The Structure of a Multi-Service Operating System

Timothy Roscoe

Queens' College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

April 1995

Summary

Increases in processor speed and network bandwidth have led to workstations being used to process multimedia data in real time. These applications have requirements not met by existing operating systems, primarily in the area of resource control: there is a need to reserve resources, in particular the processor, at a fine granularity. Furthermore, guarantees need to be dynamically renegotiated to allow users to reassign resources when the machine is heavily loaded. There have been few attempts to provide the necessary facilities in traditional operating systems, and the internal structure of such systems makes the implementation of useful resource control difficult.

This dissertation presents a way of structuring an operating system to reduce crosstalk between applications sharing the machine, and enable useful resource guarantees to be made: instead of system services being located in the kernel or server processes, they are placed as much as possible in client protection domains and scheduled as part of the client, with communication between domains only occurring when necessary to enforce protection and concurrency control. This amounts to multiplexing the service at as low a level of abstraction as possible. A mechanism for sharing processor time between resources is also described. The prototype Nemesis operating system is used to demonstrate the ideas in use in a practical system, and to illustrate solutions to several implementation problems that arise.

Firstly, structuring tools in the form of typed interfaces within a single address space are used to reduce the complexity of the system from the programmer's viewpoint and enable rich sharing of text and data between applications.

Secondly, a scheduler is presented which delivers useful Quality of Service guarantees to applications in a highly efficient manner. Integrated with the scheduler is an inter-domain communication system which has minimal impact on resource guarantees, and a method of decoupling hardware interrupts from the execution of device drivers.

Finally, a framework for high-level inter-domain and inter-machine communication is described, which goes beyond object-based RPC systems to permit both Quality of Service negotiation when a communication binding is established, and services to be implemented straddling protection domain boundaries as well as locally and in remote processes.

To my family.

All of it.

Preface

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

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

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

Acknowledgements

I would like to thank my supervisor Ian Leslie for his help and support with the right advice at the right moments, and to the members of the Systems Research Group past and present for many helpful discussions, in particular Paul Barham, Simon Crosby, David Evers, Robin Fairbairns, Andrew Herbert, Eoin Hyden, Mark Hayter, Derek McAuley and Ian Pratt. Martyn Johnson also deserves a mention for system management above and beyond the call of duty.

For reading and commenting on drafts of this dissertation, I am indebted to Paul Barham, David Evers, Robin Fairbairns, Eoin Hyden, Ian Leslie, Bryan Lyles, Simon Moore and Cormac Sreenan. I am particularly grateful to Robin Fairbairns for his help and encyclopaedic knowledge of LaTeX.

I would also like to thank the staff and interns of the Computer Science Laboratory at Xerox PARC for the tremendously enjoyable and rewarding time I spend there in the summer of 1993. In particular Lyn Carter, Cynthia DuVal, Lorna Fear, Berry Kercheval, Bryan Lyles and Dan Swinehart deserve particular thanks.

This work was supported by a grant from the EPSRC (formerly SERC), and a CASE studentship from Olivetti Research Ltd. I am grateful to Andy Hopper for his support and the opportunity to work for three months at ORL. Thanks are also due to Frazer Bennett, Damian Gilmurray, Andy Harter and Cosmos Nicolaou.

Finally, special thanks are due to Ian Jones and Jinny Crum-Jones for their friendship and accomodation, and to Elaine Vautier for all her love, understanding and support over the last months.

Contents

List of Figures

List of Tables

Glossary of Terms

The list below is a brief glossary of terms used in this dissertation. Most are specific to the Nemesis operating system, though some general terms have been included for clarity.

activation The upcall to the entry point of a *domain* as a result of the kernel scheduler selecting the domain to execute.

ADT Abstract Data Type. A collection of operations, each with a name and a signature defining the number and types of its arguments.

application domain A *domain* whose purpose is to execute an application program.

binding An association of a name with some value; in *IDC*, the local data structures for invoking an operation on an *interface*.

class A set of *objects* sharing the same implementation. An object is an instance of exactly one class.

closure The concrete realisation of an *interface*. A record containing two pointers, one to an operation table and the other to a state record.

concrete type A data type whose structure is explicit.

constructor An operation on an interface which causes the creation of an *object*, and returns the *interface*s exported by the object.

context A collection of *bindings* of names to values.

context slot A data structure used to hold processor execution state within a domain.

domain The entity which is *activated* by the kernel scheduler. Domains can be thought of as analogous to UNIX processes. A domain has an associated *sdom* and *protection domain*.

event channel An inter-domain connection established by the system Binder between two *event end-points*. Event channels are described fully in section 4.6. They are implemented by the kernel.

event end-point A data structure within a domain representing one end of an *event channel*.

event count A synchronisation primitive, often used with a *sequencer*. Event counts and sequencers are the basic *intra-domain* synchronisation mechanism in Nemesis: they are implemented by the user-level thread scheduler. See [Reed79] for a general description.

execution context The processor state corresponding to an activity or *thread*.

IDC Inter-Domain Communication.

interface The point at which a service is offered; a collection of operations on exactly one *object*. An instance of an *interface type*.

interface reference An entity containing the engineering information necessary to establish a *binding* to an interface. In the local case, this is a pointer.

interface specification A definition of the abstract type of an interface, which (in MIDDL) can also include definitions of *concrete types* and exceptions.

interface type The abstract type of one or more interfaces. An interface type is defined by an *interface specification*.

invocation reference A name which can be used to invoke operations on an *interface*. In the local case, this is the same as the *interface reference* and is a pointer to the interface *closure*. In the remote case, it is a pointer to a surrogate closure created from the interface reference by establishing a *binding* to the interface.

latency hint A parameter used by the scheduling algorithm when unblocking an *sdom* to determine when the sdom's new *period* will end.

MIDDL Mothy's Interface Definition and Description Language. A language for writing *interface specifications*.

module A unit of loadable code. A module contains no unresolved symbols and includes one or more interface *closures* whose state is constant.

object A computational entity consisting of some state which is manipulated solely via the *interfaces* exported by the object. An object may export several interfaces.

period The real time interval over which an *sdom* is allocated CPU time.

pervasive interfaces A set of useful *interfaces*, references to which are considered part of the execution context of a *thread*.

protection domain A function from virtual addresses to access rights.

sdom The entity to which CPU time is allocated. Otherwise known as a *scheduling domain*.

sequencer A synchronisation primitive, often used with an *event count*.

slice The quantity of CPU time allocated to a *sdom* within its *period*.

thread A path of execution within a single *domain*; a unit of potential concurrency within a domain.

Chapter 1

Introduction

This dissertation is concerned with techniques for building operating systems to support a wide range of concurrent activities, in particular time-sensitive tasks processing multimedia data.

1.1 Motivation

General-purpose workstations and operating systems are increasingly being called on to process continuous media such as video and audio streams in real time. At least two properties of continuous media set them apart from the kind of data traditionally handled by general-purpose computers.

- The validity of a computation is dependent on the timeliness with which it is performed. This problem is exacerbated by these time constraints typically being quite strict, and the volume of data to be processed imposing a high load on the system.
- To some extent the loss of information in a continuous media stream can be tolerated. This does not hold in all cases, but can still be usefully exploited by an application required to handle such media.

Whilst specialised peripheral devices have been developed to capture, encode, decode and present multimedia data in standard formats, general purpose processing of such data within a traditional workstation environment is still very difficult. The lack of resource control within conventional interactive operating systems results in behaviour under load which is often unacceptable for time-sensitive applications. This is particularly the case when the machine is being used to perform traditional, computationally intensive jobs as well as process multimedia data at the same time.

This dissertation describes operating system technology and principles which address a specific problem. This problem can be characterised as follows:

- Executing a number of time-sensitive tasks concurrently with the usual selection of interactive and batch processes on a typical workstation.
- Running the machine with a set of processes which can utilise more than the available processor cycles.
- Allowing resources to be dynamically reallocated within the system.
- Preventing *crosstalk* between applications so that one task cannot hog resources or violate the timing requirements of another.
- Ensuring that when the resources allocated to time-sensitive applications are reduced, application performance can degrade in a graceful, application-specific manner.

The scenario is that of a desktop workstation, with a high-speed network interface and associated peripherals such as video and audio input devices, which is being used to process multimedia data while at the same time executing a mix of more traditional interactive applications such as text editors and browsers, and batch jobs such as program compilation and numerical analysis. The emphasis is also on processing continuous media rather than merely present it: applications such as real time video indexing, speaker tracking, voice and gesture input, and face recognition are envisaged.

1.2 Background

Several extensions have been made to existing workstation operating systems to assist the execution of multimedia applications. These usually take the form of a "real-time" scheduling class with a higher static priority than other tasks in the system. This solution is inadequate: in practice, several tasks running at such a priority still interfere in an unpredictable manner [Nieh93]. Furthermore, lower priority tasks only run when the “real-time” tasks have no work to do, and the nature of continuous media applications means that this is infrequent. Thus batch and interactive jobs in the system (even system daemons) are starved of CPU time.

One of the fundamental problems with conventional operating systems is that decisions as to which task should receive a given resource are based on a measure (e.g. priority) which does not permit control over the actual quantity of a resource to be allocated over a given time period, particularly when this time period is small.

Systems which have been developed to specify resource requirements more accurately have typically borrowed techniques from the field of real-time systems, such as deadline-based scheduling [Oikawa93, Coulson93]. Such systems can deliver resource guarantees to kernel threads or processes, but do not address the problems caused by the interaction of kernel threads with one another, nor do they allow applications to internally redistribute resources among their own activities. This issue is addressed more fully in chapter 2.

1.3 Quality of Service

In the last ten years, similar problems have appeared in communication networks which carry a mix of different traffic types. The term Quality of Service or QoS has been used to denote the generalised idea of explicit, quantitative allocation of network resources, principally with regard to bandwidth, end-to-end delay, and delay jitter. QoS parameters are mostly independent of the mechanisms and algorithms employed by the resource provider, and oriented more towards application requirements. The concept of QoS is very generalised, and guarantees can take many forms, including probabilistic notions of resource availability.

Typically, a client of the network negotiates with the network for resources, and the client and network agree on a particular QoS. This is a measure of the allocation the client can expect to receive and may be much more than a simple lower bound—see, for example, [Clark92]. Within the network, an *admission control* procedure ensures that the network does not over-commit its resources, and the process of *policing* prevents clients from unfairly over-using resources. If the needs of a client change, or network conditions alter the QoS which the network can deliver to a client, *renegotiation* may take place, and a new QoS agreed.

The distributed nature of many multimedia applications has resulted in the need for a way to specify and support end-to-end QoS from application to application [Nicolaou90, Campbell93]. This in turn has led to investigation of suitable interfaces between clients and the operating system to provide flexible resource allocation in the end system. In this context, the resource provider is the operating system and the clients are application tasks. If the resource is CPU time, the provider is the kernel scheduler.

1.4 Contribution

This author has built on the work in [Hyden94], investigating the wider issue of how to structure a general-purpose QoS-based operating system and associated applications.

The thesis of this work is that a general-purpose operating system which:

- allocates resources (and CPU time in particular) using a QoS paradigm,
- performs in a predictable and stable manner under heavy load,
- delivers useful resource guarantees to applications,
- allows them to utilise their resources efficiently, and
- ensures that resources can be redistributed dynamically and that applications can seamlessly adapt to the new allocation,

—can be achieved by an architecture which places much operating system functionality into application processes themselves without impinging upon protection mechanisms, and which multiplexes system resources at as low a level of abstraction as possible.

As well as outlining the architecture, this dissertation contributes to the design of such a system by presenting:

- structuring tools to aid the construction of the operating system and to provide rich sharing of data and code,

- an efficient scheduling algorithm for allocating processor time to applications in accordance with QoS specifications, even in the presence of high interrupt rates from devices, and
- a concrete model of local client-server binding that allows QoS negotiation and permits services to be migrated across protection domain boundaries.

The issues in constructing such a system are illustrated by describing the Alpha/AXP prototype of Nemesis, a multi-service operating system developed by the author in the course of his work. This system was written on the DECchip EB64 board [DEC93] over PALcode written by Robin Fairbairns, and later ported to the DEC3000/400 Sandpiper workstation [DEC94] by the author and Paul Barham.

1.5 Overview

The overall structure of Nemesis is described in chapter 2.

An issue raised by chapter 2 is the complexity and code size resulting from the architecture. Chapter 3 discusses the use of typed interfaces, closures and a per-machine, single virtual address space to provide modularity and rich sharing of data and code.

Chapter 4 describes the scheduling algorithm for allocating CPU time to applications in accordance with QoS specifications existing between application domains and the kernel scheduler. The interface presented to applications is described. This enables the efficient multiplexing of the CPU required within an application to support the architecture. The performance of the scheduler is evaluated.

Communication between applications within Nemesis is described in chapter 5. In particular, a framework for establishing *bindings* between clients and servers is discussed. This framework allows precise control over the duration and characteristics of bindings. Furthermore, it transparently integrates the communication optimisations necessary to migrate code from servers into clients. Using this binding model, services can be implemented partly in the client and partly in the server protection domains, and QoS negotiation with a server can occur.

Chapter 6 summarises the research and discusses some areas for future work.

Chapter 2

Architecture

This chapter describes the architectural principles governing the design of Nemesis: where services are located in the system. The aim is to minimise the impact of two related problems with current operating systems: the lack of fine-grained *resource control* and the presence of *application Quality-of-Service crosstalk*. Consideration of these issues leads to a novel way of structuring the system by vertically integrating functions into application programs.

2.1 Introduction

This dissertation uses the term *operating system architecture* to describe how services are organised within the operating system in relation to memory protection domains, schedulable entities and processor states. This chapter deals with a high-level view of system structure, whilst later chapters describe the low-level details of protection and scheduling in Nemesis.

The major functions of an operating system are to provide:

- *sharing* of hardware resources among applications,
- *services* for applications to use, and
- *protection* between applications.

An operating system architecture is to a large extent determined by how these functions are implemented within the system.

Most operating systems running on interactive workstations fall into one of three categories architecturally: monolithic, kernel-based, and microkernel-based.

2.1.1 Monolithic Systems

Cedar [Swinehart86] and the Macintosh [Apple85] are examples of monolithic operating systems. Typically all code in the system executes with the same access rights on memory and in the same processor mode (with the exception of interrupt masks). System services (memory management, communication, filing systems, etc.) are provided via procedure calls or a vector table accessed via a processor trap. A single, central policy is used to allocate and release resources.

Monolithic systems thus provide simplicity and high performance, while offering little in the way of protection between different activities on the same machine. The motivation for this tradeoff is twofold. Firstly, the machine is considered to be a single protection domain under the control of one human user. The danger of intrusion by other users is avoided, although the risk due to malicious programs initiated innocently by the user still exists. Secondly, programming language features (sophisticated type systems, language-level concurrency primitives and strong modularity) can reduce the chance of bugs in one application corrupting other applications on the same machine, or bringing the whole system down.

Such features are hard to justify in a general-purpose system designed to be programmed in a variety of different languages, and potentially used by several users at once. Generally speaking, hardware features must be used to provide protection between applications and operating system components. However, it is important to realise that protection from programming bugs is an insufficient reason on its own for protecting system services with hardware.

2.1.2 Kernel-based Systems

Many current workstation operating systems are descended from central multi-user timesharing systems: examples include varieties of UNIX [Bach86, Leffler89], VMS [Goldenberg92] and Microsoft Windows NT [Custer93]. Systems like these provide each application with a different address space and memory protection domain. In addition, there is a single large *kernel*, which runs in a privileged processor mode and is entered from user programs by means of a trap instruction (figure 2.1).

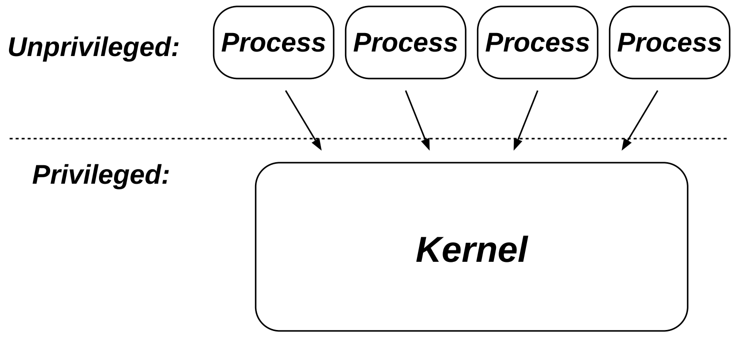


Figure 2.1: Kernel-based operating system architecture

The kernel provides most system services and presents *virtual* resources (time, memory, etc) to applications; thus each application is given the illusion of having the entire machine to itself. Protection is achieved by limiting the physical address space accessible to each application.

This virtual machine model provides robustness, provided the kernel is reliable and can be trusted. Unfortunately, as the systems have evolved kernels have become large and unwieldy, a situation not improved by their implementation in a language with few ways of enforcing modularity, such as C or C++.

Furthermore, reconfiguring the operating system involves modifying the kernel, which often entails a system restart. Despite features such as loadable device drivers, changing the configuration of a kernel-based system is a difficult business and it is still normal for a buggy piece of loadable operating system code to bring down the whole system.

2.1.3 Microkernel-based Systems

The desire for extensibility and modularity led to the development of *microkernels*, for example Mach [Accetta86] and Chorus [Rozier90]. Such systems move functionality into separate protection domains and processes, which communicate with each other and application processes via a small kernel, often using message passing (figure 2.2).

The protection boundaries between the various components can make the operating system as a whole more robust. More importantly, it is much easier to dynamically extend and reconfigure the system. This comes at some cost in performance since invoking a service now requires communication between two

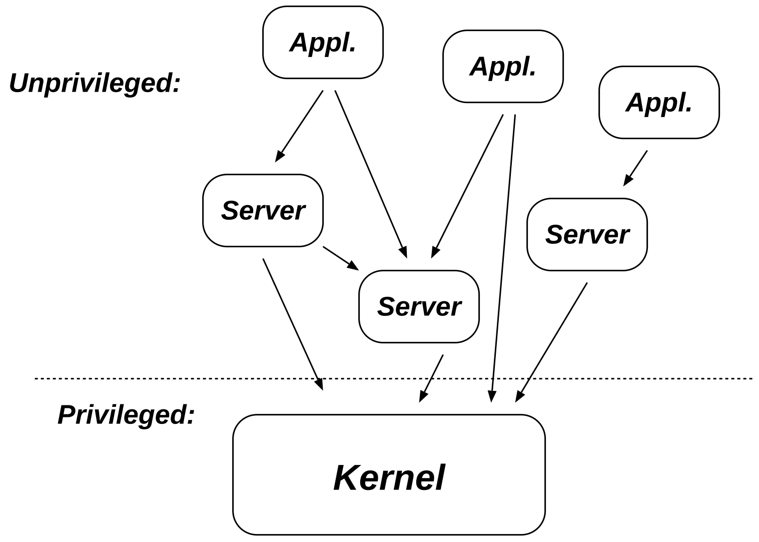


Figure 2.2: Microkernel-based operating system architecture

processes. In a microkernel system this overhead includes two context switches as opposed to a simple processor trap in kernel-based system. Much work has gone into reducing the cost of this communication, for example [Hamilton93a, Bershad90, Bershad91, Hildebrand92]. Some systems have even migrated services back into the kernel for performance reasons [Bricker91].

2.1.4 Other systems

The incomplete taxonomy above classifies systems into those with zero, 1 or many entities providing operating system services. Naturally, the boundaries between classes are blurred (for example, UNIX uses server daemons) and there have been operating systems (for example, the CAP [Wilkes79]) which do not fit in the model. However, the classification covers most workstation operating systems in use today, at least in so far as the architecture impinges on the issue of resource control.

2.2 Resource Control Issues

Resource control is concerned with limiting the consumption of resources by applications in a system so that they can all make some satisfactory progress. In this sense it is needed to guarantee the *liveness* of a system: excessive consumption by one party should not prevent the progress of others. The exercise of control can be viewed as a form of *contract* between the resource provider and user: the user 'pays' in some way for a resource, and the provider guarantees to provide the resource.

Resource control in workstation operating systems has traditionally been quite primitive. Quite simple scheduling policies can guarantee that each process receives the processor eventually, and rarely are per-process quotas enforced on resources such as physical page frames or disk file space. Indeed, in this respect traditional mainframe timesharing systems offer rather more in the way of resource control than workstations. The prime motivation in the mainframe case is the need to charge clients money for use of system resources. However, even in these systems the contracts employed do not apply over short time scales on the order of milliseconds, since the applications supported by these do not require such guarantees at this level.

It is now widely accepted that processing of continuous media in real time does require this kind of resource control. An important feature of the Nemesis operating system is that it provides fine-grained resource control. Furthermore, it supports the provision of a service to change contracts dynamically, and permits applications to adapt when this happens.

Resource control of this nature has been discussed in the field of high speed networks, particularly those designed to carry a mix of different traffic types, for some time. Such ideas have been referred to as Quality of Service or QoS, and this dissertation borrows much terminology from this field.

2.2.1 Requirements of a Resource Control System

A resource control component for an operating system must fulfill at least five functions: Allocation, Admission Control, Policing, Notification, and Internal Multiplexing support.

Allocation

An operating system is a multiplexor of resources such as processor time, physical memory, network interface bandwidth, etc. The system should try to share out quantities of bulk resource to clients in accordance with their respective contracts.

A further requirement is to control resource allocation dynamically. In a workstation where resources are limited, users may wish to redistribute resources to increase the level of service provided to certain applications at the expense of others, for example in response to an incoming video phone call.

Admission Control

Since the system aims to satisfy all its contracts with clients, it should not negotiate contracts which it will not be able to honour. Admission Control is the term used in networks for the process by which the system decides whether it will provide a requested level of service.

Systems with a very large number of clients (such as wide-area networks), can employ *statistical multiplexing* to reserve more resources in total than the system can supply instantaneously, relying on the fact that the probability that all clients will simultaneously require their entire guaranteed resource share is small.

In an operating system environment, this technique cannot be employed. Firstly, there are too few clients to permit valid statistical guarantees, and their loads are often highly correlated. Secondly, experience with early systems which attempt to integrate video and audio with the workstation environment (for example Pandora [Hopper90]) shows that in practice the system is under high load for long periods: software systems tend to use all the CPU time allocated to them.

This situation is likely to continue despite the increasing power of workstations, as software becomes more complex. However, this lack of predictability is offset by the ability to use a central resource allocation mechanism, which has complete knowledge of the current system utilisation. Ultimately, a human user can dictate large-scale resource allocation at run time.

Policing

Policing is the process of ensuring that clients of the operating system do not use resources unfairly at the expense of other applications. Policing requires an accounting mechanism to monitor usage of a resource by each application.

A well-designed resource allocation mechanism should provide a policing function. However, in the case of CPU time there are several operating systems whose applications are expected to cooperate and periodically yield the processor. Examples include the Macintosh Operating System and applications running under Microsoft Windows NT with the `/REALTIME`[1] switch.

More importantly, effective policing of CPU time can only be carried out by the system if the application (the complete entity requiring a given QoS) is the same unit as that dealt with by the scheduler, rather than individual threads.

Notification

In a system where applications are allocated resources quantitatively, the 'virtual machine' model of kernel-based systems is inappropriate. Instead of the illusion of an almost limitless *virtual* resource, clients of the operating system must deal in *real* resources.

For example, UNIX applications are given huge amounts of address space (in the form of virtual memory), and a virtual processor which they never lose and is rarely interrupted (by the signal mechanism). In reality, the system is constantly interrupting and even paging the process. The passage of real time bears little resemblance to the virtual time experienced by the application, particularly millisecond granularity. While this hiding of real time is highly convenient for traditional applications, this is precisely the information required by programs processing time-sensitive data such as video. Similar arguments apply to physical memory (when paging), network interface bandwidth, etc.

A key motivation for providing information to applications about resource availability is that the policies applied by an application both for degradation when starved of resources, and for use of extra resources if they become available, are highly specific to the application. It is often best to let the application decide what to do.

The approach requires a mapping from the application's performance metric (for example, number of video frames rendered per second) to the resources allocated by the system (CPU time). This is extremely difficult analytically, except in specialised cases such as hard real-time systems, for example [Veríssimo93, Chapman94]. However, if applications are presented with a regular opportunity

[1]Such processes execute at a priority higher than any operating system tasks.

to gauge their progress in real time, they can use feedback mechanisms to rapidly adapt their behaviour to optimise their results, provided that conditions change relatively slowly or infrequently over time.

Thus applications require timely knowledge both of their own resource allocation and of their progress relative to the passage of real time.

Internal Multiplexing Mechanisms

Simple delivery and notification of a bulk resource to an application are not in general sufficient: a program must be able to make effective use of the resource. This amounts to multiplexing the resource internally, and in such a way that when the total allocation changes the application can change its internal resource tradeoffs to achieve the best results. An operating system should provide the means for applications to do this efficiently.

The processor is, again, a good concrete example. Threads provide a convenient model for dividing the CPU time allocated to a program among its internal tasks. The UNIX operating system provides no explicit support for user-level threads, with the consequence that thread-switching in user-space takes place with no knowledge of kernel events. Furthermore, most user-level threads packages use a periodic signal to reenter the thread scheduler. This incurs a high overhead and limits the granularity of scheduling possible. At the other end of the scale, operating systems which provide kernel threads take the thread-scheduling policy away from the application, and so are incompatible with the QoS model.

Recently, systems have appeared which provide much greater support for user-level threads systems over kernel threads. The motivation for this approach in Scheduler Activations [Anderson92] was the performance gain due to reduced context switch time; in Psyche [Scott90] it was the desire to support different thread scheduling and synchronisation policies. These techniques have been adopted, in modified form, in Nemesis.

2.3 Crosstalk

A scheduler which provides the facilities discussed above can be built: chapter 4 describes the one used in Nemesis. However, scheduling *processes* in this way is not in itself sufficient to provide useful resource control for *applications*.

In a conventional operating system, an application spans several processes. Assigning QoS parameters to each process so that the application as a whole runs efficiently can be very difficult, particularly if the resource allocation in the system is dynamically changing. In effect, the application has lost some control over its internal resource tradeoffs unless it can rapidly and efficiently transfer resources from one process to another.

Furthermore, a process may be shared between several applications. This introduces the problem of *crosstalk*.

2.3.1 Protocol QoS Crosstalk

When dealing with time-related data streams in network protocol stacks, the problem of *Quality of Service crosstalk* between streams has been identified [McAuley89, Tennenhouse89]. QoS crosstalk occurs because of contention for resources between different streams multiplexed onto a single lower-level channel. If the thread processing the channel has no notion of the component streams, it cannot apply resource guarantees to them and statistical delays are introduced into the packets of each stream. To preserve the QoS allocated to a stream, scheduling decisions must be made at each multiplexing point.

When QoS crosstalk occurs the performance of a given network association at the application level is unduly affected by the traffic pattern of other associations with which it is multiplexed. The solution advocated in [McAuley89, Tennenhouse89] is to multiplex network associations at a single layer in the protocol stack immediately adjacent to the network point of attachment. This allows scheduling decisions to apply to single associations rather than to multiplexed aggregates. This idea grew out of the use of virtual circuits in ATM networks, but can also be employed in IP networks by the use of packet filters [Mogul87, McCanne93].

It is widely accepted that to be useful, QoS guarantees need to be extended up to the application so as to be truly end-to-end. By extension, we can identify *application QoS crosstalk* as a general problem which arises from the architecture of modern operating systems.

2.3.2 Application QoS Crosstalk

Application QoS Crosstalk occurs because operating system services as well as physical resources are multiplexed among client applications. This multiplexing is performed at a high level by the use of server processes (including the kernel itself).

In addition to network protocol processing, components such as device I/O, filing systems and directory services, memory management, link-loaders, and window systems are accessed via a set of high-level interfaces to client applications. These services must provide concurrency and access control to manage system state, and so are generally implemented in server processes or within the kernel.

This means that the performance of a client is dependent not only on how it is scheduled, but also on the performance of any servers it requires, including the kernel. The performance of these servers is in turn dependent on the demand for their services by other clients. Thus one client's activity can delay invocations of a service by another. This is at odds with the scheduling policy, which should be attempting to allocate time among applications rather than servers.

A particularly impressive example of this in practice is described in [Pratt94]. A Sun SparcStation 10 running SunOS received video over an ATM network and displayed it on the screen via the X server. In this case the available CPU time in the system was divided roughly equally between the kernel (data copying and protocol processing), the application itself (conversion between image formats) and the X server (copying the image to the frame buffer).

In this case over 60% of the processor time used by an application was not being accounted to it. Other clients were unable to render graphics due to the demand on the X server from the video application. The point here is not the load on the machine, but that contention for a shared service is occurring, and the service is unable to effectively multiplex its processor time among clients.

Some degree of crosstalk is inevitable in an operating system where there are data structures which are shared and to which access must be synchronised. However, identifying the phenomenon of application QoS crosstalk is important because it allows systems to be designed to minimise its impact. To reduce crosstalk, service requests should as far as possible be performed using CPU time accounted to the client and by a thread under control of the client.

2.4 The Architecture of Nemesis

Nemesis is structured so as to fulfill the requirements of a fine-grained resource control mechanism and minimise application QoS crosstalk. To meet these goals it is important to account for as much of the time used by an application as possible, without the application losing control over its resource use.

For security reasons, code to mediate access to shared state must execute in a different protection domain (either the kernel or a server process) from the client. This does not imply that the code must execute in a different logical thread to the client: there are systems which allow threads to undergo protection domain switches, both in specialised hardware architectures [Wilkes79] and conventional workstations [Bershad90]. However, such threads cannot easily be scheduled by their parent application, and must be implemented by a kernel which manages the protection domain boundaries. This kernel must as a consequence, provide synchronisation mechanisms for its threads, and applications can no longer control their own resource tradeoffs by efficiently multiplexing the CPU internally.

The alternative is to implement servers as separate schedulable entities. Some systems allow a client to transfer some of their resources to the server to preserve a given QoS across server calls. The Processor Capacity Reserves mechanism [Mercer94] is the most prominent of these; the kernel implements objects called *reserves* which can be transferred from client threads to servers. This mechanism can be implemented with a reasonable degree of efficiency, but does not fully address the problem:

- The state associated with a reserve must be transferred to a server thread when an IPC call is made. This adds to call overhead, and furthermore suffers from the kernel thread-related problems described above.
- Crosstalk will still occur within servers, and there is no guarantee that a server will deal with clients fairly, or that clients will correctly ‘pay’ for their service.
- It is not clear how nested server calls are handled; in particular, the server may be able to transfer the reserve to an unrelated thread.

Nemesis takes the approach of minimising the use of shared servers so as to reduce the impact of application QoS crosstalk: the minimum necessary functionality for a service is placed in a shared server. Ideally, the server should only perform concurrency control. In addition, to reduce the resource management which must be performed *outside* applications, resources should be allocated as early as possible, in bulk if necessary.

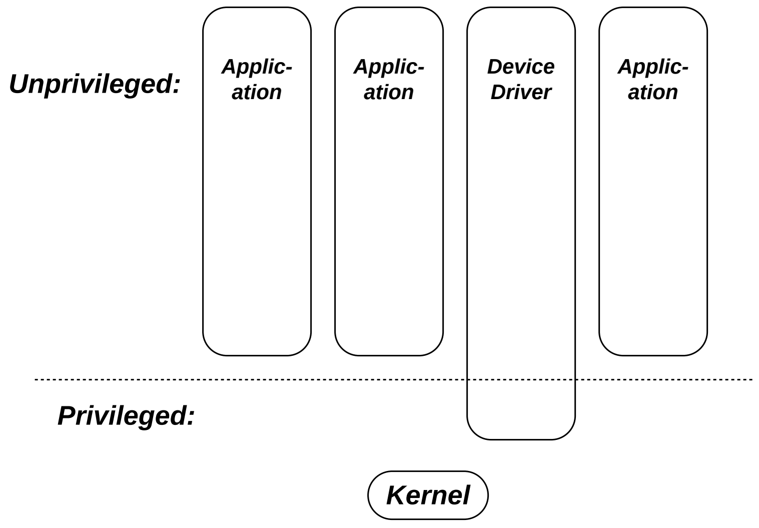


Figure 2.3: Nemesis system architecture

The result is a 'vertically integrated' operating system architecture, illustrated in figure 2.3. The system is organised as a set of *domains*, which are scheduled by a very small kernel. A Nemesis domain is roughly analogous to a process in many operating systems: it is the entity scheduled by the kernel, and usually corresponds to a memory protection domain. However, a given domain performs many more functions than a typical thread: the Nemesis kernel on DEC Alpha/AXP machines consists only of interrupt handlers (including the scheduler) and a small Alpha PALcode image.

The minimum functionality possible is placed in server domains, and as much processing as possible is performed in application domains. This amounts to multiplexing system services at the lowest feasible level of abstraction. This both reduces the number of multiplexing points in the system, and makes it easier for scheduling decisions to be made at these points. Protection within an application domain is performed by language tools.

This stands in contrast to recent trends in operating systems, which have been to move functionality away from client domains (and indeed the kernel) into separate processes.

However, there are a number of examples in recent literature of services being implemented as client libraries instead of within a kernel or server. Efficient user-level threads packages have already been mentioned.

[Thekkath93] discusses implementation of network protocols as client libraries in the interests of ease of prototyping, debugging, maintenance and extensibility, and also to investigate the use of protocols tuned to particular applications. While at Xerox PARC, the author of this dissertation investigated implementation of TCP over ATM networks in a user-space library over SunOS, in order to reduce crosstalk and aid in accounting. In principle packets must be multiplexed securely, but above this in the protocol stack there are no inherent problems in protocol processing as part of the application.

A recent version of the $8\frac{1}{2}$ window system [Pike94] renders graphics almost entirely within the client. The client then sends bitmap tiles to the window manager, which is optimised for clipping these tiles and copying them into the frame store. The frame store device for the Desk Area Network [Barham95a] provides these low level window manager primitives in hardware.

Finally, a good indicator that most of the functions of the UNIX kernel can be performed in the application is given by the Spring SunOS emulator [Khalidi92], which is almost entirely implemented as a client library.

Nemesis is designed to make use of these techniques. In addition, most of the engineering for creating and linking new domains, and setting up inter-domain communication, is performed in the application.

2.5 Summary

Resource control in operating systems has traditionally been provided over medium to long time scales. Continuous media processing requires resources to be allocated with much finer granularity over time, the failure of the system to exercise control over the resource usage of other tasks seriously impacts such applications. The related problem of application QoS crosstalk has been identified as a problem inherent in operating systems but greatly exacerbated by the high level functionality currently implemented in servers.

Nemesis explores the alternative approach of implementing the barest minimum of functionality in servers, and executing as much code as possible in the application itself. This has the dual aim of enabling more accurate accounting of resource usage while allowing programs to manage their own resources efficiently. Examples from the existing literature illustrate how many operating systems functions can be implemented in this way.

The architecture raises a number of issues. Programmers should neither have to cope with writing almost a complete operating system in every application, nor contend with the minimum level interfaces to shared servers. Binaries should not become huge as a result of the extra functionality they support, and resources must be allocated in such a way that applications can manage them effectively. The next three chapters present solutions to these problems.

Chapter 3

Interfaces and Linkage

This chapter deals with linking program components within a single domain. It presents solutions to two potential problems caused by the architecture introduced in chapter 2. The first is the software engineering problem of constructing applications which execute most of the operating system code themselves. This is addressed by the typing, transparency and modularity properties of Nemesis interfaces. The second problem is the need for safe and extensive sharing of data and code. The use of *closures* within a single address space together with multiple protection domains provides great flexibility in sharing arbitrary areas of memory.

The programming model used when writing Nemesis modules is presented, followed by the linkage model employed to represent objects in memory. In addition, auxiliary functions performed by the name service and runtime type system are described, together with the process by which a domain is initialised.

3.1 Background

The linkage mechanism in Nemesis encompasses a broad range of concepts, from object naming and type systems to address space organisation. Below is a survey of selected operating systems work which has relevance to linkage in Nemesis in one area or another.

3.1.1 Multics

In Multics [Organick72], virtual memory was not organised as a flat address space but as a set of segments, integrated with the filing system. Any process could attach a segment and access it via a process-specific identifier (in fact, an entry in the process segment table). Thus, a great deal of code and data could be shared between processes.

Related procedures were organised into segments. *Linkage segments* were used to solve the problem of allowing a procedure shared between processes to make process-specific references to data and procedures in other segments. There was one process-specific linkage segment for each text segment in a process, which mapped unresolved references in the segment to pairs of (segment identifier, offset) values. This segment was used by the GE645 indirection hardware and was filled in on demand via a faulting mechanism.

The idea worked well (albeit slowly) as a means of linking conventional procedural programs. However, the scheme does not sit happily with an object-based programming paradigm where members of a class are instantiated dynamically: Linkage segments are inherently per-process, and work best in situations where there is a static number of inter-segment references during the lifetime of a process.

3.1.2 Hemlock

The state of the art in UNIX-based linking is probably Hemlock [Garrett93]. Hemlock reserves a 1GB section of each 32-bit address space in the machine for a region to hold shared *modules* of code, with the obvious extension to 64-bit address spaces. A great deal of flexibility is offered: modules can be *public* (shared between all processes), or *private* (instantiated per-process). They can also be *static* (linked at compile time) or *dynamic* (linked when the process starts up, or later when a segment fault occurs).

Hemlock is geared towards a UNIX-oriented programming style, thus modules are principally units of code. The definition of interfaces between modules is left to programming conventions, and the data segments of private modules are instantiated on a one-per-process basis.

3.1.3 Spring

Along with Nemesis, Spring [Hamilton93a, Hamilton93b] is one of the few operating systems to use an interface definition language for all system interfaces. Spring is implemented in C++ and employs the UNIX model of one address space per process. Shared libraries are used extensively, but the need to link one library against another has led to copy-on-write sharing of text segments between address spaces. With the C++ programming language, the number of relocations is quite large, even with position-independent code. This reduces the benefits of sharing and results in increased time to link an image [Nelson93]. The solution adopted has been to cache partially linked images—combinations of libraries linked against one another—on stable storage.

Linkage is carried out mainly by the parent domain through memory mapping, at a different address from that at which the code must execute. However, the model is essentially the same as UNIX, with a small number of memory areas holding the state for all objects in the address space.

Spring is also unusual in providing a name service that is uniform and capable of naming any object. Naming contexts are first-class objects and can be instantiated at will. The scope of the naming service is broad, encompassing access control and support for persistent objects. This requires that all objects must either provide an interface for requesting that they become persistent, or a way of obtaining such an interface.

3.1.4 Plan 9 from Bell Labs

Plan 9 [Pike92] is a UNIX-like operating system but with a novel approach to naming. Plan 9 has several different kinds of name space, but the main one is concerned with naming *filing systems*, which are the way many system interfaces present themselves.

This approach has a number of problems:

- Instead of the typed interfaces of systems such as Spring, Plan 9 constrains everything to look like a file. In most cases the real interface type comprises the protocol of messages that must be written to, and read from, a file descriptor. This is difficult to specify and document, and prohibits any automatic type checking at all, except for file errors at run time.

- Filing systems are heavyweight: access to them must be through the kernel. Instantiating interfaces dynamically is impossible.

- There are limits to what can be named. For instance, both the initial I/O channels available to a domain and the space of network addresses comprise name spaces separate from the principal one.

- Instead of providing a service for mapping strings to pointers to interfaces (which are essentially low-level names for the services), in Plan 9 a path name relative to a process' implicit root context is the *only* way to name a service. Binding a name to an object can only be done by giving an existing name for the object, in the same context as the new name. As such, interface references simply *cannot* be passed between processes, much less across networks. Instead, communication has to rely on conventions, which are prone to error and do not scale.

3.1.5 Microsoft OLE 2

OLE 2 [Brockschmidt94] is a system built on top of the Windows 3.1 environment to provide object-based facilities. Objects export one or more interfaces, which appear as C++ objects with virtual member functions. Objects are shared between applications in the operating system by the use of stubs and the dynamic link libraries provided by Windows, though shared memory can be used as a transport mechanism. Above the basic OLE 2 infrastructure are built several complex subsystems to provide object naming and persistent object storage.

Interface types in OLE 2 are never explicitly defined. Instead, the runtime system deals only in globally unique type identifiers allocated centrally by Microsoft, which by programmer convention refer to particular revisions of C or C++ function definitions. As a consequence, type conformance relations are not supported. Also, runtime information about the structure of types is not available.

OLE 2 is a large and complex body of software. Much of the complexity of OLE 2 arises from the need to expose the underlying Windows operating system, which has no specified interfaces or idea of modularity. A further problem is that while OLE 2 provides object services to application writers, these services themselves are not provided through objects but, like Windows, employ a flat C programming interface. An operating system designed using a consistent object model from the ground up, such as Spring, can be much simpler and more cohesive while offering superior functionality.

3.1.6 Single Address Space Operating Systems

Recently, there have been a number of research projects to build single address space operating systems. These projects have generally aimed at providing an environment with rich sharing of data and text. They try to achieve this by giving each process a different memory protection domain within a single, system-wide address space. Two representative systems are discussed here.

Opal

Opal [Chase93] is an experimental, single address space system implemented as a Mach server process. Linkage in Opal is based around modules similar to those in Hemlock. Domain-specific state for a module is stored at an offset from the Alpha global pointer (GP), a general-purpose register reserved in OSF/1 for accessing data segment values. Modules can contain per-domain, initialised, mutable state: when the module is attached this state is copied into a per-domain data segment.

However, this means that modules may only be instantiated once per domain, and the domain may have only one private data segment. Also, the GP register must be determined on a per-domain basis on every cross-module procedure call; at present it is fixed for the lifetime of a domain. The format of the data segment is constrained to be the same for all domains.

Angel

The Angel microkernel [Wilkinson93] aims to provide a single distributed address space spanning many processors. The designers wished to reduce the cost of context switching with virtual processor caches, and to unify all storage as part of the address space. While UNIX-like text segments are shared, there is no sharing at a finer granularity. The traditional `data` and `bss` segments are still present, and the C compiler is modified to use a reserved processor register to address them.

Like Opal, Angel represents an attempt to use the UNIX notion of a process in a single address space. However, UNIX processes are based on the assumption of a private address space: absolute addresses are used for text and data, and references between object files are resolved statically by the linker. In effect, the environment in which any piece of code executes is the whole address space.

When the address space is shared between processes, this assumption no longer holds. In both Opal and Angel, the process-wide environment using absolute addresses is simply replaced by another using addresses relative to a single pointer. This precludes most of the potential benefits of sharing code and data between protection domains.

3.2 Programming Model

The programming model of Nemesis is a framework for describing how programs are structured; in a sense, it is how a programmer thinks about an application. In particular, it is concerned with how components of a program or subsystem interact with one another.

The goal of the programming model is to reduce complexity for the programmer. This is particularly important in Nemesis where applications tend to be more complex as a consequence of the architecture. The model is independent of programming language or machine representation, though its form has been strongly influenced by the model of linkage to be presented in section 3.3.

In systems, complexity is typically managed by the use of *modularity*: decomposing a complex system into a set of components which interact across well-defined *interfaces*. In software systems, the interfaces are often instances of *abstract data types* (ADTs), consisting of a set of *operations* which manipulate some hidden state. This approach is used in Nemesis.

Although the model is independent of representation, it is often convenient to describe it in terms of the two main languages used in the implementation of Nemesis: the interface definition language MIDDL and a stylised version of the programming language C.

3.2.1 Types and Middl

Nemesis, like Spring, is unusual among operating systems in that all interfaces are strongly typed, and these types are defined in an interface definition language. It is clearly important, therefore, to start with a good type system, and [Evers93] presents a good discussion of the issues of typing in a systems environment. As in many RPC systems, the type system used in Nemesis is a hybrid: it includes notions both of the abstract types of interfaces and of concrete data types. It represents a compromise between the conceptual elegance and software engineering benefits of purely abstract type systems such as that used in Emerald [Raj91], and the requirements of efficiency and inter-operability: the goal is to implement an operating system with few restrictions on programming language.

Concrete types are data types whose structure is explicit. They can be predefined (such as booleans, strings, and integers of various sizes) or constructed (as with records, arrays, etc). The space of concrete types also includes typed references to interfaces[1].

Interfaces are instances of ADTs. Interfaces are rarely static: they can be dynamically created and references to them passed around freely. The type system includes a simple concept of *subtyping*. An interface type can be a subtype of another ADT, in which case it supports all the operations of the supertype, and an instance of the subtype can be used where an instance of the supertype is required.

The operations supported by interfaces are like procedure calls: they take a number of *arguments* and normally return a number of *results*. They can also raise *exceptions*, which themselves can take arguments. Exceptions in Nemesis behave in a similar way to those in Modula-3 [Nelson91].

Interface types are defined in an interface definition language (IDL) called Middl [Roscoe94b]. Middl is similar in functionality to the IDLs used in object-based RPC systems, with some additional constructs to handle local and low-level operating system interfaces. A Middl specification defines a single ADT by declaring its supertype, if any, and giving the *signatures* of all the operations it

[1]The term *interface reference* is sometimes used to denote a pointer to an interface. Unfortunately, this can lead to confusion when the reference and the interface are in different domains or address spaces. Chapter 5 gives a better definition of an interface reference. In the local case described in this chapter, interfaces references can be thought of as pointers to interfaces.

supports. A specification can also include declarations of exceptions, and concrete types. Figure 3.1 shows a typical interface specification, the (slightly simplified) definition of the `Context` interface type.

3.2.2 Objects and Constructors

The word *object* in Nemesis denotes what lies behind an interface: an object consists of state and code to implement the operations of the one or more interfaces it provides. A *class* is a set of objects which share the same underlying implementation, and the idea of object class is distinct from that of type, which is a property of interfaces rather than objects.

This definition of an object as hidden state and typed interfaces may be contrasted with the use of the term in some object-oriented programming languages like C++ [Stroustrup91]. In C++ there is no distinction between class and type, and hence no clear notion of an interface[2]. The type of an interface is always purely abstract: it says nothing about the implementation of any object which exports it. It is normal to have a number of different implementations of the same type.

When an operation is invoked upon an object across one of its interfaces, the environment in which the operation is performed depends only on the internal state of the object and the arguments of the invocation. There are no global symbols in the programming model. Apart from the benefits of encapsulation this provides, it facilitates the sharing of code described in section 3.3.

An object is created by an invocation on an interface, which returns a set of references to the interfaces exported by the new object. As in Emerald, constructors are the basic instantiation mechanism rather than classes. By removing the artificial distinction between objects and the means used to create them, creation of interfaces in the operating system can be more flexible than the 'institutionalised' mechanisms of language runtime systems. This is particularly important in the lower levels of an operating system, where a language runtime is not available.

[2]C++ abstract classes often contain implementation details, and were added as an afterthought [Stroustrup94, p. 277].

```
Context : LOCAL INTERFACE =
  NEEDS Heap;
  NEEDS Type;
BEGIN

  --
  -- Interface to a naming context.
  --

  Exists     : EXCEPTION [];
    -- Name is already bound.

  -- Type used for listing names in a context:
  Names : TYPE = SEQUENCE OF STRING;

  -- "List" returns all the names bound in the context.
  List : PROC    []
         RETURNS [ nl : Names ]
         RAISES Heap.NoMemory;

  -- "Get" maps pathnames to objects.
  Get : PROC    [ IN  name : STRING,
                  OUT o    : Type.Any ]
        RETURNS [ found : BOOLEAN ];

  -- "Add" binds an object to a pathname.
  Add : PROC    [ name : STRING, obj : Type.Any ]
        RETURNS []
        RAISES Exists;

  -- "Remove" deletes a binding.
  Remove : PROC [ name : STRING ] RETURNS [];

END.
```

Figure 3.1: Middl specification of the `Context` interface type

3.2.3 Pervasives

The programming model described so far enforces strict encapsulation of objects: the environment in which an interface operation executes is determined entirely by the operation arguments and the object state. Unfortunately, there are cases where this is too restrictive from a practical point of view. Certain interfaces provided by the operating and runtime systems are used so pervasively by application code that it is more natural to treat them as part of the thread context than the state of some object. These include:

- Exception handling
- Current thread operations
- Domain control
- Default memory allocation heap

Many systems make these interfaces 'well-known', and hardwired into programs either as part of the programming language or as procedures linked into all images. This approach was rejected in Nemesis: the objects concerned have domain-specific state which would have to be instantiated at application startup time. This conflicts with the needs of the linkage model (section 3.3), in particular, it severely restricts the degree to which code and data can be shared. Furthermore, the simplicity of the purely object-based approach allows great flexibility, for example in running the same application components simultaneously in very different situations.

However, passing references to all these interfaces as parameters to every operation is ugly and complicates code. The references could be stored as part of the object state, but this still requires that they be passed as arguments to object constructors, and complicates the implementation of objects which would otherwise have no mutable state (and could therefore be shared among domains as is).

Pervasive interfaces are therefore viewed as part of the context of the currently executing thread. As such they are always available, and are carried across an interface when an invocation is made. This view has a number of advantages:

- The references are passed implicitly as parameters.

- Pervasives are context switched with the rest of the thread state.
- If necessary, particular interfaces can be replaced for the purposes of a single invocation.

3.2.4 Memory Allocation

The programming model has to address the problem of memory allocation. An invocation across an interface can cause the creation of a concrete structure which occupies an area of memory. There needs to be a convention for determining:

- where this memory is allocated from, and
- how it may be freed.

In many systems the language runtime manages memory centrally (to the domain) and all objects may be allocated and freed in the same way. Some systems provide a garbage collector for automatic management of storage.

Unfortunately, Nemesis does not provide a central garbage collector[3] and a domain typically has a variety of pools to allocate memory from, each corresponding to an interface of type `Heap` (multiple heaps are used to allocate shared memory from areas with different access permissions). Moreover, it is desirable to preserve a degree of *communication transparency*: wherever possible, a programmer should not need to know whether a particular interface is exported by an object local to the domain or is a *surrogate* for a remote one.

Network-based RPC systems without garbage collection use conventions to decide when the RPC runtime has allocated memory for unmarshalling large or variable-sized parameters. Usually this memory is allocated by the language heap, although some RPC systems have allowed callers to specify different heaps at bind time (for example, [Roscoe94c]). To preserve transparency, in all cases the receiver of the data is responsible for freeing it. This ensures that the application code need not be aware of whether a local object or the RPC run time system has allocated memory.

[3]The problems of garbage collection in an environment where most memory is shared between protection domains is beyond the scope of this thesis. This issue is touched upon in section 6.2.

In systems where caller and object are in different protection domains but share areas of memory, the situation is complicated because of the desire to avoid unnecessary memory allocation and data copies. Ideally, the conventions used should accommodate both the cases where the caller allocates space for the results in advance, and where the callee allocates space on demand from caller memory during the invocation.

Nemesis uses parameter passing modes to indicate memory allocation policy: each parameter in a MIDDL operation signature has an associated mode, which is one of the following:

`IN`: Memory is allocated and initialised by client.
Client does not alter parameter during invocation.
Server may only access parameter during invocation, and cannot alter parameter.

`IN OUT`: Memory is allocated and initialised by client.
Client does not alter parameter during invocation.
Server may only access parameter during invocation, and may alter parameter.

`OUT`: Memory is allocated but *not* initialised by client.
Server may only access parameter during invocation, and is expected to initialise it.

`RESULT`: Memory is allocated by server, on client pervasive heap, and result copied into it. Pointer to this space is returned to the client.

The `OUT` mode allows results to be written by a local object into space already allocated by the client (in the stack frame, for example). In the remote case, it is more efficient than the `IN OUT` mode because the value does not need to be transmitted to the server; it is only returned.

These modes are all implemented on the Alpha processor using call by reference, except `RESULT`, which returns a pointer to the new storage. For values small enough to fit into a machine word, `IN` is coded as call by value and `RESULT` returns the value itself rather than a reference to it.

These conventions have been found to cover almost all cases encountered in practice. As a last resort, MIDDL possesses a `REF` type constructor which allows pointers to values of a particular type to be passed explicitly.

3.3 Linkage Model

The linkage model concerns the data structures used to link program components, and their interpretation at runtime. An early version of the linkage mechanism was described in [Roscoe94a]. Its goal is twofold:

1. To support and implement the Programming Model.
2. To reduce the total size of the system image through sharing of code and data.

A *stub compiler* is used to map MIDDL type definitions to C language types. The compiler, known as `middlc`[4] processes an interface specification and generates a header file giving C type declarations for the concrete types defined in the interface together with special types used to represent instances of the interface.

3.3.1 Interfaces

An interface is represented in memory as a *closure*: a record of two pointers, one to an array of function pointers and one to a state record (figure 3.2).

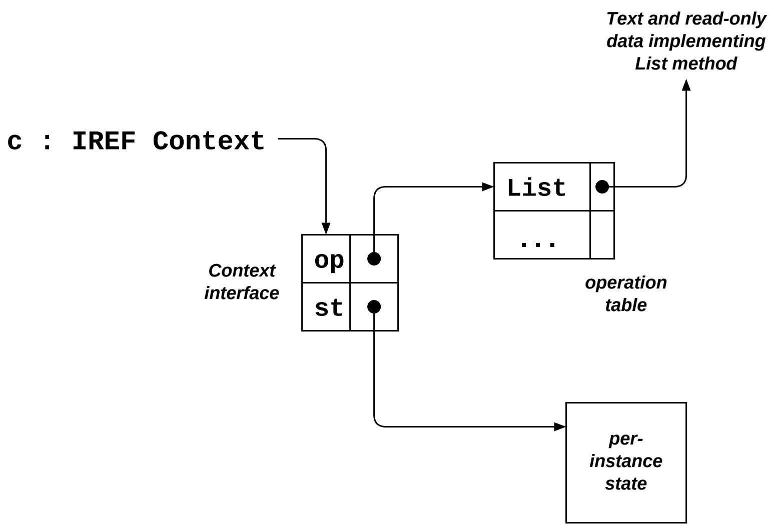


Figure 3.2: `Context` interface closure

[4]`middlc` was written by the author and David Evers.

To invoke an operation on an interface, the client calls through the appropriate element of the operation table, passing as first argument the address of the closure itself. The `middlc` compiler generates appropriate C data types so that an operation can be coded as, for example:

```
b = ctxt->op->Get( ctxt, "modules/DomainMgr", &dma );
```

In this case, `ctxt` is the interface reference. `middlc` generates C preprocessor macros so one may use the CLU-like syntax:

```
b = Context$Get( ctxt, "modules/DomainMgr", &dma );
```

3.3.2 Modules

A Nemesis *module* is a unit of loadable code, analogous to an object file. All the code in Nemesis exists in one module or another. These modules are quite small, typically about 10 kilobytes of text and about the same of constant data. The use of constructor interfaces for objects rather than explicit class structures makes it natural to write a module for each kind of object, containing code to implement both the object's interfaces and its constructors. Such modules are similar to CLU clusters [Liskov81], though 'own' variables are not permitted.

Modules are created by running the UNIX `ld` linker on object files. The result is a file which has no unresolved references, a few externally visible references, and no uninitialised or writable data[5].

All linkage between modules is performed via pointers to closures. A module will *export* one or more fixed closures (for example, the constructors) as externally visible symbols, and the system loader installs these in a name space (see section 3.4) when the module is loaded. To use the code in a module, an application must locate an interface for the module, often by name lookup. In this sense linking modules is entirely dynamic.

If a domain wishes to create an object with mutable state, it must invoke an operation on an existing interface which returns an interface reference of the required type and class.

[5]In other words, there is no `bss` and the contents of the `data` segment are constant.

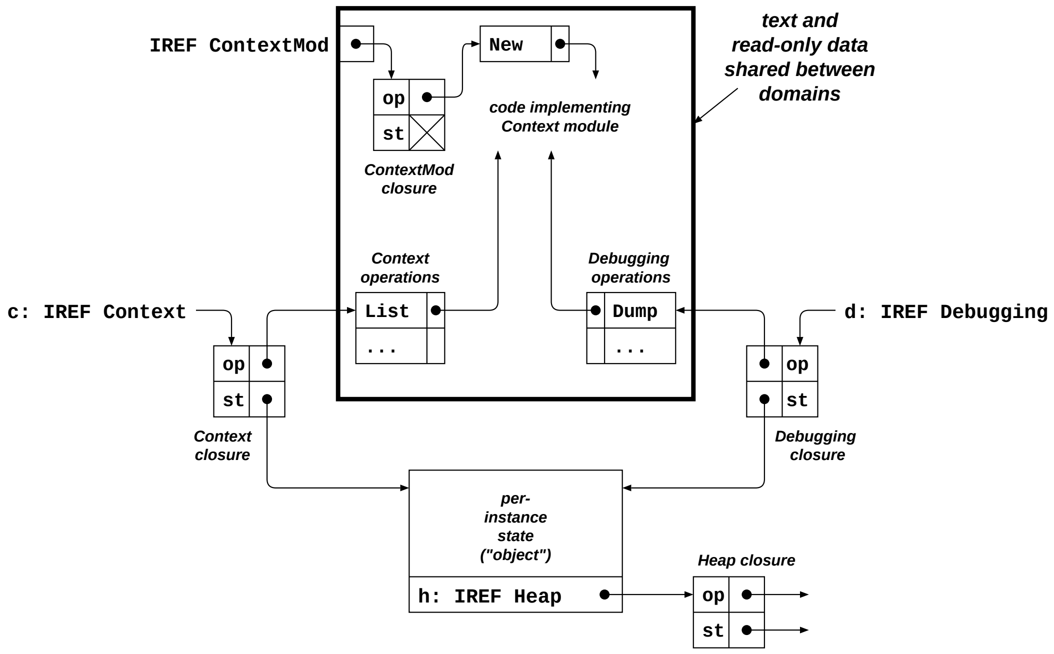


Figure 3.3: Module and instantiated object

Figure 3.3 shows an example where a domain has instantiated a naming context by calling the `New` operation of an interface of type `ContextMod`. The latter is implemented by a module with no mutable state, and has instantiated an object with two interfaces, of types `Context` and `Debugging`. The module has returned pointers to these in the results `c` and `d`. The state of the object includes a heap interface reference, passed as a parameter to the constructor and closed over.

3.3.3 Address Space Structure

The use of interfaces and modules in Nemesis permits a model where all text and data occupies a single address space, since there is no need for data or text to be at well-known addresses in each domain. The increasing use of 64-bit processors with very large virtual address spaces (the Alpha processor on which Nemesis runs implements 43 bits of a 64-bit architectural addressing range) makes the issue of allocating single addresses to each object in the system relatively easy.

It must be emphasised that this in no way implies a lack of memory *protection* between domains. The virtual address *translations* in Nemesis are the same for all domains, while the *protection* rights on a given page may vary. Virtual address space in Nemesis is divided into segments (sometimes called *stretches*) which have access control lists associated with them. What it does mean is that any area of memory in Nemesis can be shared, and addresses of memory locations do not change between domains.

3.4 Naming and Runtime Typing

While simple addresses in the single address space suffice to identify any interface (or other data value) in the system, a more structured system of naming is also required.

The name space in Nemesis is completely independent of the rest of the operating system. While some operating system components do implement part of the name space, most naming contexts are first-class objects: they can be created at will and are capable of naming any value which has a MIDDL type.

There are few restrictions on how the name space is structured. The model followed is that of [Saltzer79]: a *name* is a textual string, a *binding* is an association of a name with some value, and a *context* is a collection of bindings. *Resolving* a name is the process of locating the value bound to it. Name resolution requires that a context be specified.

Context Interfaces

Naming contexts are represented by interfaces which conform to the type `Context`. Operations are provided to bind a name to any value, resolve the name in the context and delete a binding from the context. The values bound in a context can be of arbitrary type, in particular they can be references to other interfaces of type `Context`. *Naming graphs* can be constructed in this way, and a *pathname* may be presented to a context in place of a simple name. A pathname consists of a sequence of names separated by distinguished characters, either '`/`' or '`>`'. To resolve such a pathname, the context object examines the first component of the name. If this name resolves to a context, this second context is invoked to resolve the remainder of the pathname.

Ordered Merges of Contexts

The `MergedContext` interface type is a subtype of `Context`, modelled after a similar facility in Spring [Radia93]. An instance of `MergedContext` represents a composition of naming contexts; when the merge is searched, each component context is queried in turn to try and resolve the first element of the name. Operations are provided to add and remove contexts from the merge.

An Example

Figure 3.4 illustrates part of a naming graph created by the Nemesis system at boot time. Context *A* is the first to be created. Since one must always specify a context in which to resolve a name, there is no distinguished root. However *A* serves as a root for the kernel by convention. Context *B* holds local interfaces created by the system, thus '`Services>DomainMgr`' is a name for the Domain Manager service, relative to context *A*. Any closures exported by loaded modules are stored in context *C* ('`Modules`'), and are used during domain bootstrapping.

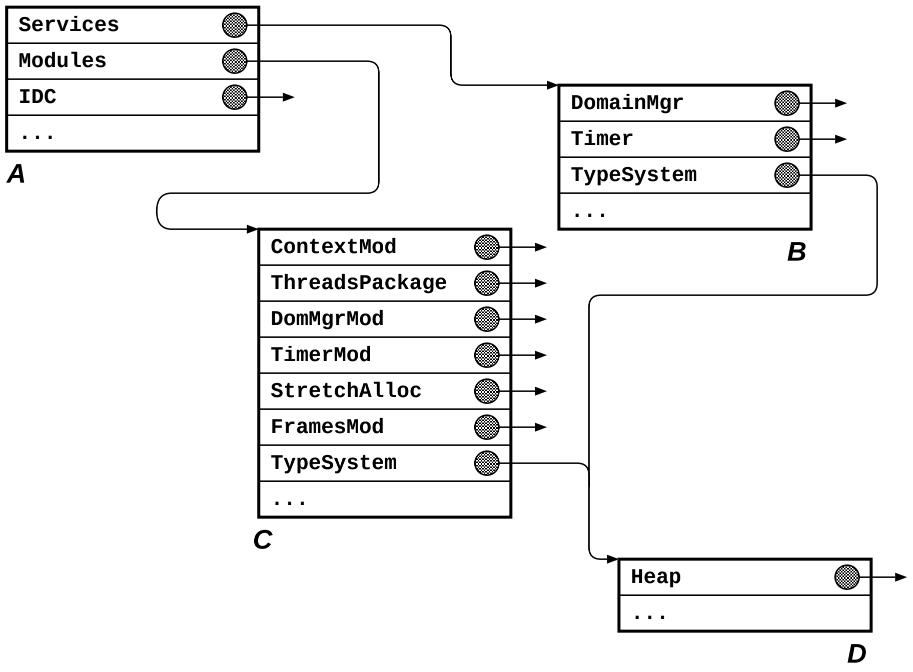


Figure 3.4: Example name space

Context *D* has two names relative to the root, '`Services>TypeSystem`' and '`Modules>TypeSystem`'. This context is not in fact implemented in the usual way, but is part of the runtime type system, described in the next section.

3.4.1 Run Time Type System

The Type System is a system service which adds a primitive form of dynamic typing, similar to [Rovner85]. Each MIDDL type is assigned a unique `Type.Code`, and the `Type.Any` type provides support for data values whose type is not known at compile time. The `TypeSystem` interface provides the operations `IsType`, to determine whether a `Type.Any` conforms to a particular type, and `Narrow`, which converts a `Type.Any` to a specified type if the type equivalence rules permit. A major use of `Type.Any` is in the naming interfaces: values of this type are bound to names in the name space.

The Type System data structures are accessible at run time through a series of interfaces whose types are subtypes of `Context`. For example, an operation within an interface is represented by an interface of type `Operation`, whose naming context includes all the parameters of the operation. Every MIDDL interface type is completely represented in this way.

3.4.2 CLANGER

A good example of how the programming and linkage models work well in practice is CLANGER[6] [Roscoe95] , a novel interpreted language for operating systems. CLANGER relies on the following three system features:

- a naming service which can name any typed value in the system,
- complete type information available at runtime, and
- a uniform model of interface linkage.

In CLANGER a variable name is simply a pathname relative to a naming context specified when the interpreter was instantiated. All values in the language are represented as `Type.Any`s. The language allows operations to be invoked on

[6] CLANGER has been implemented by Steven Hand.

variables which are interface references by synthesising C call frames. Access to the Type System allows the interpreter to type-check and narrow the arguments to the invocation, and select appropriate types for the return values.

The invocation feature means that the language can be fully general without a complex interpreter or the need to write interface 'wrappers' in a compiled language. This capability was previously only available in development systems such as Oberon [Gutknecht] and not in a general-purpose, protected operating system. CLANGER can be used for prototyping, debugging, embedded control, operating system configuration and as a general purpose programmable command shell.

3.5 Domain bootstrapping

The business of starting up a new domain in Nemesis is of interest, partly because the system is very different from UNIX and partly because it gives an example of the use of a single address space to simplify some programming problems.

The traditional UNIX `fork` primitive is not available in Nemesis. The state of a running domain consists of a large number of objects scattered around memory, many of which are specific to the domain. Duplicating this information for a child domain is not possible, and would create much confusion even if it were, particularly for domains with communication channels to the parent. In any case, `fork` is rarely used for producing an exact duplicate of the parent process, rather it is a convenient way of bootstrapping a process largely by copying the relevant data structures. In Nemesis, as in other systems without `fork` such as VMS, this can be achieved by other means.

The kernel's view of a domain is limited to a single data structure called the *Domain Control Block*, or DCB. This contains scheduling information, communication end-points, a protection domain identifier, an upcall entry point for the domain, and a small initial stack. The DCB is divided into two areas. One is writable by the domain itself, the other is readable but not writable. A privileged service called the *Domain Manager* creates DCBs and links them into the scheduler data structures.

The arguments to the Domain Manager are a set of Quality of Service (QoS) parameters for the new domain, together with a single closure pointer of type `DomainEntryPoint`. This closure provides the initial entry point to the domain in its sole operation (called `Go`), and the state record should contain everything the new domain needs to get going.

The creation of this DCB is the only involvement the operating system proper has in the process. Everything else is performed by the two domains involved: the parent creates the initial closure for the domain, and the child on startup locates all the necessary services it needs which have not been provided by the parent. Figure 3.5 shows the process.

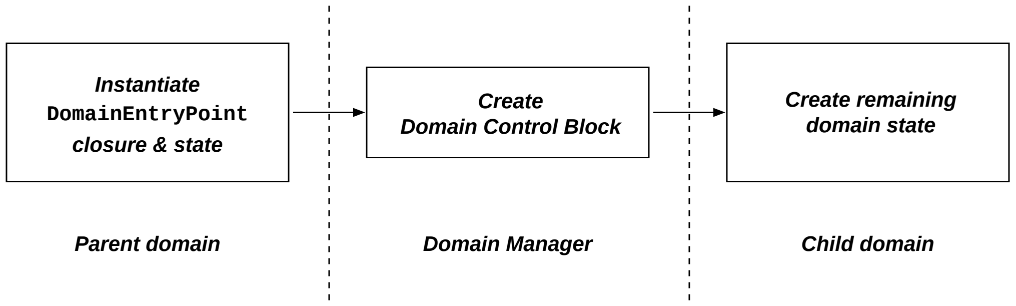


Figure 3.5: Creation of a new domain

The `DomainEntryPoint` closure is the equivalent of `main` in UNIX, with the state record taking the place of the command line arguments. By convention the calling domain creates the minimum necessary state, namely:

- A naming context.
- A heap for memory allocation.
- The runtime type system (see below).

From the name space, a domain can acquire all the interface references it needs to execute. One useful consequence of this is that an application can be debugged in an artificial environment by passing it a name space containing bindings to debugging versions of modules. The type system is needed to narrow types returned from the name space. The heap is used to create the initial objects needed by the new domain.

The Builder

To save a programmer the tedium of writing both sides of the domain initialisation code, a module is provided called the *Builder*. The Builder takes a

`ThreadClosure`[7] which represents the initial thread of a multi-threaded domain to be created. The Builder instantiates an initial heap for the new domain. Most heap implementations in Nemesis use a single area of storage for both their internal state and the memory they allocate, so the parent domain can create a heap, allocate initial structures for the child within it, and then hand it over in its entirety to the new domain.

The Builder returns a `DomainEntryPoint` closure which can be passed to the Domain Manager. When started up, the new domain executes code within the Builder module which carries out conventional initialisation procedures, including instantiating a threads package. The main thread entry point is also Builder code, creating the remaining state before entering the thread procedure originally specified. Figure 3.6 shows the sequence of events.

This illustrates two situations where a module executes in two domains. In the first, part of the Builder executes the parent and another part executes in the child. In the second, a single heap object is instantiated and executes in the parent, and is then handed off to the child, which continues to invoke operations upon it. In both cases the programmer need not be aware of this, and instantiating a new domain with a fully functional runtime system is a fairly painless operation.

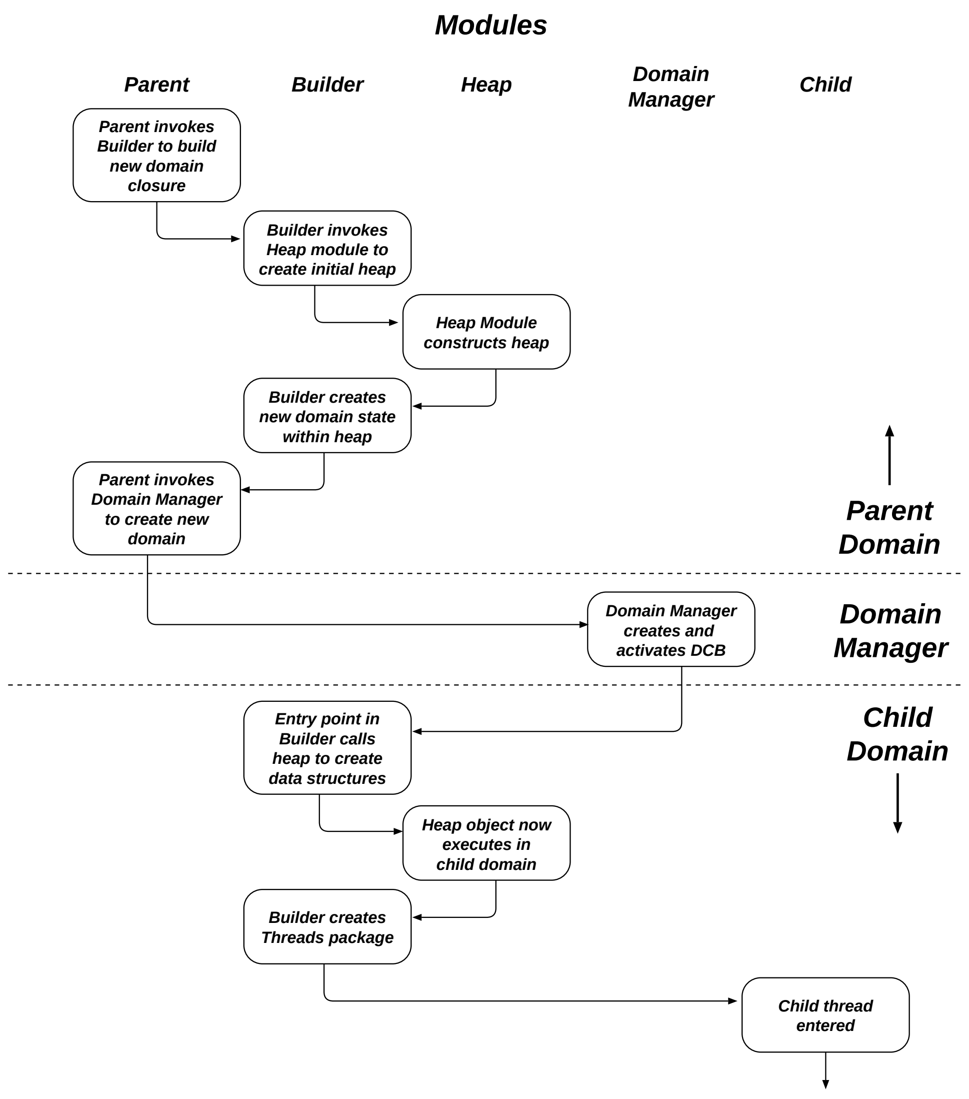
3.6 Discussion

Nemesis has adopted a different method of achieving sharing from Hemlock: objects are used throughout and there are no real ‘global variables’ in a Nemesis program. Instead, state is held either in closures or as part of an extended thread context containing pervasive interfaces in addition to processor register values.

However, the combination of an entirely interface-based programming model, and a per-machine single address space works well in practice. The code for Nemesis is highly modular, and there are many situations where use is made both of the ability of an object to export multiple interfaces, and of the same interface type to be implemented by several classes of object. Unlike Opal and Angel, the single address space is fully exploited.

The typing and code reuse benefits of interfaces are achieved independently

[7] The user-level threads equivalent of a domain entry point.



Modules
Parent
Builder
Heap
Domain Manager
Child
Parent invokes Builder to build new domain closure
Builder invokes Heap module to create initial heap
Heap Module constructs heap
Builder creates new domain state within heap
Parent invokes Domain Manager to create new domain
Parent Domain
Domain Manager creates and activates DCB
Domain Manager
Child Domain
Entry point in Builder calls heap to create data structures
Heap object now executes in child domain
Builder creates Threads package
Child thread entered

Figure 3.6: Action of the Builder

of any particular programming language. No runtime system is strictly required, although dynamic typing does require the Type System module. This has enabled the use of interfaces throughout the low levels of the operating system. As an aside, this in turn enables Clanger to be used at a very basic level in the system, and almost the whole operating system to be *virtualised*: to run entirely inside a Nemesis domain.

System development is greatly simplified by the ability to pass pointers between domains. This is particularly useful in situations involving a large quantity of data and where it is undesirable to copy it, for example the processing of video streams. The architecture of Nemesis tries to discourage pipelines of domains, since it is preferable to do all processing on a stream within a single application and thereby preserve QoS guarantees. However, when information must be passed to another domain (for example, a frame store driver), code on both sides of the protection boundary can use the same addresses. Indeed, many code modules in Nemesis straddle protection domain boundaries. This idea is returned to in chapter 5.

As another example, interrupt service routines (ISRs) are entered with a register loaded with a pointer to their state. The device driver domain assigned this pointer when it installed the ISR, and the address is valid regardless of which domain is currently scheduled. The maintenance of scatter-gather maps to enable devices to DMA data to and from virtual memory addresses in client domains is similarly simplified.

3.6.1 Overhead of using interfaces

The primary concern with the linkage model is the overhead of passing closures with interface operations. Table 3.1 shows the results of an experiment to measure null procedure call times. The machine used was a DEC Alpha 3000/400 Sandpiper running OSF/1. The compiler used in these experiments (as for all the results in this dissertation) was GCC 2.6.3, with optimisation on (`-O2`).

	min.	mean	std. dev.
Procedure call	34	34.6	32.50
Nemesis closure	39	39.4	25.58
Dynamic C++	39	39.5	30.15
Static C++	34	34.4	28.75

Table 3.1: Call times in cycles for different calling conventions

The overhead of passing a closure in a procedure call is 5 machine cycles (about 37ns in this case). Not surprisingly this corresponds with the overhead of a C++ virtual function call, which is generally regarded as an acceptable price to pay for modularity. The final line of the table illustrates the advantage to be gained from the compiler being able to optimise across invocation boundaries: if the called object is static, GCC can use a simple procedure call to implement the method invocation. Nemesis forgoes this performance advantage in favour of full dynamic linking.

3.6.2 Effects of sharing code

In theory, the performance overhead of using closures for cross-interface invocations should be compensated for to some extent by the increased cache performance resulting from decreased code size: The granularity at which text is shared in Nemesis means that the code portion of the working set of the complete system is much smaller than in a statically-linked, multiple address space system,

Unfortunately, observing the effect of image size upon execution speed proved to be extremely difficult due to the cache architecture of the machines available. The DECchip EB64 development board and the DEC3000/400 Sandpiper workstation both use a DECchip 21064-AA processor with 8k of instruction cache and 8k of data cache. Both these caches are physically addressed and direct mapped, as is the unified secondary cache of 512k bytes.

The non-associativity of the cache system means that the likelihood of 'hot spots' occurring in the cache during normal operation is very high. The result is that minor changes in the arrangement of code in the image can have a large effect on the performance of the system as a whole.

Figure 3.7 shows the result of altering the order in which modules are loaded into the address space. A single, reasonably large module (42k text, 15k data) implementing the front end of `middlc` was moved through the load order, and for each configuration a benchmark performed. The benchmark consisted of compiling a set of interfaces from an in-memory filing system 2000 times. The compiler was the only application domain executing for the duration of the benchmark.

The slowest run recorded took over 65% longer than the quickest. Altering the order in which object files were linked into the module while keeping the module's position in memory constant produced a similar wide distribution. With as much variation as this it is difficult to make comparisons, but an identical experiment was performed with a version of the `middlc` module which contained the complete runtime statically linked in, and accessed via procedure calls rather than through

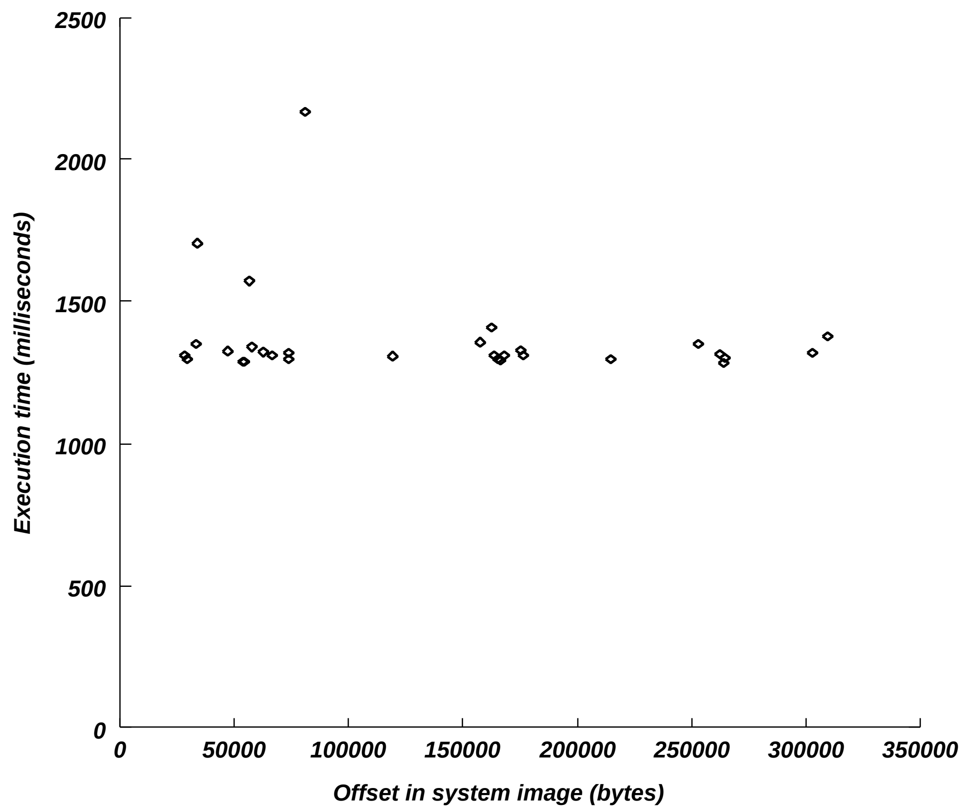


Figure 3.7: Variation in execution time for `middlc`

closures. The results were broadly similar, with the norm around 1220ms as opposed to 1300ms for the shared runtime version, making the overhead of using closures in this case about 6.5%.

Optimising memory layout of code and data for cache performance on a system-wide basis is a large research topic, and beyond the scope of this dissertation. Intuitively, however, sharing code between domains should improve performance in Nemesis as a result of the increased cache performance, as it does in other systems. The finer granularity of sharing in Nemesis may cause this performance gain to outweigh the overhead of interface calls, however, accurately quantifying the benefit of such sharing is difficult in a system as sensitive as the one used here. Increasing the associativity of one or more caches should dramatically improve the predictability of the system as well as its performance.

3.7 Summary

Nemesis programs and subsystems are composed of objects which communicate across typed interfaces. Interface types are ADTs defined in the MIDDL interface definition language, which also supports definitions of concrete data types and exceptions. Objects are constructed by invocations across interfaces. There is no explicit notion of a class. There are no well-known interfaces, but a number of *pervasive* interfaces are regarded as part of a thread context.

Interfaces are implemented as closures. The system is composed of stateless modules which export constant closures. All linking between modules is dynamic, and the system employs a single address space to simplify organisation and enhance sharing of data and text.

A uniform, flexible and extensible name service is implemented above interfaces, together with a run time type system which provides dynamic types, type narrowing, and information on type representation which can be used by the command language to interact with any system components.

The single-address space aspect of Nemesis together with its programming model based on objects rather than a single data segment prohibit the use of a fork-like primitive to create domains. Instead, runtime facilities are provided to instantiate a new domain specified by an interface closure. The single address space enables the parent domain to hand off stateful objects to the child.

The performance overhead of using closures for linkage is small, roughly equivalent to the use of virtual functions in C++. However, it is clear that the cache design of the machines on which Nemesis currently runs presents serious obstacles to the measurement of any performance benefits of small code size.

Chapter 4

Scheduling and Events

This chapter describes the problem of scheduling applications in an operating system from a QoS perspective. It discusses some existing techniques which have been applied to scheduling multimedia applications, and then describes the Nemesis scheduler in detail. This scheduler delivers guarantees of processor bandwidth and timeliness by transforming the problem into two components: a soluble real-time scheduling problem and the issue of allocation of slack time in the system. The client interface is designed so as to support multiplexing of the CPU within application domains, a requirement made all the more important by the use of an architecture which places much system functionality in the application. The use of event channels to implement inter-domain communication and synchronisation is described, and finally the problems of handling interrupts are mentioned, together with the solution adopted in Nemesis.

4.1 Scheduling Issues

The function of an operating system scheduler is to allocate CPU time to activities in such a way that the processor is used efficiently and all processes[1] make progress according to some policy.

In a traditional workstation operating system this policy is simply that all activities should receive some CPU time over a long period, with some having

[1]The terms *process* and *task* are used interchangeably in this chapter to denote an activity schedulable by the operating system.

priority over others, but in a multi-service system the policy adopted must now have additional constraints based on the passage of real time.

True real-time operating systems have quite strict constraints: in a hard real-time system correct results *must* be delivered at or shortly before the correct time, or the system can be said to have failed. In a soft real-time system, the *temporal* constraint on correctness is relaxed (results can be allowed to be a little late), though the results must still be *logically* correct (the computation must have completed).

Systems handling continuous media frequently have different constraints: not only can results sometimes be late, they can sometimes be incomplete as well. [Hyden94] gives examples of situations where computations on multimedia data need not complete for the partial results to be useful, and a useful taxonomy of algorithms which can make use of variable amounts of CPU time is given in [Liu91].

Both the additional constraints on CPU allocation and the tolerant nature of some multimedia applications is apparent from attempts to capture and present video and audio data using conventional UNIX workstations, for example the Medusa system [Hopper92]. Even with some hardware assistance, the audio is broken and the picture jerky and sometimes fragmented when other processes (such as a large compilation) are competing for CPU time. However, sufficient information generally does get through to allow users to see and hear what is going on.

Nemesis represents an attempt to do better than this: firstly, to reduce the crosstalk between applications so that the results are less degraded under load; secondly, to allow application-specific degradation (for example, in a way less obvious to the human eye and ear); and thirdly to support applications which cannot afford to degrade at all by providing real guarantees on CPU allocation.

4.1.1 QoS Specification

Before specifying mechanisms for multiplexing the processor among applications within Nemesis, it is important to consider the representation of QoS used between the operating system allocation mechanisms and applications. As in the rest of this dissertation, CPU time is taken as an example since it is usually the most important resource. However, the principles given here apply to most system resources. The representation, a QoS specification, must serve two purposes.

Firstly, it must allow the application to specify its requirements for CPU time. From the application's point of view, the more sophisticated this specification can be, the better. However, at odds with the desire for expressiveness is the second function of a QoS specification: to enable the resource provider (in this case the scheduler) to allocate resources between applications efficiently while satisfying their requirements as far as possible. A key part of this is to be able to schedule applications quickly: the overhead of recalculating a complex schedule during a context switch is undesirable in a workstation operating system. For this reason, it is difficult (and may be unwise to try) to fully decouple the specification of CPU time requirements from the scheduling algorithm employed.

There is a further incentive to keep the nature of a QoS specification simple. Unlike the hard real-time case, most applications' requirements are not known precisely in advance. Furthermore, these requirements change over time; variable bit-rate compressed video is a good example. In these cases statistical or probabilistic guarantees are more appropriate. Furthermore, the application being scheduled is typically multiplexing its allocation of CPU time over several activities internally in a way that is almost impossible to express to a kernel-level scheduler. Any measure of QoS requirements will be approximate at best.

To summarise, the type of QoS specification used by a scheduler will be a compromise between the complexity of expressing fully the needs of any application, and the simplicity required to dynamically schedule a collection of applications with low overhead.

4.2 Scheduling Algorithms

As well as the (relatively simple) code to switch between running domains, the Nemesis scheduler has a variety of functions. It must:

- account for the time used by each holder of a QoS guarantee and provide a policing mechanism to ensure domains do not overrun their allotted time,
- implement a scheduling policy to ensure that each contract is satisfied,

- block and unblock domains in response to their requests and the arrival of events,
- present an interface to domains which makes them aware both of their own scheduling and of the passage of real time,
- provide a mechanism supporting the efficient implementation of potentially specialised threads packages within domains.

The algorithm used to schedule applications is closely related to the QoS specification used, and for Nemesis a number of options were considered.

4.2.1 Priorities

Priority-based systems assign each schedulable process an integer representing its relative importance, and schedule the runnable process with the highest priority. They are generally unsuitable for supporting multimedia applications: [Black94] provides a comprehensive discussion of the problems of priority-based scheduling. While scheduling algorithms which are based on priority are often simple and efficient, priority does not give a realistic measure of the requirements of an application: it says nothing about the quantity of CPU time an application is allocated. Instead a process is simply given any CPU time unused by a higher-priority application.

Despite this, several operating systems which use priority-based scheduling, provide so-called real-time application priority classes, intended for multimedia processes. Examples include Sun Microsystems' Solaris 2 and Microsoft's Windows NT. Applications instantiated in this class run at a higher priority than operating system processes, such as pagers and device drivers. They are required to block or yield the CPU voluntarily every so often so that the operating system and other applications can proceed. Failure to do this can cause the system to hang—the application has effectively taken over the machine. Furthermore, as [Nieh93] points out, it is nearly impossible in the presence of several real-time applications to assign priorities with acceptable results.

4.2.2 Rate Monotonic

[Liu73] describes the Rate Monotonic (RM) algorithm for scheduling off-line a set of periodic hard real-time tasks, which essentially involves assigning static priorities to the tasks such that those with the highest frequency are given the highest priority. The schedule calculated by RM is always feasible if the total utilisation of the processor is less than $ln2$, and for many task sets RM produces a feasible schedule for higher utilisation. It relies on the following assumptions about the task set[2]:

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

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

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

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

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

4.2.3 Earliest Deadline First

The Earliest Deadline First (EDF) algorithm also presented in [Liu73] is a dynamic scheduling algorithm which will give a feasible schedule when the CPU utilisation is 100%. It, too, relies on the assumptions of Section 4.2.2, and works by considering the deadline of a task to be the time at which the results of its computation are due.

EDF scheduling is used by the Sumo project at Lancaster University to support continuous media applications over the Chorus microkernel [Coulson93]. The system uses EDF to schedule a class of kernel threads over which user-level

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

threads are multiplexed. The deadlines are presented to the kernel scheduler by the user tasks, a decision which has two consequences. Firstly, the guarantees provided by the EDF algorithm are now hints at best; deadlines can frequently be missed due to the unexpected arrival of a new task and deadline. Secondly, user-level schedulers are expected to cooperate and not present difficult deadlines to the kernel. If a user process (through error or malicious design) presents deadlines requiring more of the CPU than the kernel expects to allocate, all tasks may be disrupted. In other words, the policing mechanism is inadequate over short to medium time periods.

At first sight, the assumptions required by the static RM and dynamic EDF algorithms seem to rule out their use in a general-purpose operating system: tasks which are periodic and independent with fixed run times are not the norm. In particular, the independence of tasks in the presence of shared server tasks poses a particular difficulty. However, in a system such as Nemesis, where shared server tasks are rarely called, EDF may have some benefit, especially if the algorithm is viewed as a means of *sharing the processor between domains* rather than *completing tasks from a changing set*.

4.2.4 Processor Bandwidth

The Nemesis scheduler builds on many ideas in the Nemo system built by Eoin Hyden and described in [Hyden94]. In Nemo, applications negotiated contracts with the system for *processor bandwidth* (PB) in a manner analogous to modern high speed networks. The concept of PB consisted of a percentage share of the processor time together with some measure of the granularity with which the share should be allocated. It can be represented as a pair (p, s), where the application will receive s seconds of processor time every p seconds, and so contains measures of both required bandwidth and acceptable jitter. An admission control mechanism ensures that the system never contracts out more than 100% of the available PB.

Nemo investigated use of both RM and EDF algorithms to schedule processes, and also optionally allowed applications to specify their own deadlines. It worked well in practice, although it did not address the issue of handling interrupts from devices, and the problems of communicating domains. Nevertheless, the PB idea (together with elements of Nemo's client interface) have been used in Nemesis. In particular, it represents a highly appropriate form of QoS specification.

4.2.5 Fawn Jubilees

The Fawn system [Black94] adopts a novel scheme whereby each process is allocated a particular slice of time over a system-wide, fixed period (31.5 milliseconds in the system reported), called a *jubilee*. All processes are scheduled in turn within a jubilee. At the end of the jubilee all allocations of CPU time are reset.

Extra time remaining towards the end of a jubilee is dealt out using a hierarchy of queues: when a process runs out of time in a queue it is moved to the next lower queue. Each process has a number of different allocations of time corresponding to different queues it may find itself on. When one queue is empty, the scheduler starts on the next queue down until the jubilee is over, thus guaranteed time is merely the top-level time allocation.

This approach was rejected at an early stage in the development of Nemesis due to its inflexibility[3]. No guarantees are given about how often a process is scheduled other than the system-wide jubilee length, thus all processes must be scheduled with this frequency. Accounting is cheap since allocations only occur on jubilee boundaries, but if one process absolutely must be scheduled at some high frequency, then all processes must, resulting in an unacceptably high number of context switches. For n processes, there must be n context switches per jubilee, and it may be impossible to schedule at this granularity.

There are environments (such as the switch line cards on which Fawn was developed) where this kind of scheduling mechanism is highly appropriate. However, it is less useful in a general-purpose operating system intended for workstations.

4.3 Scheduling in Nemesis

Scheduling in Nemesis is discussed in the next few sections. First, the service model and architecture are presented: how clients of the scheduler view the services it provides. Then the algorithm itself is described, followed by the interface between the scheduler and the client. Finally, the handling of event channels and device interrupts is discussed.

[3]It should be noted that the algorithm referred to under the title 'Interprocess Scheduling in Nemesis' in [Black94] bears no resemblance to the Nemesis operating system described herein.

4.3.1 Service Model and Architecture

The scheduler deals with entities called *scheduling domains*, or *sdoms*. An sdom corresponds to either a single Nemesis domain or a set of domains collectively allocated a share of the available processor time. Each sdom receives a QoS from the system specified by a tuple $\{s, p, x, l\}$. The slice s and period p together represent the processor bandwidth to the sdom: it will receive at least s seconds of CPU time in each period of length p. x is a boolean value used to indicate whether the sdom is prepared to receive extra CPU time left over in the system. l, the *unblocking latency*, is described in Section 4.4.5 below.

The precise nature of guarantee the Nemesis scheduler provides to an sdom is this: the scheduler will divide real time into a set of periods of length p for the sdom in question, and during each period the sdom will receive the CPU for some number of slices whose total length will be at least s.

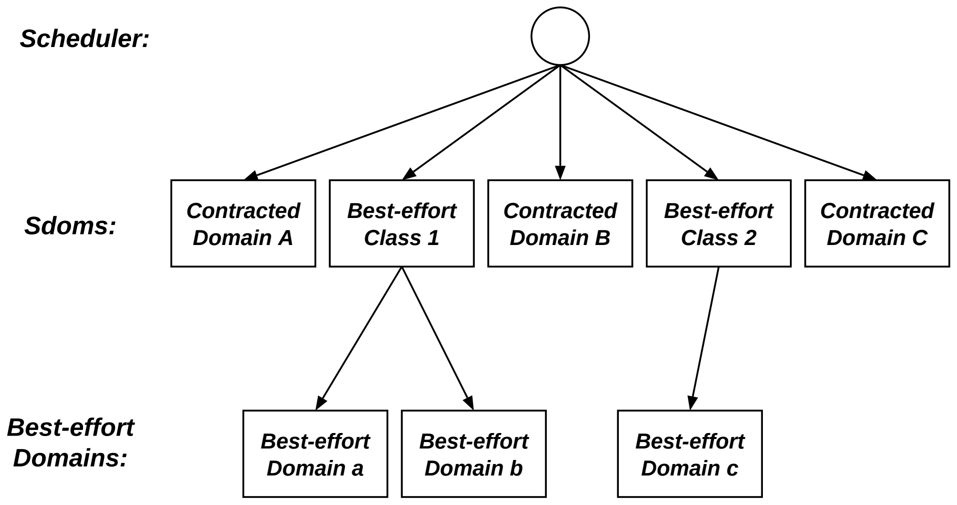


Figure 4.1: Scheduling service architecture

The service architecture is illustrated in figure 4.1. Sdoms usually correspond to contracted domains, but also correspond to best-effort classes of domains. In the latter case, processor time allotted to the sdom is shared out among its domains according to one of several algorithms, such as simple round-robin or multi-level feedback queues.

The advantage of this approach is that a portion of the total CPU time can be reserved for domains with no special timing requirements to ensure that they are not starved of processor time. Also, several different algorithms for scheduling best-effort domains can be run in parallel without impacting the performance of time-critical activities.

4.3.2 Kernel Structure

The Nemesis kernel consists almost entirely of interrupt and trap handlers; there are *no* kernel threads. When the kernel is entered from a domain (as opposed to another interrupt handler) a new kernel stack is created afresh in a fixed (per processor) area of memory. The domain operations described below such as `block`, `yield` and `send` are implemented entirely in PALcode [Sites92], though they may cause the scheduler to be entered. Having the operations entirely in PALcode reduces the number of full context switches required (the penalty for a PALcode trap is only two pipeline drains), and simplifies the implementation since PALcode executes with all internal chip registers available and all interrupts masked. Interrupt dispatching is also performed in PAL mode.

The scheduler is implemented as an Alpha/AXP software interrupt handler [DEC92], and so executes in the protection domain of the currently running domain. The scheduler is always the last pending interrupt to be serviced, and executes with all interrupts masked.

4.4 The Nemesis Scheduler

The operation of the scheduler can now be described.

4.4.1 Scheduler Domain States

An sdom can be in one of five states:

- running
- runnable
- waiting
- running optimistically
- blocked

Running sdoms have one of their domains being executed by a processor, in time that they have been guaranteed by system. Runnable sdoms have guaranteed time available, but are not currently scheduled. Waiting sdoms are waiting for a new allocation of time, which will notionally be the start of their next period. During this time they may be allocated spare time in the system and 'run optimistically'. Finally, they may be blocked until an event is transmitted to one of their domains.

4.4.2 The Basic Scheduling Algorithm

With each runnable sdom is associated a deadline d, always set to the time at which the sdom's current period ends, and a value r which is the time remaining to the sdom within its current period. There are queues Q_r and Q_w of runnable and waiting sdoms, both sorted by deadline, and a third queue Q_b of blocked sdoms.

The scheduler requires a hardware timer that will cause the scheduler to be entered at or very shortly after a specified time in the future; ideally a microsecond-resolution interval timer. Such a timer is used on the EB64 board, but has to be simulated with a 122μs periodic ticker on the Sandpiper.

When the scheduler is entered at time t as a result of a timer interrupt or an event delivery:

1. the time for which the current sdom has been running is deducted from its value of r.
2. if r is now zero, the sdom is inserted in Q_w.
3. for each entry on Q_w for which $t > d$, r is set to s, and the new deadline d' is set to $d + p$. This sdom is moved to Q_r again.
4. a time is calculated for the next timer interrupt depending on which of d_r and $d_w + p_w$ is the lower, where d_x is the deadline of the head of Q_x, and p_x is the period of the head of Q_x.
    - if d_r is the lower, the time is $t + r_r$. This is the point when the head of Q_r will run out of time.

- otherwise, the time is d_w. This is the time that the head of Q_w will become runnable and take over from the head of Q_r.

The interval timer is set to interrupt at this time.

5. the scheduler always runs the head of the run queue: the sdom with the earliest deadline.

This basic algorithm will meet all contracts provided that the total share of the CPU does not exceed 100% (i.e. $\sum s_i/p_i < 1$). Moreover, it can efficiently support domains requiring a wide range of scheduling granularities.

Firstly, note that the scheduler is only entered when a change of domain is potentially necessary. Step 4 ensures that extra CPU time is only allocated to sdoms when the scheduler has been called for some other reason. The overhead for actually *allocating* the time is very small indeed: the operation is a comparison and an addition.

Secondly, the existence of a feasible schedule (one which satisfies all contracts) is guaranteed by the admission control algorithm since the total share of the processor is less than 100%, and slices can be executed at any point during their period. In the limit, all sdoms can proceed simultaneously with an instantaneous share of the processor which is constant over time. This limit is often referred to as *processor sharing* [Coffman73][4].

Finally if we regard a 'task' as 'the execution of an sdom for s nanoseconds', this approach satisfies the conditions required for an EDF algorithm to function correctly: requests for tasks are periodic with fixed interval between requests and constant run-time. All tasks are truly independent and non-periodic tasks do not exist, providing no new tasks are introduced into the system (see below). The EDF result in [Liu73] shows that the algorithm does, in fact, work: all contracts will be met.

This argument relies on two simplifications: firstly, that scheduling overhead is negligible, and secondly that the system is in a steady state with no sdoms being introduced to or removed from the queues. These points are addressed below as well as the other elements of the scheduling algorithm, such as blocking and the use of slack time in the system.

[4]I am indebted to Simon Crosby for this line of argument

4.4.3 Taking overhead into account

Scheduling overhead is currently made up for by 'slack' time in the system (100% of the CPU is never contracted out), and by not counting time in the scheduler as used by anybody. This has worked very satisfactorily in practice under quite high load, with a reasonable number of domains. It is conceivable that a pathological collection of periods and slices might induce the highest possible reschedule frequency, which is the frequency of the domain with the smallest period times the number of domains. However, this is intuitively highly unlikely, and with more analysis this might be avoided in the admission control system. Alternatively it could be detected and dealt with at runtime, for example by renegotiating contracts to increase the slack time in the system.

4.4.4 Removing Domains from the System

An sdom can cease to be considered by the scheduler in one of two ways. The first is that it can simply be killed: it is unlinked from its queue, its contract annulled and the storage it occupied returned to the free pool. This poses no particular problems: the system will continue to schedule things as normal with more slack time.

Secondly, a domain can issue a `block` PALcode call, which sets a flag in the domain's state indicating that it has requested a block, and enters the scheduler. The domain is descheduled as normal and placed on the blocked queue Q_b. As above, no special action needs to be taken by the scheduler.

If a domain has no further useful work to perform in its current period, it can issue a `yield` call, which simply sets $r := 0$ for the domain and causes a reschedule.

4.4.5 Adding Domains to the System

By contrast, adding a domain to the set considered by the scheduler, whether by creating a new domain or unblocking an existing one, is more complex since the total resource demand increases. The key issue is deciding the values of d and r for the new domain.

If the sdom is to be introduced at time t, a safe option is to set $d := t + p$ and $r := s$. This introduces the sdom with the maximum scheduling leeway; since a feasible schedule exists no deadlines will be missed as a result of the new domain.

For most domains this is sufficient, and it is the default behaviour for almost all domains. In the case of device drivers reacting to an interrupt, sometimes faster response may be required. When unblocking an sdom which has been asleep for more than its period, the scheduler sets $r := s$ and $d := t + l$, where l is the latency hint. For most sdoms l will be equal to p to prevent deadlines being missed. For device drivers l may be reduced.

The consequences of reducing l in this way are that if such an sdom is woken up when the system is under heavy load, some sdoms may miss their deadline for one of their periods. The scheduler's behaviour in these circumstances is to truncate the running time of the sdoms: they lose part of their slice for that period. Thereafter, things settle down.

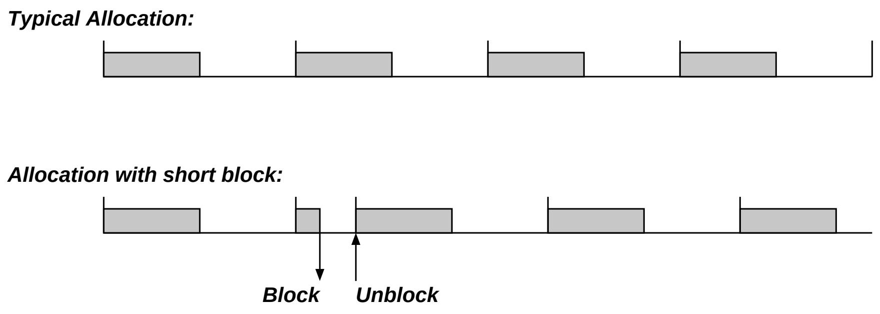


Figure 4.2: Unfairness due to short blocks

Even with sdoms for which $l = p$, a problem can arise if a domain is unblocked before the end of the period in which it was originally blocked (see figure 4.2). The policy above would give it a fresh allocation and period immediately, which is not entirely fair. One approach is leave r and d unchanged over the short block, but this might cause other domains to miss their deadlines when the driver unblocks with a large allocation and very short deadline. Such situations are therefore treated as if a `yield` had been executed, and the sdom is given its allocation at the start of its next period (see figure 4.3).

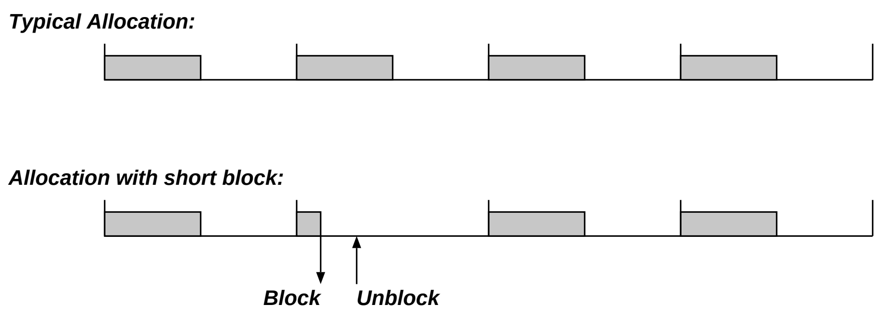


Figure 4.3: Fairer unblocking

4.4.6 Use of Extra Time

As long as Q_r is non-empty, the head is due some contracted time and should be run. If Q_r becomes empty, the scheduler has fulfilled all its commitments to sdoms until the head of Q_w becomes runnable. In this case, the scheduler can opt to run some sdom in Q_w for which x is true, i.e. one which has requested use of slack time in the system. Domains are made aware of whether they are running in this manner or in contracted time by a flag in their control block.

The current policy adopted by the scheduler is to run a random element of Q_w for a small, fixed interval or until the head of Q_w becomes runnable, whichever is sooner. Thus several sdoms can receive the processor ‘optimistically’ before Q_r becomes non-empty. The optimal policy for picking sdoms to run optimistically is a subject for further research. The current implementation allocates a very small time quantum (122 μs) to a member of Q_w picked cyclically. This works well in most cases, but there have been situations in which unfair ‘beats’ have been observed.

4.5 Client Interface

The runtime interface between a domain and the scheduler serves two purposes:

- It provides the application with information about when and why it is being scheduled, and feedback as to the domain’s progress relative to the passage of real time.
- It supports user-level multiplexing of the CPU among distinct subtasks within the domain, for example by supporting a threads package.

Time

The abstraction of time used in the system is the same throughout: the system assumes the presence of a world-readable clock giving time in nanoseconds since the machine started. In practice this will inevitably be an approximation, but it can be provided very simply and efficiently by means of a single 64-bit word in memory. There is no guarantee that this time value runs at *precisely* the same rate as time outside the machine, nor is such assurance needed. Domains synchronising to events clocked externally from the machine will need to make domain-specific long-term adjustments anyway, and the passage of true, planet-wide time falls into this category also.

Context Slots

A Nemesis domain is provided with an array of *slots*, each of which can hold a processor context. In the case of the Alpha/AXP implementation, a slot consists of 31 integer and 31 floating-point registers, plus a program counter and processor status word. Two of the slots are designated the *activation context* and *resume context* respectively; this designation can be changed at will by the domain. A domain also holds a bit of information called the *activation bit*.

Descheduling and Activation

A mechanism similar to Nemo's is used when the domain is descheduled. The context is saved into the activation context or the resume context, depending on whether the activation bit is set or not. When the domain is once again scheduled, if its activation bit is clear, the resume context is simply resumed. If the activation bit is set, it is cleared and an upcall takes place to a routine previously specified by the domain (in fact, an invocation occurs across an interface of type `DomainEntryPoint`). This entry point will typically be a user-level thread scheduler, but domains are also initially entered this way[5]. Figure 4.4 illustrates the two cases.

The upcall occurs on a dedicated stack (in the domain control block) and delivers information such as current system time, time of last deschedule, reason for upcall and context slot used at last deschedule. The state pointer for the

[5]Indeed, a new domain to be started is specified *solely* by its `DomainEntryPoint` closure.

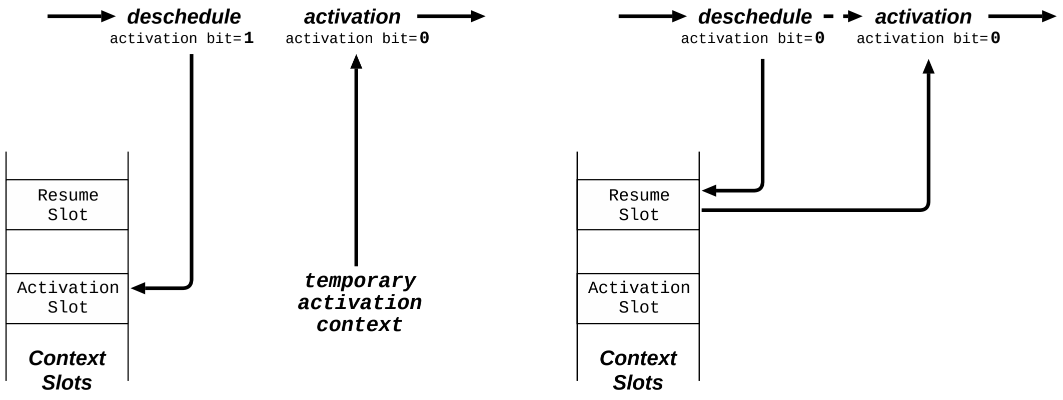


Figure 4.4: Deschedules, Activations and Resumptions

closure will contain enough information to give the domain a sufficient execution environment to schedule a thread.

A threads package will typically use one context slot for each thread and change the designated activation context according to which thread is running. If more threads than slots are required (currently 32), slots can be used as a cache for thread contexts. The activation bit can be used with appropriate exit checks to allow the thread scheduler to be non-reentrant, and therefore simpler.

Implementing threads packages over the upcall interface has proved remarkably easy. A Nemesis module implementing both preemptive and non-preemptive threads packages, providing both an interface to the event mechanism and synchronisation based on event counts and sequencers comes to about 900 lines of heavily commented C (much of which is closure boilerplate) and about 20 assembler opcodes. A further module providing the thread synchronisation primitives described in [Birrell87] comes to 202 lines of C, including comments. For comparison, the POSIX threads library for OSF/1 achieves essentially the same functionality over OSF/1 kernel threads with over 6000 lines of code, with considerably inferior performance.

4.6 Scheduling and Communication

Communication between domains is relevant to the scheduler because it may well affect the optimal choice of domain to run. However, it is important not to allow communication to influence scheduling decisions to the extent that resource guarantees are violated.

Inter-process communication generally has two aspects: firstly the presentation of information by one entity to another, and secondly a synchronisation signal by the sender to indicate that the receiver should take action. These components are orthogonal, and in Nemesis they are clearly separated. Information transfer occurs through areas of shared memory, and is discussed in chapter 5. Signalling between domains is provided by *event channels*. An event channel is a unidirectional connection capable of conveying single integer values and influencing the scheduler.

In the context of this dissertation, the key points about event channels are as follows:

- They provide a communication and synchronisation mechanism which does not rely on a server (such as the kernel).
- They impose no particular synchronisation policy on either of the domains using a channel. The effect of the arrival of an event for a domain is limited to unblocking the domain if necessary, and causing a reactivation if the domain is running.

Event channels are more primitive than traditional communication mechanisms such as semaphores, event counts and sequencers, and message passing, in that they tend to transfer less information and are less coupled to the scheduler. Such mechanisms can be built on top of event channels, and chapter 5 describes how RPC, the most common form of inter-domain communication used in Nemesis, is implemented over them.

4.6.1 Channel End-Points

Domains are provided with arrays of transmit- and receive-side event channel *end-points*, analogous to sockets. An end-point may be in one of four states, shown in figure 4.5. When the domain starts up all but two end-points are initially *free*. The domain may `Allocate` an end-point of either type, which may subsequently become *connected* as a result of either a `Connect` operation initiated by the domain or the domain replying to an incoming connection request. If a connection is closed down, the end-point enters a *dead* state, from which it can be `Freed`.

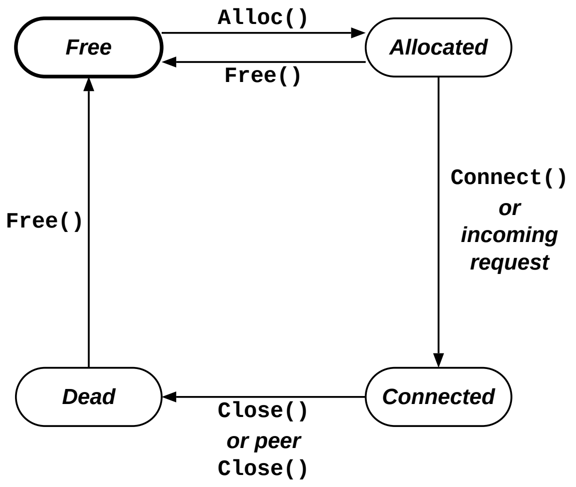


Figure 4.5: Event Channel End-Point States

As well as its current state, a receive-side end-point contains two 64-bit values called *received* and *acknowledged*. If an end-point (of either flavour) is in the *connected* state, it also contains a (*domain*, *index*) pair giving its peer. Figure 4.6 shows the user writable and user read-only portions of event end-points. It is important to note that the state of an end-point is represented by a combination of a state word and the values of the peer fields, in such a way that the two transitions which require privileged actions (`close` and `connect`) rely only on fields which cannot be written by the domain itself. In effect, the only end-point states seen by privileged code are 'connected' and 'not connected'.

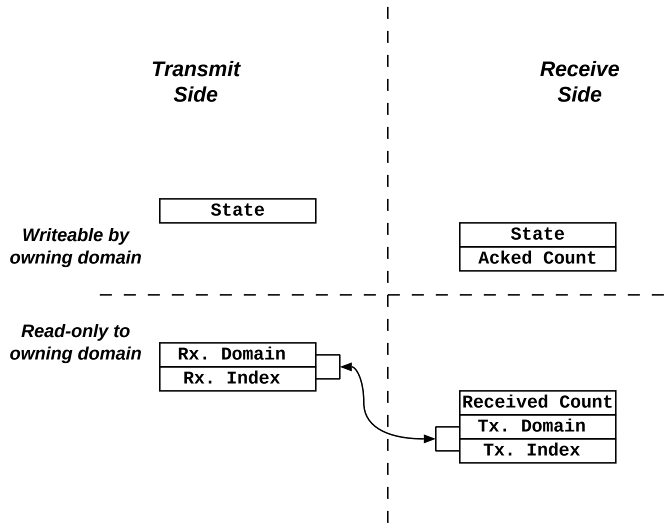


Figure 4.6: Event Channel End-Point Data Structures

4.6.2 Sending Events

An invocation to transmit an event takes as arguments the transmitter's channel end-point, and a value to add to the receiver's count. The event delivery mechanism is required to perform sanity checks on the specified event channel, increment the remote count, and if necessary signal to the scheduler that the target domain requires attention. The exact procedure is as follows:

1. Validate the event channel end-point in the transmitter. Event channels are specified by an index in an array, so this involves a range check and ensuring that the receive domain pointer is non-zero.

2. The relevant end-point in the receiving domain is located and its received count incremented by the value specified in the call.

3. Each domain has a FIFO holding receive-side event end-point indices, to aid in demultiplexing incoming events. If this FIFO is not full, the receive end-point is entered into the FIFO.

4. A flag is set in the domain to indicate that it has received one or more events.

5. If this flag was previously clear, a reschedule is requested.

Only information which is read-only to the user is examined during the call, much reducing the number of consistency checks that need to be performed at invocation time. The procedure is implemented as the `event` PALcode call and the entire code, including checks and error conditions, consists of 87 machine instructions. For a multiprocessor version the code would be slightly longer, to include spinlocks on the event structures.

The event delivery mechanism in the version of Nemesis described herein is quite conservative about reschedules: it requests a schedule whenever the target domain has not received any events since it last executed. Event delivery was expected to be more common than reschedules, so it was important to make the `event` operation very fast.

With experience, this tradeoff has proved inappropriate. Domains tend to be scheduled quite frequently and so the scheduler is entered as a result of most `event` calls, often unnecessarily. The new version of Nemesis has a PALcode image[6] which only enters the scheduler if the target is blocked or actually running. Otherwise, event processing by the scheduler is deferred until it is entered for other reasons.

An important aspect of the event mechanism is that while it acts as a hint to the scheduler, causing it to unblock or reactivate a domain as necessary, the sending of an event does not in itself force a reschedule. Thus communication between domains is decoupled from scheduling decisions to the extent that resource contracts are not affected by the transmission of events.

4.6.3 Connecting and Disconnecting End-Points

The process of event channel setup is carried out as much as possible within the two domains involved. A privileged third party is required to perform two functions:

- Acting as an exchange for routing initial connection requests.
- Filling in the event fields not writable by the domains.

This third party is called the Binder. Every domain is started up with initial event channels to and from the Binder, and these are used for connection requests. Figure 4.7 shows the interaction with the Binder when domain d_1 wishes to set up a connection to send events to d_2.

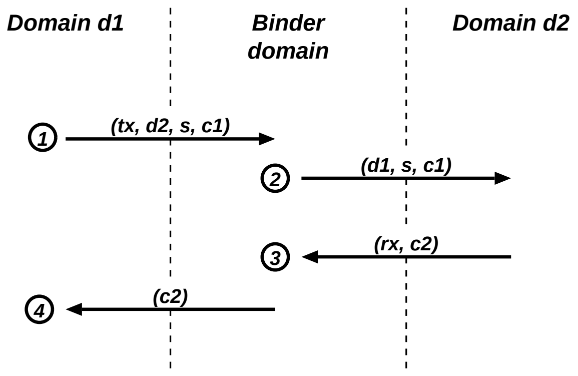


Figure 4.7: Event Channel Connection Setup

[6]Written by Paul Barham.

A similar thing happens when the initiator wishes to receive events, or set up two event channels, one each way. Connection setup takes the form of two inter-domain RPC calls, one nested within another:

1. Domain d_1 sends a request to the Binder specifying a transmit end-point it has allocated for itself (tx), the target domain d_2, a service identifier s, and a cookie c_1. s is a 64-bit identifier used to identify what domain d_1 actually wants to talk to in d_2. c_1 is another 64-bit number typically used by the Inter-domain Communication mechanism to pass shared memory addresses, but it is ignored by the Binder.
2. The Binder then calls a method of an interface in domain d_2 passing the identifier of d_1, s and c_1. This interface has previously been registered with the Binder by d_2.
3. If the connection request is accepted, d_2 returns to the Binder a receive end-point that it has allocated (rx) together with c_2, a cookie of its own.
4. The Binder fills in the fields of the two end-points tx and rx, thus creating the channel.
5. The first call returns with the cookie c_2. The connection has now been made.

This is the only interaction required with the Binder. Close down of event channels is performed by the `close` PALcode routine, which zeroes the peer information in both end-points. The representation of end-point states within the domain is chosen so that this represents the *dead* state. If the caller is on the transmit side, the PALcode also sends an event of value zero to the receiving domain. This has the effect of alerting the domain to the demise of the channel. A channel closed by a receiver will cause an exception to the sender next time it tries to `send` on it.

4.7 Device Handling and Interrupts

The Nemesis scheduler as described provides efficient scheduling of domains with clear allocation of CPU time according to QoS specifications. Interrupts present a problem to the scheduler, however, because CPU cycles used in the execution of an interrupt service routine are difficult to account to a particular domain.

Interrupts cause other problems for a system which attempts to give guarantees on available time. In most existing operating systems, the arrival of an interrupt usually causes a task to be scheduled immediately to handle the interrupt, preempting whatever is running. The scheduler itself is usually not involved in this decision: the new task runs as an interrupt service routine.

The interrupt service routine (ISR) for a high interrupt rate device can therefore hog the processor for long periods, since the scheduler itself hardly gets a chance to run, let alone another process. [Dixon92] describes a situation where careful prioritising of interrupts led to high throughput, but with most interrupts disabled for a high proportion of the time.

Sensible design of hardware interfaces can alleviate this problem, but devices designed with this behaviour in mind are still rare, and moreover they do not address the fundamental problem: scheduling decisions are being made by the interrupting device and interrupt dispatching code, and not by the system scheduler, effectively bypassing the policing mechanism.

The solution adopted in Nemesis decouples the interrupt itself from the domain which is handling the interrupt source. Device driver domains register an interrupt handler[7] with the system, which is called by the interrupt dispatch PALcode with a minimum of registers saved. This ISR typically clears the condition, disables the source of the interrupt, and sends an event to the domain responsible. This sequence is sufficiently short that it can be ignored from an accounting point of view. For example, the ISR for the LANCE Ethernet driver on the Sandpiper[8] is 12 instructions long.

Since any domain can be running when the ISR is executed, a PALcode trap called `kevent` is used by the ISR to send the event. This call is similar to `event` but bypasses all checks and allows the caller to specify a receive end-point directly. It can only be executed from kernel mode.

4.7.1 Effect of Interrupt Load

At low load, the unblocking latency hint l can be used by the device driver domain to respond to interrupts with low latency if necessary, while interrupts which do not need to be serviced quickly (such as those from serial lines) do not disturb the scheduling of other tasks.

[7]Actually a closure of type `KernelEntryPoint`.

[8]Written by Paul Barham.

At a high interrupt rate from a given device, at most one processor interrupt is taken per activation of the driver domain, and the scheduling mechanism prevents the driver from hogging the CPU. As the activity in the device approaches the maximum that the driver has time to process with its CPU allocation, the driver rarely has time to block before the next action in the device that would cause an interrupt, and so converges to a situation where the driver polls the device whenever it has the CPU.

When device activity is more than the driver can process, overload occurs. Device activity which would normally cause interrupts is ignored by the system since the driver cannot keep up with the device. This is deemed to be more desirable than having the device schedule the processor: if the driver has all the CPU cycles, the 'clients' of the device wouldn't be able to do anything with the data anyway. If they could, then the driver is not being given enough processor time by the domain manager. The system can detect such a condition over a longer period of time and reallocate processor bandwidth in the system to adapt to conditions.

4.8 Comparison with Related work

The client interface to the Nemesis scheduler is a development of that used in the Nemo system; [Hyden94] gives an extensive survey of related schemes as well as describing Nemo in detail. This section presents a selection of scheduling systems not already mentioned which are of relevance to Nemesis.

4.8.1 Scheduling

The Psyche system [Scott90] aims to support a number of different scheduling policies and process models over the same hardware. Psyche uses an upcall mechanism to notify a user-level scheduler that it had received service by a *virtual processor*, which is analogous to a kernel thread in a traditional system. Upcalls are also used to support user-level threads packages in systems such as Scheduler Activations [Anderson92], but these systems rely on kernel threads in some form, adding to the number of register context switches needed and the amount of state required in the kernel. The underlying kernel-level scheduler in all cases does not use fine-grained allocation of time, and time is not presented explicitly to the user-level schedulers as a useful aid to scheduling. Psyche also provides a re-activation prior to de-scheduling as a hint to the user process that it is about to lose the processor. Such a mechanism was thought unnecessary in Nemesis, where applications are expected to adapt their behaviour over longer time scales than single time slices.

Various algorithms have been produced to deliver a share of the CPU rather than priorities. A recent example, Lottery Scheduling [Waldspurger94] employs an interesting scheduler which uses a random probabilistic allocation of CPU time. The abstraction used is that of *tickets* in a lottery: the more tickets a process has, the more likely it is to 'win' the next scheduling slot, and so the greater the share of the CPU it receives in the long run. The lottery model copes nicely with nested allocations of CPU time, but does not give a notion of the passage of real time, or allow domains to specify a particular granularity of allocation. Like the Fawn system, allocation is based on a fixed ticker (100ms in this case), allowing no precision in scheduling below this level. Furthermore, extra time in the system is implicitly handed out equally to all domains: there is no room for a tailor-able allocation policy.

Processor Capacity Reserves [Mercer93] is a scheme similar to Nemesis in specifying the service required by an application in terms of processor bandwidth. However, the problem of shared servers is addressed by having clients transfer resource reservations to the server, with the server charging time to the client. This can create problems of QoS crosstalk between domains and also fails to address the problem of blocking synchronisation between domains. Nor is the allocation of slack time addressed: all processes are scheduled as best-effort during slack time in the system.

4.8.2 Communication and Interrupts

The method of interrupt handling in Nemesis somewhat resembles 'structured interrupts' [Hills93], though the presence of a QoS -based scheduler gives considerably more incentive to use them.

At least one recent ATM interface [Smith93] has resorted to polling on a periodic timer interrupt to alleviate the problem of high interrupt rates. Nemesis naturally converges to a polling system at high loads but retains the benefits of interrupt-driven operation when system load is low.

In terms of inter-process communication, few systems have separated signalling, synchronisation and data transfer to the extent that Nemesis has. Most kernels (such as Mach [Accetta86] or Chorus [Rozier90]) provide blocking synchronisation primitives for kernel threads and message passing for inter-domain communication. Nemesis has no kernel threads and relies on user-space functionality for data transfer. This difference is addressed in more detail in chapter 5.

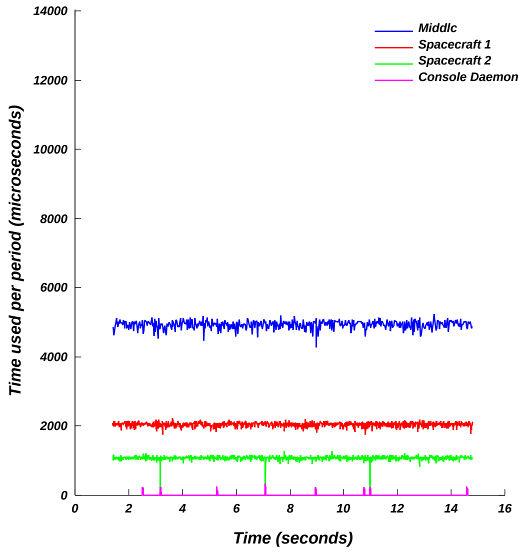
4.9 Evaluation

To examine the behaviour of the scheduler under load, a set of domains were run over the Sandpiper version of the kernel, with a scheduler which used the processor cycle counter to record the length of time each domain ran between reschedules. The mix of domains was chosen to represent device drivers, applications using blocking communication, and 'free-running' domains able to make use of all available CPU cycles. The full set was as follows:

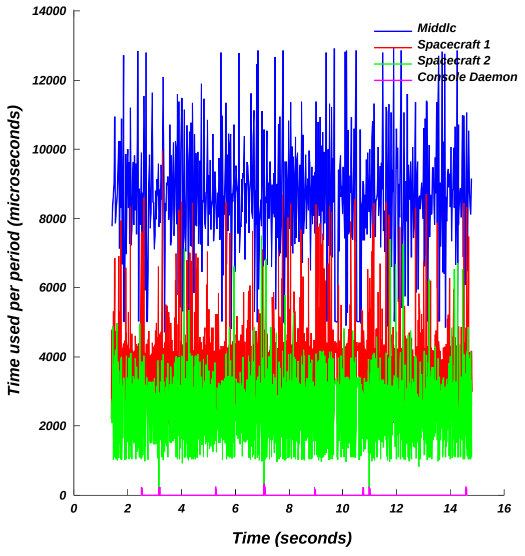
- The `middlc` compiler, in a loop compiling a set of interface definitions. This domain did not perform any communication, and simply ran 'flat out'.
- An application to draw an animation of a spacecraft on the screen, and print logging information to the console driver using blocking local RPC. Two instantiations of this application were employed.
- A console daemon, consisting of an interrupt driven UART driver and an RPC service used by the spacecraft domains.
- An Ethernet monitor, which responded to each packet received from the Ethernet interface in promiscuous mode and generated a graph of network load on the screen.

For an initial run, the QoS parameters used in table 4.1 were used. The results are shown in figure 4.8.

In these graphs each data point represents the CPU time used by a domain since the previous point. Points are plotted when a domain passes a deadline (and so receives a new allocation of CPU time), or when it blocks or unblocks. This representation is chosen because it makes clear exactly what the scheduler is doing at each point, information that would be obscured by integrating the time

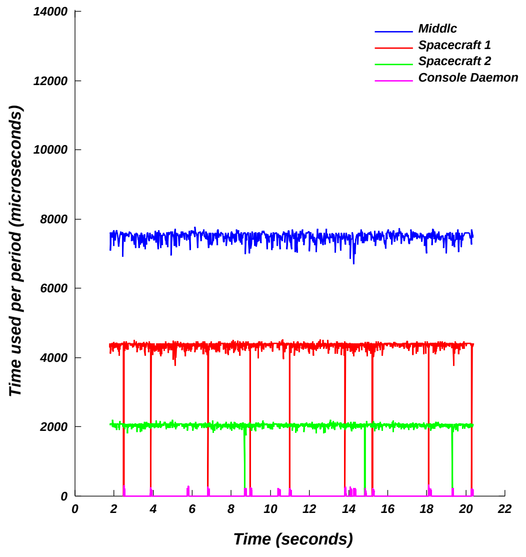


(a) Contracted time

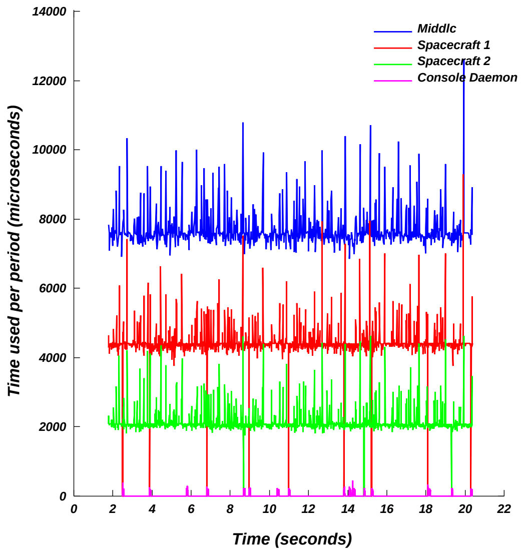


(b) Total time

Figure 4.8: CPU allocation under 70% load.



(a) Contracted time



(b) Total time

Figure 4.9: CPU allocation under 100% load.

	QoS parameters (μs)			
	p	s	l	CPU share
Console daemon	14000	1400	14000	10%
Ethernet monitor	2000	200	200	10%
Spacecraft 1	10000	100	10000	10%
Spacecraft 2	10000	2000	10000	20%
MIDDL compiler	25000	5000	25000	20%
			Total:	70%

Table 4.1: QoS parameters used in figure 4.8

	QoS parameters (μs)			
	p	s	l	CPU share
Console daemon	14000	350	14000	2.5%
Ethernet monitor	4000	160	160	4%
Spacecraft 1	10000	2000	10000	20%
Spacecraft 2	10000	4350	10000	43.5%
MIDDL compiler	25000	7500	25000	30%
			Total:	100%

Table 4.2: QoS parameters used in figure 4.9

used over one or more allocation periods. Short vertical troughs in the traces for spacecraft domains correspond to blocks.

Allocation of guaranteed CPU time is clearly fairly accurate. Accounting and scheduling within Nemesis is performed at the granularity of the system clock, which in this case is 122μs. Thus a small amount of jitter is introduced into each domain's allocation on every reschedule. This is the reason why `middlc` has a higher jitter than the other domains: since its period is longer, it will experience more reschedules per period.

Figure 4.8 shows that the pseudo-random algorithm for allocating extra time leaves something to be desired: the allocation to a given domain is fair over a long time scale, but can vary wildly from period to period.

4.9.1 Behaviour Under Heavy Load

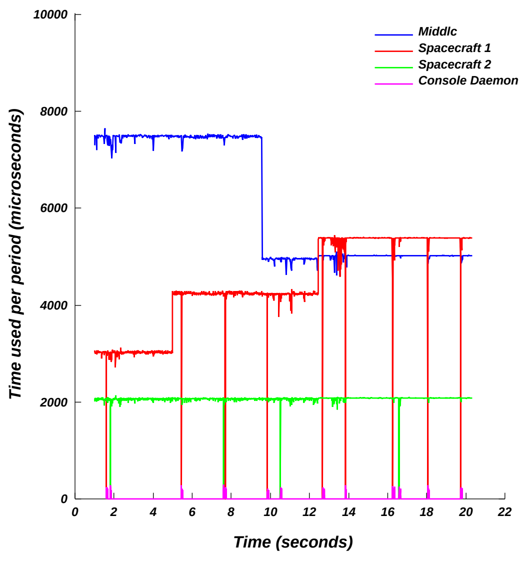
The next run used the QoS parameters shown in table 4.2.

In theory all the processor time in the system was committed at this point, although since the console driver is blocked for much of the time there is still a small amount of leeway. Figure 4.9 shows the result: contracted time allocation is still very stable, even when the amount of slack time available to domains is small.

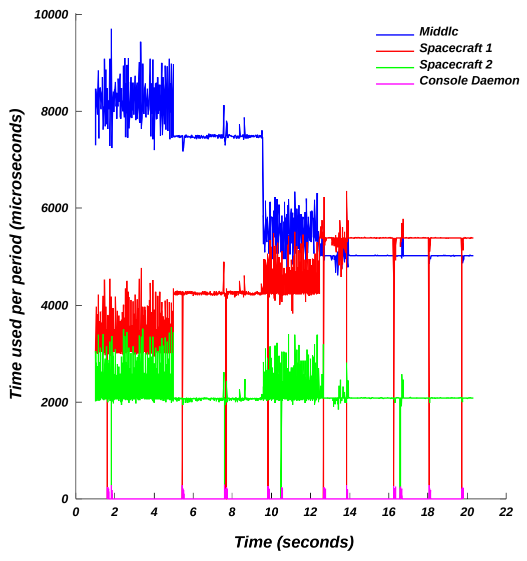
4.9.2 Dynamic CPU reallocation

An important feature of Nemesis is its ability to reallocate resources dynamically, under user or program control. Figure 4.10 shows this in action. The experiment used the same QoS parameters in table 4.2, except the slice length for some domains was altered at roughly 5, 10 and 12 seconds into the experiment. The values of s used are shown in table 4.3.

This reallocation was achieved by simply altering the values of the s field in the domain control blocks, and shows how the scheduler can cope immediately with the new distribution of resources, even when the system load is pushed up to 100%. In practice this reallocation would be subject to an admission control procedure to ensure that the processor was never over-committed.



(a) Contracted time



(b) Total time

Figure 4.10: Dynamic CPU reallocation.

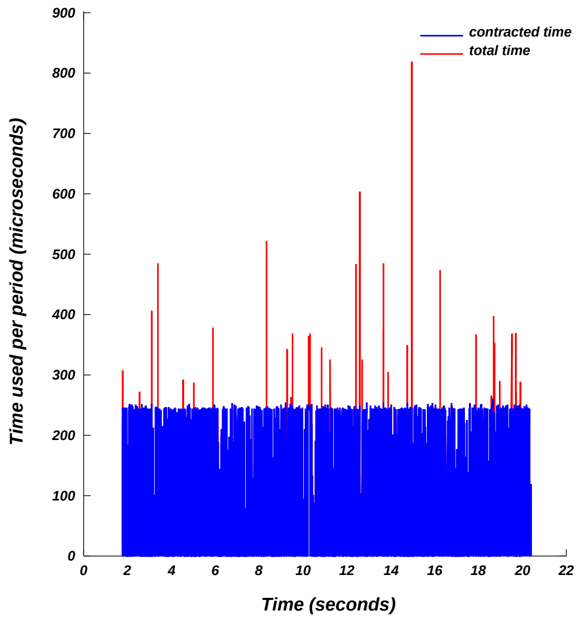


Figure 4.11: CPU allocation to Ethernet driver

	Value of s (μs)			
	0-5 sec.	5-10 sec.	10-12 sec.	12-22 sec.
Spacecraft 2	3000	4250	4250	5350
MIDDL compiler	7500	7500	5000	5000
	Total CPU share committed			
	86.5%	99%	89%	100%

Table 4.3: Changes in CPU allocation in figure 4.10

4.9.3 Effect of Interrupt Load

Figure 4.11 illustrates the effective decoupling of domain scheduling from interrupt handling in Nemesis. It shows a trace of time allocated to the Ethernet load monitor during the experiment shown in figure 4.9. The load monitor uses the LANCE Ethernet interface on the Sandpiper in promiscuous mode to draw a bar on the screen corresponding to current Ethernet usage. The reception of an Ethernet packet causes an interrupt, which in turn causes an event to be sent to the domain.

The first point to note is that most of the time, a reschedule is occurring upon the receipt of every packet on the network. Despite this, and the relatively low CPU allocation given to the domain (4% of the total available), almost all Ethernet packets are processed.

The second and more important point is that this activity is not interfering with the rest of the system. The load monitor is running with slightly less CPU time guaranteed to it than it really requires: during the 18 seconds of the run about 12 buffer overruns occurred[9]. Since the system as a whole is heavily committed, the time available to the domain has been limited and the interrupt rate on the device is prevented from impacting on the performance of the other domains in the system.

[9]The monitor is capable of processing every packet if it is given about 5% of the processor.

4.9.4 Scheduler Overhead

To obtain an idea of how much of the processor's time was spent scheduling, the scheduler was instrumented using the processor cycle counter on the 21064 processor. The system was run with a set of 7 domains: 6 application domains which drew animations on the screen and performed RPCs to write logging information to a 7th domain implementing an interrupt-driven console driver. Reschedules were therefore being caused by time-slicing interrupts, events sent by domains, block requests from domains, and events generated by the UART interrupt service routines. This provided a plausible approximation to the system under real load.

Each of the application domains was receiving a guaranteed 400μs of processor time every 3600μs, and the console driver was receiving 1400μs every 14000μs. The system was therefore about 77% committed. An 8th domain, the Domain Manager, was blocked throughout the run. 30000 passes through the scheduler were observed; this took about 5 seconds. The cache artifacts reported in chapter 3 caused some problems, but a coherent picture emerged. Figure 4.12 shows the distribution of times taken to calculate a new schedule.

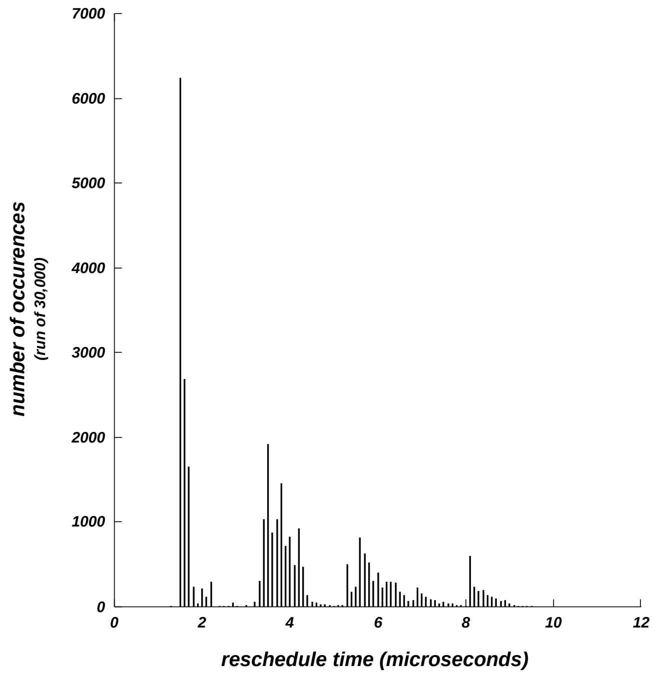


Figure 4.12: Distribution of reschedule times

The scheduler can execute extremely fast: the lowest time observed so far is 1.37μs not including context switch overhead, and most schedules take under 4μs. The four peaks in the graph correspond to tasks the scheduler may or may not need to perform during the course of a reschedule, namely:

1. The case when no priority queue manipulation is required; it is this case which is most likely to be produced by the event delivery optimisations mentioned in section 4.6.2.
2. Moving one sdom between Q_r and Q_w.
3. Handling the arrival of events, including blocking and unblocking domains. Also moving more than one domain between queues.
4. Combinations of 2 and 3.

The scheduler is not a well-tuned piece of software at present: is was written with comprehensibility and ease of experimentation in mind more than raw performance. Despite this, it represents on average less than 2% overhead at the fastest reschedule rate possible on a Sandpiper (about once every 122μs). The cost of unblocking domains could be significantly reduced by re-implementing the algorithm which finds the domain to unblock: it is currently a linear scan of the queue[10].

4.9.5 Scheduler Scalability

The execution time of the scheduler depends on the number of domains being scheduled. Aside from the unblocking search mentioned above, the dependency is entirely due to the queue manipulation functions. The queues are implemented as heaps, so one would expect the relation between reschedule time and number of domains to be logarithmic.

Figure 4.13 shows the result of performing the experiment in section 4.9.4 with varying numbers of domains, altering the allocation period for the application domains to keep their total processor bandwidth at 67%, with slices of 400μs. The results tend to support the hypothesis that scheduling overhead is logarithmic with the number of domains. Furthermore the incremental cost of extra domains

[10]At time of writing, a new version of Nemesis incorporates modifications to the scheduler by David Evers to remove this bottleneck.

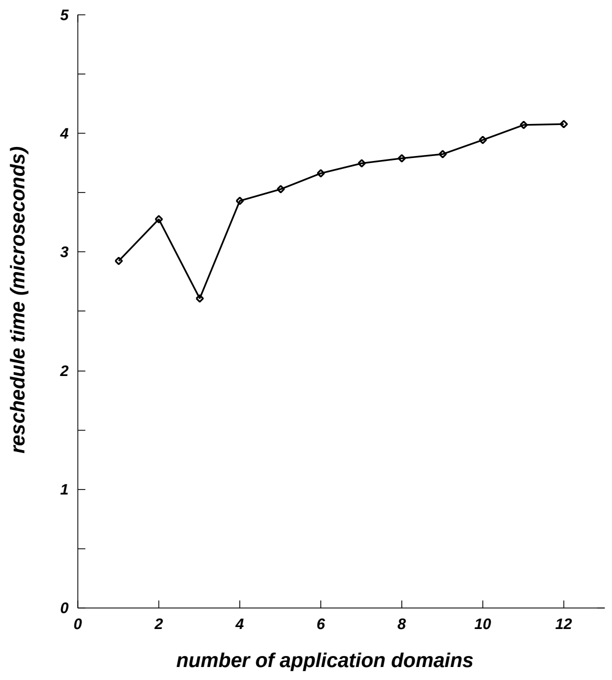


Figure 4.13: Cost of reschedule with increasing number of domains

above about 10 is very small (a few nanoseconds), resulting in an highly scalable as well as efficient scheduler.

The anomaly in the case of 3 domains is believed to be an artifact of the heap manipulation procedures.

4.9.6 Event delivery

The time taken to deliver an event to a domain has been measured at roughly 2.3μs. This does not include the scheduler processing required. In a heavily loaded system event processing for several domains will be performed in a single pass through the scheduler, resulting in a reduction in this overhead over all domains.

Section 5.5.1 presents the results of timing RPC calls built over the event mechanism.

4.10 Summary

This chapter has discussed scheduling in operating systems, and in particular the requirements of continuous media applications, which frequently occupy a region distinct from hard and soft real-time applications but are still time-sensitive. The form and meaning of a QoS specification with regard to CPU time has been discussed.

Various approaches to scheduling in multi-service operating systems have been described, culminating in the scheduler used in Nemesis. The latter separates the general QoS-based scheduling problem into two components: that of delivering guaranteed CPU time and that of allocating slack time in the system. The first problem is mapped onto one which is solved using Earliest Deadline First techniques whilst still enabling strict policing of applications. The present solution to the second problem is simple and reasonably effective; refining it is a subject for future research.

The scheduler is fast, scalable, and permits domains to be scheduled efficiently subject to QoS contracts even when nearly all processor time has been contracted out to domains. Furthermore, allocation of processor share to domains can be easily varied dynamically without requiring any complex scheduling calculations.

The client interface to the scheduler is based on the idea of activations. It makes applications aware of their CPU allocation and provides a natural basis for the implementation of application-specific threads packages, of which several have been produced.

The only kernel-provided mechanism for inter-domain communication is the event channel, which transmits a single value along a unidirectional channel. Event arrival unblocks a blocked domain and causes reactivation in a running domain.

Interrupts are integrated at a low level by means of simple first-level interrupt handlers with most processing occurring within application domains. The activation interface allows this to be achieved with a small interrupt latency at low machine load, and the policing mechanism ensures that high interrupt rates do not starve any processes under high load.

Chapter 5

Inter-Domain Communication

The basic method for communication between Nemesis domains is the event channel mechanism described in chapter 4. However, it is clearly desirable to provide facilities for communication at a higher level of abstraction. These communication facilities come under the general heading of Inter-Domain Communication (IDC).

Nemesis provides a framework for building IDC mechanisms over events. Use is made of the run-time type system to allow an arbitrary interface to be made available for use by other domains. The basic paradigm is dictated by the MIDDL interface definition language: Remote Procedure Call (RPC) with the addition of 'announcement' operations, which allow use of message passing semantics.

This chapter discusses the nature of IDC in general, and then describes in detail the Nemesis approach to inter-domain binding and invocation, including optimisations which make use of the single address space and the system's notion of real time to reduce synchronisation overhead and the need for protection domain switches.

5.1 Goals

Most operating systems provide a basic IDC mechanism based on passing messages between domains, or using RPC [Birrell84]. The RPC paradigm was chosen as the default mechanism for Nemesis, because it fits in well with the use of interfaces, and does not preclude the use of other mechanisms.

The use of an RPC paradigm for communication in no way implies the traditional RPC implementation techniques (marshalling into buffer, transmission of buffer, unmarshalling and dispatching, etc.). This should not be surprising, since RPC is itself an attempt to make communication look like local procedure call. There are cases where the RPC programming *model* is appropriate, but the underlying *implementation* can be radically different. In particular, with the rich sharing of data and text afforded by a single address space, a number of highly efficient implementation options are available.

Furthermore, there are situations where RPC is clearly not the ideal paradigm: for example, bulk data transfer or continuous media streams are often best handled using an out-of-band RPC interface only for control. [Nicolaou90] describes early work integrating an RPC system for control with a typed stream-based communication mechanism for transfer and synchronisation of continuous media data.

The aim in building RPC-based IDC in Nemesis was not to constrain all communication to look like traditionally-implemented RPC, but rather to:

1. provide a convenient default communication mechanism,
2. allow a variety of transport mechanisms to be provided behind the same RPC interface, and
3. allow other communication paradigms to be integrated with the IDC mechanism and coexist with (and employ) RPC-like systems.

5.2 Background

The design of an RPC system has to address two groups of problems: the creation and destruction of *bindings*, and the *communication* of information across a binding.

Operating systems research to date has tended to focus on optimising the performance of the communication systems used for RPCs, with relatively little attention given to the process of binding to interfaces. By contrast, the field of distributed processing has sophisticated and well-established notions of interfaces and binding.

5.2.1 Binding

In order to invoke operations on a remote interface, a client requires a local interface encapsulating the engineering needed for the remote invocation. This is sometimes called an *invocation reference*. In the context of IDC a *binding* is an association of an invocation reference with an interface instance. In Nemesis IDC an invocation reference is a closure pointer of the same type as the remote interface, in other words, it is a *surrogate* for the remote interface.

An *interface reference* is some object containing the information needed to establish a binding to a given interface. To invoke operations on a remote interface, a client has to have acquired an interface reference for the interface. It must first establish a binding to the interface (so acquiring an invocation reference), and then use the invocation reference to call operations in the remote interface.

An interface reference can be acquired in a variety of ways, but it typically arrives in a domain as a result of a previous invocation. *Name servers* or *traders* provide services by which clients can request a service by specifying its properties. An interface reference is matched to the service request and then returned to the client. Such services can be embedded in the communication mechanism, but if interface references are first-class data types (as they are in Nemesis) traders are simply conventional services implemented entirely over the IDC mechanism. This leads to a programming model where it is natural to create interface references dynamically and pass them around at will.

In the local case (described in chapter 3), an interface reference is simply a pointer to the interface closure, and binding is the trivial operation of reading the pointer. In the case where communication has to occur across protection domain boundaries (or across a network), the interface reference has to include rather more information and the binding process is correspondingly more complex.

Strictly speaking, there is a subtle distinction between creating a binding (simply an association) and *establishing* it (allocating the resources necessary to make an invocation). This distinction is often ignored, and the term *interface reference* used to refer to an invocation reference. This leads to systems where the true interface reference itself is hidden from the client, which only sees invocation references.

Implicit binding

An implicit binding mechanism creates the engineering state associated with a binding in a manner invisible to the client. An invocation which is declared to return an interface reference actually returns a closure for a valid surrogate for the interface. Creation of the surrogate can be performed at any stage between the arrival of the interface reference in an application domain and an attempt by the application to invoke an operation on the interface reference. Indeed, bindings can time out and then be re-established on demand.

The key feature of the implicit binding paradigm is that information about the binding itself is hidden from the client, who is presented with a surrogate interface indistinguishable from the ‘real thing’.

Implicit binding is the approach adopted by many distributed object systems, for example Modula-3 Network Objects [Birrell93] and CORBA [Obj91]. It is intuitive and easy to use from the point of view of a client programmer. For many applications, it provides all the functionality required, provided that a garbage collector is available to destroy the binding when it is no longer in use.

The Spring operating system [Hamilton93a] is one of the few operating systems with a clear idea of binding. Binding in Spring is implicit. It uses the concept of *doors*, which correspond to exported interfaces. A client requires a valid local *door identifier* to invoke an operation on a door; an identifier is bound to a door by the kernel when the door interface reference arrives in the domain. Binding is hidden not only from the client but also from the server, which is generally unaware of the number of clients currently bound to it. Spring allows a server to specify one of a number of mechanisms for communication when the service is first offered for export. These services are called *subcontracts* [Hamilton93b]. However, there is no way for the mechanism to be tailored to a particular type of service.

Explicit binding

Traditional RPC systems have tended to require clients to perform an explicit bind step due to the difficulty of implementing generic implicit binding. The advent of object-based systems has recently made the implicit approach prominent for the advantages mentioned above.

However, implicit binding is inadequate in some circumstances, due to the hidden nature of the binding mechanism. It assumes a single, 'best effort' level of service, and precludes any explicit control over the duration of the binding. Implicit binding can therefore be ill-suited to the needs of time-critical applications.

Instead, bindings can be established explicitly by the client when needed. If binding is explicit, an operation which returns an interface reference does not create a surrogate as part of the unmarshalling process, but instead provides a local interface which can be later used to create a binding. This interface can allow the duration and qualities of the binding to be precisely controlled at bind time with no loss in type safety or efficiency. The price of this level of control is extra application complexity, which arises both from the need to parametrise the binding and from the loss of transparency: acquiring an interface reference from a locally-implemented interface can now be different from acquiring one from a surrogate.

Some recent research, notably ANSA Phase III [Otway94], is developing sophisticated binding models which are type-safe and encompass both implicit and explicit binding to support QoS specification at the level of RPC invocations. Much terminology used in this chapter is borrowed from the ANSA Binding Model, and Nemesis IDC binding shares many concepts with ANSA.

Finally, note that the behaviour of the server is independent of whether client binding is performed explicitly or implicitly, since the same communication mechanism is likely to be used in both cases for setting up the binding.

5.2.2 Communication

The communication aspect of IDC (how invocations occur across a binding) is independent of the binding model used. Ideally, an IDC framework should be able to accommodate numerous differing methods of data *transport* within the computational model. Current operating systems which support RPC as a local communications mechanism tend to use one of two approaches to the problem of carrying a procedure invocation across domain boundaries: message passing and thread tunnelling.

Message Passing

One approach is to use the same mechanism as that used for remote (cross-network) RPC calls: a buffer is allocated, arguments are marshalled into it, and the buffer is 'transmitted' to the server domain via the local version of the network communication mechanism. The client then blocks waiting for the results to come back. In the server a thread is waiting on the communication channel, and this thread unblocks, processes the call, marshals results into a buffer and sends it back. The client thread is woken up, unmarshals the results, and continues.

Much recent research has concentrated on reducing the latency for this kind of invocation, mostly by raising the level at which the optimisations due to the local case are performed. Since the local case can be detected when the binding is established, an entirely different marshalling and transmission mechanism can be used, and hidden behind the surrogate interface. Frequently one buffer is used in each direction, mapped into both domains with appropriate protection rights. This means the buffer itself does not need to be copied, values are simply written in on one side and read out on the other.

The ultimate latency bottleneck in message-passing comes down to the time taken to copy the arguments into a buffer and perform a context switch, thereby invoking the system scheduler.

Thread tunnelling

This bottleneck can often be eliminated by leaving the arguments where they are when the surrogate is called (i.e., in registers), and 'tunnelling' the thread between protection domains. Care must be taken with regard to protection of stack frames, etc, but very low latency can be achieved. The scheduler itself can be bypassed, so that the call simply executes a protection domain switch.

Lightweight RPC [Bershad90] replaced the Taos RPC mechanism for the case of same machine invocations with a sophisticated thread-tunnelling mechanism. Each RPC binding state includes a set of shared regions of memory maintained by the kernel called *A-stacks*, which hold a partial stack frame for a call, and a set of *linkage records*. Each thread has a chain of linkage records, which hold return addresses in domains and are used to patch the A-stack during call return for security. LRPC uses a feature of the Modula-2+ calling conventions to hold the stack frame for the call in the A-stack, while executing in the server on a local stack called the E-stack, which must be allocated by the kernel when the call is made. Various caching techniques are used to reduce the overhead of this operation so that it is insignificant in the common case. To address the problem of threads being ‘captured’ by a server domain, a facility is provided for a client to create a new thread which appears to have returned from a given call.

For RPC calls which pass arguments too large to be held in registers, or which require kernel validation, other mechanisms must be used. For example, Spring falls back on message passing for large parameters, and the nucleus must perform access control and binding when doors are passed between domains.

A slightly different approach is used by the Opal system [Chase93]. Opal uses binding identifiers similar to Spring doors, conveniently called portals. Calling through a portal is a kernel trap which causes execution at a fixed address in the server protection domain. Unlike doors, however, portals are named by system-wide identifiers which can be freely passed between domains, and so anyone can try to call through a portal. Security is implemented through check fields validated at call time, and password capabilities. An RPC system based on surrogates is built on top of this mechanism. This approach reduces kernel overhead over Spring at the cost of call-time performance.

In all cases, the performance advantage of thread tunnelling comes at a price: since the thread has left the client domain, it has the same effect as having blocked as far as the client is concerned. All threads must now be scheduled by the kernel (since they cross protection domain boundaries), thus applications can no longer reliably internally multiplex the CPU. Accounting information must be tied to kernel threads, leading to the crosstalk discussed in chapter 2.

5.2.3 Discussion

RPC invocations have at least three aspects:

1. The transfer of information from sender to receiver, whether client or server
2. Signalling the transfer of information
3. The transfer of control from the sender to the receiver

The thread tunnelling model achieves very low latency by combining all components into one operation: the transfer of the thread from client to server, using the kernel to simulate the protected procedure calls implemented in hardware on, for example, Multics [Organick72] and some capability systems such as the CAP [Wilkes79]. These systems assumed a single, global resource allocation policy, so no special mechanism was required for communication between domains.

With care, a message passing system using shared memory regions mapped pairwise between communicating protection domains can provide high throughput by amortising the cost of context switches over several invocations, in other words by having many RPC invocations from a domain outstanding. This separation of information transfer from control transfer is especially beneficial in a shared memory multiprocessor, as described in [Bershad91].

Of equal importance to Nemesis is that the coupling of data transfer and control transfer in tunnelling systems can result in considerable crosstalk between applications, and can seriously impede application-specific scheduling.

5.2.4 Design Principles

A good RPC system provides high throughput and low latency, and should be as easy as possible for a programmer to use without compromising flexibility or expressiveness. A number of design principles can be identified:

1. The invocation path should be fast. In cases where it is acceptable to sacrifice security and other guarantees in the interest of performance, this should be possible without introducing undue complexity into the API.
2. Since much code is shared, and re-compilation of code is much less frequent than the creation and destruction of services, as much optimisation and checking as possible should be performed at compile time.
3. Since creation and destruction of connections to services is less frequent than invocations, as much runtime optimisation and checking as possible should be performed well before any calls between from a particular client to a server are actually made.
4. To support an object-based (or interface-based) programming paradigm, it should be easy to create and destroy services dynamically, and pass references to them freely around the system.

5. The common case in the system should be very simple to use, without compromising the flexibility needed to handle unusual cases.

Communication in Nemesis has been designed with these goals in mind.

5.3 Binding in Nemesis

This section describes how IDC bindings are created between domains in Nemesis. Although designed for a single machine and address space, the architecture has many similarities with ideas developed in the field of distributed naming and binding, in particular ANSA Phase III. As a local operating system IDC mechanism, it has a number of novel features:

- All interfaces are strongly typed. Most type checking is done at compile-time. Run-time type checking is highly efficient.
- Multiple classes of communication mechanism are supported. The particular implementation of IDC transport is chosen by the server domain when a service is exported.
- Optimisations can be integrated transparently into the system. These include the elimination of context switches for invocations which do not alter server state, and relaxation of synchronisation conditions and security in certain circumstances.
- Both implicit and explicit binding are supported. The binding model is determined by the client independently of the class of IDC communication employed. Precise control over the duration and qualities of a binding is possible.

5.3.1 Infrastructure

The system-wide infrastructure for IDC in Nemesis consists of the Binder and event delivery mechanism (discussed in chapter 4) and various modules, each of which implements a class of IDC transport. In addition, there are stub modules which encapsulate code specific to the remote interface type, and a number of support objects which are instantiated by domains wishing to communicate. These include object tables and gatekeepers.

Object Tables

A domain has an *object table* which can map an interface reference either to a previously created surrogate or to a 'real' interface closure, depending on whether the service is local or not. It is used in a similar way to the Modula-3 object table [Evers93], except that Nemesis does not implement garbage collection.

Gatekeepers

Most classes of IDC communication mechanism use shared memory buffers for communication of arguments and results. Thus when establishing a binding, both client and server need to acquire regions of memory which are mapped read-only to the other domain. A per-domain service called a *gatekeeper* maps protection domains to memory heaps. The precise mapping is domain-dependent: for example, a domain may use a single, globally readable heap for all communication when information leakage is not a concern, or can instantiate new heaps on a per-protection domain basis. This allows a domain to trade off security for efficiency of memory usage.

Stub modules

A *stub module* implements all the type-specific IDC code for an interface type. This includes a number of components:

- An operation table for the client surrogate. Each operation marshals the relevant arguments, signals that the data is ready, and blocks the thread waiting for a reply. When this arrives, it unmarshals the results, and if necessary dispatches any exceptions which have been raised.
- The dispatch procedure for the server side. This is called by the server thread corresponding to a binding when an invocation arrives. It unmarshals arguments for the appropriate operation, invokes the operation in the true interface, catches any exceptions and marshals them or the results into the buffer.
- A *stub record*. This includes information on the type of interface this is a stub module for, together with information useful at bind time such as the size of buffers needed for the binding.

A stub module such as this can be generated automatically by the MIDDL compiler. Other stubs, implementing caching, buffering or the more specialised optimisations mentioned below can be built from a mixture of generated and hand-written code. Stub modules are installed by the system loader in a naming context which allows them to be located based on the type they support.

5.3.2 Creating an offer

To export an IDC service, a domain uses an instance of an `IDCTransport` closure type. Typically there will be several available, offering different classes of IDC transport. The class of transport used determines the underlying communication implementation to be employed. The domain also uses an object table (of type `ObjectTbl`), which provides two closures. The first is used by the domain to register and lookup interface references and offers, and the second is invoked by the system Binder when a client wishes to bind to an interface that the domain has exported.

The situation in figure 5.1 shows a situation in which a domain has decided to offer for export an interface of type `FileSystem`. The domain invokes the `Offer` operation of the `IDCTransport`, passing the `FileSystem` closure as a `Type.Any`. The `ObjectTbl` is accessible through the pervasive record.

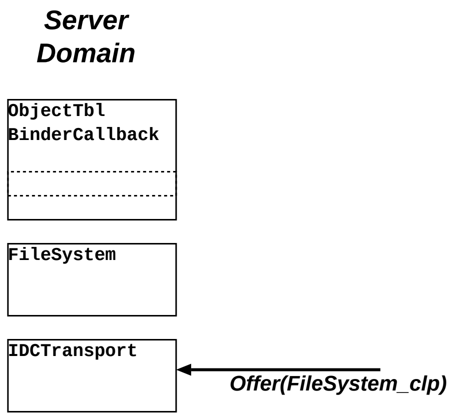


Figure 5.1: Creating a service offer

If all goes well, the result should be as shown in figure 5.2. The `IDCTransport` has created two new closures of types `IDCService` and `IDCOffer`. The `IDCService` closure allows the domain to control the operation of the offer, for example it allows the offer to be withdrawn and the state associated with it destroyed. It also provides an operation used internally for binding to the service. Not shown is the stub module, located at this time by looking up the name 'FileSystem' in an appropriate naming context maintained by the loader.

The IDCOffer itself is what is handed out to prospective clients, for example it can be stored as a Type.Any in some naming context. In other words, it functions as an interface reference. It is effectively a module (its operations can be invoked locally in any domain), and it has a Bind operation which attempts to connect to the real service. The type of service referred to by an offer is available as a type code. The closure (code and state) is assumed to be available read-only to any domain which might wish to bind to it.

A final consequence of offering a service is that the offer is registered in the server domain's object table. This is so that if the server domain receives the offer from another domain at some later stage, it can recognise it as a local interface and need not perform a full bind to it.

5.3.3 Binding to an offer

Given an IDCOffer closure, a client can invoke the Bind operation to attempt to connect to the server. The IDCOffer operates as a local interface, and uses

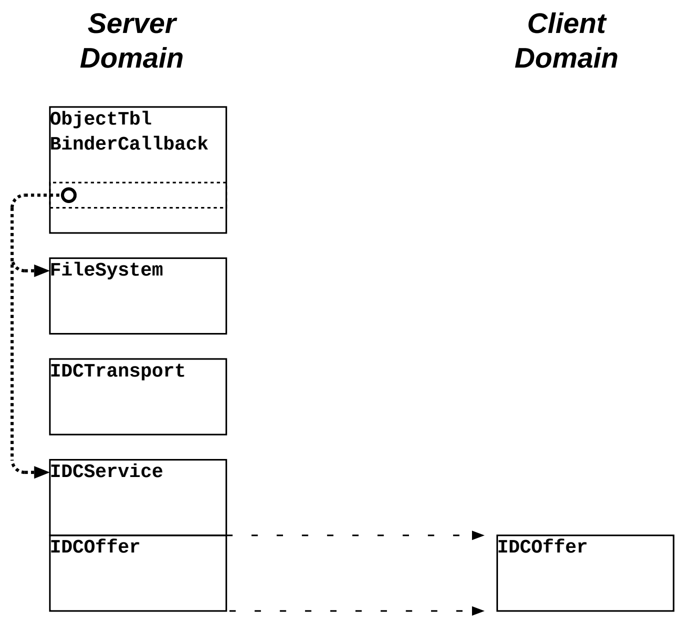


Figure 5.2: The result of offering a service

the client's local state to call the Binder with a request to set up event channels to the server domain, passing the offer interface reference as an identifier (figure 5.3).

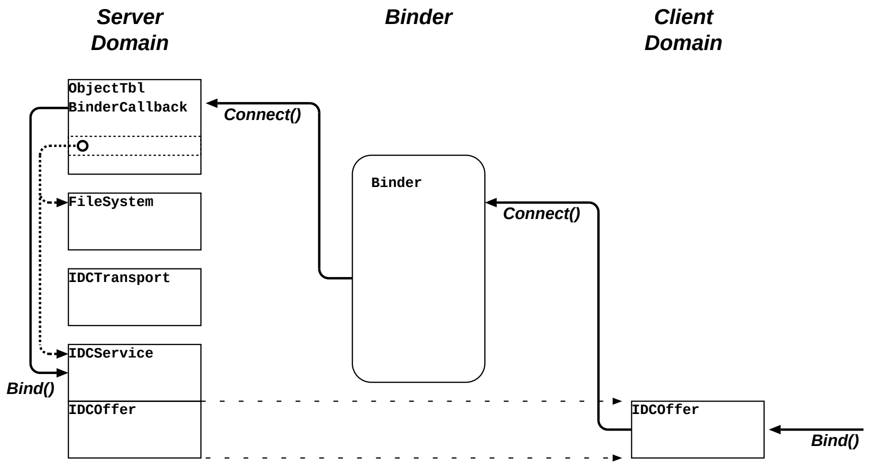


Figure 5.3: An attempt to bind to an offer

Since the client is effectively executing some code supplied by the server in the client's protection domain, there is conceivably a security problem here. However, since all offers generated by a particular transport class share the same operation table[1], and the number of different transport classes in the system is small, it is a simple matter for a concerned client to copy the closure record and validate the address of the operation table. A similar mechanism is used in Spring.

The Binder in turn calls the object table in the server domain, which determines which `IDCService` closure corresponds to the offer. The `Bind` operation on this closure is called. This invocation creates the state (shared memory buffers, event channel end-points, threads, etc.) for the binding. It also creates a closure of type `IDCServerStub`, which allows the server domain to close down a binding, for example.

It is at this point that access control on the interface is exercised. How this is performed is, once again, up to the server. The object table could hold access control lists, for example. Alternatively, the `IDCService` could implement a rather more application-specific policy. The exchange of cookies can be used for more secure authentication if necessary.

Information for creating the connection between the client and server domains is passed back to the Binder. The Binder connects the event channel end-points

[1]Regardless of type

in client and server domains, and returns from the call by the `IDCOffer` in the client's domain. Finally, the `Bind` call to the offer returns after creating two closures: the first is of the same type as the service and acts as the surrogate. The second is of type `IDCClient` and allows the client domain to manipulate the binding in certain ways (for example, to close it down). At this stage the situation looks like figure 5.4. All the state required for the binding has been created in both client and server domains.

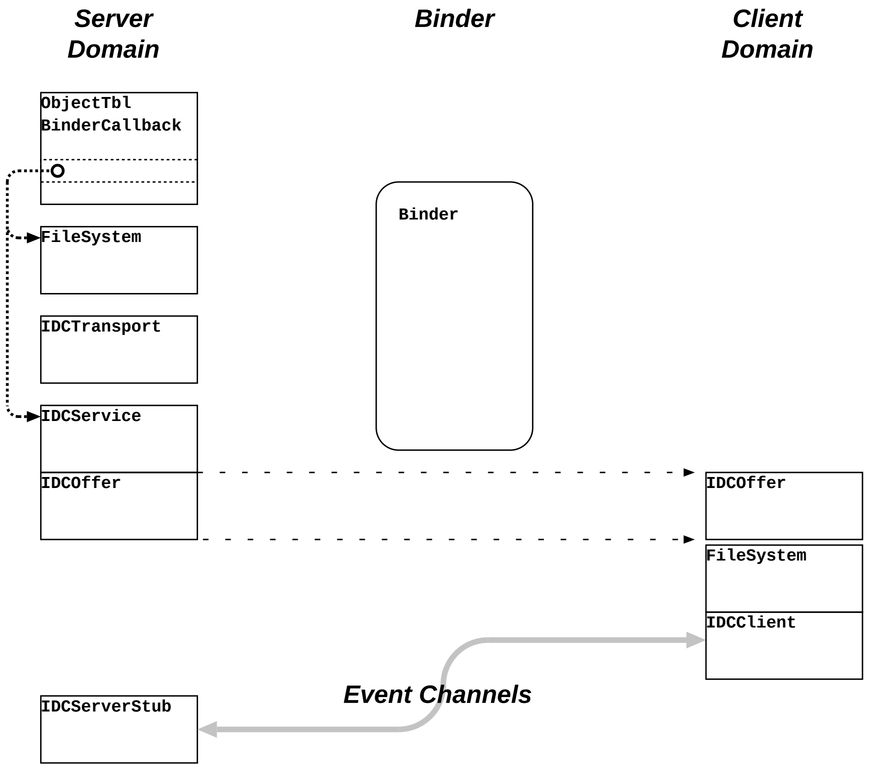


Figure 5.4: The result of a successful bind

5.3.4 Naming of Interface References

In this binding system the interface reference which is passed between domains is a pointer to the `IDCOffer` closure. Possession of an interface reference does not imply any kind of access rights to the service. Rather, the interface reference is simply a low-level name for the service. Access control is carried out by the server domain at bind time, so the kernel does not need to enforce restrictions on how interface references are passed between domains[2].

[2]In this respect, the statement concerning unguessable interface references in Nemesis on page 56 of [Black94] is incorrect.

As with any other value in the system, the `IDCOffer` may be installed at will in the name space. This is made possible by the highly orthogonal nature of the Nemesis naming scheme: any value can be named, and because of the explicit nature of interface references (closure pointers in the local case and `IDCOffer`s in the remote case) there is little reliance on name space conventions, with their associated management problems.

5.4 Communication

One of the principal benefits of the binding model is that by allowing the server to dictate the transport mechanism used, it allows great flexibility in implementing communication across a binding.

Coupled with the use of a single address space, a number of useful optimisations are possible in the case of communication between domains on a single machine, without affecting the performance of conventional RPC. In this section several increasingly specialised optimisations are described, starting with the default Nemesis local RPC transport.

5.4.1 Standard mechanism

The 'baseline' IDC transport mechanism (and the first to be implemented) operates very much like a conventional RPC mechanism. The bind process creates a pair of event channels between client and server. Each side allocates a shared memory buffer of a size determined from the stub record of the offer and from a heap determined by the domain's gatekeeper, which ensures that it is mapped read-only into the other domain. The server creates a thread which waits on the incoming event channel.

An invocation copies the arguments (and the operation to be invoked) into the client's buffer and sends an event on its outgoing channel, before waiting on the incoming event channel. The server thread wakes up, unmarshals the arguments and calls the concrete interface. Results are marshalled back into the buffer, or any exception raised by the server is caught and marshalled. The server then sends an event on its outgoing channel, causing the client thread to wake up. The client unmarshals the results and re-raises any exceptions.

Stubs for this transport are entirely generated by the MIDDL compiler, and the system is good enough for cases where performance is not critical. The initial bindings domains possess to the Binder itself use this transport mechanism.

5.4.2 Constant Read-only Data

For information which does not change, or which is guaranteed to be read and written atomically (for example, single machine words), data can simply be made readable in the client domain. All 'IDC' transport code is executed within the client's domain, and no communication need occur. The `Domain` interface (which presents the client interface to the kernel scheduler and the domain data structures), the system local clock (a 64-bit ticker), and the initial implementation of the Type System use this optimisation.

If the data involved is readable globally, no bind step is technically necessary and a ready-made surrogate (requiring no per-domain state) may be exported instead of the `IDCOffer`. Alternatively, a trivial `IDCOffer` could return the surrogate.

A more useful function of the `IDCOffer` in this case is to ensure that the data is available, or request that it be made so. The (constant) surrogate is only returned if the data is readable.

5.4.3 Optimistic Synchronisation

In many cases it may be possible for a server domain to modify a data structure in place in such a way that a client which is reading it does not cause any exceptions. Version numbers can then be employed to implement a form of optimistic synchronisation: a client wishing to read the data structure notes the version number, reads the data structure, and then looks to see if the version number has changed. If it has, then some invariant on the data structure as read by the client may not hold and it must retry the operation. If updates to the structure are rare, this technique can be very fast.

If it is desired to share write access to a data structure between mutually trusting domains, more sophisticated optimistic synchronisation methods can employed. Such an approach has been found to work well in the Synthesis operating system [Massalin89]. As with access to immutable data, the details of the mechanism can be hidden with a surrogate object.

5.4.4 Timed Critical Sections

A more exciting possibility is to use the knowledge of the passage of time to allow safe read access to data structures which may change, and which might normally cause exceptions (such as bad pointer references) in clients if they changed in the middle of a read sequence.

The basic idea is that a data structure is always changed in such a way that a client in the process of traversing it will not encounter a bad reference until a fixed period of time has passed from the time of update.

A simple example is to have a single location holding a pointer to a structure. To update the structure, a new copy is made, the pointer changed in an atomic write, and then the old copy deleted after a certain period of time.

A *timed critical section* is a programming construct that causes a thread to register the start of the traversal with the threads package. Since the user-level thread scheduler is entered with a guaranteed frequency (given by its QoS parameters), it can observe modifications to the data structure and halt the thread (by raising an exception on it) before it has a chance to encounter a bad reference if the time limit is passed.

Timed critical sections have yet to be implemented, and while clearly inappropriate in some circumstances they are mentioned here as an intriguing line of future work.

5.4.5 Specialist Server Code and Hybrid Solutions

As a final point, note that all the techniques described in the previous sections can coexist within the same interface stub. For example, information can be read from a server simply by reading shared memory while cross-domain events are used to transmit updates to a data structure. For ease of prototyping it may be easy to use a compiler to generate a standard set of stubs for a given service, and then at a later date optimise the stubs when the performance requirements of the interface and its effect on the rest of the system are better understood.

5.5 Discussion

It is vital to make the distinction between the interfaces that a programmer sees to a particular service, and the interfaces placed at the boundaries of protection domains or schedulable entities. A noticeable feature of most modern operating systems is that they usually confuse these two types of interface. This is a major contributing factor to the problem of application crosstalk.

The binding model described above enables the functionality of a service to be split arbitrarily between the protection and scheduling domains of client and server with no increase in complexity; indeed the object-based RPC invocation of Nemesis is simpler to use than the ad-hoc mechanisms in most traditional operating systems.

It is interesting to note that the binding model is much closer to that of modern distributed programming environments than conventional operating systems. This is a natural consequence of Nemesis enforcing much stronger separation of resource usage (in particular CPU time) between applications than other operating systems. It is also a reflection of the fact that the fundamental concepts in binding are much more prominent in the distributed case.

Within the flexibility of the binding model, a number of techniques can be employed for communication between domains to reduce the level of synchronisation that must occur. The division in service functionality between client and server is usually dictated by the needs of security and synchronisation rather than by the abstractions used to think about the service. When server processes are eventually called, it is generally to perform the minimum necessary work in the shortest possible time.

When a conventional message exchange between domains has to occur, the separation of data transmission (shared memory buffers) from synchronisation (events) allows high performance without unduly compromising the QoS guarantees or scheduling policies of both client and server.

Finally, Nemesis bindings are one-to-one and visible in the server. This means that a server can attach QoS parameters to incoming invocations according to contracts negotiated at bind time. This is in marked contrast to systems such as Spring: in Spring the kernel is aware of bindings and threads but the server is not, whereas in Nemesis the server is aware of bindings and threads but the kernel is not.

This feature of Nemesis enables the use of small schedulers within the servers to reduce crosstalk, and gives client applications qualitative bounds on the jitter they experience from a service. Operating system servers which provide QoS in this way, particularly window systems, are currently being investigated within Nemesis [Barham95b].

5.5.1 Performance

Figure 5.5 shows the distribution of same-machine null RPC times between two domains on an otherwise unloaded machine. Most calls take about 30μs, which compares very favourably with those reported in [Chase93] for Mach (88μs) and Opal (122μs) on the same hardware. The calls taking between 55μs and 65μs experience more than one reschedule between event transmissions.

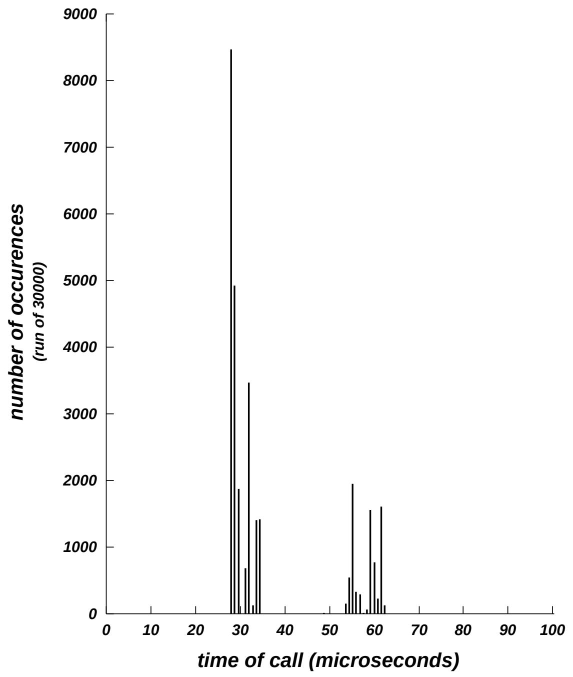


Figure 5.5: Distribution of Null RPC times

Nemesis does not currently implement full memory protection domains; the cost of a full protection domain switch consists of a single instruction to flush the 21064 data translation buffer (DTB), followed by a few DTB misses. This cost of a DTB fill on the current hardware has been estimated at less than 1μs [Fairbairns95].

It must be emphasised that figure 5.5 represents measurements of absolutely standard, non-optimised user-thread to user-thread RPC calls across an interface of type `NullRPC`. The RPC involved calling through generated stubs, two complete passes through the scheduler, plus an activation to both client and server domains, each of which was running the default vanilla threads package. The call even went through a call dispatcher at the server end. A highly optimised Nemesis domain has been observed to send a message to another domain and receive a reply in under 14μs though it is unreasonable to claim this is a null RPC call. It does, however, illustrate the efficiency of the event mechanism in addition to its flexibility.

	percentage
Scheduler and context switch	29
Event delivery	20
Activation handler	47
Stubs	4

Table 5.1: Breakdown of call time for same-machine RPC

The cost of a same-machine null RPC call breaks down roughly as shown in table 5.1 (measured using the processor cycle counter). There are no unexpected figures, except that the efficiency of the user-level thread scheduler in dispatching events clearly leaves something to be desired.

5.6 Summary

Communication between domains in Nemesis is object-based and uses invocations on surrogate interfaces. As in the case of a single domain, interfaces can be created and destroyed easily, and interface references can be passed around at will.

Establishing a binding to an interface in another domain is an explicit operation, and is type-safe. This explicit bind operation allows the negotiation of QoS with the server and returns control interface closures which allow client and server to control the characteristics of the binding. Implicit binding of interface references can be performed if desired in generated stub code. Passing interface references between domains requires no intervention by the operating system, and the single address space facilitates the process of binding so that the system binder's involvement in the process is minimal.

Communication between domains over a binding is in the default case performed with shared memory buffers, using events to convey synchronisation information. This provides low kernel overhead and to a large extent decouples remote invocation from the scheduler, preventing crosstalk due to IDC operations. The normal invocation path is very fast, more so if the cost of a reschedule can be amortised over several invocations.

The flexibility and abstraction of the binding model also permits the transparent integration of a number of local-case RPC optimisations, including a novel technique to use domains' knowledge of the passage of time to relax synchronisation constraints on data structures.

This leads on to the more general value of the Nemesis IDC architecture: since the interfaces over which invocations are performed are not the same as those between protection domains or schedulable entities, functionality can be moved between server and client. In particular, as much of an operating system service can be executed in the client domain as the requirements of security and synchronisation will allow.

Chapter 6

Conclusion

This dissertation has presented a way of structuring an operating system better suited to the handling of time-sensitive media than existing systems. This chapter summarises the work and its conclusions, and suggests future areas of study.

6.1 Summary

Chapter 2 discussed the requirements for an operating system to process multimedia. The use of a Quality of Service paradigm to allocate resources, in particular the processor, has been shown to give the kind of guarantees required. However, implementing such a resource allocation policy in a conventional kernel- or microkernel-based operating system is problematic for two reasons, both arising from the fact that operating system facilities are provided by the kernel and server processes, and hence are shared between applications.

The first problem is that of accounting for resource usage in a server. Current attempts to solve this problem fall into two categories:

- Accounting can be performed on a per-thread basis, in which case threads must be implemented by the kernel and cross protection domain boundaries to execute server code.
- Alternatively, accounting can be performed on a per-domain basis, in which case some means of transferring resources from a client to a server is required.

Both these solutions are shown to be inadequate. The former approach prevents the use of application-specific scheduling policies, while the latter is difficult to make efficient in practice since resource requirements are difficult to determine. Both approaches also allow badly behaved servers to capture clients' resources.

The second problem is that of application crosstalk, first identified in protocol stack implementations but extended in this dissertation to cover all shared operating system services. Crosstalk has been observed in practice and it is important to design an operating system to minimise its effect.

The approach proposed in this dissertation is to multiplex system services as well as resources at as low a level as possible. This amounts to implementing the minimum functionality in servers, migrating components of the operating system into the client applications themselves. The rest of the dissertation is concerned with demonstrating that it is possible to construct a working system along these lines, by describing the Nemesis operating system.

Chapter 3 presented the model of interfaces and modules in Nemesis, which address the two principal software engineering problems in constructing the system: managing the complexity of a domain which must now implement most of the operating system, and sharing as much code and data between domains as possible. The use of closures within a single address space allows great flexibility in sharing, while typed interfaces provide modularity and hide the fact that most system services are located in the client application.

An unexpected result from chapter 3 is that the gain in performance from small image sizes is very difficult to quantify. The fully direct-mapped cache system in the machines used meant that effects of rearranging code within the system overwhelmed the effects of sharing memory, even with image sizes much larger than the cache. The overhead of closure passing in Nemesis is also swamped by the cache effects.

Chapter 4 addressed the problem of scheduling, and how CPU time can be allocated to domains within a system such as Nemesis. Existing systems either do not allow sufficient flexibility in the nature of CPU time guarantees, or else do not permit adequate policing of processor usage by domains. Allocation of CPU time in Nemesis is based on a notion of a time slice within a period best suited to an individual domain's needs. An algorithm is devised which transforms the problem of meeting all domains' contracts into one which can be solved using an Earliest Deadline First (EDF) scheduler. A mechanism developed from that of the Nemo system is used to present domains with information about their resource allocation, and to provide support for internal multiplexing of the CPU within each domain via a user-level threads package. Communication between domains is designed so as not to violate scheduling constraints. Processor interrupts are decoupled from scheduling so as to prevent high interrupt rates from seriously impacting system performance.

The scheduler is shown to be very fast, and to scale well with the number of schedulable entities in the system. Furthermore, it can efficiently schedule a job mix where resource guarantees have effectively committed all the available processor resources.

Chapter 5 discusses the design of an inter-domain communication facility. A model of binding is presented which allows great freedom in splitting service functionality between client and server domains, and permits negotiation of quality of service parameters at bind time where server domains implement resource scheduling between clients.

A conventional local RPC system built over this framework, using shared memory and the events mechanism from chapter 4, is shown to be significantly faster than comparable systems on the same hardware. Furthermore, several optimisations are discussed which use the single address space structure of Nemesis.

In the design of an operating system for multi-service applications, there is a tension between the need for predictability and accurate accounting, and the desire for efficiency of resource usage and allocation in the system. Monolithic and kernel-based systems can make highly efficient use of resources but give little in the way of fine-grained guarantees to applications. Nemesis demonstrates that an operating system can make useful Quality of Service guarantees without compromising system performance.

6.2 Future Work

Nemesis as described in this dissertation is a working prototype, and while it appears capable of achieving its aims, much work needs to be done before it can be used reliably as a workstation operating system: virtual memory, network protocol stacks, etc. The system is also currently geared towards uniprocessor machines, and the changes required to the scheduling mechanism on a multiprocessor require consideration.

The issue of how to collect garbage automatically in a single address space operating system remains problematic. Not all pointers can be traced from a given protection domain, and without a central policy or convention for object creation and destruction it is difficult to imagine an effective, system-wide collector. Ideas from the field of distributed systems may help here: it might be possible to run a local per-domain collector with communication between domains to handle inter-domain references.

Some of these issues are being addressed in a new version of the operating system being produced in the Computer Laboratory. This system will be made available for general release, and is one of the platforms used in the DCAN project, a collaborative venture between the Laboratory, APM Ltd. and Nemesys Research Ltd. to investigate the distributed control and management of ATM networks.

Research into the design of applications which can adapt to changing conditions is at an early stage. Of particular interest is the design of real-time threads packages for applications with particular requirements. The implementation of the timed critical sections outlined in section 5.4.4 also falls into this category. It is expected that experience with the system will make clear the issues in the design of a Quality of Service Manager, to provide system wide resource allocation and admission control. This service, and its user interface, are crucial to the success of QoS as a resource allocation paradigm.

Bibliography

[Accetta86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. *Mach: A New Foundation for* UNIX *Development.* In USENIX, pages 93–112, Summer 1986. (pp 8, 70)

[Anderson92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism.* ACM Transactions on Computer Systems, 10(1):53–79, February 1992. (pp 13, 68)

[Apple85] Apple Computer. *Inside Macintosh*, volume 1. Addison-Wesley, 1st edition, 1985. (p 7)

[Bach86] Maurice J. Bach. *The Design of the* UNIX *Operating System.* Prentice-Hall International, 1986. (p 7)

[Barham95a] P. Barham, M. Hayter, D. McAuley, and I. Pratt. *Devices on the Desk Area Network.* IEEE Journal on Selected Areas in Communication, 13, March 1995. To appear. (p 18)

[Barham95b] Paul Barham. *Devices in a Multi-Service Operating System.* PhD thesis, University of Cambridge Computer Laboratory, 1995. in preparation. (p 98)

[Bayer79] R. Bayer, R. M. Graham, and G. Seegmuller, editors. *Operating Systems: an Advanced Course*, volume 60 of *LNCS.* Springer-Verlag, 1979. (p 112)

[Bershad90] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. *Lightweight Remote Procedure Call.* ACM Transactions on Computer Systems, 8(1):37–55, February 1990. (pp 9, 16, 85)

[Bershad91] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. *User-Level Interprocess Communication for Shared Memory Multiprocessors.* ACM Transactions on Computer Systems, 9(2):175–198, May 1991. (pp 9, 87)

[Birrell84] Andrew D. Birrell and Bruce Jay Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, 2(1):39–59, February 1984. (p 80)

[Birrell87] A.D. Birrell, J.V. Guttag, J.J. Horning, and R. Levin. *Synchronisation Primitives for a Multiprocessor: A Formal Specification.* Technical Report 20, Digital Equipment Corporation Systems Research Centre, August 1987. (p 61)

[Birrell93] Andrew Birrell, Greg Nelson, Susan Owicki, and Ted Wobber. *Network Objects.* Proceedings of the 14th ACM SIGOPS Symposium on Operating Systems Principles, Operating Systems Review, 27(5):217–230, December 1993. (p 83)

[Black94] Richard J. Black. *Explicit Network Scheduling.* PhD thesis, University of Cambridge Computer Laboratory, 1994. (pp 49, 52, 93)

[Bricker91] A. Bricker, M. Gien, M. Guillemont, J. Lipkis, D. Orr, and M. Rozier. *A New Look at Microkernel-based* unix *Operating Systems: Lessons in Performance and Compatibility.* Technical Report CS/TR-91-7, Chorus Systemes, February 1991. (p 9)

[Brockschmidt94] Kraig Brockschmidt. *Inside OLE 2.* Microsoft Press, 1994. (p 23)

[Campbell93] Andrew Campbell, Geoff Coulson, Francisco Garcia, David Hutchison, and Helmut Leopold. *Integrated Quality of Service for Multimedia Communications.* In Proceedings of IEEE INFOCOMM '93, volume 2, pages 732–739, March/April 1993. (p 4)

[Chapman94] Roderick Chapman, Alan Burns, and Andy Wellings. *Integrated program proof and worst-case timing analysis of SPARK Ada.* In Proceedings 1994 ACM Workshop on Language, Compiler, and Tool Support for Real-Time Systems, June 1994. (p 12)

[Chase93] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. *Sharing and Protection in a Single Address Space Oper-*

ating System. Technical Report 93-04-02, revised January 1994, Department of Computer Science and Engineering, University of Washington, Seattle, Washington 98195, USA, April 1993. (pp 24, 86, 98)

[Clark92] David D. Clark, Scott Shenker, and Lixia Zhang. *Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism.* In Proceedings of SIGCOMM '92, pages 14–26, 1992. (p 3)

[Coffman73] E. Coffman and P. Denning. *Operating Systems Theory.* Prentice-Hall Inc., Englewood Cliffs, N. J., 1973. (p 56)

[Coulson93] Geoff Coulson, G. Blair, P. Robin, and D. Shepherd. *Extending the Chorus Micro-kernel to support Continuous Media Applications.* In Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video, pages 49–60, November 1993. (pp 3, 50)

[Custer93] Helen Custer. *Inside Windows NT.* Microsoft Press, 1993. (p 7)

[DEC92] Digital Equipment Corporation. *DECchip 21064-AA Microprocessor Hardware Reference Manual*, 1st edition, October 1992. Order Number EC-N0079-72. (p 54)

[DEC93] Digital Equipment Corporation. *DECchip 21064 Evaluation Board User's Guide*, May 1993. Order Number EC-N0351-72. (p 5)

[DEC94] Digital Equipment Corporation. *DEC3000 300/400/500/600/700/800/900 AXP Models: System Programmer's Manual*, 2nd edition, July 1994. Order Number EK-D3SYS-PM.B01, and good luck to you. (p 5)

[Dixon92] Michael Joseph Dixon. *System Support for Multi-Service Traffic.* PhD thesis, University of Cambridge Computer Laboratory, January 1992. Available as Technical Report no. 245. (p 67)

[Evers93] David Evers. *Distributed Computing with Objects.* PhD thesis, University of Cambridge Computer Laboratory, September 1993. Available as Technical Report No. 332. (pp 26, 89)

[Fairbairns95] Robin Fairbairns. *Cost of an Alpha Nemesis protection domain switch.* personal communication, March 1995. (p 98)

[Garrett93] W. E. Garrett, M. L. Scott, R. Bianchini, L. I. Kontothanassis, R. A. McCallum, J. A. Thomas, R. Wisniewski, and S. Luk. *Linkage Shared Segments.* In Proceedings of Winter USENIX, pages 13–28, January 1993. (p 21)

[Goldenberg92] Ruth E. Goldenberg and Saro Saravanan. *VMS for Alpha Platforms Internals and Data Structures*, volume 1. Digital Press, preliminary edition, 1992. (p 7)

[Gutknecht] Jürg Gutknecht. *The Oberon Guide.* (version 2.2). (p 38)

[Hamilton93a] Graham Hamilton and Panos Kougiouris. *The Spring Nucleus: A Microkernel for Objects.* Technical Report 93-14, Sun Microsystems Laboratories, Inc., April 1993. (pp 9, 22, 83)

[Hamilton93b] Graham Hamilton, Michael L. Powell, and James G. Mitchell. *Subcontract: A Flexible Base for Distributed Programming.* Technical Report 93-13, Sun Microsystems Laboratories, Inc., April 1993. (pp 22, 83)

[Hildebrand92] Dan Hildebrand. *An Architectural Overview of QNX.* In USENIX Workshop Proceedings : Micro-kernels and Other Kernel Architectures, pages 113–126, April 1992. (p 9)

[Hills93] Ted Hills. *Structured Interrupts.* ACM Operating Systems Review, 27(1):51–68, January 1993. (p 69)

[Hopper90] Andy Hopper. *Pandora: An Experimental System for Multimedia Applications.* ACM Operating Systems Review, 24(2):19–34, April 1990. Available as ORL Report no. 90-1. (p 11)

[Hopper92] Andy Hopper. *Digital Video on Computer Workstations.* In Proceedings Eurographics '92, September 1992. Available as ORL Technical Report 92-6. (p 47)

[Hyden94] Eoin Hyden. *Operating System Support for Quality of Service.* PhD thesis, University of Cambridge Computer Laboratory, February 1994. Available as Technical Report No. 340. (pp 4, 47, 51, 68)

[Khalidi92] Yousef A. Khalidi and Michael N. Nelson. *An Implementation of UNIX on an Object-oriented Operating System.* Technical Report 92-3, Sun Microsystems Laboratories, Inc., December 1992. (p 18)

[Leffler89] S.J. Leffler, M. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD* UNIX *Operating System.* Addison-Wesley, 1989. (p 7)

[Liskov81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *LNCS*. Springer-Verlag, 1981. (p 33)

[Liu73] C. L. Liu and James W. Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.* Journal of the Association for Computing Machinery, 20(1):46–61, January 1973. (pp 50, 56)

[Liu91] J. Liu, J. Lin, W. Shih, A. Yu, J. Chung, and Z. Wei. *Algorithms for Scheduling Imprecise Computations.* IEEE Computer, 24(5):58–68, May 1991. (p 47)

[Massalin89] Henry Massalin and Calton Pu. *Threads and Input/Output in the Synthesis Kernel.* In Proceedings of the 12th ACM Symposium on Operating Systems Principles, December 13-16 1989. (p 96)

[McAuley89] Derek McAuley. *Protocol Design for High Speed Networks.* PhD thesis, University of Cambridge Computer Laboratory, September 1989. Available as Technical Report no. 186. (p 14)

[McCanne93] S. McCanne and V. Jacobson. *The BSD Packet Filter: A New Architecture for User-level Packet Capture.* In USENIX Winter Technical Conference, pages 259–269, January 1993. (p 14)

[Mercer93] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. *Processor Capacity Reserves: An Abstraction for Managing Processor Usage.* In Proc. Fourth Workshop on Workstation Operating Systems (WWOS-IV), October 1993. (p 69)

[Mercer94] Clifford W. Mercer, Stefan Savage, and Hideyuki Tokuda. *Processor Capacity Reserves: Operating System Support for Multimedia Applications.* In Proceedings of the IEEE International Conference on Multimedia Computing and Systems, May 1994. (p 16)

[Mogul87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. *The Packet Filter: An Efficient Mechanism for User-level Network Code.* Research Report 87/2, Digital Equipment Corporation, Western

Research Laboratory, 100 Hamilton Avenue, Palo Alto, California 94301, November 1987. (p 14)

[Mullender93] Sape Mullender, editor. *Distributed Systems.* Addison Wesley/ACM Press, 2nd edition, 1993. (p 113)

[Nelson91] Greg Nelson, editor. *Systems Programming With Modula-3.* Prentice-Hall, Englewood Cliffs, NJ 07632, 1991. (p 26)

[Nelson93] Michael N. Nelson and Graham Hamilton. *High Performance Dynamic Linking Through Caching.* Technical Report 93-15, Sun Microsystems Laboratories, Inc., April 1993. (p 22)

[Nicolaou90] Cosmos Nicolaou. *A Distributed Architecture for Multimedia Communication Systems.* PhD thesis, University of Cambridge Computer Laboratory, December 1990. Available as Technical Report no. 220. (pp 4, 81)

[Nieh93] J. Nieh, J. Hanko, J. Northcutt, and G. Wall. *SVR4* UNIX *Scheduler Unacceptable for Multimedia Applications.* In Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio Video, pages 35–47, November 1993. (pp 3, 49)

[Obj91] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, Draft 10th December 1991. OMG Document Number 91.12.1, revision 1.1. (p 83)

[Oikawa93] Shuichi Oikawa and Hideyuki Tokuda. *User-Level Real-Time Threads: An Approach Towards High Performance Multimedia Threads.* In Proceedings of the 4th International Workshop on Network and Operating Systems Support for Digital Audio Video, pages 61–75, November 1993. (p 3)

[Organick72] E.I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, 1972. (pp 21, 87)

[Otway94] Dave Otway. *The ANSA Binding Model.* ANSA Phase III document APM.1314.01, Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge, CB3 0RD, UK, October 1994. (p 84)

[Pike92] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. *The Use of Name Spaces in Plan 9.* Technical Report, AT&T Bell Laboratories, Murray Hill, New Jersey 07974, 1992. (p 22)

[Pike94] Rob Pike. $8\frac{1}{2}$ *in Brazil.* Personal communication, December 1994. (p 18)

[Pratt94] Ian Pratt. *Hardware support for operating system support for continuous media.* PhD Thesis proposal, July 1994. (p 15)

[Radia93] Sanjay Radia, Michael N. Nelson, and Michael L. Powell. *The Spring Name Service.* Technical Report 93-16, Sun Microsystems Laboratories, Inc., November 1993. (p 36)

[Raj91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. *Emerald: A General-Purpose Programming Language.* Software—Practice and Experience, 21(1):91–118, January 1991. (p 26)

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

[Roscoe94a] Timothy Roscoe. *Linkage in the Nemesis Single Address Space Operating System.* ACM Operating Systems Review, 28(4):48–55, October 1994. (p 32)

[Roscoe94b] Timothy Roscoe. *The* MIDDL *Manual.* Pegasus Working Document (4th Edition), available from `ftp://ftp.cl.cam.ac.uk/pegasus/Middl.ps.gz`, August 1994. (p 26)

[Roscoe94c] Timothy Roscoe, Simon Crosby, and Richard Hayton. *The MSRPC2 User Manual* . In SRG [SRG94], chapter 16. SRG Technical Note. (p 30)

[Roscoe95] Timothy Roscoe. CLANGER *: An Interpreted Systems Programming Language.* ACM Operating Systems Review, 29(2):13–20, April 1995. (p 37)

[Rovner85] Paul Rovner. *On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language.* Technical

Report CSL-84-7, Xerox Corporation, Palo Alto Research Center, July 1985. (p 37)

[Rozier90] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. *Overview of the CHORUS Distributed Operating Systems.* Technical Report Technical Report CS-TR-90-25, Chorus Systemes, 1990. (pp 8, 70)

[Saltzer79] J. H. Saltzer. *Naming and Binding of Objects.* In Bayer et al. [Bayer79], chapter 3.A, pages 100–208. (p 35)

[Scott90] Michael L. Scott, Thomas J. LeBlanc, and Brian D. Marsh. *Multi-Modal Parallel Programming in Psyche.* In Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming, pages 70–78, March 1990. (pp 13, 68)

[Sites92] Richard L. Sites, editor. *Alpha Architecture Reference Manual.* Digital Press, 1992. (p 54)

[Smith93] Jonathan M. Smith and C. Brenden S. Traw. *Giving Applications Access to Gb/s Networking.* IEEE Network, 7(4):44–52, 1993. (p 69)

[SRG94] Systems Research Group. *ATM Document Collection.* University of Cambridge Computer Laboratory, 3rd (Blue Book) edition, March 1994. SRG Technical Note. (p 111)

[Stroustrup91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, 1991. (p 27)

[Stroustrup94] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, 1994. (p 27)

[Swinehart86] D. Swinehart, P. Zellweger, R. Beach, and R. Hagemann. *A Structural View of the Cedar Programming Environment.* Technical Report CSL-86-1, Xerox Corporation, Palo Alto Research Center, June 1986. (published in ACM Transactions on Computing Systems 8(4), October 1986). (p 7)

[Tennenhouse89] David L. Tennenhouse. *Layered Multiplexing Considered Harmful.* In Protocols for High Speed Networks, IBM Zurich Research Lab., May 1989. IFIP WG6.1/6.4 Workshop. (p 14)

[Thekkath93] Chandramohan A. Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D. Lazowska. *Implementing Network Protocols at User Level.* Technical Report 93-03-01, Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, 1993. (p 18)

[Veríssimo93] Paulo Veríssimo and Hermann Kopetz. *Design of Distributed Real-Time Systems.* In Mullender [Mullender93], chapter 19, pages 511–530. (p 12)

[Waldspurger94] Carl A. Waldspurger and William E. Weihl. *Lottery Scheduling: Flexible Proportional-Share Resource Management.* In Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 1–11, Novemeber 1994. (p 69)

[Wilkes79] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System.* North Holland, 1979. (pp 9, 16, 87)

[Wilkinson93] Tim Wilkinson, Ashley Saulsbury, Tom Stiemerling, and Kevin Murray. *Compiling for a 64-bit Single Address Space Architecture.* Technical Report TCU/SARC/1993/1, Systems Architecture Research Centre, City University, London, March 1993. (p 24)