tasks to synchronise via semaphores dispenses with (A3). This can lead to *priority inversion* where a higher priority task is blocked on a semaphore which is held by a lower priority task — a situation which violates the assertion that at any time the highest priority, runnable task is running. [Sha90] and [Nakamura93] describe protocols which can be used to limit the amount of priority inversion experienced by tasks which use semaphores for synchronisation.

Assumption (A4) is still required in many techniques for scheduling hard real-time systems. For tasks whose run-time varies, the maximum possible run-time required by the tasks is determined and used in any scheduling calculations.

Assumptions (A1) and (A5) constrain the analysis to periodic processes; dispensing with them enables the introduction of sporadic tasks into the system.

2.1.4 Sporadic Tasks

[Lehoczky87] presents the *Deferrable Server* (DS) algorithm for processing sporadic tasks in a system of periodic tasks which has been scheduled using RM. DS creates a server task τ_s with period T_s and computation time C_s, which is allocated a static priority P_s according to RM. At the beginning of each period, τ_s is allocated C_s processor time with which to process any sporadic tasks which may arrive during

that period. Any time which is remaining at the end of the period is lost. DS provides a predictable response to sporadic tasks while maintaining the deadlines of the periodic tasks.

[Dertouzos74] shows that the Earliest Deadline First (EDF) algorithm is optimal in the sense that if any other algorithm can produce a feasible schedule for a set of tasks whose request times, computation times and deadlines are known, then so can EDF. In contrast to static priority algorithms, EDF is suitable for use in dynamic systems because it requires only a knowledge of the task deadlines to produce optimal schedules.

2.1.5 Overload Behaviour

It is useful to compare the behaviour of some of these scheduling algorithms when the system is overloaded. Using RM, the task with the longest period will be the first to miss its deadline as the system becomes overloaded. This may not be what is wanted and is a result of the priorities used by RM failing to take into account the *importance* of tasks. [Sha86] presents *period transformation* as a solution to this problem. Using period transformation, the priority which a task is assigned by RM is controlled by altering its period and computation time. A task with run-time C and period T which is assigned priority P by RM is transformed into an equivalent task with shorter run-time C/k and period T/k for some $k > 0$ to which RM will assign a priority $P^* > P$. Priorities can also be decreased by increasing C and T.

Using EDF, the manner in which the system degrades is not easily predictable. [Miller90] presents a predictive deadline scheduler which uses EDF to schedule tasks, and additionally assigns to each task an explicit measure of the importance of the task to the system. Tasks are required to maintain an estimate of their execution times and the scheduler uses these times in conjunction with deadlines and importance to create feasible schedules during transient overload situations.

Both of these methods are retroactive; they allow the system to overload first then shed load according to some specification. Period transformation is a (somewhat artifical) means of conveying this specification to the scheduling algorithm. On the other hand, the deferrable server DS is proactive and avoids transient overload by explicitly controlling the amount of processor time which a sporadic task is allowed to consume.

Both RM and the predictive deadline scheduler need to know the computation time of tasks and if these times are bursty, RM will produce schedules with a low processor utilisation. Both cause less important tasks not to receive any processor time at all

during overload. The construction of some CM applications can be simplified if they are given some minimal amount of processor time during transient overloads; at the very least, they can detect the passing of time and maintain synchronisation. In principle, DS affords such behaviour during overload; regardless of the arrival rate of the sporadic task, the computation time allocated to DS will always afford a certain minimum amount of service. In practice though, DS is scheduled using RM and is as susceptible to processor starvation during overload as other periodic tasks.

2.2 Continuous Media and Real-Time

The preceding discussion has presented a number of techniques which have been developed in the context of HRT systems for handling temporally sensitive applications. It is necessary to consider the prospect of incorporating into a workstation operating system, sufficient HRT capability to be able to support CM applications in addition to the workstation's usual job mix. This could be done by a simple partitioning of the processor resources using priorities. Assigning to HRT tasks priorities which are higher than best effort tasks would enable the HRT tasks to obtain their execution time guarantees, while enabling the best effort tasks to make use of any of the remaining cycles. Before running a HRT application, a schedulability check such as that provided with the RM priority assignment could be used to decide whether running the new application would cause the current schedule to become infeasible.

2.2.1 Hard Real-Time and Best Effort Job Mixes

It has been shown [Tindell93] that, for a job mix consisting of 50% hard real-time periodic processes whose deadline equals their period and 40% sporadic soft real-time load, the response times of the sporadic load would be the same as those observed if the processor were running only the 40% sporadic soft load. This result could be applied to the design of a multi-media workstation to produce a system in which, provided that the multi-media activities consume 50% or less of the processor cycles, they will not be noticed by someone working on the same machine in a sporadic manner, such as when using an editor.

An example of a HRT, CM application environment is described in [Jeffay92], where the YARTOS real-time kernel is being used to support live digital audio and video within a conferencing environment. The requirements placed upon this system are that it provide a service as good as that which can be provided by a conventional analogue system. This means that video frames have to be displayed every 33 milliseconds, buffering has to be kept to a minimum, zero or one buffer being allowed

within a connection, and great importance is placed on not dropping any frames.

These requirements are extreme and are likely only to be encountered when it is desired to produce a high quality display of live video. To satisfy them, YARTOS draws heavily upon the techniques used in the design of hard real-time systems to provide:

- support for shared resources to ensure that priority inversion does not occur;
- an algorithm for determining whether a given workload can be guaranteed correct execution and;
- an algorithm for sequencing tasks on a processor which guarantees that tasks will meet their deadlines.

This system is capable of handling demanding application requirements but obtains this capability at the cost of a low utilisation of system resources; there are feasible sets of resource requirements which the schedulability checking algorithm will not guarantee.

Such approaches may be acceptable in an isolated machine, where high quality CM presentations are required, but they have a number of drawbacks in a dynamic environment such as a networked workstation. Firstly, they afford only a low level of resource utilisation for the system's real-time task set. Secondly, the techniques used in HRT scheduling algorithms assume that the worst case resource requirements of each HRT process are known. In the case of processing compressed data from a live video source, it is difficult to estimate accurately such worst case requirements. Thirdly, if the applications which are being scheduled make use of a multi-service network to transport CM, and the network provides temporal and logical guarantees which are weaker than those which are required by a distributed HRT application, then the end point applications will be unable to meet their deadlines due to the data not being delivered on time. Finally, as the HRT load increases, the SRT tasks become starved of resources. In such a system, the only way to ensure some degree of timely behaviour from a process is to include it in the HRT task set.

To gain an insight into a more flexible approach to resource allocation, the following section examines some of the methods proposed for the allocation of bandwidth in the multi-service networks which are being designed to transport CM data.

2.3 Networks

A considerable amount of research is currently being undertaken to develop multi-service networks which are suited to the timely transport of high bandwidth data such as CM. This section examines some of the methods used for allocating network resources to provide a suitable means of transport for CM data.

2.3.1 Transfer Mode

In networking terminology, a channel's *transfer mode* refers to the manner in which messages are multiplexed onto the channel. Using *Synchronous Transfer Mode* (STM), a fixed proportion of the channel bandwidth is allocated to each of the messages waiting to be sent. This is done when the network is configured by dividing each period of time on the channel into a number of slots, allocating a slot for the transmission of messages from each source, and sending a slot's worth of data from each source once every period. Using *Packet Transfer Mode* (PTM), the whole of the channel bandwidth is allocated to a message for the duration of its transmission. *Asynchronous Transfer Mode* (ATM) is a compromise between these two in which channel time is divided into a number of small, fixed-length slots or cells.[2] Any number of cells may be allocated at the next cell boundary for the transmission of a message.

Each of these transfer modes corresponds to a particular scheduling strategy. STM effects a preemptive, round-robin strategy, transmitting slices of multiple messages so that they appear to be being sent at the same rate. PTM corresponds to a non-preemptive strategy — once the transmission of a message has been started, the channel is allocated until the entire message has been sent. The regular, frequent cell boundaries characteristic of the ATM provide points throughout the transmission of a message at which the transmission may be preempted, so ATM corresponds to a preemptive scheduling strategy, which allows bandwidth allocation decisions to be made at every cell boundary.

In their raw forms, continuous media typically exhibit a Constant Bit Rate (CBR); Audio can be represented as a periodic stream of 8-bit or more samples and video as a periodic stream of frames of fixed size. They also consume a large amount of bandwidth so use is often made of the fact that they contain redundant information to transform them into compressed representations. Silence suppression within an audio stream is a simple compression mechanism. Video compression techniques such

[2] A 5-byte header and a 48-byte payload have been chosen for B-ISDN, giving a 53-octet cell size.

as H.261 [CCITT90] and MPEG [ISO/IEC91] make use of more complex mechanisms such as chrominance sub-sampling, quantisation and selective discarding of spatial frequencies and motion compensation to produce reduced bandwidth representations of the original stream. These techniques work because of the informational property of CM. Two important consequences of such encodings are that the data streams which they produce may not necessarily be periodic and often will have a Variable Bit Rate (VBR). In the case of silence suppressed audio, the data samples may only be periodic during the talk intervals. In the case of motion compensated video, data may not be produced when there is no difference between a frame and a number of its successors; the picture content of differing frames may also require different numbers of bits to encode.

While STM networks are well suited to carrying CBR traffic and are able to provide delay and jitter guarantees when doing so, they are less well suited to carrying VBR traffic due to their inability to allocate bandwidth dynamically. While PTM networks are able to allocate bandwidth dynamically, their inability to do so at short notice and preempt long packets means that they cannot provide the delay and jitter guarantees which STM networks can. The ability to dynamically allocate bandwidth at every cell boundary makes ATM networks suitable for carrying VBR traffic but in order to maintain delay and jitter guarantees, the network bandwidth has to be allocated appropriately. Figure 2.4 shows the effect of transmitting two temporally constrained messages m_1 and m_2 using each of the three transfer modes. m_1 arrives at $t = 0$, requires 10 units of time to transmit and has a deadline of

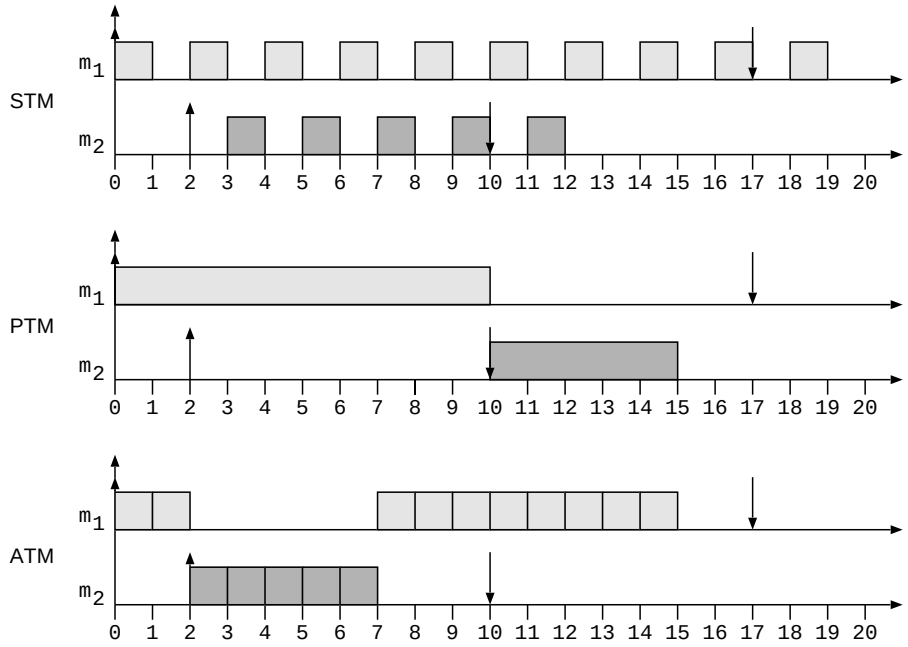


Figure 2.4: Transmission of two temporally constrained messages.

$t = 17$. m_2 arrives at $t = 2$, requires 5 units of time to transmit and has a deadline of $t = 10$.

2.3.2 Bandwidth Allocation

As has been mentioned, the success of ATM networks will depend on how well network resources can be allocated to the traffic sources which use it. The following sections review some of the concepts that have been developed to facilitate this process. [Bae91] provides a more detailed coverage of the issues.

2.3.2.1 Quality of Service

From the point of view of an application, it is desirable for the network to behave predictably. For the network to do this, it needs to have some knowledge of the behaviour of each of the traffic sources it is carrying; the more detailed the knowledge about each of the sources, the better the network is enabled to allocate resources to them in a predictable manner. This situation is commonly abstracted by viewing the network as a provider of services and applications as users of these services. The level of service required by an application and provided by the network is described by a QOS specification. Essentially, a QOS specification has two purposes; firstly, it is used by applications to specify the behaviour which they require from any traffic they send over the network and secondly, it is used by the network to allocate the resources it needs to allocate in order to effect that desired traffic behaviour. Typical QOS parameters include delay, jitter, bandwidth and error rate.

2.3.2.2 Admission Control

Having been presented with a QOS requirement by an application requesting a connection, the network has to decide whether or not it has available sufficient resources to provide the desired service quality. If it does, then the application traffic can be accepted; if not, the application should be informed. This process of deciding whether or not to accept a traffic source is called *admission control.*

2.3.2.3 Policing

Once the network and application have agreed upon a particular QOS for some traffic, a two-way agreement is in force. The network, by accepting the traffic source and agreeing to provide the desired QOS to it, should try very hard to maintain

that level of service to the application. Similarly, the application, having requested a certain amount of bandwidth in its QOS specification, should not be able to consume any more bandwidth unless it is available and doing so will not interfere with other users of the network.[3] A *policing* function is required in the network to prevent this sort of interference.

2.3.3 Bursty Traffic

Much of the data handled by ATM networks will be bursty; figure 2.5 shows a plot of bits of data per frame of a video sequence which has been coded using an MPEG encoder.

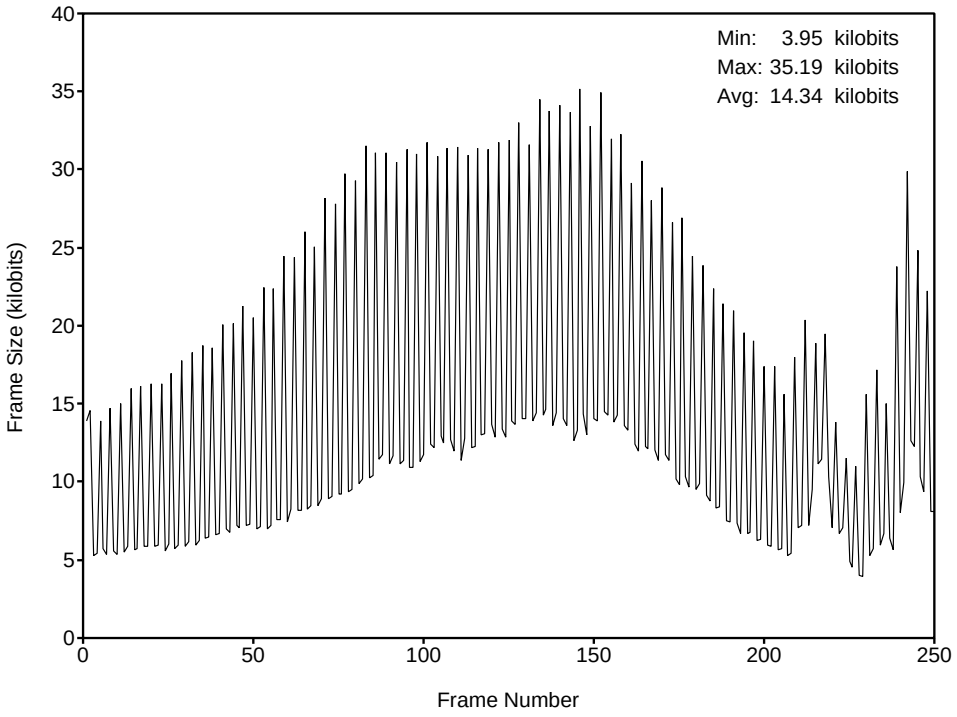


Figure 2.5: Bits per frame for video encoded with MPEG codec.

The encoder has produced three types of frames: I frames contain all the information required to reconstruct a single frame of the original video; P frames contain the information required to predict one frame of the original video from the previous I frame and; B frames contain the information required to predict the original frame

[3]This depends considerably on the viewpoint of the network designers. It may be the intention that only the requested amount of bandwidth ever be available regardless of the current utilisation of the network, or it may be that applications may try to make use of any spare bandwidth which fortuitously becomes available.

using the previous I or P frame and the next I or P frame. The video has been encoded using the sequence $[IBBPBB]^+$.[4,5] In its original form, each of the frames of this video sequence is approximately 164 kilobits in size. As shown by the plot, the bandwidth of the compressed data varies from 3.9 kilobits per frame to 35.2 kilobits per frame. This burstiness raises the question of how much bandwidth ought to be allocated within a network to carry such traffic.

2.3.4 Traffic Descriptors

A simple strategy for admission control and bandwidth allocation would be to allocate the peak bandwidth required by a traffic source. This would have the benefits of always leaving sufficient bandwidth available for carrying the traffic and would require only one number to specify the traffic's bandwidth requirements. One disadvantage of this as an allocation strategy is that allocation of peak bandwidth requirements for a large number of bursty sources leads to unacceptably low utilisation of the overall network bandwidth. Another is that there are applications where the peak bandwidth requirement of such a stream is unknown; if the source of video were a camera recording a live scene, for example, then the bandwidth would depend on the nature of the scene and in such a circumstance, the stream's peak bandwidth would have to be guessed.

2.3.5 Statistical Multiplexing

Network designers are investigating the use of *statistical multiplexing* techniques which allow less than peak bandwidth to be allocated to traffic sources and rely on carrying a large number of streams so that the short-term surplus bandwidth requirements of some of the streams can be met by allowing them to use bandwidth which is not currently being used by the remainder of the streams. Bandwidth in such networks can be allocated in a number of ways, each of which will provide different forms of QOS contracts to an application.

If the peak bandwidth B_p of a stream is known for the lifetime of the stream, then $B_a \geq B_p$ could be allocated and the cell loss rate for the stream would be close to

[4]The syntax $[pattern]^+$ is used to represent one or more occurrences of *pattern*.

[5]Note that, because the second frame is encoded as a B frame and this requires the previous I frame (frame number 1) as well as the next P frame (frame number 4) to decode, the transmission and decoding order is $I[PBBIBB]^+$. This accounts for the two adjacent peaks at frames 1 and 2 in the figure.

that of the hardware bit error rate.[6] Alternatively, some bandwidth $B_b < B_p$ could be allocated, resulting in a much higher cell loss rate. More often, the problem is stated in terms of knowing a tolerable cell loss rate, which is dictated by an application, and having to determine the bandwidth to be allocated, based on some knowledge of the traffic source's bandwidth requirements. This may be exact, but typically it will be in the form of some function of probabilities.

2.3.6 Service Degradation

The very fact that less than peak bandwidth has been allocated to a traffic source means that it is possible for such a source to produce more data than can be handled by the connection that it has established. When this happens, the network is forced to offer the user a degraded service quality. The manner in which service quality degrades is specific to the service user, and for this reason should be part of the original QOS requirement parameters.

For example, a video stream being viewed requires timely delivery of data, but is able to tolerate some loss of the data for small amounts of time. So an acceptable method of degrading service quality during bursts in the video traffic is simply to discard any excess data.[7] File system data, on the other hand, is sensitive to data loss, but often does not have very strict timeliness requirements, so an acceptable method of degrading service quality for this type of traffic would be to wait until sufficient bandwidth becomes available before sending any more data.

2.4 Real-Time and QOS

The HRT and QOS paradigms might appear to provide two different approaches to allocating resources in a timely manner — HRT provides stringent, deterministic guarantees that deadlines will always be met, and much of the current research focusses on the use of QOS to provide a probabilistic assurance that resource requirements will be satisfied a certain fraction of the time. It is worth remembering though, that an application's QOS requirement is a means of specifying the QOS which the application requires from the system, and can encompass requests for service qualities with such stringent guarantees as are provided by HRT. Given that

[6] The loss of 1 cell in 10^9 is a commonly cited figure.

[7] This has an effect on the coding method used. Coding techniques which strive for high compression ratios tend to be intolerant of errors within the coded data stream. In the example of figure 2.5, the effects of discarding data will propagate only as far as the next I frame.

in any real system, there will be a finite probability of failure due to possible software defects and a certain mean time between failures for any piece of hardware, one might also consider the difference between the "guarantees" offered by HRT and those offered by QOS when asked to provide an error rate close to that of the underlying hardware.

A comparison might also be made between SRT and QOS. From the definition provided in section 2.1, a SRT system is one in which deadlines are *sometimes* allowed to be missed. Perhaps it is because of the subjective nature this definition, or the perception that SRT systems are generally easier to build than HRT systems and not as useful, that much of the research in real-time systems has been directed at the HRT case. Many of the systems which might employ QOS for resource allocation are SRT. The desire to maintain some form of assertion (albeit one involving probabilities) about the long term frequency with which deadlines will be missed provides a means to reason about the behaviour of the system. From this perspective, QOS can be seen as being roughly equivalent to quantifiable, soft real-time.

2.5 Extending QOS

The suitability of QOS as a paradigm for resource allocation for CM traffic in multi-service networks has been identified, as has the relative inflexibility of HRT for this purpose. This will have implications for the software which is running on systems connected to these networks.

2.5.1 IMAC

[Nicolaou91] presents an extensive survey of multi-media systems and applications, then uses this to guide the design of an Integrated Multi-media Applications Communication (IMAC) architecture, the aim of the architectural approach being to describe the components within the system and how they interact. The IMAC architecture recognises the fact that, by their nature, most CM applications are distributed and has as one of its major goals the integration of CM data within a distributed computing environment. So the work presented focusses on system components for generating and presenting media (devices), transporting media (networks and protocols), and distributed processing (providing a base set of services which applications can use for distributed processing). The scope of the architecture is quite broad, but of particular relevance to the work which is presented in this dissertation is the treatment of QOS within the architectural design (section 5.2).

Within the IMAC architecture, the communications system is viewed as a service provider and applications are viewed as service users. In addition to this, system servers export interfaces which describe the set of operations which can be performed on them. The underlying system makes a series of *QOS offers* representing the QOS it can support. Each operation within an interface has associated with it a *QOS specification*, which is used to select the set of QOS offers which are acceptable for use with that particular operation. The invoker of an operation specifies a *QOS request*, which identifies a single QOS from those selected by the QOS specification for use with the current operation.

Throughout the system, QOS is specified in terms which are appropriate to the current level of abstraction, so at the application level for example, QOS appropriate to a video stream might be described simply as `compressed_pal`, while at the communications protocol level, it would more appropriately be described in terms of peak bandwidth, maximum acceptable delay and allowable error rate. To implement the conversion from one form of QOS to another, a number of constructs are introduced: a *QOS domain* delineates the scope of QOS offers, specifications and requests; QOS domains are implemented as a set of *QOS layers*, which represent the points at which QOS is provided; and *QOS mapping* is provided to convert QOS specifications from one layer to an underlying layer.

Two algorithms are provided for the purposes of negotiating QOS: one takes a desired QOS and recursively evaluates a set of conforming QOS specifications; the other determines a suitable QOS request from this set. In IMAC, these algorithms operate on the layers of a communications protocol stack and the results which they return are a private instance of a protocol stack which will provide the QOS desired by the application.

Two important points derive from the QOS paradigm presented in IMAC. The first is that QOS is always understood to extend between the end points of any operation; this is a simple consequence of end-to-end arguments in system design. The second is that, while the discussion of QOS within the IMAC architecture focusses on the services provided by a communications system, the algorithms used are quite generic and allow other system components to be represented as QOS layers for the purposes of end-to-end QOS negotiation.

2.5.2 QOS-A

[Campbell93] presents an integrated *Quality of Service Architecture* (QOS-A) in which QOS is used as the single paradigm for resource allocation and scheduling throughout a system. The discussion focusses on the provision of QOS guarantees

to support CM communications and presents a number of requirements for a QOS-A. A number of points germane to the current discussion are raised.

Within QOS-A, application QOS requirements need to be mapped through all system layers to the network so that support for QOS can be provided at all layers in the form of end-to-end QOS negotiation, admission control, policing and monitoring. To do this will require support from the operating system in the form of scheduling resources such as processor cycles and memory.

While much of the current work concentrates on the use of QOS in networks and protocols to provide communications guarantees, a generalised QOS-A ought to be extensible to include other areas of QOS provision such as real-time control systems.

A consequence of focussing on the use of QOS in networks is that QOS is being seen as a service provider issue, and little attention is being given to service user issues. This is illustrated by the observation that current notions of QOS provide little or no feedback to applications when QOS changes.

2.5.3 End-to-End QOS

Both IMAC and QOS-A emphasise that QOS must be maintained between the end points of distributed CM applications. There is little point in having the network deliver CM data according to some QOS requirement if the devices or applications at the end points are not given the resources to handle the data in a corresponding manner. This provides the motivation for extending QOS from the network into the operating system. To this end, the chapters following investigate the consequences of using within the operating system, resource allocation strategies similar to those used within the network.

2.6 Summary

The use of HRT and QOS to schedule resources in a timely manner has been reviewed, and both have been compared with respect to their suitability for scheduling CM data. While HRT is capable of satisfying stringent timing requirements, it does not readily afford the flexibility required to take advantage of the unique properties of CM. While SRT tolerates missed deadlines, little attention is paid to quantifying the frequency with which deadlines are missed. The use of QOS for scheduling CM within networks has shown that it proves flexible enough to accommodate a wide range of scheduling demands and so is more useful for managing CM. End-to-

end QOS requirements in distributed applications dictate that the operating system be able to provide QOS guarantees at all of the layers used by the applications. This includes scheduling of the system resources used within the communications protocol stacks as well as those needed by applications. The remainder of this dissertation will consider the use of QOS as a means of scheduling applications within a workstation operating system.

Chapter 3

QOS in Operating Systems

The previous chapter has compared QOS with real-time scheduling paradigms and discussed the advantages of QOS when used to schedule CM applications. The purpose of this chapter is to investigate in some more detail, the means by which QOS might be supported within an operating system together with some of the implications of doing so. To begin with, a simple CM application is described. This will be used as an example in later sections.

3.1 Example Decoder Application

The example application decodes a compressed video stream and displays it directly on a memory mapped frame buffer. Each frame of the video stream is compressed using the JPEG [Wallace91] still picture compression algorithm and the decoder performs decompression and display of the frames at the rate at which the video was originally sampled. The application decodes compressed frames in a number of steps. The input bit stream is expanded into a sequence of 8×8 tiles of Discrete Cosine Transform (DCT) coefficients using a Huffman decoder and a run-length decoder. Each of the coefficients in a tile is multiplied by a corresponding entry in an 8×8 quantiser matrix. An 8×8 Inverse DCT (IDCT) is performed on the coefficients to retrieve a tile of level-shifted pixel values which are then adjusted to absolute pixel values. The process is repeated until all of the tiles for one frame of video have been decoded whereupon the frame is displayed.

The encoding process is essentially the reverse of decoding; a tile of absolute pixel values is level-shifted, transformed using the DCT and each of the coefficients divided, using integer division, by a corresponding entry in the quantiser matrix. The quantising step truncates small coefficients of the higher spatial frequencies, produc-

ing large numbers of zero-valued coefficients in the quantised tile. These coefficients are run-length encoded then Huffman encoded to produce the output bit stream.

The degree of compression obtained can be controlled by scaling the quantiser matrix during the encoding phase. The implementation used in the encoder controls this scaling by means of the Q factor. Prior to encoding or decoding, each entry $q_{ij} (1 \leq i, j \leq 8)$ in the quantiser matrix is scaled by the Q factor so that $q_{ij}^Q = (q_{ij} \times Q)/100$ with the resulting values truncated to the range $(1, 32767)$, and the q_{ij}^Q are used during the encoding and decoding process. The same Q factor is used to decode an image as is used to encode the image. High Q factors produce high compression ratios and more distorted images; low Q factors produce low compression ratios and less distorted images. Typical Q factors used for compressing video range from 30 to 200.

Section 2.3.3 demonstrates that one of the results of compressing a video stream is that the bandwidth required to transport the compressed representation is bursty. The JPEG decoder application exhibits this property and reveals another interesting fact. A video segment was sampled at 15 frames per second and a resolution of 160 pels wide and 112 pels high, each pel being represented by 8 bits of grey-scale information. In its raw form, this stream requires a bandwidth of $160 \times 112 \times 8 = 143.36$ kilobits per 67 milliseconds to deliver the stream to an application. The application then requires 653 microseconds of processor time if it is to display each frame by copying it into a frame buffer.[1] In this case, a CBR stream is being delivered to an application which requires a fixed rate of processor resource to display the stream. The usages of the bandwidth and processor resources are both periodic and constant.

The stream was then compressed using the JPEG encoder and presented to the JPEG decoder for displaying. The graph in figure 3.1 plots frame number versus the number of bits in the compressed frame and the time required to decode the frame. The graph shows that not only has compression made the bandwidth of the video stream bursty, but that the execution time required by the decoder to reconstruct the frames has also become bursty.

The decoder is a simple application, but has the properties which are typical of the applications which this work is aimed at scheduling. Each frame must be displayed at the appropriate time and the resource requirements of the application vary considerably with time. The temporal requirement would make it a good candidate for scheduling using conventional real-time techniques but the burstiness would cause them to obtain poor resource utilisation. The remaining sections discuss how QOS can be incorporated into an operating system and how its use interacts with applica-

[1] The measurements presented in this section were made on a DECstation 5000/25.

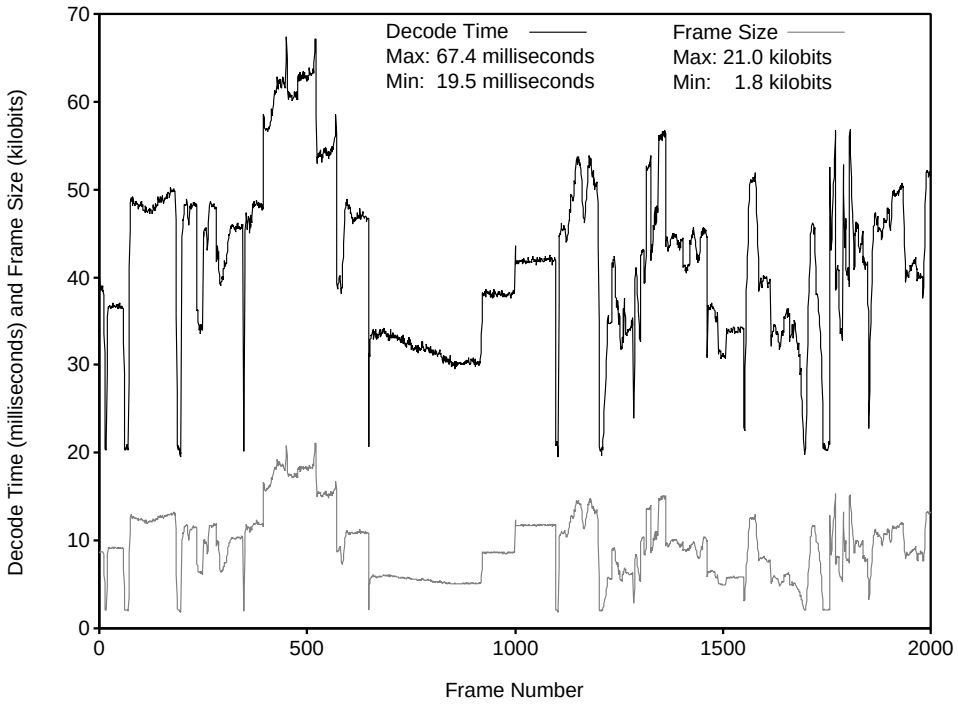


Figure 3.1: Decode time and bits per frame of compressed video.

tions such as the decoder. The discussion commences by presenting some definitions.

3.2 A Definition of QOS

In its common, accepted usage, *quality* connotes a particular characteristic, as well as the degree to which a characteristic is present. Within the context of the operating system being a provider of services such as processor cycles, memory, I/O operations and so on, *quality of service* is used to refer both to the kind of a service being provided, and to the extent to which that service is made available to an application. This extent is quantified by the amount of the service which is provided and the times at which it is provided.

The provision of services in a timely manner requires appropriate allocation of system resources so, in their most basic forms, the QOS q_d desired by an application will describe the application's resource requirements, and the QOS q_p provided by the system will be the result of the resource allocation scheme used within the system. q_d and q_p can be represented by vectors of resource requirements $\vec{q_d} = \left[\ r_1^d(t) r_2^d(t) \cdots r_n^d(t)\ \right]$ and $\vec{q_p} = \left[\ r_1^p(t) r_2^p(t) \cdots r_n^p(t)\ \right]$ where $r_i^d(t)$ is the amount of resource r_i desired by the application and $r_i^p(t)$ is the availability of resource r_i which was provided by the system. The $r_i(t)$ are called *QOS parameters*.

3.3 Elements of a QOS Implementation

Figure 3.2 shows how an operating system might be constructed to provide QOS for its applications. In this diagram, the application is either a client or a server process.

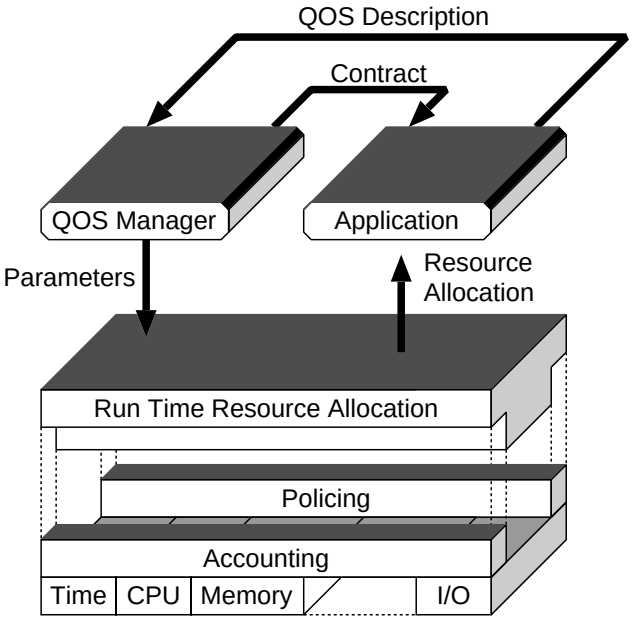


Figure 3.2: QOS management entities and their interactions.

The purpose of the QOS manager is to come to some agreement with processes about the QOS which will be delivered to them. An application presents to the QOS manager a description of its requirements and the QOS manager converts this to a set of parameters. The QOS manager then decides whether or not it can accommodate this request; it may be able to do this using only the QOS description or it may need the information provided by the QOS parameters. If the QOS can not be provided, the application is informed and the user may be informed by the application or the QOS manager. If the QOS can be provided the resources required to provide that QOS are noted and the application is informed of their availability. A contract then exists between the application and the system. The QOS manager communicates the application's QOS parameters to the Run-Time Resource Allocator (RTRA), which uses them in conjunction with some knowledge of the current time to allocate the resources to the application. Effective resource allocation will require some means of accounting for the resources used by an application. This will enable the resource allocator to police an application's use of system resources so that, when resources are scarce, it cannot use more than it has been allocated.

3.4 Contracts

Among the most important factors in a QOS implementation is the nature of the contractual agreement which exists between the system and an application. The existence of such a contract enables certain assertions to be made about the run-time behaviour of an application. These assertions are useful to the writers of applications, because they supplement the semantics of the standard definition of the operating system interface.

3.4.1 Virtual Processors

One view of an operating system is that of a layer of software which abstracts the physical hardware to a Virtual Processor (VP) more amenable to the applications which run on it. Applications are written so that they are aware only of the Virtual Processor Interface (VPI) provided by the operating system and do not necessarily know about the existence of any other activities within the system. For example, [Leffler89] describes the UNIX VPI as consisting of a set of system calls, a vector of interrupt or signal handlers and a mask for enabling and disabling the delivery of interrupts. The base semantics of this VPI are defined by the programmer's reference manual. These include neither the time which any of the requested operations will take nor any notion of the rate of progress which the application will make during its execution. Without this information it is difficult for the application writer to make any assertions about the temporal behaviour of an application.

The purpose of a QOS contract is to augment the semantics provided by the system VPI so that stronger assertions may be made about an application's run-time behaviour whenever the contract is in force. The terms of the contract are specific to a particular instance of an application and determine the assertions which can be made. To gain a better understanding of the QOS contract, it is worth noting that the ability of a scheduling algorithm to decide whether or not an application will run on time is determined by the amount of knowledge which it has about current availabilities of system resources and the resource requirements of the application.

3.4.2 Hard Real-Time Systems

In a HRT system, application resource requirements are known exactly for all time so a scheduler can, given sufficient time, determine whether or not a feasible schedule exists and if so, produce a schedule. In such an environment, the contract between the system and the application is quite rigid; the application, when it runs, will

always receive the resources it needs and if it does not, a fatal system error has occurred. The QOS afforded by such a contract is always good enough for the application to behave correctly both logically and temporally. Applications are written so that, once they are scheduled, there is never any reason to expect that their resources will not be available at the times they are needed, and timeliness is always guaranteed.

3.4.3 Best Effort Systems

A *Best Effort* (BE) system, such as is typically run on current workstations, assumes almost no knowledge of application resource requirements and provides no mechanism for applications to specify their temporal requirements. The schedulers in such systems make use of multilevel feedback queues to try to improve interactive response times by giving a higher priority to programs which do a lot of I/O, while also trying to give compute bound programs access to the system resources for comparitively longer times. The contract which exists between such a system and an application asserts that the VPI will be logically correct, but does not make any assertions regarding the timeliness with which resources will be made available to the application. The QOS afforded by such a contract is variable and fluctuates with system load. Applications are written without any requirement for resources to be made available to them in a timely manner and the operating system provides little or no direct support for an application to determine its progress.

3.4.4 Dynamic Real-Time Systems

In between these two extremes lay a range of systems in which less than complete knowledge of application resource requirements is available. This may be due to the inability to measure them accurately, or simply the inability to predict the future. Overall, the exact behaviour of such systems may be unpredictable, but the applications which they support still have temporal requirements. The nature of the contract which exists between the system and an application is less than absolute, in that the application may not always be guaranteed to obtain all of its resources every time it needs them. The assertions which may be made about the execution of such an application are of the form "99% of the time, this program will meet its deadlines," and are less than absolute but do serve to quantify the application's run-time behaviour.

Application writers will need to keep in mind that the quantity of resources available to them will change with time. This can be made easier if the system makes

available to applications some information about the amount of resources which are being made available to them whenever they are running. The information could be presented to the application as a sequence of upcalls or signals incorporated into the VPI for this purpose.

This represents a significant departure from the traditional view of the VPI provided by real-time systems. Rather than having the system guarantee a required level of performance to applications and writing applications which assume their resources will always be available, these systems try to provide resources in a timely manner informing applications of their availability, and applications attempt to produce the best results they can using the notification and resources which they are given.

3.4.5 Implications

The definition of QOS given in section 3.2 is quite broad and accommodates all three types of system just described, so it ought to be possible in a QOS based system to request any QOS ranging from BE to HRT. The former is relatively straightforward to provide and the ability of a system to provide the latter will depend on the current knowledge of resource requirements and the scheduling algorithms in use within the system. Both BE and HRT have been studied extensively and considerable experience has been gained with them [Coffman73] [Stankovic88], so they will not be considered in any detail in the remainder of this discussion except where necessary. Rather, attention will be focussed on the more interesting case of the type of QOS which might be used to support applications with dynamic real-time resource requirements.

3.5 Negotiation and Renegotiation

An important system parameter is the length of time for which any single QOS contract remains valid. A system in which contracts remained valid for the lifetime of all processes would not be suitable for a workstation because of the variability of the job load in such an environment.

Consider a situation in which 90% of a workstation's processor had been reserved and the user wished to run an application which requires 15% of the processor; there are two possible courses of action. The system can recognise that allowing the new process to run could adversely affect the contracts currently in force and inform the application that, while it can not have its desired 15%, it could be given the remaining 10%. The application then would have to decide, either by executing

some default decision or by consulting the user, whether it would want to proceed with only two thirds of the processor time which it ideally would like and then accept or reject the contract which it has been offered. These actions constitute a *negotiation* between the application and the system about the amount of processor time which will be made available to the application as part of its contract.

If this application could not run with any less than 15% of the processor and contracts were not allowed to be broken, the system would have to refuse to run it. If contracts are allowed to be broken, then the user can direct the QOS manager to release resources previously reserved and make them available to the new application. To help the other applications respond to this, they need to be informed of the fact that the contract which they held at the time was just broken. They can use this information to engage in *renegotiation* with the QOS manager for a new contract.

QOS renegotiation can also be instigated by an application. Consider the decoder application described in section 3.1. Suppose the input data were being produced by a camera viewing a live scene and encoding it for delivery to the decoder via a network. The processor time of the decoder is dependent upon the data which it is receiving, which is directly related to the scene which the camera is viewing. This dependency is compounded by the use of encoders which use motion compensation techniques to achieve high compression ratios. In such circumstances, an initial estimate of processor time which was based upon the data rate produced by a uniform, motionless scene would be wholly inadequate for decoding the data produced by a popular music video clip. It would then be appropriate for the decoder to note the number of frames which are successfully decoded on time, and renegotiate for more processor time when this number falls below a specified level. To ensure sensible use of system resources, applications ought to aim at maintaining the amount of processor resource requested as close as is reasonable to their actual processor requirements.

In the examples just described, scheduling decisions are made at two levels. At the negotiation phase, long term scheduling decisions are being made and, because they are made comparatively infrequently, they may be relatively complex and take longer to make. The RTRA makes only very simple decisions (which process to run next, for example), based on simple rules such as highest priority, earliest deadline or as recorded in a state table. Negotiation and RTRA within the operating system are parallelled in ATM networks by call acceptance control and cell-level scheduling respectively. Negotiation and call acceptance control both have the effect of factoring out the harder scheduling decisions and amortising their cost over the time between negotiations. In the limiting case, where an application is required to negotiate for resources at every arrival, the scheduling problem becomes that of scheduling a totally dynamic real-time system.

3.6 Parameters

In the system shown in figure 3.2 in section 3.3, the QOS manager converts QOS descriptions into a set of parameters which describe at a low level an application's resource requirements.

3.6.1 Resources Requiring Parameterization

To describe all of the resource requirements of an application would require a QOS parameter for each resource. In the case of the video decoder example, two obvious QOS parameters would describe processor and input bandwidth requirements. Because paging in virtual memory systems can introduce large amounts of jitter, the decoder might also like to specify that a certain minimum number of frames of physical memory be made available to it for its execution; this would constitute another QOS parameter. Some criteria are required for determining the set of resources for which QOS parameters are needed. An argument similar to that of section 2.1.1 would indicate that if, within a particular system, a resource is so plentiful that its utilisation remains sufficiently low, then there is no need to reserve it and no corresponding requirement to parameterise its usage. This set is specific to a system; resources which are scarce on one system may be plentiful on another and *vice versa*. The remainder of this discussion will be concerned with only the processor resource.

3.6.2 Parametric Complexity

Another matter which arises is the amount of detail which ought to be included in the QOS parameters. In the context of scheduling bandwidth in a network, the B-ISDN signalling protocol [CCITT92] currently acknowledges peak bandwidth and end-to-end transmission delay as being significant parameters. Other parameters such as jitter and burstiness are yet to be decided upon and it is not likely that this decision will result in an orthogonal set of individual parameters. More realistically, the service user will be allowed to select from a number of QOS classes. The evidence suggests that the more accurately the description of resource requirements is known, the more accurately resources can be reserved, but that doing so will require more detailed calculations. This motivates the use of simple, parametric descriptions of service requirements.

3.6.3 Processor Time Parameters

An application's *Processor Bandwidth* (PB) can be used to quantify its processor requirements. Such a bandwidth may be expressed as a percentage of the total available processor resource to obtain a measure of the utilisation of the processor by an application. A percentage however, does not provide any indication of *when* the processor resource is required, so the pair (c, t) is used as an equivalent representation, where c is the processor time allocated to the process every t seconds.

The decoder application is a periodic process and it is straightforward to see how a bandwidth can be specified for it. The upper timing diagram in figure 3.3 shows three arrivals of a sporadic process τ_1 whose minimum inter-arrival time is 4 milliseconds and whose deadline is 5 milliseconds after arrival. The process requires 2 milliseconds of processor time, so its peak processor bandwidth is $(2/4) \times 100 = 50\%$. The lower

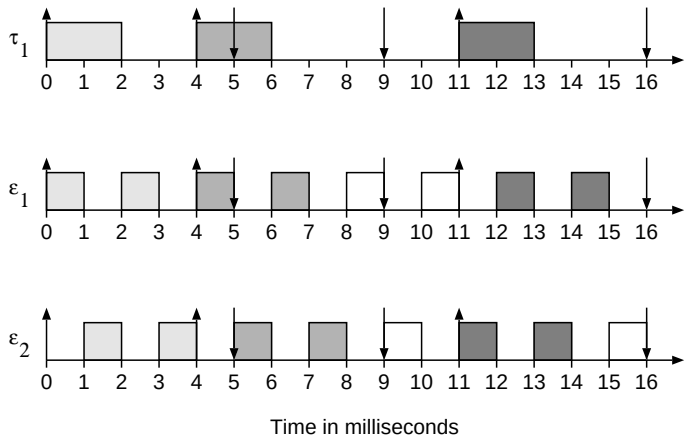


Figure 3.3: Sporadic process τ_1 and execution profiles ϵ_1 and ϵ_2.

two timing diagrams show possible execution profiles ϵ_1 and ϵ_2 of τ_1 when it has been allocated a processor bandwidth of $(1, 2)$. From the figure, it can be seen that the processor needs to be allocated to τ_1 both within a certain minimal time and at a certain minimum rate for ϵ_1 to meet its deadline. This places some restrictions on the granularity of the bandwidth parameters which can be used for ϵ_1 if it is not to be tardy. If bandwidth is periodically allocated as (c, t), and τ_1 requires C cycles after its arrival at time A to meet a deadline at time D, then a bound for (c, t) when $t \leq (D - A)$ would be $\lfloor (D - A)/t \rfloor \times c \geq C$. When $t > (D - A)$, this is no longer sufficient, and additional constraints that $c \geq C$ and C of the c cycles be allocated before D are needed.

While it is possible to meet temporal requirements by simply allocating a sufficient processor bandwidth, this will not always result in an efficient use of the processor. Figure 3.4 shows a feasible schedule for two sporadic tasks τ_1 and τ_2. A simple as-

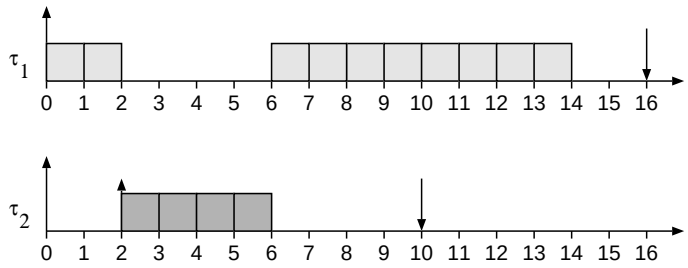


Figure 3.4: Two sporadic processes τ_1 and τ_1.

signment of processor bandwidths to these processes would allocate $(10, 16) = 62.5\%$ to τ_1 and $(4, 8) = 50\%$ to τ_2 which would overcommit the processor by 12.5%. In other words, to meet these deadlines by assigning a processor bandwidth to each task would require a processor which is 1.125 times faster than the one used to obtain the timing diagram of figure 3.4. Using a faster processor, the execution times could be reduced by a factor of 1/1.25, so that the bandwidths required would become $(10/1.125, 16) = 55.56\%$ and $(4/1.125, 8) = 44.44\%$ for τ_1 and τ_2 respectively. In this situation, processor bandwidth is too simple a description of the behaviour of the processes; use of a constant processor bandwidth assumes that the processes are periodic, while those in the figure are sporadic. In the special case of a periodic process whose deadline is the same as its period, processor bandwidth can be an accurate description of processor time requirements, and the process can be scheduled using RM. While processor bandwidth provides a convenient means of describing processor requirements, it must be remembered that, for some applications it is a simplified description of the actual requirements.

3.7 QOS Description

[Nicolaou91] and [Campbell93] suggest that, when requesting a particular QOS from a network, an application ought to provide a description of *what* it requires rather than how the QOS is to be provided. An example given is that of an application requiring a network connection over which a standard format video stream is to be carried; the application simply requests a video source with a QOS description of `StandardVideo` and it is left to the network QOS management functions to map this description to a set of network QOS parameters. This is reasonable because the network QOS management entities are likely to have some knowledge of what resources are required to support a frequently requested QOS such as a `StandardVideo` stream. Both sources recognise that this will not always be the case and that applications ought to be able to supply low level QOS parameters whenever necessary.

Were the QOS manager able to convert a description as provided by an application into a suitable set of QOS parameters, a number of advantages would be obtained. Firstly, an application could be moved among machines of varying speeds without having to concern itself with how much of the processor it needs on each of the machines; the QOS managers on various machines could contain machine specific information and scale the run time allocated to the application accordingly. Secondly, it provides a convenient mechanism for isolating the application's data dependent resource requirements.

3.7.1 Describing the Accuracy of Results

One difficulty when presenting the QOS manager with a description of some desired QOS lies in actually constructing the description. In the case of a segment of video or audio, subjective judgements of the quality of presentation need to be quantified. With respect to the decoder application, the Q factor provides a means of directly specifying the amount of compression desired, but does not provide any means of quantifying the image quality.

A simple method of measuring the quality of an image which has been processed using lossy techniques is to measure the absolute difference between the processed image and the original. This could be averaged over all pixels in the image, or a maximum value might be used. This would provide a simple, efficient indication of how accurately the measured image represents the original.

The temporal requirements of CM, together with the notion of an acceptable accuracy represent a two dimensional space within which the quality of CM may be quantified. Figure 3.5 shows the region of acceptable temporal and logical error for a hypothetical CM application. The application is deemed to have computed an

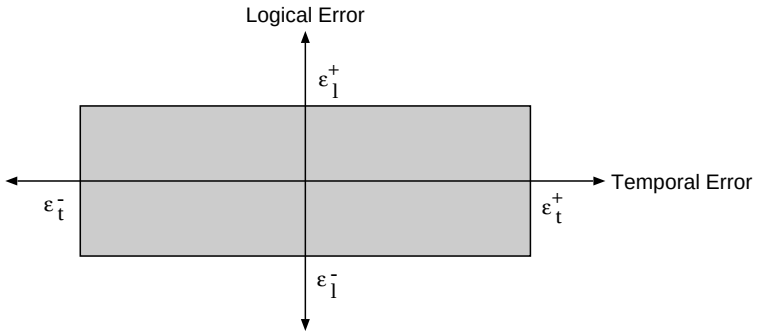


Figure 3.5: Region of acceptable error for a computation.

output of sufficient subjective quality if the output is within $[\epsilon_l^-, \epsilon_l^+]$ of the logically correct or ideal output, and it does this within $[\epsilon_t^-, \epsilon_t^+]$ of its deadline.

3.7.2 Mapping QOS Descriptions

A QOS description is useful if it can be used to infer some indication of the resources required by an application to process a particular set of data. For live data, a guess based on some previous experience will have to suffice. If the data is stored, it is possible to produce a profile of the data offline and use this in conjunction with a model of the application to produce an estimate of resource requirements.

Inspection of the example decoder application reveals that the time required to decode a frame of video is approximately proportional to the size of the frame. Figure 3.6 shows a plot of frame sizes in kilobits versus the time required to decode the frames for 2000 frames of video. Using the method of least squares, the line of

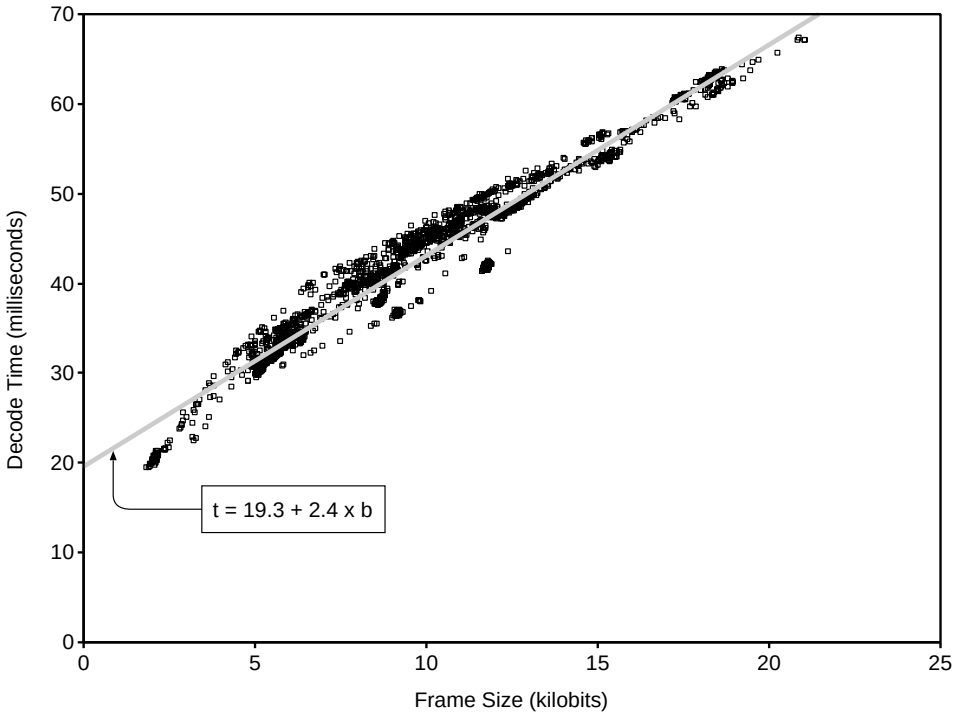


Figure 3.6: Frame size versus time to decode.

best fit is given by $t = 19.3 + 2.4b$ where t is the time required to decode the frame and b is the number of kilobits of compressed data in the frame. This line is a model of the data dependent execution time of the decoder application which, given the right information about the data which is to be processed, can provide an estimate of the time required to do so.

Figure 3.7 plots the predicted decode time per frame for a different segment of video along with the decode time measured. The model tends to predict less decode time than is required for some of the frames. This is a result of the line derived by the

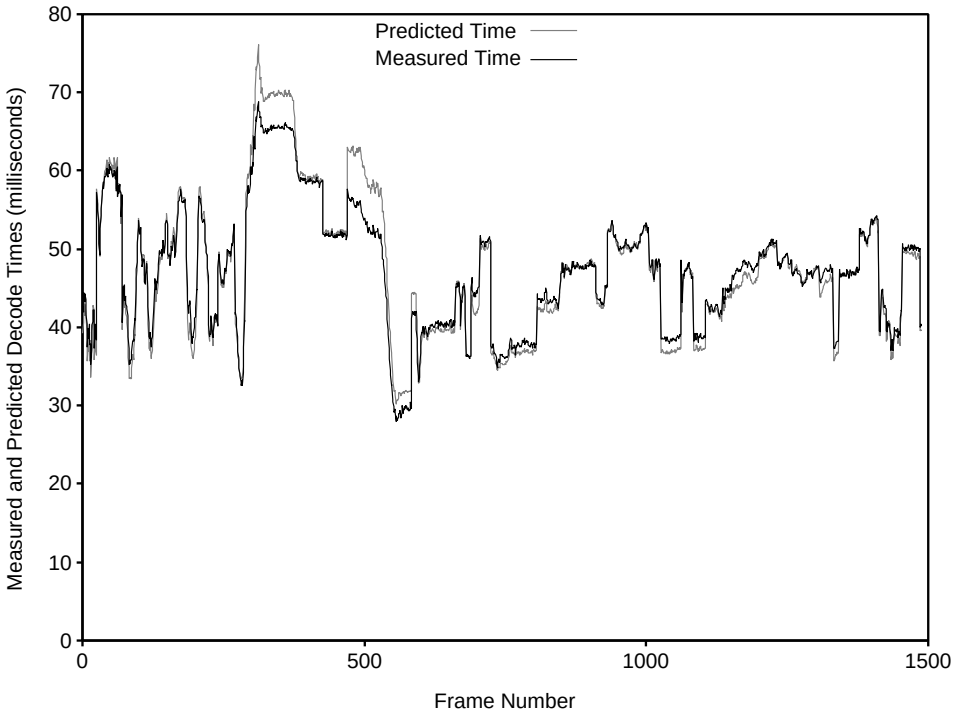


Figure 3.7: Predicted and measured decode times.

method of least squares not allowing for the large number of points which lie above it. While the model is too simple to predict an accurate decode time for a single frame, it does make reasonable predictions over a large number of frames. Raising the height of the line by increasing the value of the intercept on the decode time axis will cause the model to predict times which allow a greater percentage of the frames to be decoded completely.[2] A more conservative model such as $t = 25 + 2.4b$ can be used to determine an upper bound on the decode times. This particular model is quite primitive and more accurate models could be constructed based on a more detailed analysis of the decoding algorithm and data, but it does provide an estimate of the required processor time which is more realistic than the maximum value of 67.4 milliseconds.

Such execution time models can be used to map QOS descriptions onto a set of QOS parameters. For a particular decoder, an execution time model can be constructed. This model can then be used in conjunction with a stored video segment to predict the execution times required to decode the frames and these times used individually, or summarised in the form of a distribution of execution times. The key point here is that such a distribution would provide more information about the run-time behaviour of the decoder than just a simple maximum decode time. The prediction can be stored as a set of metadata associated with the compressed video segment.

[2]Section 3.8.3 will discuss the usefulness of this.

When requested to display a video segment, the decoder can identify itself and the video segment to the QOS manager, which can locate the relevant metadata and use this in its calculations to map QOS descriptions onto parameters.

For many applications, this amount of offline processing may not be warranted, but it is possible to envisage situations which might make use of it. A video distribution centre might store online a number of movies in a compressed format, and offer to a large number of mass-produced (and therefore identical) client systems a video-on-demand service. An execution time model can be created for the clients and the decoder which they run, and the model used to predict the execution time characteristics of the decoder when decoding a specific segment of video. The prediction might be done more quickly than in real-time by machines more powerful than the clients. In such an application, given the bandwidth required to transport the compressed video, it is not unreasonable to ensure that all of the clients run the same version of the decoder by storing a master version at the centre and transmitting a copy to the clients before the video data. A QOS manager within one of the clients can then be provided with the metadata for a video segment and this can be used to calculate parameters of the QOS required by the local instance of the decoder.

3.8 Run Time Resource Allocation

The method used by a system implementation to allocate resources at runtime will determine the type of VP perceived by applications. At the lowest level, the Run Time Resource Allocator (RTRA) multiplexes the available resources among those applications which need to use them. It does this using information in the form of QOS parameters supplied by the QOS manager. In the case of the processor resource, run time allocation is performed by the system dispatcher, a model of which is depicted by the diagram of figure 3.8. The dispatcher sees application resource requests as a series of task arrivals τ_i, which are placed in individual queues to wait until the processor can be allocated to them. Whenever one request has been serviced, the dispatcher, based on some service discipline which may be inbuilt or dictated by the QOS manager, selects a task from the head of one of the queues, then allocates the processor to it for some amount of time. The service discipline and allotted service times need to be chosen so that application QOS contracts are honoured.

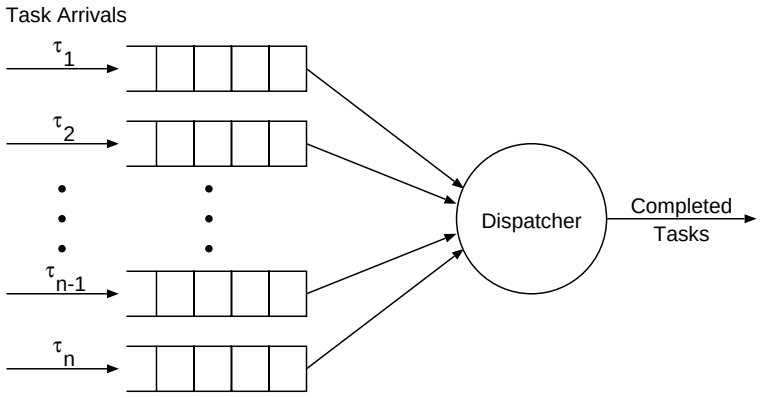


Figure 3.8: Run time resource allocation model.

3.8.1 Notation

This is a queueing system whose operation can be described using the notation $W/X/Y/Z$, where W is the distribution of interarrival times, X the distribution of service times, Y the number of servers and Z the service discipline. Among the values for W and X are D, indicating deterministic times and G indicating stochastic times. The use of G here is deliberate as there is at present little or no real data concerning the distributions of execution times and these are likely to be specific to each application. For the purposes of this discussion, the only resource which will be considered for allocation is processor cycles and the discussion will focus on single processor systems, so Y will represent the number of processors in the system which will usually be one. Service disciplines include First Come First Served ($FCFS$) and priority based (PR). The latter is extended to include nonpreemptive, priority based (NPR) and preemptive, priority based (PPR) disciplines.

This type of model has been used extensively in studies of the performance of time-sharing systems. [Coffman73] provides an introductory treatment of the subject, proposing that important performance measures for a given system include: the distribution of the number of tasks in a system; the distribution of the time a task will be in the system and; the distribution of the lengths of *busy periods*[3] in the system. In essence, the type of questions answered by performance analyses are the inverse of those required to maintain QOS contracts. Given a particular system configuration, performance analyses can determine the QOS which an application will receive. In providing QOS contracts, the desired application QOS is known, and a system configuration which maintains that QOS needs to be determined.

[3]A busy period begins when a job arrives to find an empty system and ends when the system again becomes empty.

3.8.2 Deterministic Models

A system in which arrival and service times are known could be modelled as a $D/D/1/PR$ queue. This type of model reflects the behaviour of a hard real time system such as might be scheduled using the RM algorithm and in which task arrivals are assumed to be periodic, of known period and task service times are fixed at the maximum of any of a task's service times. Priorities are assigned to each of the queues according to RM and the PR service discipline always services the task of highest priority. In such a system, the contract provided to an application is absolute.

3.8.3 Stochastic Models

It is often the case that exact or even maximal execution times for an application are not known; if the video being viewed by the decoder application were produced by a camera viewing a live scene, it would be difficult to estimate the maximum time required to decode a frame.

The frame decode times plotted in figure 3.1 show a large amount of variation from a minimum of 19 milliseconds to a maximum of 67 milliseconds; since the video was sampled at a rate of 15 frames per second giving an interframe period of 67 milliseconds, allocating the peak processor bandwidth of $(67, 67)$ to the application would consume 100% of the processor resource. The graph of figure 3.9 plots the run time allocated to the decoder versus the percentage of frames which it can decode in that time. It can be seen from this graph that allocating $(54, 67) = 79\%$ of the processor to the application enables 91% of the frames to be decoded on time.

Allocating less than peak resources means that the application now misses some of its deadlines (9% in this case). Even though the method of resource allocation is similar to that used by periodic HRT systems with priorities assigned according to RM, the system has become SRT. In this type of system, resource requirements (and hence QOS parameters) are not known exactly, but their distributions may be known. QOS requirements could be described in terms of the probability that a deadline will be met. Given the distribution of run times either derived from a model, or as a histogram resulting from a previous run, it is possible to calculate how much resource to allocate to meet the desired percentage of deadlines.

Allocation of a fixed amount of processor time to an application as done in the previous section is an example of a reservation strategy. A disadvantage of such strategies is that, when used in conjunction with bursty resource requests, their use can result in poor resource utilisation. Figure 3.10 plots frame number versus the

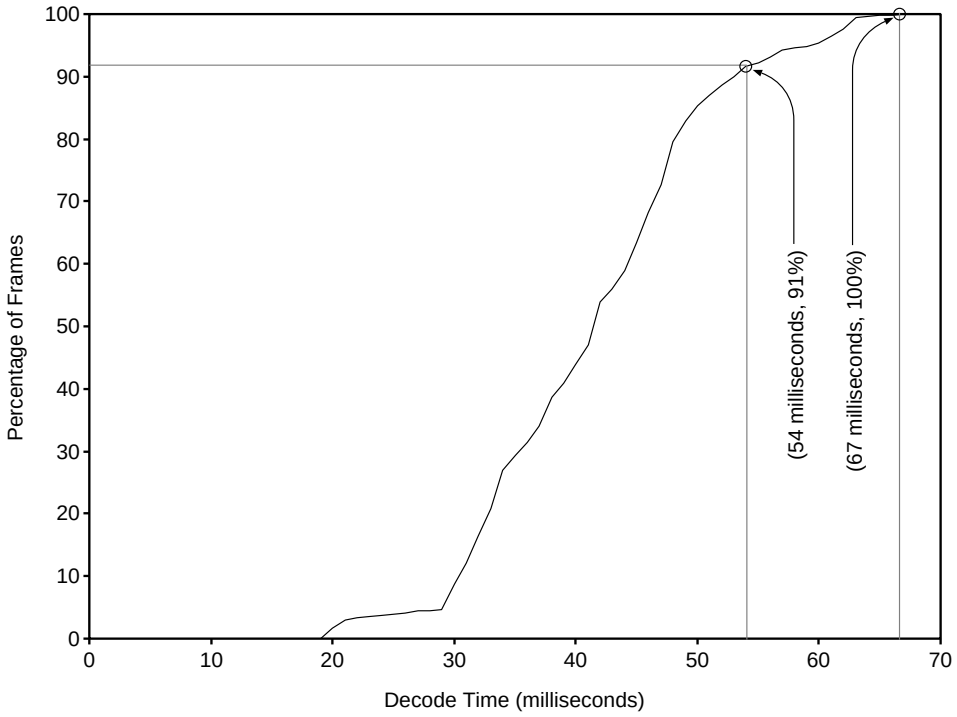


Figure 3.9: Run time allocated to decoder versus percentage of frames which can be decoded within that time.

time to decode the frame for two different video streams, along with the total time to decode both streams (this is just the sum of the two times). From this graph, it is seen that reserving peak processing times for the two streams would require $67 + 69 = 136$ milliseconds per frame, but the peak total processing time required is only 120 milliseconds per frame. Statistical multiplexing aims to exploit this property by allocating resources based on the combined requirements of a number of sources rather than their individual requirements. If statistical multiplexing is to be used for processor allocation, run time resource allocation can be modelled as a $D/G/1$ queue, assuming deterministic (periodic) process arrivals and generally distributed processing times. Within this model, an important QOS parameter would be the probability that the service time of a given arrival will exceed its deadline. A typical implementation would select a priority based service discipline on the grounds that these are known to provide higher resource utilisations. Additionally, the models presented so far have assumed that the arrival processes are deterministic and periodic. If CM data are presented to applications as motion compensated video or silence suppressed voice for example, then arrivals will be sporadic, requiring the incorporation of stochastic arrivals into the model.

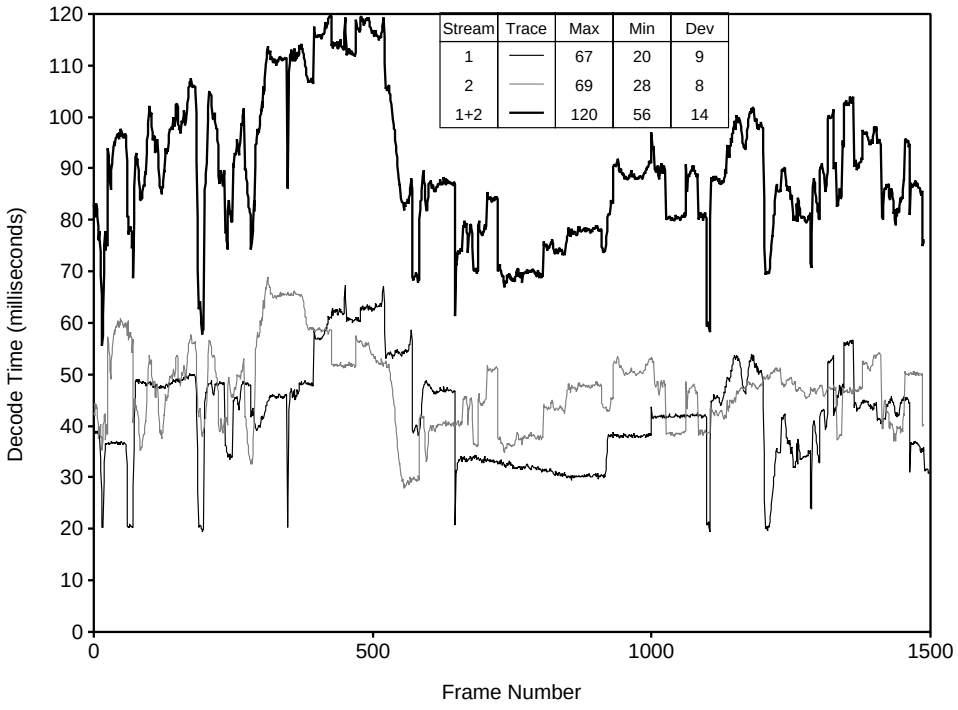


Figure 3.10: Frame number versus time to decode two compressed video streams.

3.9 Accounting

Ensuring that an application receives the QOS it requested requires a means of accounting for the resources used by the application. Current monolithic kernels already do this when they maintain a running total of the time spent by an application in both user and supervisor mode. Accounting is performed by charging the cost of the resources used to the accounting structure which is associated with the currently executing user-mode protection domain. In the case of a system structured as a number of clients and servers running on top of a microkernel, accounting becomes complicated by the fact that a single client may need to interact with a number of servers during its execution.

3.9.1 Credits

One approach to solving this problem is via a system of credits. An application is allocated credits at a rate determined by its PB and, whenever it interacts with a server, it nominates a number of its credits which are to be used for the interaction. These credits are debited from the application's account and deposited into the server's account so that the server may do work for each of its clients on the basis of the credits it has received from them. This approach separates the accounting

function from a process's address space. Its disadvantages include:

- it is possible for a server to use resources given to it by one client for other purposes;
- it requires some knowledge of whether a particular server is on the same system as the client or on a remote one and;
- it increases the overhead required for a client-server interaction.

3.9.2 QOS

Another approach is to have the client request a connection to a server with a particular QOS. The client and server then perform the negotiation and reservation actions described in previous sections. Thereafter, if the QOS can be supplied, the server knows the resources which will be required to service any requests which arrive on that connection. This has the advantages of not adding to the overhead of client-server interactions and fitting in well with the notion of end-to-end QOS even when the client and server are located on different machines. Among its disadvantages are that a server has to perform some of the QOS management functions itself, and that these include the policing functions, so poorly constructed servers may promise qualities of service which they do not deliver. This last problem is common to both approaches and currently may be addressed by observing that people will tend not to use badly-behaved servers.

3.10 Policing

Policing occurs at two levels in the system. At the low level, a mechanism is required to ensure that an application does not exceed its allocation and start consuming resources which are reserved by another application. An appropriately designed dispatcher can ensure that processor time is accurately meted out. At a higher level, a means is required to ensure that the writers of applications do not request excessive resources to ensure that their applications perform acceptably. Such behaviour will result in poor resource utilisation and the informed user of a workstation will tend to prefer programs which are more frugal with workstation resources. In a commercial environment, being charged for the resources used to run a program may achieve similar results.

3.11 Application Design

Since the QOS environment often does not guarantee absolutely to give an application all of the resources it requires, applications must be able to produce as good a result as possible with the resources which they are given. This is in contrast to real-time approaches of resource allocation, but in many respects is more realistic; to be able to offer absolute guarantees requires knowledge of the future. Applications are written either so that they assume they have a minimal resource availability as described by their QOS contract and try to make use of resources which fortuitously become available, or they always assume the best possible QOS and have the means of producing an acceptable result of poorer quality when peak resource requirements cannot be met. In either case, applications may be written so that they can produce results of varying quality.

3.11.1 Temporal Degradation

In timesharing systems, when resources are scarce, process execution degenerates by taking longer to complete. This form of degeneration is not appropriate for some CM applications, so an alternative is to decouple the rate at which processing is performed from the data arrival rate, select the most recent data and process them to completion. The decoder application could reduce its processor bandwidth from $(67, 67) = 100\%$ to $(33, 67) = 50\%$ by halving its frame rate.

3.11.2 Spatial Degradation

The processor bandwidth required by the decoder application could also be reduced by shrinking the size of the video format being presented. This could be done at either the video sink or the video source by subsampling.

3.11.3 Logical Degradation

The performance of some algorithms may degrade logically if they are capable of producing results of reduced but acceptable accuracy on time. These algorithms may be implemented as *imprecise computations*. [Liu91] identifies a number of different types of imprecise computation and presents approaches for scheduling them. A task is *monotone* if the quality of its intermediate results does not decrease as it executes. The *milestone method* records intermediate results and associated errors

at defined points in a computation, so that the intermediate result of a task is its value at the last completed milestone. A task can be constructed as a sequence of *seive functions*, one or more of which may be skipped during execution to reduce the amount of time it requires. The *multiple version method* employs a primary and an alternate version of a task. The primary version takes longer to produce a precise result while the alternate version produces an imprecise result more quickly. During transient overload, the system can execute alternate versions of tasks.

Tasks τ_i of these types are modelled as a mandatory subtask and an optional subtask o_i. Algorithms are presented for scheduling subtasks such that the m_i are guaranteed to run to completion. For each of the o_i, a processor time σ_i is allocated so that some function of their errors $\epsilon_i = o_i - \sigma_i$ is minimised. It is assumed that release times r_i, deadlines d_i and the computation times required by the subtasks m_i and o_i are known *a priori*. Typically the m_i are scheduled using a fixed priority assignment generated by RM which will guarantee their execution. In a dynamic environment, the o_i may be scheduled using, for example, EDF.

This type of scheduling can be provided by a QOS contract which offers applications a guaranteed minimum processor bandwidth with the condition that, should additional bandwidth become available, it will be offered to the contracted applications.

The success of these scheduling strategies will depend on an application's ability to make use of them, and the informational property of CM can be exploited to do this. [Ghanbari89] presents a layered video codec for use on a slotted ring network which is capable of offering fixed, guaranteed bandwidths as well as variable, additional bandwidth. The codec uses a transform based algorithm to produce a motion compensated stream of *guaranteed packets* and a DPCM based algorithm to produce a stream of *enhancement packets*. The guaranteed packets contain the basic picture and motion information and are allocated a fixed amount of guaranteed bandwidth. The enhancement packets contain the error difference between the original image and the image reconstructed from the guaranteed packets. The enhancement packet stream is bursty and is transmitted through a VBR channel when bandwidth is available. When the VBR channel allows transmission of the entire enhancement stream, a high quality picture can be reconstructed by the decoder using both the guaranteed stream and the enhancement stream. At the very worst, when the VBR channel does not carry any enhancement packets, the received picture quality degrades to what can be reconstructed from the guaranteed stream.

In the same way that layered coding makes use of the guaranteed and additional bandwidths in a network, *layered processing* can be used to take advantage of the processor time allocations associated with QOS contracts. At the application level, an example of the use of layered processing would be the presentation of video frames

to a display application such that each frame is decomposed into a number of layers, with each layer representing a milestone and successively refining the accumulated image. Another example would be the presentation of stereo audio to an application as two streams, one containing the sum of the left and right channels ($L + R$), the other containing their difference ($L - R$). The audio application can simply use ($L + R$) to produce a monophonic signal or, given some extra processing time can compute a stereophonic signal by adding the two inputs to obtain $2L$ and subtracting them to obtain $2R$. The advantage of using these types of computations is that they provide a direct control over the quality of their results by varying the amount of processing resource they are allowed to use.

Applications which use layered processing can benefit from a knowledge of the type of processing time which they are currently receiving. Consider an application holding a contract which provides it with a guaranteed minimum processor time and affords it additional time should any become available. If the application were informed of the type of processor time it is about to receive whenever it is given the processor, it could use this information to decide whether it ought to execute mandatory or optional code. This information can be provided by the operating system as part of the VPI seen by the application. Section 4.7 discusses the effects which this has on the design of the VPI.

The fact that CM are amenable to layered processing provides a different perspective on scheduling problems. Instead of determining the processor time required by an application and constructing schedulers to accommodate this processing time completely, layered processing and imprecise computations enable the system to dictate the processing time which the application will receive, trading off quality for computational resources.

3.11.4 Data Representation

The preceding section brings to light an important point about the representation of data used by applications which can vary the quality of their results. The manner in which an application degrades is specific to the application and this will directly affect the representational requirements of the data with which the application is presented. A layered coding scheme for video data which is to be viewed by a human could split the video into high and low frequency components and transmit these to a decoder. The low frequency components could be reconstructed to form the minimal quality picture and the high frequency components used to refine the picture. If the observer of the video were a program which did edge detection, then it might be better to transmit some high and some low frequency components for the basic picture and use the intermediate frequencies to refine it.

3.12 Summary

QOS has been presented as a paradigm for resource management within an operating system. It has been shown that QOS can meet a variety of resource allocation requirements ranging from those of hard real-time to soft real-time applications; in the case of soft real-time applications, a number of metrics have been presented which enable their QOS to be quantified. A means of mapping high level QOS descriptions onto low level QOS parameters has also been presented. It is recognised that the range of QOS which a system can provide is determined by the combination of the QOS manager and RTRA; the basic mechanisms required within an operating system to support these types of resource management include accounting and policing. QOS contracts may give applications fewer resources than they require, so the quality of any results produced may degrade. Providing applications with a knowledge of the current availability of their resources can help them control the manner in which their performance degrades. The remaining chapters describe the design and implementation of a system which provides this information, along with an example application which makes use of this information.

Chapter 4

Design Considerations for QOS

The basic, low level mechanisms required to support QOS in an operating system are run time resource allocation, accounting, policing and a means by which applications can determine the QOS which they are currently obtaining. This chapter describes design of an operating system which incorporates these mechanisms.

4.1 Approach

While it would be possible to take an existing operating system or kernel and add to it the mechanisms required to support QOS, it is easy in doing so to lose track of the implementation goals; there is the possibility of having to alter the QOS mechanisms to fit them into the system thus obfuscating their exact requirements, intentions and objectives. Once they have been implemented, a large or complex system might also impede accurate measurement of their performance. Additionally, the mechanisms being discussed here are present at the very lowest levels of an operating system, so their use may have repercussions throughout large portions of the system; much of the time spent propagating these side effects could be unrewarding.

Depending on the number and complexity of services provided, the design of a complete operating system *ab initio* can be a large task. Since the scope of the work presented in this dissertation is necessarily limited by time and resources, the following sections will focus on those issues in the design which are pertinent to QOS and support for CM. Nevertheless, it is still possible to design and implement enough basic system functionality to demonstrate the viability of the approach.

This motivates the approach taken in this and the following chapters in which the design, implementation and evaluation of an experimental kernel is presented. The

kernel is tailored to the provision of mechanisms which provide the necessary system-level support for QOS guarantees.

Current experience with operating system design evinces the advantages of structuring operating systems as a group of client and server processes which are supported by a microkernel. The microkernel provides exactly the functionality required by the clients and servers to interact both among themselves and with the system hardware. This functionality consists of protection between the address spaces of different processes, some form of Inter-Process Communication (IPC), resource allocation in the form of a scheduler or dispatcher and an interface between the hardware and those processes which require access to it. These basic services are sufficient to support user level applications and system servers and more complex systems may be developed from the low level software which provides them. Consequently, in order to prove sufficient for the construction of systems of a reasonable utility and size, the design presented in this chapter will have to provide at least this basic functionality.

4.2 Design Goals

A number of supplementary requirements guide the design and are used as ultimate decision criteria to select one from a number of seemingly equivalent design possibilities should the need arise.

Firstly, the resources used by system software are unavailable to the user, so one of the goals of any system design ought to be to provide a reasonable environment in which a user's applications may run while maximising the amount of resource which is available to the user.

Secondly, at the lower levels, system software ought to provide only the most basic, necessary mechanisms. Policy decisions ought to be left to the users of the resources provided by the system.

Thirdly, it is important to keep in mind the environment in which the end system will be operating; this will aid the derivation of a suitable set of design requirements and ensure that neither too much nor too little emphasis is placed on any one aspect of the design. In the case of the design presented here, the environment is a workstation which is connected to other workstations and server machines via a high speed network. This does not mean that the design is to be so myopic that the resulting system is only useful within a workstation; the use of sufficient modularity in the design ought to make the system amenable to distribution and the basic kernel functionality ought also to be usable in other applications such as a dedicated server. Also to be noted is that a server is a relatively controlled environment when

compared to a workstation, so that some features, such as memory protection, which are often necessary in a workstation may be dispensed with in a server to obtain better performance.

4.3 Addressing

Recently, there has been some renewed interest in the construction of systems which use a Single Address Space (SAS). In the case of systems structured as sets of processes which communicate among themselves and cooperate by sharing data, the ability to refer to some data from different processes by using one address enables certain optimisations to be obtained. In particular, in the case of processes which communicate, large data structures can be passed between processes as arguments and return values by simply passing the address of the data. This obviates the need to convert addresses between address spaces or to copy the data as is done in some systems [Accetta86] [Hildebrand92] and results in a significant performance improvement over such systems. Use of a single address space will affect the design of other parts of the system.

4.3.1 Loading Executables

In a Multiple Address Space (MAS) system, program code and data can be bound to their run time addresses before execution because it is known that each program will run at the same virtual address. A similar strategy could be used in a SAS system if addresses are never reused. Whenever a program is linked for execution, it is allocated a range of addresses by the system, and it will be permanently bound to that range. The allocation has to remain valid across system reboots, and the system needs to take great care in remembering where the base of unallocated address space is. This approach is less viable in machines with a small (32 bits or less) address space as the size of the address space will limit the lifetime of the system. Recently, a number of microprocessors which have much larger (up to 64 bits) address spaces have been developed[DEC92] [MIPS91]. These large address spaces mean that addresses may not have to be reused in a typical system until it has been running for a very long time; use of addresses at the rate of 1 gigabyte per second would exhaust a 64-bit address space after approximately 500 years. It is important to keep in mind that in such a system, the size of the address space only determines, for a given rate of address consumption, the lifetime of the system after which it has to be restarted from its initial state; it is in many respects, unrelated to the fact that there is only one address space.

In a system in which addresses may be reused, the binding of programs to their run time addresses can be left until the programs are executed. At this stage, a system loader can obtain a suitable range of addresses from the system and relocate the program to use these addresses as it is loaded. In a small address space, this can be used in conjunction with some additional code which maintains a map of unallocated addresses to prolong the lifetime of the system. The ability to reassign addresses whenever a program is executed for the first time removes the need to remember the pointer to the base of unused addresses across system reboots. This pointer can be set to the start of the address space at every system initialisation.

4.3.2 Sharing Text

Allowing multiple processes to share a single text segment can improve the utilisation of memory, so some consideration has to be given to how this might be done in a SAS system. If all program addresses are resolved just prior to execution, text sharing can be implemented with the aid of a data segment base register `dsb`. At link time, the static data are coalesced into a single data segment and each datum is allocated an appropriate offset from the base of the segment. When the program is loaded into memory, the system allocates space for the data segment and initialises `dsb` to point to the base of the segment. Data are referenced from the shared text segment by using these offsets to index off the `dsb` register. Figure 4.1 shows how two processes which share a text segment access their own versions of the data segment. For the

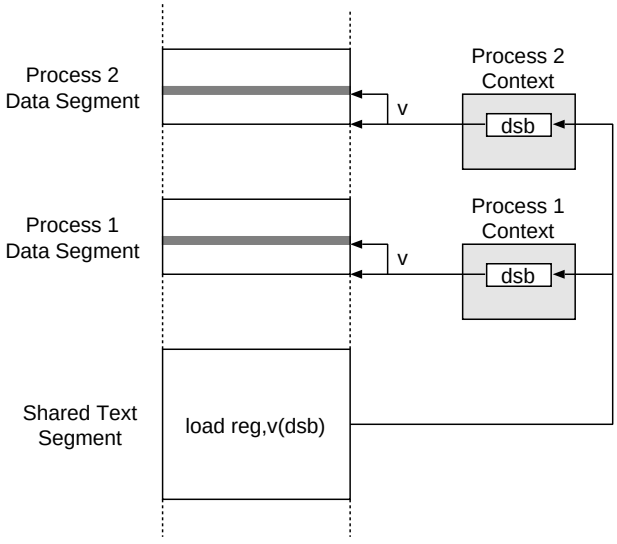


Figure 4.1: Sharing text in a single address space.

design presented here, a single address space is chosen to obtain the performance advantages outlined above and to gain some experience with its use. The resulting

system can also be used in further work to investigate some of the outstanding issues pertaining to the use of a single address space such as the implementation of more dynamic systems which allow run time binding to shared library modules.[1]

4.4 Memory Access Protection

Memory access protection in operating systems is used to limit the scope of the accesses which can be performed by a program. Typical operations which can be performed on data by a program are read, write and execute. Appropriate use of protection mechanisms can prevent both accidental and malicious accesses of all of these types. The costs of using such protection are that it requires extra hardware to implement and that a certain amount of time is required to perform the checking of accesses. Because a workstation is intended to support a single user,[2] the use of protection to guard against malicious access is not always necessary and many accidental accesses can be prevented by strict compiler checking and careful programming. Systems have successfully been constructed on these principles [Swinehart86].

The microprocessors used within current workstations usually provide some form of hardware memory protection mechanism which is often integrated with the address translation unit [Kane88]. These typically provide an unprivileged user mode, a supervisor mode in which it is possible to execute privileged instructions and the ability to set read, write and execute permissions on areas of memory, down to the granularity of the hardware page size. Current operating systems use this hardware in essentially two ways.

Monolithic architectures [Ritchie74] place the whole of the operating system in supervisor mode. In such systems, communication between system modules requires the cost of a procedure call, the lack of reinforcement of system module boundaries means that their interfaces can become blurred and more code executes in privileged mode than is necessary.

Microkernel architectures [Accetta86] [Rozier89] [Hildebrand92] are based on a kernel of reduced functionality which provides a high-level interface to the hardware, support for processes, memory management and some form of IPC which is typically based on message passing and implemented within the kernel. In such systems, the

[1] This subject is currently being investigated within the Computer Laboratory by Timothy Roscoe [Roscoe94].

[2] In an environment in which a number of "workstations" are interconnected via a LAN, this is not always the case.

operating system software is structured as a set of servers which provide system services to client programs. Message passing is a low-level facility, so often a Remote Procedure Call (RPC) mechanism is provided as a convenient programming abstraction. Characteristic of such systems is that the nature of their construction facilitates distribution over networks of processors, but while the boundaries between system modules are well enforced, communication between modules requires one or two message passing operations.

The design attempts to utilise the protection hardware by providing facilities to share data among protection processes which need access to it. The granularity of protection domains is such that a protection domain encompasses a system server or application. Intra-module protection is static and relies on compiler type checking while inter-module protection is dynamic and requires appropriate hardware. Applications can interact with the system's memory and protection management server to arrange to share a part of their domain of protection with other system entities.

The amount of code which executes in supervisor mode is minimised by the provision of a facility which allows a sufficiently privileged process to request for sections of its code to be executed in privileged mode. This is motivated by a number of factors. Moving between user and supervisor mode is becoming less expensive on modern RISC processors because the hardware does not automatically incur the overhead of saving the processor state, this task being left to hand crafted exception management code which is provided as part of the system supervisor. Conceptually, microkernels contain a small subset of the functionality required of an operating system, but in reality this requires a not inconsiderable amount of code to implement. This code executes in supervisor mode and on some architectures (for example the MIPS R3000) necessarily has unlimited access to the whole of the system address space. The ability to request that the processor be placed into privileged mode means that the majority of the code which usually constitutes a microkernel can be executed in user mode at a correspondingly lower privilege level, enabling better use to be made of the available protection hardware. A final factor which motivates this element of the design occurs as a result of having dynamically loadable system entities such as device drivers. Separating the portions of the driver code which absolutely *must* execute in supervisor mode from the rest improves the chances of being able to check this sensitive code before it is loaded into the system. The long term goal of this is that if sufficiently clever checkers can be constructed, then the system can relax its restrictions about who is allowed to use such checked, privileged code to the extent that, certainly system provided libraries and possibly even user applications, might be afforded the ability to execute short sections of code in privileged or uninterruptable mode. Figure 4.2 shows the effect which such a design has on the structure of an operating system. Most of the functionality provided by a typical microkernel has been moved into a user mode domain; requests to have the

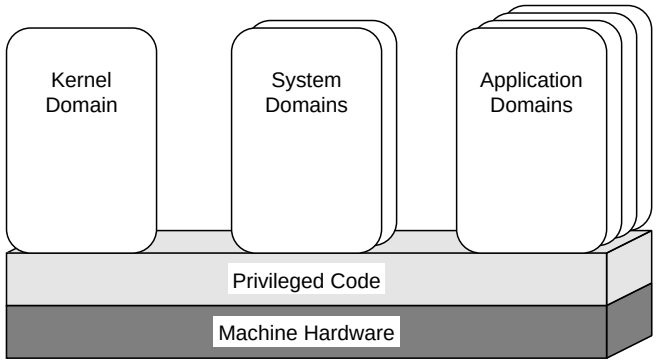


Figure 4.2: The structure of a system which uses privileged library.

processor placed into supervisor mode which can be identified as originating from the privileged kernel or system domains are granted. This identification may be based on a knowledge of the process which is currently running.

4.5 Communication

Within the system being designed, communication occurs between two processes, between a process and the system and between the hardware and system processes. These forms of communication will occur frequently, so they ought to cost as little as possible. Many systems fall prey to lack of performance caused by programming generality. Not only do many microkernel systems offer a limited set of message passing primitives, but the cost of message passing between processes on the same machine is much more expensive than a local procedure call. In at least one system, this has resulted in the migration of user mode servers into the kernel and supervisor mode purely for performance reasons [Bricker91].

Such performance gains in themselves are not directly related to the implementation of QOS within a system, but the performance of IPC primitives is of great importance in evaluating the viability of a design. Microkernel architectures offer the advantages of well enforced modularity at the expense of more costly inter-module interactions. Since IPC will be used frequently in a system, it is important that it be implemented in as efficient a manner as possible. The ultimate goal of doing so is to make it of the same cost as a subroutine call.

The aspect of IPC design which will be influenced the most by a QOS implementation is timeliness. Many current systems provide support for the timely scheduling of processes which communicate via IPC by assigning static priorities to the processes. The scheduling decisions which are based on these priorities are bound implicitly

within the IPC mechanism so applications have no means of explicitly expressing their temporal requirements or of controlling the times at which scheduling decisions are made. The design should incorporate sufficient flexibility to enable applications to express temporal requirements explicitly and allow them control over when then scheduling decisions which result from IPC operations are to occur.

4.5.1 Components of Communication

Communication between processes consists of two independent actions: data transfer and synchronisation. In an implementation which is based on message passing, these two actions are combined in the message passing primitives provided by the system. The manner in which data is transferred is determined by the maximum message size and the amount of control given to a process over the type of synchronisation desired and varies depending on whether the primitives are synchronous or asynchronous. Combining these actions into the one message abstraction incurs unnecessary overhead when it is desired to use only one of them independently. The use of messages containing no data for synchronisation and of messages to request the current value of some data between processes which are executing on the same machine are examples of this.

The desire to provide support for CM streams within the design provides additional motivation for separating the synchronisation and data transfer aspects of communication. Applications which need to synchronise their actions with CM streams often do not need access to the stream data itself. This factor guided the design of the *stream agents* presented in [Sreenan92] which can be used to filter a stream of CM data and present to an application a sequence of synchronisation events.

Recent advances in the design of workstation architectures emphasise this separation. [Hayter93] presents the Desk Area Network (DAN), a workstation architecture which is well suited to handling CM streams. DAN replaces the traditional backplane bus with an ATM switch, which is used to interconnect devices such as the processor, frame store and LAN interface. [Pratt92] presents the "ATM camera" which converts an analogue video signal into a stream of ATM cells and connects directly to the switch in the DAN. The switch can be programmed to direct a CM stream from the camera to the processor if the video data is to be delivered to an application. If the video data is only to be displayed, the switch can be configured to route the video data stream directly to the frame store, bypassing the processor. A later version of the camera [Pratt93] produces in addition to the data stream, a control stream containing synchronisation and timing information. Within the DAN, separation of the data and synchronisation components of a video stream enable the data stream to be routed directly to the frame store and the control stream

to be delivered to an application.

It can be concluded that, while message passing is a convenient abstraction, useful for building distributed systems, it can be constructed from more basic primitives which provide a lower level of abstraction. The separation of these primitives into data transfer and synchronisation operations makes them better suited to a number of applications such as the handling of CM streams. It also enables certain optimisations to be made when communication is between entities which are running on the same machine.

4.5.1.1 Sharing Data

Data can be shared between two processes provided that they can agree which data to share; the data can be protected according to how much each of the processes trusts the other and they can synchronise accesses to the data. An example of the usefulness of data sharing can be found in the information about itself which a process is able to know and which usually resides within the operating system; the current time and accumulated processor usage are examples of such information. When this is resident in the operating system a system call is required to obtain it. If the operating system made this information available to the process, it could be read directly, thus avoiding the overhead of a system call. The mechanics of such operations should be hidden behind suitable interfaces provided by system libraries so that application portability is not compromised. If the system trusts a process, the process can be given open access to the data used by the system. If the process is not trusted, memory management hardware can be programmed to allow the process only the ability to read the data and not write it. Data consistency can be provided by the use of atomic updates or some more general synchronisation primitives as might be provided by the system. The type of synchronisation best suited to a particular instance is dependent upon the type of data being shared and not all synchronisation requirements will need to use the mechanisms provided by the system. The agreement between processes as to *what* data are to be shared must therefore also include *how* it is to be shared.

4.5.1.2 Synchronisation

Synchronisation marks points in the execution of processes where scheduling decisions have to be made. In a message passing system, a scheduling decision has to be made after a message is sent. A blocking send primitive necessarily suspends the sending process, surrendering the processor, until a reply is received. A more flexible system allows the sender to return, still in possession of the processor. A synchronous

send can be implemented using asynchronous primitives by atomically implementing a send then a receive. The synchronisation primitives designed ought to provide at least the functionality of the synchronisation provided by asynchronous message passing implementations in that after a synchronisation point has been reached, a process is given the choice of whether or not it wants to retain possession of the processor.

4.5.2 Events

The synchronisation aspect of communication can be modelled as the signalling of the occurrence of an event by one process to another. Figure 4.3 depicts the phases through which the path of execution of a processor passes when processing a hypothetical event. The event is said to *arrive* when the system first becomes aware

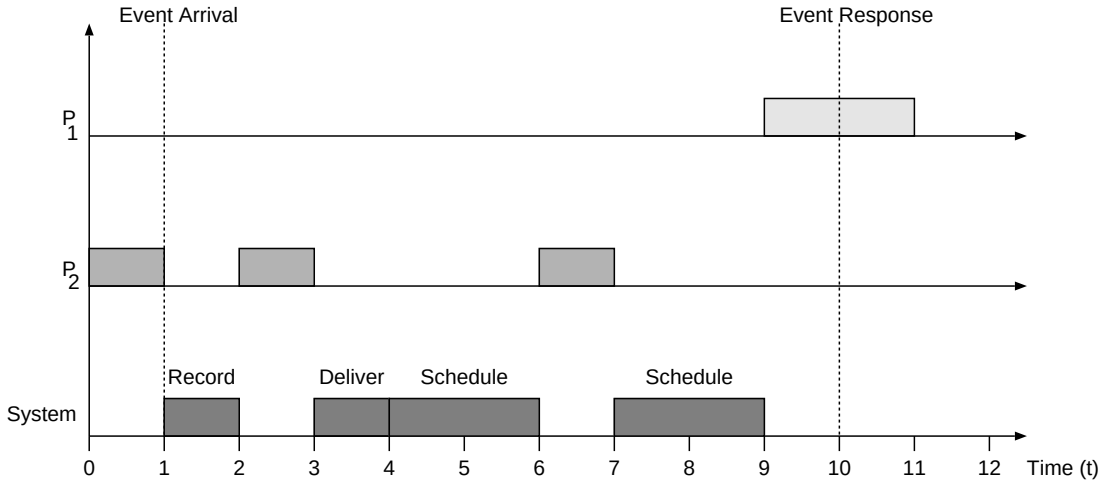


Figure 4.3: Arrival, processing and delivery of an event.

of the event's occurrence (time $t = 1$); in the case of the example, the event may be generated by a process or as the result of some external hardware interrupt. At the time of arrival, P_2 is running and P_1 is the event's target. In response to an arrival, the system must at the very least make a *logical record* of the arrival so that it is not forgotten. Some time later, (time $t = 3$), this record is examined, the event is *delivered* to its target, and a scheduling decision is made. Delivery of an event does not necessarily mean that the target process is given the processor next; this decision is made on other information and in the case of the figure, after delivery of the event to P_1, P_2 is given the processor next. Eventually, another scheduling decision is made, and P_1 gets to run, at which stage, it becomes aware of the event delivery and may choose to respond to it whenever suitable. The model can be applied with equal success to both hardware generated events and their software

equivalents.

The ability to postpone delivery and scheduling after an event has arrived enables the implementation of critical sections. P_2 can implement a critical section by setting a flag upon entering it. This flag indicates to the event handler that the current process is not to lose the processor until it has left the critical section. The event handler, upon seeing this flag set, simply records the event and resumes P_2. Upon leaving the critical section, P_2 clears the flag and then allows any pending events which may have arrived while it was in the critical section to be processed. In the case of interrupts, event masking, recording and delivery are implemented by the machine hardware.

The scheduling decisions made after event delivery (at $t = 4$ and $t = 7$) imply that the model distinguishes between the logical value of any event (i.e. the record that an event has happened) and the time at which the response to the event occurs. In cases where the scheduler is provided with some information concerning the temporal requirements of an event, it can use this information to schedule processes so that these requirements are met. The ability to make such scheduling decisions extends to any user level schedulers which might exist within a process; in the example of the figure, even though P_1 is running at $t = 9$, it postpones the response to the event until $t = 10$, because it has more urgent work to do beforehand.

4.5.3 Design of the Event Mechanism

The design includes a mechanism called an *event channel* which is a unidirectional path through which one process can communicate the occurrence of events to another. Within its own domain of execution, a process is free to accumulate records of events as it pleases, and doing so does not have any direct effect on the scheduling of the process. When it decides to, a process may communicate the occurrences of events n_i on channels e_i by executing the call `signal(vec)`, where vec is a vector of (e_i, n_i) pairs. After doing this, the events have arrived into the system. For each event channel e_i, the system determines the end point of the channel. This locates an integer counter within the protection domain of the target process which the system atomically increments by n_i; this increment records the delivery of the n_i occurrences of the event. A scheduling decision is then made. When the target process is next executed, it is informed of the arrival of events and may respond to them as it sees fit. The use of the vector argument `vec` to the `signal` call allows signalling on a single event channel or atomic signalling on a number of event channels. The latter provides support for a type of multicast signalling facility.

A process which receives events is expected to maintain a record of which events it

has processed and which are still pending. This can be done by reading the value of the event to determine the total number of events which have been delivered (d, say) and comparing this with a locally maintained count of the number of events to which a response has been generated (r, say). At any time, two invariants hold:

(1) The number of pending events is $d - r$ and;

(2) $d - r \geq 0$.

The presentation of events has so far assumed that event channels are in existence when they are used; a means is required whereby event channels can be created. The basic problem to be solved in this instance is that before a process P_1, say, is allowed to send events to another process P_2, say, P_2 must agree to accept events from P_1. The event mechanism is designed to be usable for communications between all of the active entities in a system, including communications between a process and the system itself. For this reason, when a process is created, it is provided with at least the means to communicate with any necessary system entities. An optimisation is to provide processes with communications channels to all of the frequently required system servers.[3]

A process P_1 can use these established channels to request a channel for sending events to another process P_2 by executing the system call `request(P2)`, which returns to P_1 a channel identifier e. At this stage, any attempt to signal events on e will result in an error condition. The `request` call causes P_2 to receive via its existing interface with the system an asynchronous request for a connection e from P_1 which is identified uniquely by the pair (P_1, e). P_2 can decide either to accept or to reject the request. Rejection causes an error return in P_1. Acceptance is indicated by calling `accept(P1, e, &v)` which informs the system that P_2 is willing to receive events from P_1 on channel e, and that the event deliveries are to be recorded in the variable v, whose location is given by `&v`. This causes the state of the channel to be updated so that events may be signalled on it and also causes P_1 to be notified that the channel may now be used.

An event channel may be closed by the process at either end; this causes the process at the other end of the channel to be notified of the closure and any further signals to return an error status.

[3]This is the equivalent of the provision of UNIX processes with the standard I/O connections `stdin`, `stdout` and `stderr` when they are created.

4.5.4 IPC from Primitives

Shared memory and events can be used to implement efficient IPC within the same machine. A unidirectional communications path can be established from process P_1 to process P_2 using an event and a shared memory segment with protection which allows P_1 permission to read and write and P_2 to read. P_1 can marshal one or more sets of arguments directly into the shared memory segment then signal the appropriate count of events to P_2. Operating in conjunction with a similar structure operating in the reverse direction, this implements a bidirectional communications path.

4.5.5 Hardware Interface

The IPC abstraction provided by the system can be used to interface user mode processes to hardware devices. Device driver processes are assumed to be privileged in that they are allowed access to device registers either by mapping them into their protection domain or via the execution of privileged code. The interface between a device driver and the system then consists of the device registers (and buffer memory if present) and an event which is signalled by the system as the result of a device interrupt. In cases where good performance is required, the device driver can export to other processes an IPC interface in which the shared memory segment maps some onboard device buffers, so that the processes marshal their arguments directly into the device buffers, avoiding unnecessary copying.

The ability to do this effectively is determined by the granularity of protection afforded by the system memory management unit. In the majority of current processor designs, the control of protection and cacheing is associated with virtual memory page size. Device drivers however, can utilise a granularity which allows them to protect down to the size of the device registers and consideration should be given to this in the design of the system hardware.

4.6 Scheduling

Scheduling occurs at a number of levels, the QOS manager makes long term scheduling decisions such as whether or not to admit a new process or withdraw resources from one process and give them to another, the system RTRA is responsible for presenting resources to processes in a timely manner and within the processes which use them, user-level thread schedulers are responsible for allocating resources to the

application threads. The mechanisms required to implement this resource management strategy need to perform run time resource allocation, accounting and policing; the implementation of each of these requires some knowledge of the current time.

4.6.1 Time

To allocate resources in a timely fashion, the notion of time has to be ubiquitous throughout the system. Arithmetic operations on temporal quantities will be frequent, so a sensible representation of time can limit the amount of resources required to effect these calculations. The resolution of the clock ought to be sufficient to measure the time taken to execute some code on a processor to within a few instructions for the purposes of accurate profiling.

Within the design, all times are represented as 64-bit integers which record the number of nanoseconds having elapsed since some well known epoch and can distinguish events which are less than 2^{64} nanoseconds or approximately 580 years apart. Maintaining times as a single integer reduces the number of instructions required for temporal arithmetic over representations which use counts of seconds and decimal fractions of seconds.

The system clock consists of two registers; the `current` register stores the time since the epoch and the `alarm` register stores the time at which the next clock interrupt is to be generated. A system clock module uses this (ideally hardware) clock device to implement a timeout queue which provides equivalent virtual clock devices for all processes. Processes may request that the clock module signal the occurrence of an alarm event some time into the future by programming their virtual alarm clocks. All processes are given direct, read only access to the `current` register to provide them with a cheap, accurate value of the current time.

4.6.2 Process Classes and States

The design distinguishes between the various states in which a process exists by maintaining a number of queues. Q_c contains all of the processes which hold contracts awarded by the QOS manager and are runnable. When a contracted process has used all of its allocated resources and requires more, it is placed in Q_w until more resources become available for it. Processes which have indicated that they wish to wait for some event are placed into Q_b, the queue of blocked processes and best effort processes are placed into Q_f.

The most suitable algorithm for determining which process ought to get the pro-

cessor next will be the result of considerable experience in the everyday use of the system, so a simple dispatching policy is described to demonstrate the use of the mechanisms provided. Whenever the process dispatcher is called to give the processor to the most eligible process, it chooses a process from Q_c; this aims to satisfy the requirements of the contracted processes before any other processes. If this queue is empty, a process is chosen from Q_w, the aim being to give contracted processes the chance to make use of any resources which become fortuitously available after they have already received their contracted quota. If Q_w is empty, the processor is offered to the best effort processes in Q_f. This algorithm will be used in the system presented in this dissertation, but further work is needed to refine it.[4]

For the two classes of processes (the contracted processes in Q_c and Q_w and the best effort processes in Q_f), the RTRA implements a dispatching policy. The best effort processes may be scheduled using some form of multiple level feedback queue. Recall from section 3.8 that the RTRA dispatching policy has a large impact on the type of contracts which it is possible to offer to processes, and that the designer is under no obligation to choose one particular policy over another, provided that the QOS manager is aware of the policy which is being used so that it can perform admission control and that processes are made aware of the nature of the contract which is being offered to them when they are admitted. Section 2.1.1, however, motivates the use of dispatching policies which take into account the temporal requirements of the contracted processes on the grounds that doing so will enable the system to meet more deadlines with a given amount of physical resource than otherwise would be the case.

A simple static priority scheme could be used in conjunction with the RM algorithm to effect some temporal ordering in which the processor is shared amongst the contracted processes. Static priority assignments have a number of disadvantages. Firstly, they afford a limited flexibility in expressing temporal requirements; static assignments may need to be recalculated in a dynamic system when new processes are given contracts. Secondly, the processes may be running multiple threads, the temporal requirements of which require more than a single priority to express. The design instead uses deadlines in conjunction with the EDF algorithm to sequence the execution of the processes. Each process specifies its deadline (in the case of multiply threaded processes, this will be that of the thread whose deadline is earliest) to the RTRA, which uses this information to order the queue of contracted processes such that the process with the earliest deadline is at the head of the queue. When the processor is given to a process, the process is informed of the next earliest

[4]For example, the algorithm presented does not make use of the fact that the response times of the best effort processes in Q_f can be reduced by delaying execution of the contracted processes in Q_c for as long as possible while not allowing them to miss their deadlines [Tindell93].

deadline in the system; a process scheduler then knows that it can execute all of its threads whose deadlines are earlier than this before having to give up the processor. This allows processes which want to cooperate to obtain a high utilisation of system resources.

There are times during the execution of a process when it can do no useful work even if it is given the processor; a process which is processing a video stream has no requirement for the processor until the frame of video for which it is waiting has arrived. A process may also detect that it does not require all of the processor time which has been allocated to it. In these circumstances, a user level scheduler may idle and waste processor cycles. The policing mechanism will prevent this from affecting other contracts, but best effort processes may benefit from being able to use these otherwise wasted cycles. The design allows processes to call into the kernel when they want to give up the processor until an event occurs. This causes the process to be removed from one of the runnable queues and placed into Q_b until any event occurs. Arrival of an event when a process is in this state causes the process to be rescheduled into an appropriate runnable queue. Allocation of processor cycles is important to an application, but applications which block may not always want to be informed of it. The mechanism allows processes to specify whether or not they wish to be informed of processor allocation while they are waiting for an event to occur.

Having used all of its resources, a process is removed from Q_c and placed into Q_w to wait for its next allocation of processor bandwidth. Should additional resources become available before then, a process can be selected from Q_w and given those resources. This allows processes which are able to make use of additional resources to do so.

4.6.3 Accounting

The design incorporates a basic accounting mechanism called a `timer` which records the accumulated time for which it has been running in nanoseconds. A number of primitives are provided for operating on timers: `timer_reset(t)` initialises the timer to zero; `timer_start(t)` starts the timer accumulating elapsed time; `timer_stop(t)` freezes the timer and; `timer_read(t)` returns the time accumulated by the timer. The RTRA maintains, for each process, a timer which records the processor time accumulated by that process.

4.6.4 Policing

A policing mechanism is required to ensure that contracted processes use no more cycles than they have been allocated by the QOS manager. The design incorporates a simple policing scheme which assumes that, if a process has been allocated a processor bandwidth of (C, T), then the process is eligible for C units of processor time every T units of real-time. For every process, a timer t_a is maintained which is reset to zero at the start of each period T and records the processor time accumulated by the process during that period. Whenever the process loses the processor, either because it is preempted by another process, or because the alarm clock signals that it has no more processor time left, t_a is stopped. The contracted processor time which is remaining for that process is calculated as $t_r = C - t_a$. If $t_r = 0$, the process has no more processor time left and it is removed from the runnable queue and forced to wait until the start of its next period when it is allocated another C units of processor time. Whenever the process is about to be given the processor, the system alarm clock is set to interrupt t_r into the future, timer t_a is started, and the processor is given to the process.

4.7 Virtual Processor Interface

The resource allocation strategies used to provide applications with various qualities of service sometimes give the applications fewer resources than they require. Section 3.11 describes some ways in which applications can degrade their QOS when resources are scarce. Applications can do this more easily if they are informed of the availability or otherwise of resources. This is effected within the design by having the system present applications with this information through the VPI. The interface includes a notification mechanism which informs applications when resources have been allocated, when they are available and when additional resources have been made available. The same mechanism can be used to inform processes of the delivery of events such as timer and communication events.

The purpose of the notification mechanism is to divert the path of execution of a process into the user level scheduler, which may make the decision of what to do based on the current resource availability. If the thread which was running when the notification was delivered is to be suspended and another thread run in its stead, some means of obtaining the thread's saved context from the system is required as well as a means of resuming a new thread. Notification will occur asynchronously with respect to the execution of a process, so some means is required of synchronising the delivery of such information with access to critical sections.

4.7.1 Activation

To notify a process of the occurrence of an event, the distinction is made between process *activation* and process *resumption*. Usually, when the processor is given back to a process, execution is *resumed* from the previously saved state and processes execute unaware that they lose the processor and regain it from time to time. *activation* allows a process to specify to the system the address of some code which is where it would like execution to begin whenever it is given the processor; this is typically the entry point to some user level thread scheduling code.

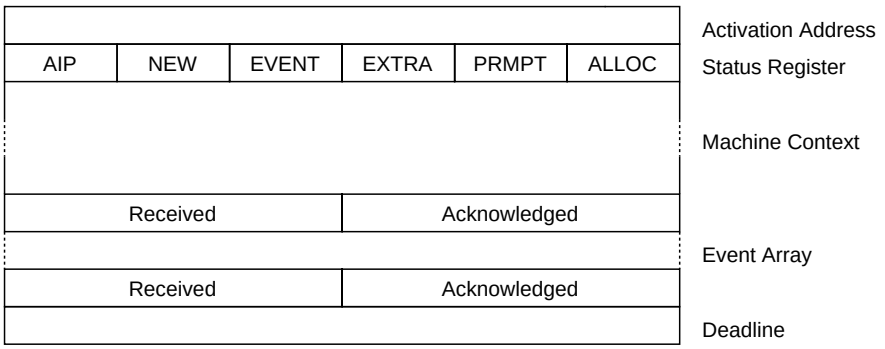


Figure 4.4: Activation section of the virtual processor interface.

Figure 4.4 shows the section of the VPI which implements activations; this consists of an activation address, a status register, and space for a saved machine context. Table 4.1 describes the functions of the bits in the status and control registers. When a process is created, its activation handler is set to be the address of the

Bit	Meaning
`ALLOC`	Processor time has been allocated
`PRMPT`	Process has been preempted
`EXTRA`	Extra processor time is available
`EVENT`	An event has been signalled
`NEW`	Events occurred during activation
`AIP`	An activation is in progress

Table 4.1: Definitions of bits in the virtual processor interface registers.

process's initialisation code. As part of its initialisation, a process installs into the `activation_address` register, the address of some activation handler code. Whenever a process loses the processor, its context is saved into the machine context field. Whenever the processor is given to the process, the status register is updated to re-

flect the reason for activation and the process is activated by forcing its execution to start at the address specified in the activation handler. Figure 4.5 shows the

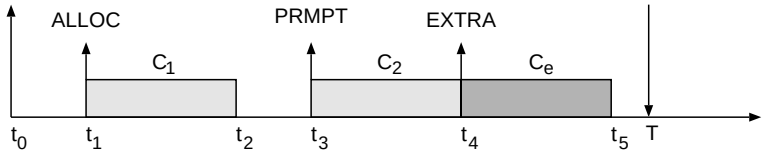


Figure 4.5: Execution of a process showing activation points and reasons.

timing diagram of a process which has a reserved processor bandwidth of (C, T) and is preempted at time t_2, so that it receives the bandwidth in two portions C_1 and C_2 such that $C_1 + C_2 = C$. At time t_0, the process is allocated C processor cycles and is eligible to run but must wait for a higher priority process. At t_1 the process is given the processor, and because it has just been allocated more resources, the `ALLOC` bit is set in the status register prior to activation. At t_2 the process is preempted and at t_3 the process is again given the processor. No resources have been allocated since the last activation but the process still has C_2 cycles available, so the `PRMPT` bit is set in the status register to reflect this. By t_4 the process has used all of its processor allocation, but the system is otherwise idle and decides to give the process additional cycles so the `EXTRA` bit is set in the status register is set and the process is activated.

During the execution of the activation handler, the `AIP` bit in the status register is set, causing any further activations to be queued. The activation handler uses the status register bits `EVENT`, `EXTRA`, `PRMPT` and `ALLOC` to determine its course of action. If this involves rescheduling threads, the state of the thread which was running when the process last lost the processor is available in the machine context area. Before continuing with the execution of the process, the activation handler must enable activations by clearing `AIP` then check the `NEW` bit in the status register to see if other events have occurred during the execution of the handler. If so, the handler must call back into the system to give it the opportunity to present any pending events.

Critical sections can be implemented using a global, boolean variable which records entry into any such sections. Before entry to a critical section, a thread sets the variable to `TRUE`, should the process be activated while in a critical section, the thread scheduling code in the activation handler can inspect this variable to determine what to do with the thread. Possible actions are to resume the thread until it leaves the critical section, or to suspend it and run another thread which is known not to be in conflict with it. Another means of implementing critical sections uses two activation handlers, the usual handler which contains a thread scheduler and an alternate handler which always resumes the current thread. Entry to a critical section can be

effected by installing the alternate handler, which makes any activations invisible during the critical section.

4.8 Summary

This chapter has presented the design of a kernel which incorporates mechanisms providing support for QOS within the operating system. The design includes an efficient IPC mechanism which makes use of a single address space, shared memory and means of signalling events between processes. The separation of data transfer and synchronisation aspects within the IPC mechanism make it better suited to some CM applications such as remote synchronisation with a stream of CM data than message based IPC mechanisms. The design also incorporates accounting and policing mechanisms, which are required if QOS contracts are to be maintained. It also augments the virtual processor interface which is seen by applications that applications can be informed of the availability of resources within the system over time.

Chapter 5

Implementation

This chapter describes the implementation of an experimental system called Nemo which was built to investigate some aspects of the design presented in the previous chapter. This implementation runs on a DECstation 5000/25 workstation, which uses a MIPS R3000 processor.

5.1 System Overview

Figure 5.1 shows the structure of a typical Nemo system. The Nemo Trusted Su-

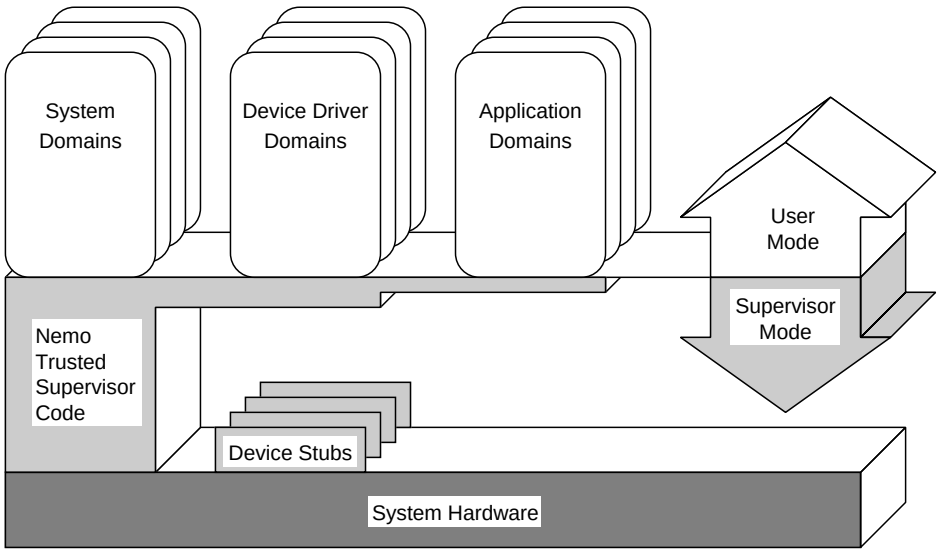


Figure 5.1: Structure of a typical Nemo system.

pervisor Call (NTSC) code implements those functions which are required by user

mode processes and which need to run in supervisor mode. It is also responsible for mapping the low level hardware interface onto the VPIs which appear in each of the processes. The NTSC code provides support for three types of processes. System processes implement the majority of the services provided by the operating system. Device driver processes are in many respects similar to system processes, but are distinguished by the fact that they are attached to device interrupt stubs which execute in supervisor mode and hence are part of the NTSC code. Application processes contain user programs. Processes interact with each other via the system IPC facilities which are implemented using events and, if required, shared memory. The following sections describe the operation of the main system entities.

5.2 NTSC Code

The NTSC code is about 2.5 kilobytes in size, written entirely in assembler and implements the routines which provide the support for VPs. Processes gain entry to the NTSC code via the standard system call mechanism and, once they have entered it, their execution is not preempted. Code is placed into the NTSC for a number of reasons:

- it needs access to privileged instructions and needs to execute within supervisor mode;
- it is executed as the result of an exception such as an interrupt, in which case the hardware has forced execution of the exception handler to be initiated in supervisor mode or;
- the code implements commonly used functionality which needs to be run without preemption.

The routines which are provided for use by processes to implement activation handlers and which will be discussed in a later section are examples of the latter. NTSC calls are separated into two classes, one containing calls which may only be executed by a suitably privileged system process such as the kernel, the other containing calls which may be executed by any process. Table 5.1 lists the NTSC code segments which currently exist and their functions. Of these segments, `sc_activate` and `sc_halt` are executable only by the kernel, the rest may be executed by any process. `sc_halt` causes the system to be halted, and execution to enter the bootstrap monitor. `sc_ici` and `sc_dci` allow a range of addresses in the instruction and data caches respectively to be invalidated. The function of the remaining code segments is described in later sections.

Name	Purpose
`sc_activate`	Activate a process
`sc_halt`	Halt the processor
`sc_rfa`	Return from activation
`sc_rfar`	Return from activation, restore context
`sc_kernel`	Activate the kernel
`sc_deliver`	Deliver any pending events
`sc_ici`	Invalidate a region of the instruction cache
`sc_dci`	Invalidate a region of the data cache

Table 5.1: NTSC code segments.

The NTSC maintains a small amount of data which include: `kvp`, the address of the kernel's VP; `cvp` the currently active VP and; `ccx` the address of the structure into which the machine context is to be saved when a process loses the processor. These are maintained as a result of the execution of NTSC code by the kernel and user processes.

5.2.1 Device Interrupt Stubs

The NTSC code is also responsible for providing an interface between device hardware and its associated driver process; an example of this is provided by the clock device. The DECstation 5000/25 provides a periodic source of interrupts which have an inter-arrival time of 1 millisecond; this source is used by the NTSC to implement the system clock. Figure 5.2 shows the interface to the Nemo system clock, which consists of two registers called `current` and `alarm`. `current` contains the number of

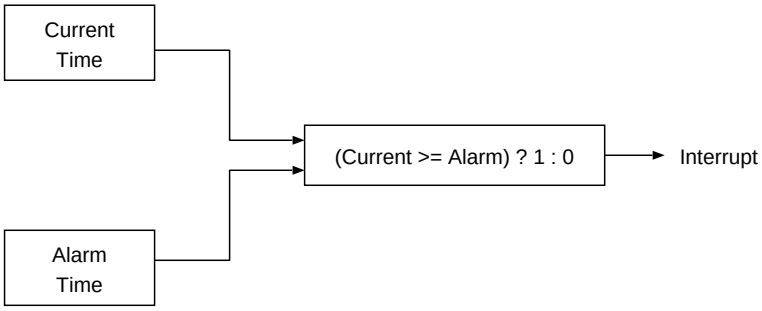


Figure 5.2: Nemo clock device interface.

milliseconds which have elapsed since the epoch. `alarm` contains the time at which the system next wants to receive a clock interrupt. The clock driver will generate an

event whenever the value of the `current` register is greater than or equal to the value of the `alarm` register. The NTSC stub for the periodic interrupt is implemented as a fast and a slow path. The entry to this code saves the minimum number of registers required, increments the `current` register and compares the updated value with the value of the `alarm` register. If the alarm has not expired, the fast path is taken; the partially saved state is restored and execution returns from the exception. If `current` is greater than or equal to `alarm`, the alarm has expired and the stub needs to generate an interrupt. In this case, the remaining context is saved and the interrupt is delivered to the kernel in the form of an event.

The `current` and `alarm` values are maintained within the NTSC code, but are made available to processes as shared memory segments. The `current` register can only be modified by the NTSC code, but may be read by any other process. Processes are given the address of this register at initialisation time and so have ready access to the current system time to millisecond resolution at the cost of a memory read operation. In addition to the address of the `current` register, the kernel is informed of the address of the `alarm` register, which it may write with the time at which it would next like to be sent an event by the NTSC.

This device driver implementation exemplifies a number of important aspects of the mechanisms whose design was presented in chapter 4. Firstly, in the case of devices which present only a low level hardware interface to the system software, code can be implemented within a device driver stub to implement a higher level interface; this allows the system implementor to trade hardware complexity and cost off against the processor cycles required to implement a high level device interface. Secondly, the implementation of the interface between the clock device registers and the processes which use them provides extremely efficient accesses to the register values without compromising the basic shared memory IPC abstraction. Thirdly, the same shared memory and event mechanisms which are used for communications between processes are also used to provide the interface between hardware devices and their device driver processes.

5.3 Processes

Processes are the active entities within a Nemo system and consist of a domain of protection and a thread of execution. A process may be multi-threaded, in which case it will implement its own thread scheduling algorithms in addition to the process scheduling which is done by the system.

5.3.1 The Virtual Processor Interface

Of interest in this discussion is the interface between a Nemo process and the system. Nemo provides each process with a virtual processor (VP); resources are allocated to each VP by the RTRA and resource availabilities are communicated to processes via the Virtual Processor Interface (VPI), which is defined by the code segment in figure 5.3. Within a process's VPI, `activate` contains the address at which

```
/*
 * Virtual processor interface
 */
typedef struct {
  void      (*activate)();  /* Activation handler         */
  u_int     status;         /* VP status (see below)      */
  u_int     disable;        /* Disable activations        */
  context_t ecx;            /* Execution context          */
  context_t acx;            /* Activation context         */
  char      *kpm;           /* Kernel call shared memory  */
  char      *pkm;           /* Upcall shared memory       */
} vp_t;

/*
 * Status register
 */
#define VP_STS_ARMSK  0x0F  /* Mask for activation reason */
#define VP_STS_PRMPT  0x01  /* Process was preempted      */
#define VP_STS_ALLOC  0x02  /* Been allocated some time   */
#define VP_STS_EXTRA  0x04  /* Obtained extra resources   */
#define VP_STS_EVENT  0x08  /* Events have been delivered */
#define VP_STS_NEW    0x10  /* New events are pending     */
#define VP_STS_AIP    0x20  /* Activation in progress     */
```

Figure 5.3: Nemo virtual processor interface.

execution is started whenever the processor is given to the process. This field can be altered with a single write by the process. Whenever a process is activated, the `status` field is updated to reflect the reason for activation as explained in section 4.7.1.

While not executing an activation handler, the NTSC code saves a process's machine context in the `ecx` field of the VPI whenever the process loses the processor. If the

machine context needs to be saved while executing an activation handler, it is saved in the `acx` field and this context is *resumed* when the process is again given the processor. This removes the need to write reentrant activation handlers, but also means that a process may not immediately be informed of scheduling events which occur during activation. `kpm` and `pkm` contain the addresses of areas of memory which are shared between a process and the kernel. To perform the equivalent of a system call, a process marshals its arguments directly into the memory pointed to by `pkm` and then executes a call to `sc_kernel`, which causes the NTSC code to deliver an event to the kernel and activate it. Conversely, whenever the kernel wishes to call a process to inform it of the results of a particular system call, it marshals the results directly into the memory pointed to by `kpm` and sends an event to the process to alert it to the presence of the results.

5.3.2 Activation and the VPI

Activation is performed by code within the NTSC as the result of:

- the kernel issuing an `sc_activate` call to give the processor to a particular process;
- an interrupt causing the NTSC to take the processor from the currently running process and activate the kernel and;
- a process issuing a `sc_kernel` call to surrender the processor and activate the kernel.

In the usual case, when a process loses the processor, its context is stored into its VP's `ecx` field. When NTSC code is called to activate the process, its `cvp` variable is updated to point to the VP to be activated, and `ccx` is set to point to the `acx` field of the current VP. The reason for activation is updated in the VPI `status` field, the `VP_STS_AIP` bit is set, and execution is continued at the address specified by the VP's `activation` field in user mode.

The activation handler can now process any pending events and make any scheduling decisions which it requires. If it decides to select a new thread for execution, the saved machine context of the thread which was running when it last lost the processor is available in `ecx`. Any external occurrences which would ordinarily result in an activation cause the `VP_STS_NEW` bit to be set if they occur while an activation is in progress.

To leave an activation handler, a process executes a call to either `sc_rfa()` or `sc_rfar()`. Both of these NTSC calls check to see if any new activations arrived

while the previous activation handler was executing and if so, clear the VP_STS_NEW bit and reactivate the process. If no new activations had arrived, cvp is set to point to the current VP's ecx field. After this, sc_rfa() causes execution to return to the instruction following the call to sc_rfa(). sc_rfar() takes one argument which is a pointer to a context which is to be resumed. A thread scheduler may use this to resume its previous execution context by calling sc_rfar(&vpp->ecx), where vpp is a pointer to its VPI, or it may nominate the address of the saved context of a new thread.

5.4 Building a Nemo System

The NTSC code and the code for each of the processes is a separately compiled and linked object module. A machine dependent configuration utility called build is used to read a configuration file and bind the specified modules together to construct a system image. Figure 5.4 shows an example configuration file. In this

```
#
# Configuration file for Nemo
#
# File    Name        Period   Cycles
#
ntsc      /ntsc
kernel    /kernel
dir       /lib/dir
video0    /proc/0     67       30
video1    /proc/1     67       30
```

Figure 5.4: Example Nemo configuration file.

configuration file, entries in the first column contain the UNIX pathname of the linked object modules which are to be used to build the system. The second column contains the ASCII strings which are used to locate the modules. In the case that the entity is a user process, its name begins with the string /proc/ and the third and fourth columns of its entry in the configuration file respectively contain the period with which processor time is allocated to the process and the amount of processor time which is allocated. The example shows that there are two contracted processes, one called video0 and the other called video1 which are both allocated a processor bandwidth of $(67, 30)$ milliseconds.

In addition to calculating the final addresses of the processes and relocating them, `build` also initialises the important system data structures. Information from the configuration file along with the final addresses of the object modules are used to initialise the directory database. A process control block (PCB) is allocated and initialised for each process in the system, and each process's VP is initialised. The `activation` field of each VPI is set to point to the first instruction in the process's program space. The `nexus` contains a list of addresses which are required by the system at startup. These include the address of the first byte of memory after the end of the image, the address of the kernel's VP and the address of the directory data and operations.

Figure 5.5 shows the layout of the bootable image created by `build`.

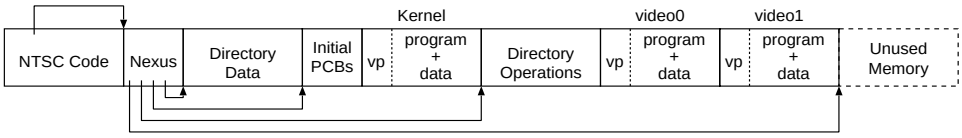


Figure 5.5: Layout of a Nemo system image as constructed by `build`.

5.5 The Bootstrap Sequence

The built image is loaded into memory and execution starts at the first instruction in the NTSC code. This code extracts the address of the kernel's VP from the `nexus` and activates the kernel, passing to it the address of the `nexus` as an argument. The kernel locates the directory data and operations from the information which it retrieves from the `nexus` and uses this to locate the PCBs of the processes which were loaded as part of the system build. Each of the PCBs found is entered into the current schedule and initialisation is completed by entering the dispatcher.

5.6 Resource Management

The Nemo scheduler is part of the kernel and maintains four process queues `cq`, `wq`, `fq` and `bq`. `cq` holds the current schedule of contracted processes sorted such that the most eligible process is at the head of the queue. `wq` contains those processes which have consumed their contracted amount of processor time and are able to make use of more time should it become available. `fq` contains any best effort processes and `bq` contains processes which are blocked, waiting for some event to occur.

Each process's PCB contains a number of fields which are used to maintain resource management information. These fields include `period`, `cycles`, `pnext` and `remaining`. `period` and `cycles` express the processor bandwidth which is allocated to the process. `pnext` is the time at which the process will next be allocated more resources and `remaining` contains the amount of contracted processor time which the process has remaining at any time.

5.6.1 Accounting and Policing

When a process is given the processor, the time is recorded from the clock `current` register and an entry is made in the clock driver timeout queue so that a policing procedure will be called at `current` + `remaining` should the process still be running when its contracted processor time has been consumed. When the processor is taken away from a process, the policing timeout is removed and `remaining` is decremented by the amount of time which the process consumed while it had the processor. If `remaining` is zero, the process is moved from `cq` to `wq`.

5.6.2 Allocation

For each contracted process, a periodic software timer is used to allocate processor time to the process by setting `remaining` to `cycles` every `period` milliseconds and updating the `status` field in the process's VPI. This timer also checks to see if the process is in `wq` because it had used all of its contracted time and if so, moves it from `wq` to `cq`.

5.7 Events

Support for signalling between processes is provided by the Nemo event mechanism.

5.7.1 Creating an Event Channel

Suppose process A wants to create an event channel for signalling events to process B, whose process identifier is 2. A can issue the system call `ev_create("/proc/2", rid)`. `rid` is an integer which identifies a particular instance of the system call; in a multithreaded process, `rid` would typically be the identifier of the thread issuing the system call.

`ev_create` locates the named process and obtains a pointer to B's PCB if the named process exists. It then allocates an entry from an array of structures which is held in A's PCB. Figure 5.6 shows the structure of one of these entries. The fields of

```
/*
 * PCB data required for an event
 */
typedef struct {
  struct pcb *pp;  /* Process receiving events  */
  u_int  *recp;    /* Address of event record   */
  u_int  state;    /* State flags - see below   */
  u_int  rid;      /* Result identifier         */
} pcb_ev_t;

#define PCB_EV_WACC  0x1  /* Waiting for accept */
#define PCB_EV_OPEN  0x2  /* Channel is open    */
```

Figure 5.6: PCB event structure.

this structure are then initialised the state set to `PCB_EV_WACC` and the kernel upcalls both processes. Process A receives a return value which identifies the event channel, along with a copy of the `rid` given in the `ev_create` call so that it can match the upcall results with the request. Process B is upcalled with a request to receive events and is given both a pointer to A's PCB and the identifier of the PCB event structure in A. This aims to provide enough information to enable B to identify the possible source of events and decide whether or not it is willing to receive events from that source. If it does not want to receive events from A, B issues the system call `ev_reject(pp, ev_id)` which causes the event structure to be deallocated in A's PCB and A to be upcalled with an error status. If B is willing to receive events from A, it issues the system call `ev_accept(pp, ev_id, &value)` where `&value` is the address of an integer which is used to record the deliveries of this particular event. This causes the state of the PCB event structure to be set to `PCB_EV_OPEN` and A to be upcalled with a good return status and the appropriate `rid`.

5.7.2 Signalling Events

Once an event channel is in place, A can signal the occurrence of `n` events to B by issuing the `ev_signal(e, n)` system call. Figure 5.7 shows the signalling of an event. The `signal` call uses the event identifier to locate the event channel's

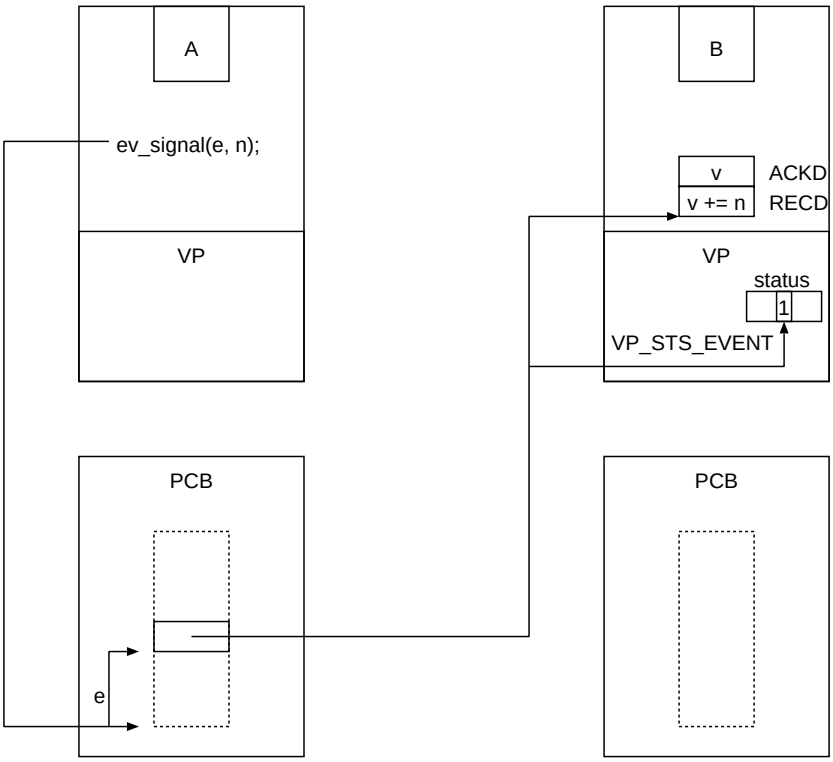


Figure 5.7: Process A signalling the occurrence of n events to process B.

structure in A's PCB. It uses the information contained in this structure to locate the address of the event record, updates the number of events received by B and sets the `VP_STS_EVENT` bit in B's VP status field. The scheduler is alerted that B has received an event, so that if B was in `bq`, waiting for an event, B is rescheduled.

5.8 Summary

An implementation of the design presented in the previous chapter has been presented in the form of the Nemo kernel. The description of Nemo has demonstrated how the mechanisms which are required to provide support for QOS within an operating system might be realised. The implementation also demonstrates that the primitives such as shared memory and events which were proposed in the design for use by application processes are also able to be used within the system itself. These primitives provide a level of abstraction which is not as high as that of message passing. However, they are better suited to a number of CM applications and their use for IPC also affords some performance gains over message passing.

Chapter 6

Evaluation

Earlier chapters have described the design and implementation of mechanisms which can be used within an operating system to provide QOS contracts between the system and applications. These mechanisms are evaluated with respect to their ability to be used by applications. The work presented is then compared with other, related work.

6.1 Experimental Assessment

Two important areas of the system which have to be evaluated are the suitability of the VPI for providing applications with the information they need to meet their own QOS requirements and the QOS mechanisms. The application which is used to test the system is a variant of the JPEG video decoder which was described in section 3.1.

6.1.1 Application Use of the System VPI

The aim of this section is to evaluate the usefulness of the VPI in terms of the ease or otherwise with which applications may make use of it. Such an assessment is necessarily subjective but does serve to demonstrate basic functionality.

When the system is built, the `activate` field of the decoder's VPI is set to point to the address of the decoder's first instruction. The code at this address initialises the application's environment and installs the initial activation handler. The decoder runs in two phases. While initialising its internal data structures, it arranges to have its saved context resumed by installing a default activation handler. When

initialisation is completed, the decoder arranges to be informed of the resources which are allocated to it by installing a second activation handler which decodes and displays video frames. Activation handlers typically consist of an assembler stub

```
        /*
         * Default activation vectoring code
         */
        .globl  jd_avec_default
        .ent    jd_avec_default
jd_avec_default:
        la      gp,_gp          /* Init global pointer */
        la      sp,jd_astack    /* Init stack pointer  */
        jal     jd_act_default /* Call handler        */
        .end    jd_avec_default
```

Figure 6.1: Assembler part of default activation handler.

and a body which is written in C. Figure 6.1 shows the default activation vectoring code used while the decoder is initialising itself and figure 6.2 shows the code which implements the body of the handler. Whenever the decoder loses the processor,

```
vp_t  *my_vpp;    /* Pointer to application's VPI */

void
jd_act_init()
{
  if (my_vpp->status & VP_STS_EVENT) {
    ev_handle();
  }
  sc_rfar(&my_vpp->ecx);
}
```

Figure 6.2: Body of default activation handler.

its context is saved in the `ecx` field of its VPI by the NTSC code. When the decoder regains the processor, the vectoring code initialises the minimum amount of environment required then calls the body of the handler. `gp` is reserved by the MIPS R3000 register usage convention and is used by the compiler to refer to data held in a global data segment. Since, in this design, activation handlers are not reentrant, an execution stack can be statically allocated and its address used to initialise the stack

pointer with two machine instructions.[1] The default activation handler checks for the arrival of new events and handles any which are pending then resumes the context saved in its VPI's `ecx` field. Using this default activation handler, the decoder executes in the same manner as a conventional process, resuming its context from where it was saved when the processor was taken from it. There is an important distinction to be made between using the default activation handler to restore the state of a process and having the system do it; use of the activation handler allows an application to decide whether or not it wants to restore its saved state, rather than having this action always forced upon it by the system. An example of when an application sometimes does not want to resume its saved context is provided by the activation handler which is used by the decoder when its initialisation has completed and it is decoding and displaying frames.

Processor cycles are allocated to the decoder periodically at the frame display rate. At the start of each period, the decoder displays the results of its attempt to decode the previous frame and then begins decoding the next frame. This strategy allows the decoder to maintain temporal correctness by sacrificing the logical correctness of its results. Logically incorrect results manifest themselves as frames of video which are displayed on time but which are not completely decoded.

The vectoring code for the decoding activation handler is similar to that of the default activation handler shown in figure 6.1. Figure 6.3 shows the body of the handler. When the handler is entered, the decoder's previous context is saved in the VPI's `ecx` field. After detecting and processing any pending events, the handler determines the reason for the current activation by examining the bits in the `status` field of its VPI.

If the `VP_STS_ALLOC` bit in the VPI `status` register is set, the decoder has just been allocated some processor cycles to decode the next frame. `sc_rfa` is called to leave activation mode and discard any context which was saved in the VPI's `ecx` field. The results (possibly incomplete) of the attempt to decode the previous frame are presented on the frame buffer by a call to `jd_show` and `jd_frame` is called to initiate the decoding of the next frame. If the current contract affords sufficient time to decode the frame completely, the kernel is informed that the decoder wants to give up the processor until the next time cycles are allocated. This is done by marshalling the argument `KC_WAIT_ALLOC` into process/kernel shared memory and executing the NTSC call `sc_kernel` which reflects the call in the kernel process as an event. This call does not return to the activation handler as would the invocation of a normal C function.

[1] The `la` (load address) opcode is expanded by the assembler into a `lui` (load upper immediate) followed by an `addiu` (add immediate unsigned).

```
char  *fr;        /* Buffer for one decoded frame */
vp_t  *my_vpp;    /* Pointer to application's VP  */
vd_t  *vd;        /* Video stream descriptor      */
pk_t  *pkp;       /* Decoder/kernel shared memory */

void
jd_act_display()
{
  if (my_vpp->status & VP_STS_EVENT) {
    ev_handle();
  }
  switch (my_vpp->status & VP_STS_ARMSK) {
    case VP_STS_ALLOC:
      sc_rfa();
      jd_show(fr, vd->wp, vd->width, vd->height);
      jd_frame(vd);
      pkp->op = KC_WAIT_ALLOC;
      sc_kernel();
    case VP_STS_PRMPT:
    case VP_STS_EXTRA:
      sc_rfar(&my_vpp->ecx);
  }
}
```

Figure 6.3: Body of the decode/display activation handler.

If the activation was caused by another process preempting the decoder, then `VP_STS_PRMPT` will be set, causing the decoder to resume its previous context from `ecx`. If, having used its contracted cycles, the decoder is allocated extra cycles by the system, `VP_STS_EXTRA` will be set, which also causes the decoder to resume its previous context.

The overall effect of these actions on execution of the decoder is that `jd_frame` is called whenever the application is allocated its share of the processor and continues to execute until either the application is next allocated resources or until it completes decoding of the current frame and calls `sc_kernel` to return any unused resources to the system. This allows the decoder to maintain the timeliness with which it delivers frames even when it does not have sufficient resources to decode them completely. In more general terms, it also demonstrates that the system can provide applications with both QOS contracts and sufficient information to enable them to control their

behaviour when they do not receive all of the resources which they require.

6.1.2 Interaction With QOS Mechanisms

To demonstrate the behaviour of the basic QOS mechanisms, the decoder is allocated a processor bandwidth of $(40, 100)$ milliseconds and executed on an otherwise unloaded system. The decoder is written so that, should it not require all 40 milliseconds to decode a frame, it returns any unused processor cycles to the system. Should the system find that it has any unused idle time, it will choose a process which is capable of using more cycles and offer them to the process. The kernel was instrumented with code to log the occurrence of events during a run and dump the log after completion of the run. During execution, the cycles accumulated by the decoder during the 100 millisecond period assigned for decoding frame f were logged at times t_f^p, when the policing function was activated and t_f^s when the decoder surrendered the processor to the kernel to wait for the next allocation of resources. Figure 6.4 shows a summary of this data. The solid line plots $\min_f(t_f^s, t_f^p)$ versus

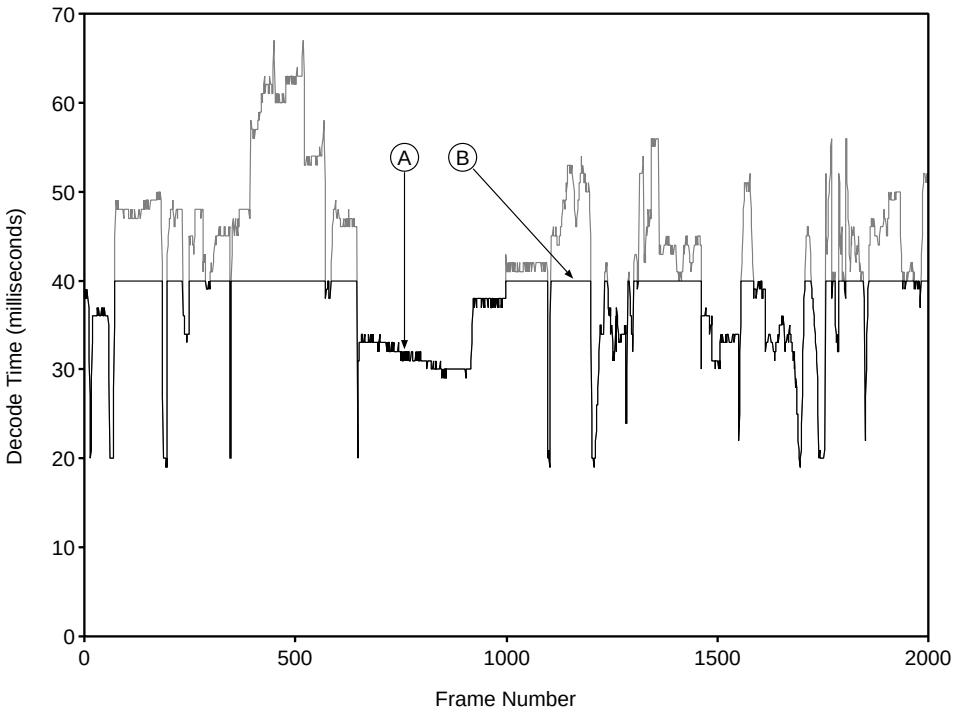


Figure 6.4: Contracted and additional processor times versus frame number.

frame number for 2000 frames of a video sequence. This represents the decoder's use of its contracted processor cycles and the fact that this line never exceeds 40 milliseconds demonstrates the actions of the system policing mechanism. The stippled line plots the t_f^s versus frame number which were recorded in the same run.

This represents the total time required by the decoder to display the previous frame and completely decode the next frame.

At point A on the graph, the decoder has decoded the frame within its contracted 40 milliseconds and surrendered the processor before it was policed. With reference to the activation handler shown in figure 6.3, execution of the handler has reached the `sc_kernel` call within the contracted amount of processor time.

At point B on the graph, the decoder was unable to decode all of the next frame within its contracted time, causing the policing mechanism to intervene while the decoder was executing the `jd_frame` routine. The system saved the current context in the `ecx` field of the decoder's VPI and tried to give the processor to another, contracted process. Since, in this experiment, there were no other processes to execute, the system allocated extra time to the decoder. This resulted in the decoder being activated with `VP_STS_EXTRA` set in its VPI's `status` field. The handler resumed the saved context and the decoder continued executing until the call to `jd_frame` completed, when it surrendered the processor to wait for the start of the next resource allocation period. This graph demonstrates the nature of resource allocation within the system. Applications will always obtain a minimum amount of processor time, and may receive additional time if the system has nothing better to do.

In the case where multiple contracted processes are able to make use of additional resources, some algorithm for choosing the processes most eligible to receive extra processor time is required. The current implementation simply gives this time to processes in a fixed order, but it is anticipated that within a workstation environment, the user will be provided with an interface which enables this order to be modified according to their directions. Figure 6.5 shows the processor times in milliseconds which were accumulated by ten decoders, each decoding the same video stream, plotted against the frame number within the stream. The stream consists of the first 100 frames of the stream used to obtain the graph of figure 6.4 and each of the decoders was allocated a processor bandwidth of $(30, 400)$ milliseconds, giving a total contracted processor bandwidth of $(30 \times 10, 400) = (300, 400)$. The maximum processor time which can be allocated to each of the decoders before the system is overloaded is $400/10 = 40$ milliseconds. Three sections of this graph are of interest. Point A marks a frame which requires less than 30 milliseconds to decode, so all applications return their unused processor time to the system and the system idles. Point B marks a frame which requires a processing time which is between 30 and 40 milliseconds; each of the decoders has consumed its contracted 30 milliseconds and then been flagged as a candidate for extra resources should they become available. The system then offers idle time to the decoders in order starting with number 9 and working down to number 0. For this frame, the system has sufficient idle time for all of the decoders to complete decoding the frame. Point C marks a frame which

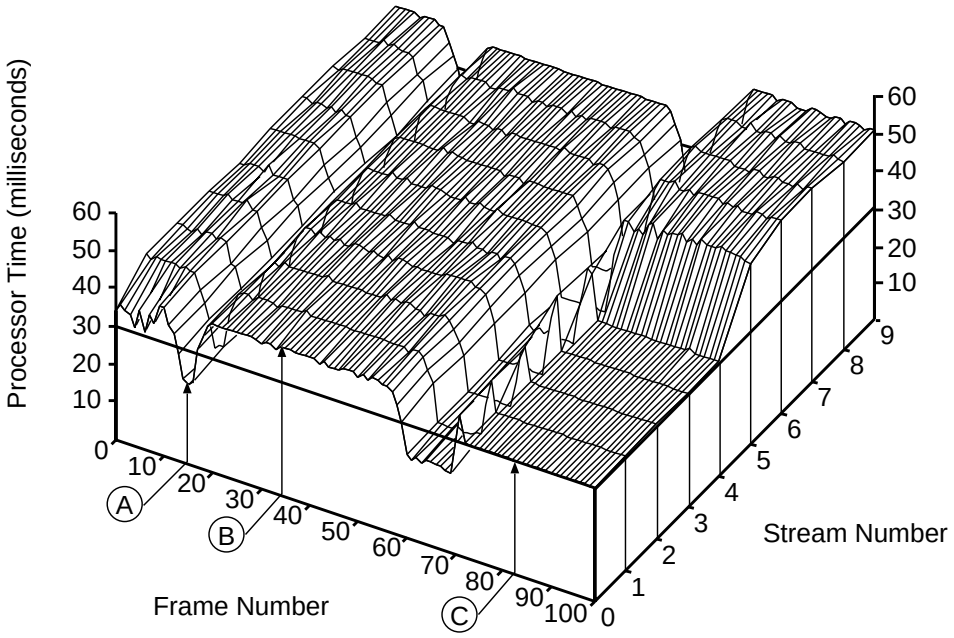


Figure 6.5: Processor time obtained by ten decoder applications.

requires more than 40 milliseconds to decode, so that there is insufficient processor bandwidth in the system to completely decode all 10 copies of this frame and the system experiences transient overload. Each decoder obtains its contracted 30 milliseconds, then additional time is offered in order of descending number. This frame requires so much time to decode completely, that only four of the decoders succeed in doing so; the rest can only decode part of the frame.

At point C, this graph also demonstrates a number of important properties which the system exhibits as a result of using QOS mechanisms to allocate resources:

- even though the system is overloaded, all of the decoders still receive their minimum contracted PB on time and;
- not only is the system degrading gracefully, but the manner in which it degrades can be directly controlled by allowing the user to specify the algorithm used for allocating extra processing time to contracted processes.

6.1.3 Varying Application QOS

It has been shown how the system can control the resources used by applications, and also how applications can detect when they are not receiving enough resources. The usefulness of such a system depends on the ability to construct applications which

are capable of producing acceptable results when resources are scarce. This section describes an implementation of the decoder application which has these properties.

6.1.3.1 Layered Processing

The video decoder uses the JPEG algorithm, an outline of which is given in section 3.1. At the centre of this algorithm is a loop which converts the incoming compressed bit stream into a sequence of 8×8 tiles of DCT coefficients. An IDCT is performed on each of these tiles to obtain an 8×8 array of pels. When the decoder is sequentially processing tiles, a shortage of processor time during a sequence of frames causes areas at the bottom of the picture not to be updated. For many video presentation applications, this form of degradation may not be acceptable.

An alternative method of processing the sequence of tiles representing one image uses *layered processing*. As each coefficient tile is reconstructed from the incoming bit stream, its individual coefficients are recorded and an approximation to the picture elements represented by the tile is generated and stored into the resultant array of image pels. When all of the tiles have been received, the decoder has an initial approximation to the image which it can further refine as more cycles are allocated to it. A version of the decoder was constructed in which the first layer of processing generates its approximation by taking the first non-zero coefficient in the tile,[2] setting all other coefficients in the approximation tile to zero and calculating the IDCT of this. An optimisation in the IDCT implementation means that calculating the IDCT of a tile containing only one non-zero coefficient can be done reasonably cheaply.

The second layer of processing refines the picture by recalling all of the tiles which contain more than one non-zero coefficient[3] and calculating their corresponding picture elements exactly. Figure 6.6 shows an image in which the first layer processing has completed and the decoder has been given enough time to refine roughly half of the tiles in the second layer. In this image, there are a total of $64 \times 64 = 4096$ tiles, 347 of which contain a single non-zero coefficient. In terms of the types of imprecise computations discussed in section 3.11.3 this version of the decoder is using the milestone method, each layer in the processing representing a milestone. This is to some extent an improvement on only processing part of the picture completely, but the lower areas of the image still need refining. The picture quality could be improved by increasing the number of DCT coefficients processed in the first layer; this would increase the cost of computing the first layer and, correspondingly increase

[2] In zigzag order.

[3] Tiles containing only one non-zero coefficient will have been processed completely by the first layer; the "approximation" in these cases being the desired result.

Figure 6.6: Partially decoded frame: milestone version.

the minimum amount of processor time below which the decoder can do little useful work. The fundamental problem with the second layer processing as just described is that it refines the picture from left to right and from top to bottom, so some of the extra processing time is spent refining "uninteresting" areas of the image.

Recognising this, a second version of the decoder was produced in which the first layer processing sorts tiles into a number of classes according to the number of non-zero coefficients they contain. At the same time, the coordinates of the tile within the resultant image are also recorded. Second layer processing then refines the tiles starting with those which have the most non-zero coefficients and working down towards those which have the least.[4] Figure 6.7 shows the image which results when the decoder has spent the same amount of time in second layer processing as was spent in figure 6.6. Ordering the processing of the second layer has converted a computation with two milestones (layers) into one which has one milestone

[4]These will be the tiles with two non-zero coefficients.

Figure 6.7: Partially decoded frame: milestone/monotone version.

(the first layer) and after reaching this milestone becomes monotone. This type of computational requirement is easy to schedule within Nemo by allocating a processor bandwidth which ensures that the first milestone will be met (or exceeded by some required amount), then using any spare processor time to refine the remaining monotone part of the computation.

The aim in choosing the second layer heuristic was to process first those parts of the picture which contain the most information, thus focussing the remaining processor time on those areas of the picture which will benefit from it the most. This is a rather arbitrary heuristic which is seen to work well in practice when applied to a number of different video streams, but there is clearly scope for further work in the development of such heuristics for use in layered processing applications. This is especially the case if the video is stored. While the decoder applies its simple heuristic in real-time as tiles are reconstructed from the input stream, the ability to preprocess a stored video stream offline allows more complex algorithms to determine

a (possibly optimal) processing order for the bits which comprise a frame and to present these bits to applications in this order.

6.1.4 User Level Threads

The decoder application does not make full use of the activation mechanism; a more complex application might want to make use of a user-level thread package. In such a case, the activation handler can be written to call into the user level thread scheduler, passing it a pointer to the context which was saved at the time the process last lost the processor. The thread scheduler can decide, based on which external events have been received, whether to use the context saved in `ecx` to save it and resume that of another thread. Whatever the decision, the chosen context can be resumed at the end of the handler by calling `sc_rfar`.

6.2 Comparison With Related Work

Support for continuous media applications is an active area of research and there is a considerable corpus of related work. The work selected for comparison in the following sections was chosen for its direct relevance to the work presented in previous chapters.

6.2.1 Scheduling

[Coulson93] describes work which aims to provide a set of low level abstractions for programming distributed CM applications and provide them with pre-specified, guaranteed QOS constraints. The basic system support for these abstractions is the Chorus [Bricker91] microkernel, which provides a number of real-time facilities such as page locking, preemptive scheduling, system call timeouts and scheduling classes. It is noted that, while the Chorus microkernel is in use within a number of real-time systems, it does not provide facilities for controlling application QOS or reserving resources to meet QOS guarantees and that, while it is possible to specify thread scheduling constraints relative to other threads, there is no way to specify absolute thread scheduling requirements. Within the system described, thread scheduling uses an EDF policy and does not guarantee that deadlines will be met, so QOS guarantees may be violated when the system becomes overloaded. The suggestion is made that this could be avoided by using a suitable admission control algorithm. The success of such an algorithm will depend on the amount of information available

about the resource requirements of the threads being scheduled. The incorporation of a policing mechanism could help prevent QOS guarantees from being violated when thread resource requirements are underestimated.

[Mercer93] describes the processor capacity reserve mechanism. A reserve represents access to a certain processor capacity, expressed as a computation time and a reservation period. Processes with reserves are given the processor in preference to time sharing processes. Processes present the system with requests for reservations and the system determines whether it can accommodate their requests using a rate monotonic admission test. At the beginning of every reservation period, a process with a reservation is allocated its reserved processor capacity. When this has been consumed, the process is scheduled under the time sharing policy until it receives its next allocation. Accounting for the use of reserves by server threads is performed by passing the client's current reserve to the server which charges its computation time to the client. This resource allocation strategy is similar to that used by Nemo, with processor capacity equating to PB. Relegation of processes which have consumed their reserved capacity to the time sharing scheduling policy loses the ability to focus additional processing time on a favoured process as might be required to take advantage of any of the benefits of statistical multiplexing.

6.2.2 Virtual Processor Interface

Viewing an operating system as a provider of a VPI is a well established concept in computing systems work; [Leffler89] describes the development of the UNIX virtual machine and the modifications which were required to remove race conditions in the signal delivery mechanism and provide signal masking facilities. This version of UNIX also makes certain kernel information available to applications by mapping their `u` areas into their address spaces and allowing them read only access to it.

The use of threads as a means of obtaining concurrency and clarity has motivated further changes in the VPI, primarily to facilitate the implementation of user level threads. [Anderson92] describes *scheduler activations*, which closely resemble kernel threads. Typically, there is one active scheduler activation per physical processor; user level threads are multiplexed on a scheduler activation by the user level thread scheduler. The kernel informs the user level scheduler of scheduling events by allocating a new scheduler activation and upcalling the user level code at a fixed address, passing the context of any blocked activations as arguments. The user level scheduler informs the kernel of user level events by calling into the system. Scheduler activations were implemented on a uniform memory architecture multiprocessor, so among the scheduling events are indications of: when a process loses or gains processors and; when a process has idle processors. A process which has a single

processor and loses it does not find out that it lost the processor until it is next given another processor, whereupon the kernel allocates a scheduler activation and upcalls the user level scheduler. Nemo provides the same information to a process by activating the process and informing the process that it has been activated because it has just obtained the processor. In the scheduler activation scheme, when a user level scheduler finds itself with an idle processor, it is obliged to surrender that processor by calling into the system; failure to do this results in an unfair usage of system resources by the process. No direct mechanism is provided for limiting the impact of such selfish processes on the rest of the system. Instead, the reactions of the multilevel feedback scheduler are relied upon to identify the process as computationally bound and prevent it from interfering with more interactive processes. Nemo's policing mechanism can be used to limit the impact of such processes on the rest of the system.

Psyche [Marsh91] provides another example of a virtual processor interface which has been augmented to provide support for large scale user level parallelism. Kernel threads are used to implement virtual processors of which typically one is allocated per physical processor. User level schedulers multiplex threads on top of these virtual processors and are informed of scheduling events by the kernel via virtual processor interrupts. These are generated in response to kernel events including: virtual processor initialisation; threads blocking and unblocking in the kernel; signals from other virtual processors and; an interrupt warning of imminent preemption. User level schedulers and the kernel communicate via a piece of shared memory which is part of the virtual processor interface. The use of shared memory to interface the user and kernel schedulers reduces the number of protection domain crossings and advantage is also taken of this mechanism in Nemo. Of particular interest in Psyche is a "two-minute warning" interrupt which alerts a process that it is about to lose the processor. This aims to provide a virtual processor with a hint so that it can clean up when it is about to lose a physical processor. This is not a guarantee that there will be enough time to complete the clean up, but it is intended to minimise the likelihood of inopportune preemption.

[Govindan91] describes the ACME continuous media I/O server as a typical CM application along with its performance when running under a typical workstation operating system. The observations are that the application's behaviour suffers from timing errors and lost data when running concurrently with other system activity, and cannot meet the low delay requirements of even moderate audio data formats. It is claimed that these problems are due in part to the overhead of the user/kernel interaction mechanisms by which user level programs invoke system functions such as CPU scheduling and I/O. Split Level Scheduling (SLS) is proposed as an operating system mechanism for supporting CM applications. SLS presents to applications a virtual processor which is implemented as a kernel thread and incorporates time in

the form of deadlines into the interface between the Kernel Level Scheduler (KLS) and the User Level Scheduler (ULS). Incorporation of thread deadlines into the virtual processor interface means that it is possible for the kernel to allocate the processor to the address space which contains a runnable thread with the earliest deadline and similar functionality is achieved within Nemo. The SLS interface is quite complex; for example, it allows the kernel to examine the contents of thread descriptor queues and I/O descriptors. This gives rise to synchronisation requirements which are met by enabling the application to disable its preemption from user level code. Interrupts presented by the SLS to the ULS include `INT_RESUME` which occurs when the address space is given the processor and `INT_TIMER` which occurs when the address space's software timer expires. The means by which interrupted thread context is made available to the ULS is not specified.

Scheduler activations, Psyche's virtual processors and the Split Level Scheduler all have the common goal of providing an efficient means by which a kernel thread scheduler can communicate with a user level thread scheduler so that the overall cost of providing user level parallelism is reduced. The incorporation of deadlines into the SLS interface makes this goal more applicable to use within CM applications. While Nemo's virtual machine interface is similar in many respects to these, the reasons for it being so are in many respects quite different. The Nemo kernel does not know about process's threads; its sole responsibility is to apportion the available processor cycles to processes in the manner dictated by the QOS manager. I/O is performed by device driver processes, and applications wishing to perform I/O communicate directly with those processes via IPC rather than through the kernel. This obviates the problem of what to do when a thread blocks in the kernel because it is waiting for slow I/O to complete. Nemo threads do not enter the kernel as do the usual kernel thread implementations. Threads are scheduled at user level within processes and when a user level thread scheduler decides that it can do no more useful work because all of its threads are blocked or because it must wait until the right time, it surrenders the processor to the kernel using the `sc_kernel` NTSC call. Execution within the kernel occurs only as part of the kernel itself and is not done as part of a user process's execution. Consequently there is no need for a process to have an associated kernel stack and all the NTSC requires of a process is a place to store a process's context when that process is deactivated. In contrast to the systems reviewed which use the scheduling information provided by virtual processor interrupts solely to implement efficient user level threads, Nemo applications can additionally make direct use of this information to control their QOS as illustrated by the example of section 6.1.1.

6.2.3 QOS

[Coulson93] describes abstractions for providing and maintaining QOS guarantees to distributed CM applications. These mechanisms include `rtports` which are end points for CM communications and `handlers`. Whenever data is sent to an `rtport`, any associated `handler` is invoked. It is claimed that real-time programming is simplified by structuring applications to react to events which are generated by the system, and that the use of `handlers` reduces the number of protection domain crossings required to deliver data to an application; the requirement of having the application call into the system to request data is removed. The functionality provided by `rtports` and `handlers` can be implemented directly from the shared memory and event mechanisms provided by Nemo. Since `rtports` which are bound between address spaces[5] use the standard Chorus IPC mechanisms, they incur the overhead of copying or remapping their data. Shared memory IPC and an appropriate event mechanism can obviate these. The QOS of data associated with an `rtport` can be specified as a vector of parameters including: `guarantee`, the desired degree of certainty with which the requested QOS is to be provided; `delay` and `jitter` which specify the temporal requirements of the `rtports`'s data, and determine the scheduling requirements of the `handler`. These QOS specifications correspond to the low level QOS parameters described in section 3.2. The discussion does not contain any information on the exact effects of the `guarantee` parameter or on how the guarantees it offers may be quantified.

[Tokuda92] presents the Capacity Based Session Reservation Protocol (CBSRP), which reserves system resources for CM applications in order to guarantee their QOS. Qualities of CM services are expressed in terms of temporal and spacial resolution: temporal resolution may be mapped to frames per second of video or samples per second for audio and spacial resolution may be mapped to bits per display pixel or maximum spacial frequency resulting from a video compression algorithm. In their terminology, spacial and temporal resolution are QOS parameters, and are chosen so that they may easily be mapped onto a reasonable set of lower level system attributes such as processor and memory allocations. As a result of this, the user is presented with a selection of QOS classes from which to choose. This definition of a QOS parameter is equivalent to what is referred to as an application or high level QOS parameter in Nemo, and the low level system attributes correspond to system or low level QOS parameters. A possible advantage of identifying system QOS parameters as such is that, if there is no mapping from an application to a system QOS parameter, then QOS may be specified directly in terms of system QOS parameters. The system entities which are involved in the provision of QOS guarantees are the Session Manager (SM), which handles creation, termination and

[5] *Actors* in Chorus terminology.

reconfiguration requests from users and renegotiates with remote session managers, System Resource Manager (SRM) and Network Resource Manager (NRM) which handle admission control and resource management. The session manager performs the equivalent function to Nemo's QOS manager, the functions of SRM and NRM being performed by equivalent management entities within the system and network domains. [Nicolaou91] makes a strong case for the construction of QOS management facilities as a collection of subsystem management domains such as SRM and NRM which engage in negotiation in terms of QOS parameters. The establishment of interfaces which are well defined in terms of QOS parameters and classes between QOS management subsystems can improve the modularity of the resulting system and, in the case of the pseudo code presented for the SM, would remove the need for a high level entity to have to calculate subsystem specific resource requirements such as MAC layer bandwidth requirements.

6.3 Summary

Nemo is evaluated with respect to both its ability to provide the resource management facilities required to support application QOS and the usability of the VP interface which it presents to applications. Correct behaviour of the resource management mechanisms is demonstrated, and an example video decoder application is presented as an example of how an application can make use of the resource availability information provided to it by the system via the VPI. The work presented in this dissertation is then compared with other, related work. It is shown that, while the facilities provided by the Nemo VPI are similar to those provided in other systems, Nemo differs in its use of the VPI to provide resource availability information for applications to use in maintaining their application-level QOS.

Chapter 7

Conclusions and Further Work

The original aim of this dissertation was to investigate how support for CM applications can be provided within a workstation operating system. It was noted that the increasing speed of microprocessors has made it possible to write programs which handle video and audio in real-time, so that CM become another data type.

7.1 Contributions

In chapter 2 it was shown that fast processors, while necessary, are not sufficient to support CM applications. If CM applications are to produce acceptable results, then their temporal requirements need to be taken into account when they are scheduled. The possibility was then explored of employing conventional real-time techniques within a workstation operating system and scheduling CM applications as a separate class of real-time processes. It was demonstrated that these techniques are not well suited to coping with the dynamic resource demands made by CM applications, or with the fluctuating resource availabilities typical of a workstation environment. Their behaviour during transient overload may be predictable, but can lead to processor starvation, which is not desirable. While hard real-time techniques allow programmers to make strong assertions about the run-time behaviour of their programs, they often lead to unacceptably poor resource utilisations when used to schedule CM applications. Soft real-time techniques are tolerant of timing errors, but they allow only weak assertions about run-time behaviour to be made.

The root cause of these shortcomings lies in the nature of the CM data themselves. Their unique temporal and informational properties set them apart from other data, and the resource allocation techniques used in conventional real-time systems do not readily allow these properties to be exploited. Within the ATM networks used to

transport CM data, resource allocation techniques based on QOS are used. QOS is in many respects better suited to handling CM data in that not only does it encompass the requirements of real-time and best effort traffic, but it also accommodates resource requirements which are less than absolute but stochastically quantifiable. This provides the motivation for investigating the suitability of QOS as a resource allocation paradigm within an operating system.

Chapter 3 shows how QOS can be supported within an operating system using a number of mechanisms including accounting, policing and suitable run-time resource allocation. In addition to this, it is shown that QOS may expressed as a high level description or as a set of low level parameters, and that it is possible to convert from one to the other. The suitability of QOS for scheduling processes whose resource requirements are bursty and may not be known exactly is demonstrated. The success of these techniques rests on an application's ability to produce results which are less than perfect, but still acceptable when it is allocated fewer resources than it would desire. Imprecise computations are presented as a means of exploiting the informational property of CM to allow programs to trade the quality of their results for the amount of processing resource which they receive. The construction of programs which make use of imprecise computations can be greatly simplified if, at run-time, they are provided with some knowledge of their current resource availabilities.

Chapter 4 presents the design of an operating system which incorporates mechanisms required to support QOS and extends the traditional concept of the virtual processor interface to incorporate information about the computational resources which are being made available to a process. The design includes an IPC mechanism which is constructed from shared memory and event mechanisms. These mechanisms provide a lower level of abstraction than message passing, but they reflect more accurately the separation of data transfer and synchronisation which is common in many CM applications.

In chapter 5, Nemo is presented as an implementation of some of the mechanisms designed in chapter 4. This implementation is evaluated in chapter 6 where the effectiveness of the QOS mechanisms is demonstrated. It is shown that the system is capable of providing an application with a minimum level of service even when overloaded. Whenever it is available, any unreserved processor time can be allocated to a process in addition to that received as part of its contract. It is shown that the QOS mechanisms allow the behaviour of the system during transient overload to be controlled. A subjective evaluation of Nemo's virtual processor interface shows that it is straightforward to use and can provide applications with the resource availability information they require. This is demonstrated by the construction of a real-time video decoder which uses this information in conjunction with imprecise computations to trade run-time for video image quality.

7.2 Further Work

While it would be possible to incorporate QOS mechanisms into an extant operating system, the approach chosen within this dissertation has been to build a new operating system from the ground up. A number of factors motivate this choice. Firstly, CM data have properties which make them different from other data, and the environment provided by an existing operating system might stifle attempts to exploit these properties. Secondly, current operating systems have not kept pace with recent advances in the design of operating systems mechanisms, workstation and processor architectures and network host interfaces. Incorporation of this technology will be essential if an operating system is to realize the full potential of CM applications. Thirdly, the current corpus of experience in building operating systems provides considerable guidance in the matter.

While this dissertation presents a set of resource allocation mechanisms suitable for scheduling CM applications and it is believed that they are complete, they form only the lowest layer of an operating system. Further work is required to complete the system and validate the design. At the system level, a QOS manager needs to be implemented which is capable of evaluating QOS contracts over multiple QOS domains, taking into account resource requirements other than just processor time. The integration of network and operating system QOS for the provision of end-to-end QOS along a path spanning multiple machines and applications needs to be investigated. At the application level, suitable representations of CM data need to be found which enable applications to exploit their temporal and informational properties. Many algorithms which manipulate CM data in real time have been designed for direct implementation in hardware. These may need to be redesigned if they are to perform well on a general purpose processor. It is also hoped that future experience will reveal the extent to which imprecise computations can be employed in applications more complex than the real-time display of video.

Bibliography

[Accetta86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. *Mach: A New Kernel Foundation for UNIX Development.* Technical Report, School of Computer Science, Carnegie Mellon University, August 1986. (pp 50, 52)

[Anderson92] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. *Schedular Activations: Effective Kernel Support for the User-Level management of Parallelism.* ACM Transactions on Computer Systems, 10(1):53–79, February 1992. (p 90)

[Audsley91] N. Audsley, A. Burns, M. Richardson, and A. Wellings. *Hard Real-Time Scheduling: The Deadline Monotonic Approach.* In Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta. IEEE, May 1991. (p 10)

[Bae91] J. Bae and T. Suda. *Survey of Traffic Control Schemes and Protocols in ATM Networks.* Proceedings of the IEEE, 79(2):170–189, February 1991. (p 16)

[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 Performence and Compatibility.* Technical Report CS/TR-91-7, Chorus Systemes, February 1991. (pp 54, 89)

[Campbell93] A. Campbell, G. Coulson, F. García, D. Hutchison, and H. Leopold. *Integrated Quality of Service for Multimedia Communications.* Technical Report MPG-93-17, Department of Computing, Lancaster University, Lancaster, LA1 4YR, U.K., 1993. (pp 21, 34)

[CCITT90] CCITT. *Video Codec for Audiovisual Services at $p \times 64$ kbit/s.* CCITT Recommendation H.261, December 1990. (p 15)

[CCITT92] CCITT. *BISDN User Network Interface Layer 3 Specification for Basic Call/Bearer Control.* Draft Text for Q.93B — Siemens-typed version, October 1992. (p 32)

[Coffman73] E. Coffman and P. Denning. *Operating Systems Theory.* Prentice-Hall Inc., Englewood Cliffs, N. J., 1973. (pp 30, 39)

[Coulson93] G. Coulson and G. Blair. *Micro-kernel Support for Continuous Media in Distributed Systems.* Technical Report MPG-93-04, Department of Computing, Lancaster University, Lancaster, LA1 4YR, U.K., 1993. (pp 89, 93)

[DEC92] DEC. *Introduction to Designing a System with the DECchip(tm) 21064 Microprocessor.* Technical Report, Digital Equipment Corporation, Maynard, Massachusetts, April 1992. (p 50)

[Dertouzos74] M. Dertouzos. *Control Robotics; The Procedural Control of Physical Processes.* In Information Processing 74, pages 807–813. North-Holland Publishing Company, 1974. (p 11)

[Geppert93] L. Geppert. *Not Your Father's CPU.* IEEE Spectrum, 30(12):20–23, December 1993. Special Report: Platforms. (p 1)

[Ghanbari89] M. Ghanbari. *Two-Layer Coding of Video Signals for VBR Networks.* IEEE Journal on Selected Areas in Communication, 7(5):771–781, June 1989. (p 45)

[Govindan91] R. Govindan and D. Anderson. *Scheduling and IPC Mechanisms for Continuous Media.* ACM Operating Systems Reviews, 25(5):68–80, October 1991. (p 91)

[Greaves92] D. Greaves and L. French. *ATM Drop Cable Interface for the DEC TURBOchannel.* Technical Report, Olivetti Research Ltd., Cambridge, England, February 1992. (p 2)

[Hayter91] M. Hayter and D. McAuley. *The Desk Area Network.* ACM Operating Systems Reviews, 25(4):14–21, October 1991. (p 2)

[Hayter93] M. Hayter. *A Workstation Architecture to Support Multimedia.* Ph.D. thesis, University of Cambridge, September 1993. (p 55)

[Hildebrand92] D. Hildebrand. *An Architectural Overview of QNX.* In USENIX Workshop Proceedings Micro-kernels and Other Kernel Architectures, pages 113–126, April 1992. (pp 50, 52)

[Hopper90] A. Hopper. *Pandora — An Experimental System for Multimedia Applications.* ACM Operating Systems Review, 24(2):19–34, April 1990. (p 1)

[ISO/IEC91] ISO/IEC. *Coded Representation of Picture, Audio and Multimedia/Hypermedia Information.* Committee Draft ISO/IEC CD 11172, December 1991. (p 15)

[Jardetzky92] P. Jardetzky. *Network File Server Design for Continuous Media.* Ph.D. thesis, University of Cambridge, October 1992. (p 2)

[Jeffay92] K. Jeffay, D. Stone, and F. Smith. *Kernel Support for Live Digital Audio and Video.* Computer Communications, 15(6):388–395, July/August 1992. (p 12)

[Kane88] G. Kane. *MIPS Risc Architecture.* Prentice-Hall Inc., Englewodd Cliffs, NJ 07632, 1988. (p 52)

[Leffler89] S. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989. (pp 28, 90)

[Lehoczky87] J. Lehoczky, L. Sha, and J. Strosnider. *Enhanced Aperiodic Responsiveness in Hard Real-Time Environments.* In Proceedings Real-Time Systems Symposium, pages 261–270, December 1987. (p 10)

[Lehoczky89] J. Lehoczky, L. Sha, and Y. Ding. *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour.* In Proceedings Real-Time Systems Symposium, pages 166–171, December 1989. (p 10)

[Leslie91] I. Leslie and D. McAuley. *Fairisle: An ATM Network for the Local Area.* Computer Comunications Review, 21(4):327–336, September 1991. (p 2)

[Liu73] C. Liu and J. Layland. *Scheduling Algorithms for Multprogramming in a Hard Real-time Environment.* Journal of the Association for Computing Machinery, 20(1):46–61, February 1973. (pp 9, 10)

[Liu91] J. Liu, J. Lin, W. Shih, A. Yu, J. Chung, and Z. Wei. *Algorithms for Scheduling Imprecise Computations.* I.E.E.E. Computer, 24(5):58–68, May 1991. (p 44)

[Marsh91] B. Marsh, M. Scott, T. LeBlanc, and E. Markatos. *First-Class User-Level Threads.* ACM Operating Systems Review, 25(5):110–121, October 1991. (p 91)

[Mercer93] C. Mercer, S. Savage, and H. Tokuda. *Processor Capacity Reserves: An Abstraction for Managing Processor Usage.* In Proceedings of the Fourth Workshop on Workstation Operating Systems (WWOS-IV), October 1993. (To Appear). (p 90)

[Miller90] F. Miller. *Predictive Deadline Multi-Processing.* ACM Operating Systems Review, 24(4):52–63, October 1990. (p 11)

[MIPS91] *MIPS R4000 Processor Introduction.* MIPS Computer Systems, Inc., 950 DeGuigne Drive, Sunnyvale, CA 94088., 1991. Mfg. Part Number M8-00041. (p 50)

[Mok83] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* Ph.D. thesis, MIT, 1983. (p 9)

[Nakamura93] A. Nakamura. *An Investigation of Real-Time Synchronisation.* Ph.D. thesis, University of Cambridge, December 1993. Submitted for the degree of Doctor of Philiosophy. (p 10)

[Nicolaou91] C. Nicolaou. *A Distributed Architecture for Multimedia Communication Systems.* Ph.D. thesis, University of Cambridge, May 1991. (pp 2, 20, 34, 94)

[Pratt92] I. Pratt. *The ATM Camera.* University of Cambridge Computer Laboratory Part II Project, July 1992. (p 55)

[Pratt93] Ian Pratt. *A Brief Description of the ATM Camera Version 2.* Personal Communication, August 1993. (p 55)

[Ritchie74] D. Ritchie and K. Thompson. *The UNIX Time-Sharing System.* Communications of the ACM, 17(7):365–375, July 1974. (p 52)

[Roscoe94] T. Roscoe. *The Structure of a Multi-Service Operating System.* Ph.D. thesis, University of Cambridge, 1994. Dissertation in preparation. (p 52)

[Rozier89] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. *CHORUS Distributed Operating Systems.* Technical Report CS/TR-88-7.8, Chorus Systemes, February 1989. (p 52)

[Sha86] L. Sha, J. Lehoczky, and R. Rajkumar. *Solutions for Some Practical Problems in Prioritized Preemptive Scheduling.* In Real-Time Systems Symposium, pages 181–191. The Computer Society of the IEEE, IEEE Computer Society Press, December 1986. (p 11)

[Sha90] L. Sha, R. Rajkumar, and J. Lehoczky. *Priority Inheritance Protocols: An Approach to Real-Time Synchronisation.* IEEE Transactions on Computers, 39(9):1175–1185, September 1990. (p 10)

[Sreenan92] C. Sreenan. *Synchronisation Services for Digital Continuous Media.* Ph.D. thesis, University of Cambridge, October 1992. (pp 2, 55)

[Stankovic88] J. Stankovic and K. Ramamritham. *Hard Real-Time Systems.* IEEE Computer Society Press, 1988. (pp 3, 30)

[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, 3333 Coyote Hill Road, Palo Alto, California 94304, June 1986. (p 52)

[Tindell93] K. Tindell. *Personal Communication*, August 1993. (pp 12, 62)

[Tokuda92] H. Tokuda, Y. Tobe, S. Chou, and J. Moura. *Continuous Media Communication with Dynamic QOS Control Using ARTS with an FDDI Network.* Computer Communication Review, 22(4):88–98, October 1992. (p 93)

[Wallace91] G. Wallace. *The JPEG Still Picture Compression Standard.* Communications of the ACM, 34(4), April 1991. (p 24)