A Dynamically Reconfigurable and Extensible Operating System

by

Alistair Craig Veitch

B.Sc., University of Waikato, 1988
M.Sc. (Hons), University of Waikato, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

Department of Computer Science

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

July 1998

In presenting this thesis/essay in partial fulfillment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Computer Science

The University of British Columbia

2366 Main Mall

Vancouver, BC

Canada V6T 1Z4

Date: 17 July 1998

Abstract

Operating systems are constantly getting more complex in the functionality they support, due to the increasing demands made by modern hardware and software innovations. Basing the kernel design on co-operating and modular services incorportating a flexible communications infrastructure with run-time binding makes the operating system dynamically configurable and extensible. These features aid in the management of system complexity, while also resulting in several software engineering and performance benefits.

Configurability gives the operating system designer and implementor the freedom to build a large number of components, which can be composed into different configurations depending upon the final system requirements. System components can be built and debugged in a user address space, and then transparently migrated into the kernel address space for performance once they have been demonstrated correct. This removes one of the major obstacles to developing kernel services, that of the necessity to reboot the system after each change to the service code. The system administrator can also reconfigure the system, providing similar advantages, and allowing dynamic system upgrades to be made, reducing system downtime.

Extensibility lets new functionality be integrated into the operating system. This can be done on an application specific basis. This enables the development of in-kernel applications in cases where high performance is required, such as for dedicated file servers. It is also possible for applications to interpose specialised kernel services, allowing them to dramatically increase their performance and aggregate system throughput when the default system policies are ill-matched to their behaviour.

The Kea operating system has been designed and implemented to be dynamically configurable and extensible. The design of the system features that make these features possible are described. Experimental results are shown that demonstrate that Kea offers comparable performance to a traditional operating system on the same hardware, and that extensibility can be used to increase performance for selected applications.

Table of Contents

Acknowledgements

As with any effort that takes almost five years, a lot of people have contributed in one way or another. It feels as if I could write another thesis just on those who have helped along the way, whether it be academically, socially, or just by being there. Certainly one page isn't enough to do you all justice.

Most important to the thesis was my supervisor, Norm Hutchinson. Norm was a never-ending source of advice and encouragement, and has become not only a mentor, but a friend.

Several people have helped in implementing Kea. Andrew Agno, Geoff Burian, Davor Cubranic, Peter Smith and Christian Vinther each contributed something. Peter Smith deserves extra mention as office-mate, wild ideas listener and proof-reader.

My parents, Ron and Isobel Veitch, deserve special thanks for always encouraging me in learning. My brother John and his wife Anita, and my in-laws, Alan and Gael Jellyman, Jeremy and Andrea Blackmore, and Spencer Jellyman, have always been supportive.

I've shared some of the best times away from campus with my various climbing partners. For the opportunity to escape, and the joy provided, thanks to Adrian Burke, Kory Fawcett, Patrick King, Lee Purvis and Murray Rudd.

Many, many other people have provided help, a positive environment, or a chance to escape the thesis for a while. In alphabetical order, you are: Don Acton, Kelly Booth, Dave Brent, Ian Cavers, Terry Coatta, Richard Dearden, Mike Donat, Adrienne Drobnies, Brad Duska, Mike Feeley, Dave Finkelstein, Jean Forsythe, Alain Fournier, Murray Goldberg, Mark Greenstreet, Scott Hazelhurst, Iris Koch, Andrew Lesperance, Dwight Makaroff, David Marwood, Lucy Meagher, Roland Mechler, Holly Mitchell, Gerald Neufeld, Raymond Ng, Margaret Petrus, Carlin Phillips, George Phillips, Peter Phillips, Frank Pronk, Jackie Purvis, Jennifer Ryan, Mike Sanderson, Chris Simpson, George Tsiknis, Alan Wagner, Sue and Wally Walker and Carol Whitehead.

Last, but definitely not least, my wife, Dallas, deserves thanks for always being there for me.

ALISTAIR C. VEITCH

The University of British Columbia

July 1998

CHAPTER 1

Introduction

1.1 Operating System Challenges

The operating system is the interface between a computer's hardware and the applications that run on that computer. As a consequence, operating system designers must continually respond to demands made by new and faster hardware, the applications which require access to that hardware for increased performance, new application technologies and the various classes of people who will be using the system.

Since the development of the first electronic computers, computer hardware technology has advanced rapidly. Modern computer systems have more and faster processors, larger and faster memories and disks, faster and wider buses, and higher network bandwidths with lower latencies. In the past five years alone, the speed and/or size of all of these technologies has grown by at least an order of magnitude, and this growth shows no sign of slowing for at least another decade. As a consequence, the assumptions under which an operating system was designed and developed may become less valid, or even false [Rosenblum et al. 95]. Differing rates of devel-

opment in computer subsystems may also lead to a shifting of performance bottlenecks. As examples, consider that inter-process communication (IPC) speed, while still very important, has been shown to be increasingly dominated by cache design [Bershad 92], or that many filesystems restrict the maximum size of files or the filesystem itself to some size (e.g. 4 Gb) which is smaller than the size of modern disk drives. The original development of timesharing systems necessitated software techniques through which the hardware could be not only used, but also managed. In particular, the need to share hardware resources among multiple users, and provide security to those users, resulted in a great increase to system software complexity.

New hardware may also require the operating system to undergo redesign. The transition from 32 to 64 bit processors required extensive redesign of several operating systems. Initial versions of Unix were designed to accommodate only one sort of filesystem, necessitating a revision when several filesystem types needed to be supported [McKusick et al. 96]. Currently, gigabit networking technology is forcing reevaluation of the design of low level network layers.

The demands made on operating systems by modern applications are at least as important a consideration as those made by hardware. Multimedia applications require large amounts of storage, high-bandwidth access to that storage, and fast, low latency network access. Database systems require efficient access to extremely large storage. An increasing dependance on distributed applications requires the operating system to support new computing paradigms. Mobile computing is forcing operating systems to deal with a dynamically changing operating environment and disconnected operation.

Applications may also require services that have not been foreseen by operating systems designers, or that have been designed in such a way as to be incompatible to the applications needs. As examples, consider that database systems often implement, or reimplement, services normally performed by the operating system, such as threads, memory management, or filesystems. Real-time applications demand specialised scheduler support and performance guaran-

tees that most general purpose operating systems cannot supply. The page-replacement algorithms of most operating systems interact badly with the memory access patterns of applications that must do significant garbage collecting, such as Java, Lisp or Emerald.

The disparate types and requirements of applications essentially force the operating system designer to make a choice between trying to satisfy the needs of a specific class of application, and consequently producing a system that may not perform well on other applications, or trying to build a general purpose system that will work with a large number of application classes, but will not provide optimal performance for any of them. It is impossible for any group of designers to foresee all possible application demands.

Other challenges to an operating system designer relate to the conflicting needs of various classes of user. Generally speaking, application developers do not, on the whole, care about the underlying system structure. Their only demands are that the system provide them with the abilities (programming interfaces) to develop applications, and that the system be efficient. This can be contrasted with system developers, who want to be able to easily reason about a system, and simplify the process of developing and debugging code to be incorporated into the system. Often these two viewpoints are in conflict – a more modular system structure may benefit designers, but may not support the efficiency requirements of developers. Other users also have different requirements of an operating system. System administrators want to be able to control the configuration of their systems, and in particular want to be able to easily change that configuration when necessary, e.g. when new software releases become available or a machine has new hardware added. Researchers wish to be able to experiment with totally new components which may incorporate new concepts, but do not wish to have to change the entire system to accommodate this experimentation.

1.2 Problem Definition

An operating system design – the paradigm which dictates how the various components making up the system are structured, the dependencies between them, and the means through which they interact – should be such that the operating system itself is flexible, is able to accommodate new hardware and software technologies, and meet the demands made by specific applications and users. Given the demands made by new hardware and software technologies, it is desirable that operating systems be capable of evolving in a timely manner. In particular, the operating system developers must be able to quickly and easily design, implement, debug and install new operating system services, or modify existing services. System administrators also need to be able to easily reconfigure systems. Application developers require a system that lets them extend the system in order to give their application the best performance possible. Building a system that satisfies all of these requirements was the goal of the research described by this thesis. In particular, the thesis describes the design, implementation and evaluation of a *reconfigurable* and *extensible* operating system. For the purposes of this thesis, these terms are defined as:

Reconfigurable: A system is reconfigurable if the components comprising the system can be rearranged in some manner. With respect to operating systems, this is restricted to mean the replacement of services, or the migration of services between address spaces. These operations may be done to enhance the systems performance, to replace components which have been found to contain errors, or to provide richer functionality.

Extensible: A system is extensible if new functionality can be added to that system. With respect to operating systems, this concept is further subdivided. *General* or *global* extensibility refers to the ability of system administrators or developers to add new services, which then become available to all users of the system. This is closely related, but not identical to, reconfigurability. *Application specific* extensibility is the ability of application developers to insert code that replaces or modifies the behaviour of a service, for a single application only, with other applications continuing to use the default

system provided service. This allows applications to increase both their own performance and the total system throughput.

It should be noted that an important property of both reconfigurable and extensible systems is that these actions can be performed dynamically at run-time.

A reconfigurable and extensible system has many tangible advantages. For developers, it eliminates the need to rebuild and reboot a system when changing any portion of the code. Instead, new services can be developed, compiled and downloaded directly into the kernel, replacing older versions. When services are still unreliable, and need debugging, they can be installed into their own address space, isolating them from other parts of the system. Once debugged, they can be incorporated into the kernel itself. This saves time previously needed to recompile, reinstall and reboot a system. Administrators can also install services in order to support new hardware, without having to recompile an entire kernel, or can reconfigure a system in order to meet changed operating requirements. Costly downtime can be avoided by the dynamic installation of system upgrades. Application developers can modify the system, so that their application can increase its performance.

There are several properties that are desirable for a reconfigurable and extensible system:

- *Fine grain construction* – system components must be built on as fine a grain as possible. This lets the system develop in an incremental manner, and restricts the scope of changes to those strictly necessary for the desired extension.

- *Flexible communications infrastructure* – the various components of the system must be able to communicate with each other, and the communications system used must be capable of supporting changes to the components, including those made at run-time.

- *No performance degradation* – compared to conventional operating systems offering similar functionality, there must be little, or no, performance degradation. As perfor-

mance is often the yardstick by which operating systems are measured, this is a critical property.

- *Run-time service replacement* – for the resulting system to be truly flexible, it must be possible to dynamically replace parts of the system. This is also important for rapid development, as dynamic replacement removes the necessity for a system rebuild and reboot with every change to a services code.
- *Application transparency* – changes made to kernel services should be transparent to both the applications using the system, and to other kernel services (even in the case where they are using the service which is being changed).
- *Security mechanism* – there must be some means through which arbitrary users or applications are prevented from modifying vital system components, but which also allows them to make changes that are deemed to be "safe" by the systems administrators.
- *Protection mechanism* – there must be some means through which the system is protected from damage by dynamically inserted code, particularly when that code has been provided by an application.
- *Ease of use* – the interface to install and modify system components should be relatively easy to use. Additionally, there should be support for determining the current system configuration.

1.3 Research Contributions

An operating system, Kea[1], incorporating the properties listed above has been developed. During implementation, it was also observed that the structure of the kernel itself strongly encouraged the development of system services in a highly modular and structured fashion. Based on these results, the following thesis statement is made:

1. The Kea is a native New Zealand bird, with several interesting properties of its own [Temple 94].

An operating system based on modular system services and incorporating a flexible communications infrastructure with run-time binding can be built without sacrificing performance. Kernels employing this paradigm are fully reconfigurable and extensible, as services can be dynamically installed, migrated and replaced, both globally and on an application specific basis.

Kea is the first operating system that satisfies all of the properties listed in the previous section. The important innovations that make Kea unique are its communications structure, which allows transparent run-time binding, the capabilities for dynamic modification of the system's structure, and the unification of the kernel and user programming environments, which further enhances Keas flexibility, as well as conferring substantial software engineering benefits.

Kea was also designed with the goal of application specificity in mind, and demonstrates that this can be accomplished with this type of system architecture. The thesis also defines some of the inherent problems associated with any form of application specificity, due to the conflict between an application's local resource requirements, and the need of the operating system to balance these with both the global resource availability and the demands of other applications. The thesis determines that some types of system changes only make sense when they are global, and then only for highly specialised applications which will be the primary load on the machine in question.

Another thesis contribution is the specification of one possible modular decomposition of operating system structure, and the identification of those parts of the system which "cut across" all other such boundaries, or are fundamentally tied to the system in such a way that it is impossible to replace them without major changes to all the other parts of the system. The most important system components with this property are the scheduler and protection mechanisms. In the case of scheduling, the thesis specifies a unique low-level interface that enables the construction of many types of scheduler. This interface is sufficiently complete that development of a new scheduler is relatively easy. It also reduces the number of functions that depend on the

properties of any individual scheduler to two, which can be easily found and modified in any code depending on scheduling properties.

With regard to protection and security[1], the Kea architecture proposes that these concepts are separate, and are in fact largely orthogonal both to each other and to the modularity of the system as a whole. While it has been successfully argued that modularity and protection should be separate [Druschel et al. 92a], as the partitioning of the system into modules is a matter of configuration, not design, Kea is the first design that also tries to isolate the security model of the system. Instead of implementing a set security policy, Kea isolates security information into a small number of structures that are defined by the system designer. By only providing base functions that treat these as anonymous structures, designers can implement as much (or as little) security checking as they desire. Hooks have been set in the code wherever security decisions need to be made, and, as proof of concept, a simple Unix-like protection system has been implemented and used where deemed appropriate.

Kea also directly supports, or substantially eases the development of, several other capabilities, including, but not limited to, interposition agents [Jones 93], operating system emulation [Golub et al. 90, Malan et al. 91], shared libraries [Gingell 89, Sabatella 90, Orr et al. 93], and continuous operation [Anderson et al. 92].

1.4 Design Overview

There are three important facets of the Kea design. The first of these is the overall system structuring, i.e. how the individual system components are coalesced into the whole. The second is the communications structure that binds these components. The third is the specification of the functionality for extensibility and application specificity. Each of these is examined in the subsequent sections.

1. Protection is regarded as the hardware enforced means by which parts of the software system are separated from each other, typically implemented using address spaces. Security is the set of policies used to determine user access to various parts of the system, and may be implemented using either hardware protection or software.

1.4.1 Overall Systems Structure

Before discussing the structure of the Kea system, it is necessary to briefly review some of the existing paradigms for operating system structuring. The issue of how best to design, configure and build an operating system kernel continues to be a somewhat contentious issue in the research community. While many paradigms are possible, the primary two are the *monolithic* and *microkernel* designs.

Traditionally, operating system kernels have been constructed as a single monolithic image, compiled from a large body of kernel code. This code is often many hundreds of thousands (or even millions) of lines and implements many disparate sets of functionality, such as virtual memory, process management, network protocols and file systems, many of which interact in subtle ways. Additionally, kernel code is often inherently more complex than application code, due to synchronisation, scheduling and timing constraints, and the need to manipulate privileged processor state. For the same reasons, kernel debuggers are also more difficult to develop and use. Because of this complexity, kernel code requires a proportionally higher level of maintenance and development than ordinary, or application, code.

Partly in order to combat the cost of monolithic kernel development, another kernel design technique, microkernels, has been pursued by researchers. As the name implies, microkernels are designed to be "small", where the kernel itself provides only the needed functionality to implement "servers" – programs that run in their own address spaces in user mode, and provide the system with those services not provided by the microkernel. The microkernel design philosophy offers the developer advantages in modularity, as different services, e.g. file systems and network protocols, can be implemented in separate servers, and can often be debugged as user level programs, using standard debuggers.

Despite their advantages, microkernels incur a performance penalty that, to a large extent, renders them unpalatable to many developers, for whom performance is all-important. This performance penalty arises out of the means through which different servers communicate with

one another. In a monolithic kernel, this is done with procedure calls, but this is impossible within a decomposed system, where various parts are running in different address spaces. Instead, microkernels must use some form of inter-process communication (IPC) between servers and kernel. This usually takes the form of message passing, often with some type of remote procedure call (RPC) [Birrell 89] layered on top. The disadvantages imposed by this architecture – where passing messages often means marshalling arguments, composing the message, copying the message between address spaces and context switching between those address spaces – limit the performance to be strictly less than that of an equivalent monolithic system, even for very highly tuned implementations [Härtig et al. 97].

Microkernels do, however, have the promise of increased flexibility, due to the decomposition of a single kernel into multiple servers. In reality, this promise is seldom realised, as a typical system only includes one server, providing all the standard operating system functionality not provided by the microkernel. Compared to the equivalent monolithic systems (from which they are normally derived), these "single-server" systems are no more flexible or extensible, have the same internal complexity, and often consume more resources. Furthermore, when the systems are decomposed, the performance penalties due to IPC costs become more important, because of the greater amounts of cross-address space communication.

The architecture proposed in this thesis resolves much of this conflict between modularity, system decomposition, and performance, by implementing a hybrid scheme. The system is composed of a set of *services*, which together implement a complete operating system. A service is essentially a well defined interface through which some entity can be accessed and asked to perform a task for the caller. There are several ways to view a service. As seen by the programmer, there are two basic parts to each service, an *interface* and an *implementation*. The interface describes the procedures, constants and data types that the service offers to clients, while the implementation refers to the compiled code which implements the interface. A reasonable comparison to make is that the interface is analogous to a C header (".h") file, while the implementation is like a library. As an example, the set of procedures that manipulate the virtual

memory of an address space might be grouped into a single service, the "vm" service. The service guarantees to provide the given procedures, with specified semantics. From the applications viewpoint, a service appears as a number of pointers to functions – one for each of the service entry points. By dereferencing these pointers and calling the specified function, the service can be accessed. To both the programmer and application, this appears as a local function call – the underlying function pointer implementation is effectively invisible[1]. In the case where the services are in separate address spaces, the functions pointed to are automatically generated stubs, which call the underlying communications system. Where the services are co-located (that is, located in the same address space), the destination function address is called directly. The mechanisms for this are discussed in sections 1.4.2 and 1.4.3, on the communications structure and extensibility features, respectively.

Services are built up in a highly modular fashion, and each can be run in a separate address space. However, for performance purposes, services can be co-located into any other address space, including the kernel. In the case where a service is loaded into an address space in which another service is already resident, the underlying communications system optimises any inter-service interactions into the appropriate procedure calls, thus ensuring optimum performance. This is transparent to both the services and their developers. While some other systems [Rozier et al. 92, Lepreau et al. 93, Condict et al. 94] have allowances for co-location, Kea is the first to make this completely transparent. Additionally, Kea makes this co-location dynamically available. At any time, services can be migrated between address spaces, transparently to clients of the service. The same facility also allows services to be dynamically replaced, or removed from the system entirely (although the latter may not be advisable if other services depend on the one being removed).

The dynamic run-time binding of services is what gives the Kea system its extensibility, as at any time services can be extended, or new services added, in order to increase the systems func-

1. This is at least true for the C language, in which Kea has been developed.

tionality or capabilities. It is also highly beneficial to system developers, as the operating system no longer needs to be rebuilt and rebooted every time a services code is changed.

1.4.2 Inter-Service Communications

As described, services essentially appear as a set of procedures, which can be invoked by other services or applications in order to accomplish some task. This argues strongly for a communications paradigm that directly supports procedure calls. This can be contrasted with typical microkernels, which use message passing, usually with an additional RPC layer. Although message passing is arguably more general, there are several advantages to directly supporting procedure calls:

- *Familiarity* – procedure calls are more familiar to the average programmer. Directly supporting them allows for easier development.
- *Efficiency* – while many systems can hide the underlying message passing mechanism using automatically generated stubs that marshall and unmarshall procedural arguments, this also introduces a layer of unnecessary inefficiency. By making "procedure call" the IPC mechanism, some cost is incurred due to the more complex operation (as opposed to passing a simple contiguous message), but an ultimate gain in efficiency is made by getting the kernel to perform tasks normally done (and duplicated) in every server in a conventional microkernel. Consider the section of pseudo-code below (Figure 1.1), that could come from a server in such a system:

```
while(true) {
    receive message;
    decode message;
    call procedure to perform operation;
    send reply message;
}
```

Figure 1.1: Server Pseudo-code

In the Kea system, this loop, and its receive/decode/send components, are entirely eliminated, as these operations are carried out in the kernel, with the service procedures being invoked directly.

- *System throughput* – When a call is made, the thread making the call effectively transfers into the destination address space. This is accomplished through a mechanism akin to the "migrating threads" model used in Mach 3.0 [Ford & Lepreau 94] and Springs "doors" [Hamilton & Kougiouris 93]. These papers describe several performance advantages of this idea, due to the lack of scheduler interaction, simpler code paths, and reduced thread interactions. This model also removes any artificial barriers to the parallelism inherent in the service. Instead of only processing one request at a time (as in Figure 1.1), any service can have a potentially unlimited number of threads executing in parallel within itself, allowing greater system throughput in many cases. This natural parallelism also goes some way towards reducing priority inversion problems caused by high priority threads having to wait on low priority threads whose request is currently executing in the server. With Kea the high priority thread can run in parallel with the low priority one, and is not blocked from entering the service.

An IPC mechanism that directly supports procedure call also considerably simplifies the construction of the Kea system as a number of independent services. As each of the services is specified by an interface, which in turn consists of a number of procedural entry points, the generation of client-side stubs is simple. More importantly, the stubs themselves are very short, and therefore efficient[1].

Kea's IPC mechanism also considerably eases the implementation of service co-location. When being built, each service is compiled into a single object file (the implementation file). When loaded into a new address space, the implementation file is dynamically linked against any necessary libraries. When the service is loaded into, or migrated to, an already extant address space, it is first dynamically linked against the existing application code and/or service

1. Stub generation and structure are discussed in detail in Sections 3.8 and 3.10.1.

implementations in that address space. This prevents any duplication of routines within address spaces, while keeping service code separate. To accomplish this, the kernel must keep symbol information for each address space and service available, but this has been found to be only a minor cost – for example, the storage space needed for the symbols in Kea's standard C library is only 5 Kb.

As a performance optimisation, the kernel detects when services are co-located, and optimises the service invocations into a single indirect procedure call, eliminating the kernel trap overhead completely. This is easily accomplished because, as described, the access points to each service are kept in function pointers. Because the kernel has access to the symbol table which specifies these pointers, and knows the names of the procedures comprising the interface, it can adjust the pointers to point to the service directly, rather than to the client stub. This alleviates one of the standard problems with microkernel systems, namely that the performance overheads imposed by IPC/RPC are too costly.

1.4.3 Extensibility

The structure of the system as a number of independent services, combined with some extensions to the service loading mechanisms described above, allow the implementation of a number of features which make Kea extensible. These features are referred to as service *migration*, *replacement* and *interposition*.

Service Migration

As the system is used, it may be desirable to move the service into another address space. The primary reason to do this is performance. If a service can be migrated into the address space of the most frequent client of the service, then the time needed for each service invocation can be reduced by the time taken for changing address spaces, which is a potentially expensive operation. Alternatively, invocations into the kernel are far cheaper than those into another address space. Thus, for trusted services such as device drivers, filesystems, and network protocols, it is desirable to move, or migrate, these into the kernel once they have been debugged. This

offers several advantages to the service developer, particularly if they are developing kernel services. Firstly, the strictly defined service interfaces enforce modularity, which makes the system as a whole more maintainable. Secondly, it is not uncommon for newly developed kernel services to cause a system crash, necessitating a tedious and time-consuming program/crash/reboot development cycle. Within Kea, services can initially be developed within a private address space, which restricts the crash (if any) or consequences of a programming error, to that address space. Finally, since the service is running in a separate address space, it can be easily debugged with standard tools.

Supporting service migration between user and kernel address spaces has the interesting consequence of unifying the user and kernel programming interfaces. This is required, as to be executable in both environments, the environments themselves must both conform to the same interfaces. In particular, the behaviour of synchronisation, memory allocation, and asynchronous event handling have to be identical. This unification has the beneficial side effects of reducing the complexity of the kernel programming environment (at the one-time cost of increasing the complexity of the low-level kernel code), and simplifying the software engineering process, particularly in documentation and testing.

Service Replacement

Given the ability to migrate services between address spaces, it is simple to also support the dynamic replacement of services, as almost exactly the same operations need to take place. Replacement is actually slightly easier, as there is only one address space involved. There are three reasons why a developer or system administrator might wish to replace a service:

- *Performance* – compared to the original, the replacement service may offer improved performance. This could be as simple as a reduced memory footprint or smaller CPU usage, or as complex as a changed trade-off in efficiency between various procedures in the service (e.g. a faster speed for some of the frequently invoked service procedures, at the cost of slower times on some of those that are used less often).

- *Correctness* – the replacement service may fix a bug in the original service. Replacement allows the system administrator to ensure system correctness, without potentially expensive down-time. It would also be possible for system vendors to ship upgrades to the system, without requiring the system administrator to perform a full system rebuild and installation.

- *Testing* – During development, services must be tested and debugged. Service replacement enables this process to proceed in a more timely manner, as faulty services are easily replaced.

Figure 1.2 illustrates service migration and replacement. In this figure, several services and an application (solid boxes) are shown as existing in several different address spaces (dashed boxes). Thin arrows indicate calls made on services. Calls that cross address space boundaries will be accomplished using a form of RPC. Other calls, within address spaces (such as that shown between services B' and C) will be optimised to direct procedure calls. Two of the basic service operations that make Kea extensible are also shown (thick arrows). The first, service migration, is when a service is moved transparently between address spaces. In this case, service A can reside in either its own or the kernel's address space. The second, service replacement, shows service B being replaced by B' (i.e. service B will no longer exist). These changes are transparent to both the service clients and the services themselves. Calls between services are optimised when possible. Thus, after replacement, service B' will now make upcalls [Clark 85] to C, but will have its calls to D optimised.

Service Interposition

Service interposition refers to the ability to interpose a new service between any existing pair of services. Interposition can be usefully applied in many areas. At the interface between the

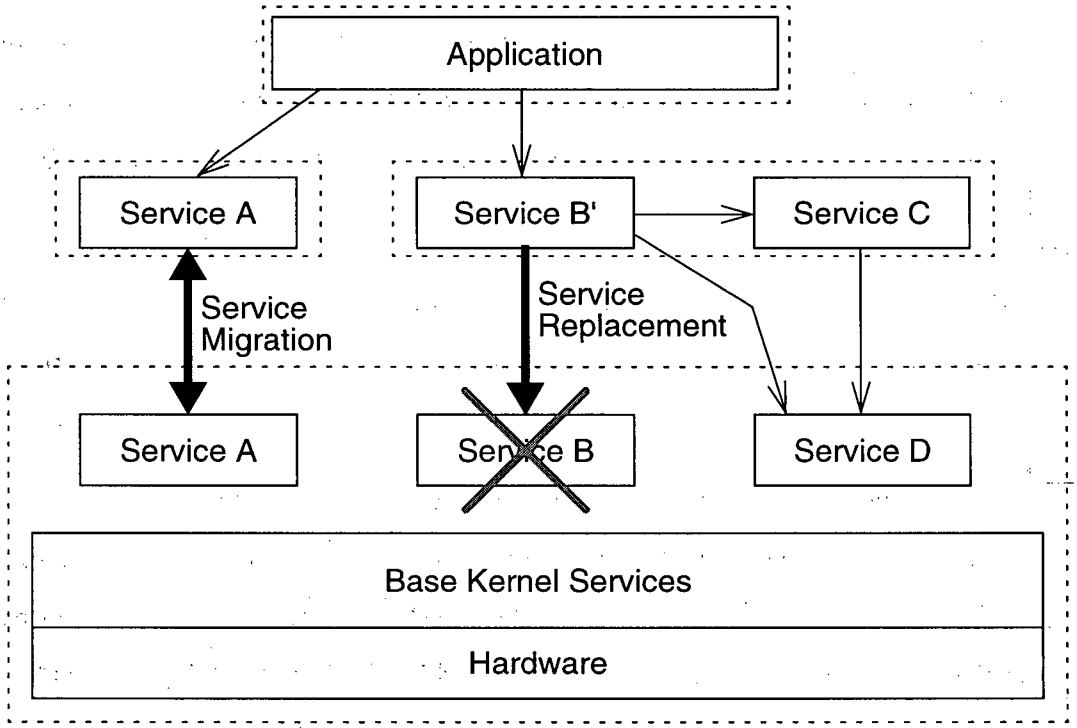

Figure 1.2: Service Migration and Replacement
Dotted boxes indicate address spaces. Thick arrows are service operations, thin arrows are calls on services.

operating system and applications alone, [Jones 93] identified several possible uses of this type of facility:

- Call tracing/monitoring – an interposed service can record or monitor an application's use of the underlying service.
- Operating system emulation – a service can be used to translate calls made by an application written for a "foreign" operating system into those used by the native system.
- Protected environments – a service wrapper can be developed that limits the actions of an untrusted binary.
- Alternate/enhanced semantics – a service could offer a interface that extends that of the underlying service, e.g. a transaction-based file system.

By interposing services at different points in the service hierarchy (not just at the upper layer), and allowing this to occur dynamically, the set of applications to which interposition applies is greatly expanded. In particular, the primary uses envisaged for service interposition are in interposing on system services that implement policies. For example, a service that, when interposed on Kea's buffer cache, implements preferential caching of specified file blocks has been developed.

The first parts of Figure 1.3 shows an example of a simple service interposition. Various service configurations are shown from left to right. The initial configuration, (a), shows two applications (A and B), using a chain of services S_1 through S_3. In (b), a new service, S_4, has been inserted between the applications and S_1. S_4 offers the same semantics as S_1, and the applications will not be able to tell that the underlying service structure has changed, except for a possible performance increase. As an example, if S_1 was a filesystem, S_4 could offer compression services, transparently uncompressing and compressing the data for read and write calls respectively. Another example could be adding an encryption layer onto a standard network protocol.

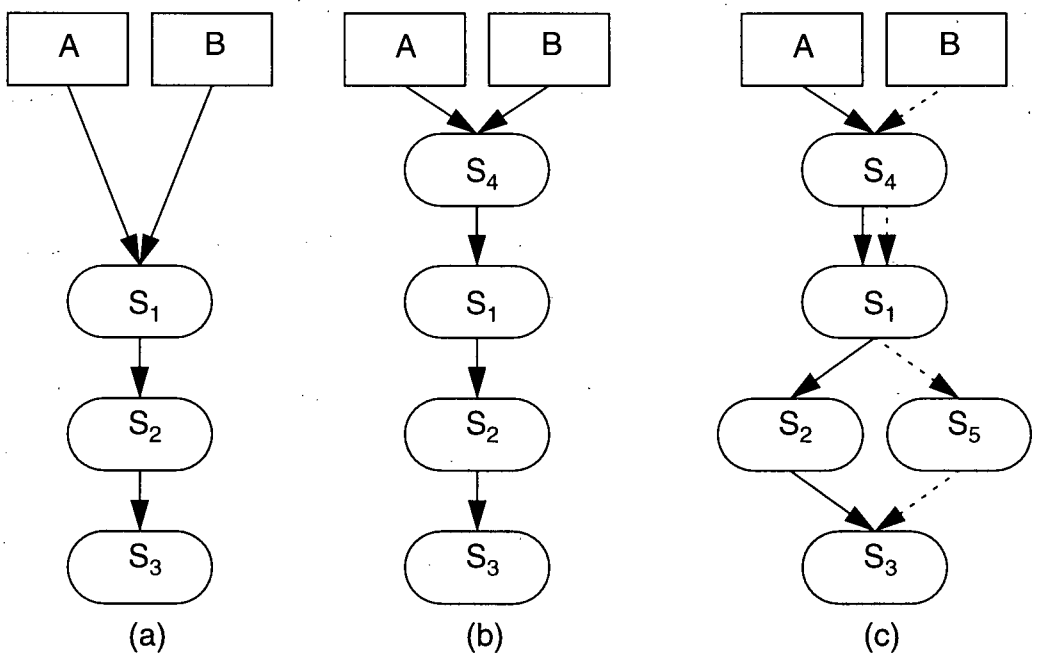

Figure 1.3: Service Reconfigurations

1.4.4 Application Specificity

Kea also specifies that each of the above operations can take place on an application specific basis. That is, if an application wishes to provide its own service, equivalent to one already existing in the system, but having different performance characteristics or enhanced semantics, it can specify the use of that service for its computations only. This ability can be applied both to services on which the application directly depends, but also to services further down the application's call chain. This capability lets the application effectively install its own code into the system, in order to enhance its own performance, while leaving other system clients unaffected. Part (c) of Figure 1.3 shows an application specific interposition. Here a service S_5 has been inserted, taking the place of S_2 for all calls from application B. Any calls that originate in application B (shown by dotted lines), as well as those made on its behalf by other services, are redirected through S_5. This is transparent to all other clients. In particular, there is no way for service S_1 to tell that it will be directed to another service – it continues to make the same calls, and the underlying communications infrastructure handles the destinations.

1.5 Thesis Organisation

The remainder of this thesis is organised as follows:

- Chapter 2 is a detailed survey of related work, and contrasts this work with Kea. In particular, the evolution of some of the important concepts are discussed, together with a comparison with their expression in other systems.

- Chapter 3 describes the low-level details of the Kea kernel architecture. It focuses on the organisation of the system as a whole, and provides details on the design and capabilities of the core services. In particular, the IPC mechanism and service manipulation functionality is discussed in detail. The special facilities needed for the unification of the user and kernel programming environments are also described. A section deals with the problems of protection, both in general terms, i.e. how it can be provided in a

decomposed system, and in terms of the special capabilities needed for the extensible and application specific features. Another section describes scheduling functionality, and how different scheduling code can be made, if not dynamically replaceable, at least easy to develop and configure into the system. The chapter concludes with a discussion of the performance of the base services.

- Chapter 4 describes some of the higher level services built using this architecture, concentrating on the design of the Kea filesystems and their supporting services. It then discusses how the design of these services both demonstrates and facilitates Kea's flexibility.

- Chapter 5 describes several experiments which measure the performance of the system for each of the developed services. These experiments allow an evaluation of the system reconfigurability to be made.

- Chapter 6 discusses the application of the Kea architecture to the problem of application specificity. Several experiments and their results are described and discussed. The chapter concludes with a discussion on the limitations of application specificity both for Kea specifically, and operating systems generally, and considers how these might be overcome.

- Chapter 7 concludes the body of the thesis, summarising the experimental results and main contributions of the research. The possibilities for future work arising out of the thesis are also discussed.

- Appendix A provides details on the interfaces to selected low level Kea services.

- Appendix B describes the interfaces to selected high level services.

- Appendix C details Kea's internal scheduler interface.

CHAPTER 2

Related Work

Several of the concepts on which Kea is based are similar to, or have been extracted from, a number of related systems. These can be (roughly) grouped into two different classes – systems that influenced ideas on operating systems structure and general extensibility, and those that were explicitly designed for application specific extensibility. This chapter describes influential systems in both areas, and examines where they differ from Kea.

2.1 System Structure

Since the advent of computer systems, the question of how best to structure the operating system has been of interest to systems architects and researchers. As described in the introduction, the default, monolithic kernel approach, in which all kernel code is built and configured into a single image, does not satisfy all the needs of modern systems for flexibility and extensibility. The key structural ideas that Kea is based on can be identified as:

- Fine grain modular construction

- Procedural IPC
- IPC optimisation when services are co-located
- Transparent service migration and replacement
- The orthogonality of protection, modularity and security.

The following sections examine other systems that incorporate some of these, or similar, ideas and compare those systems with Kea, pointing out both similarities and differences in approach or structure.

2.1.1 Multics

The Multics system [Organick 72, Corbato et al. 72] was extremely influential on systems design. Using special hardware, the GE 645, Multics was a system that depended heavily on segmentation. A Multics process consisted of a collection of segments, each of which had its own protection attributes, and could be shared with other processes. Thus, the Multics system let designers build system software as a set of segments, which were linked into applications as necessary. Every segment belonged to a protection ring, with inner segments (low ring number) protected from outer segments. The boundary between user and kernel mode was blurred, with only a few "supervisor" segments at ring 0 able to execute privileged instructions. Other system segments provided functionality such as filesystems, and the application segments were composed with these.

There are two aspects of Multics that are pertinent to Kea. The first is the construction of the system as a set of segments. This is superficially similar to the Kea philosophy of system construction as a set of cooperating services. The major difference is in the binding between system components. In Multics, this binding is completely static – the number and types of segments comprising the system are fixed at the time the system is compiled and linked. In Kea, service binding is completely dynamic. Services can be manipulated in a number of different ways at run-time. Additionally, the protection in Kea is active, being based on address space location,

and the current protection given to a service. Multics segments are statically configured with a protection ring, which cannot be changed. Finally, Multics relied on specialised hardware to support calls between segments and enforce segment protection boundaries, whereas the Kea system is architecture neutral.

The second major contribution of Multics was that there was no real differentiation between user and kernel modes, and in particular, the programming environment was the same for all developers. "It is worth reemphasizing that the only differentiation between Multics systems programmers and user programmers is embodied in the access control mechanism which determines what on-line information can be referenced; therefore, what are apparently two groups of users can be discussed as one" [Corbato et al. 72]. This simple, yet powerful, concept has been revitalised in Kea, where it offers significant development advantages. Concurrently with Kea, this idea has also been developed in the Rialto system [Draves & Cutshall 97], where significant software engineering advantages have also been observed.

2.1.2 Hydra

Hydra [Wulf et al. 74, Levin et al. 75, Wulf et al. 81] is, in many ways, the ancestor of both object-oriented and microkernel systems. In particular, Hydra was one of the first systems to experiment with the modular construction of operating systems, the "small kernel" idea, and the separation of policy and mechanism. This was accomplished by separating functionality into different "objects", and providing a capability based call mechanism between these objects. The call mechanism was very costly however, and, like Multics, Hydra was statically configured, which severely limited both its flexibility and extensibility.

2.1.3 Microkernels

While the "microkernel" description can cover a wide variety of systems, the general usage is generally presumed to mean a small kernel, exporting a minimal set of functionality, on which various higher-level "servers" can be built. Servers are developed in separate address spaces,

and communicate using RPC. While the microkernel design facilitates the construction of a modular system, there are two important limitations with such a system with respect to extensibility. The first is that even the most aggressively optimised systems still incur a performance overhead with respect to the equivalent monolithic systems [Härtig et al. 97]. The second is that, typically, the number of servers implemented are few, and of large granularity. For instance, typical Mach based systems only implement a single server, which provides the entire range of normal operating system functionality [Golub et al. 90]. In general, the greater the number of servers, the worse the performance, due to the increased number of IPCs between servers. This may account for the fact that, to date, only one microkernel based system, QNX [Hildebrand 92], has achieved commercial success. However, several microkernel based systems, primarily Mach and Chorus, implement techniques that partially overcome these problems.

2.1.4 Chorus

The Chorus system [Guillemont et al. 91, Rozier et al. 92, Walpole et al. 92] supports supervisor tasks that can be co-located in the kernel address space. IPC between these tasks can be optimised to procedure calls, but this is not transparent to the code, and must be explicitly specified when the system is configured. While co-location gives Chorus some of the advantages inherent in Kea, the static nature of the system reduces both its flexibility and extensibility. Additionally, supervisor tasks have access to additional kernel interfaces, that are not visible to other tasks. Coupled with the lack of transparency for IPC optimisation, the location transparency of Chorus servers is far less than that of Kea services.

2.1.5 Mach

Several versions of Mach support kernel co-location of privileged subsystems. In-kernel servers [Lepreau et al. 93], allows servers to be loaded into the kernel at run time. When this is done, calls between client and server are optimised from the high overhead RPC mechanism to use trap-based calls, which can execute substantially faster. There are several differences

between this system and Kea. Firstly, and most importantly, there is no optimisation of server-server or server-kernel interactions. Secondly, this work concentrates on improving the efficiency of the existing Mach system, unlike Kea, where the focus is on providing a complete infrastructure for extensibility.

Another project [Condict et al. 94] also extends the Mach system, allowing server co-location in the kernel, and using a form of optimised RPC (although not direct procedure calls, as in Kea). Also, server to kernel calls are not optimised, as the Mach kernel continues to require certain calling conventions, unlike Kea, where all kernel interfaces have exactly the same calling semantics as other (non-kernel) interfaces. A further difference is in the RPC semantics. This system builds RPC out of one-way messages, and optimises for this, whereas Kea assumes a round-trip procedure call, and provides this facility directly. Finally, while the issue of optimising server-server interactions is addressed, this facility has not been tested, as the system implemented uses only the OSF/1 single server.

Neither of the systems described above support the arbitrary co-location and grouping of services, but are only concerned with locating servers in the kernel space. While this supports some common configurations, it may not suit others, where a system developer or administrator may want to place certain groups of servers in independent address spaces for reliability or testing purposes. Finally, these systems to not have any of the dynamic placement or migration features of Kea.

2.1.6 Lipto

Lipto [Druschel et al. 91, Druschel et al. 92a, Druschel et al. 92b, Druschel 93] is perhaps the system most like Kea in its design. Lipto is an object-oriented system, and the servers implementing objects can reside in any configuration of address spaces, including the kernel. When object servers are co-located, the object invocation mechanism detects this, and optimises the object invocations to use a local stub and indirection mechanism, resulting in near procedure call efficiency. Other than the minor difference of the RPC mechanism being object based,

rather than procedural in nature, there are two major differences between Kea and Lipto. The first is the level of granularity at which RPC optimisation takes place. In Lipto, the unit on which sharing takes place is the object. For each object reference obtained by a client, the system has to provide an appropriate (either local or remote) proxy object. Kea optimises not on objects, but at the interface level – that is, one layer of abstraction higher. This eliminates some overhead due to proxy objects, and also allows the system to optimise directly on the interface's procedural entry points.

The second major difference is that Lipto's configuration is statically determined at boot-time, and is not dynamic like Kea's. It should also be noted that the Lipto project only progressed to the implementation of a prototype of the object invocation method. Kea is the first system to apply the principle of fine-grain decomposition within a fully functional operating system. Despite these differences, Liptos' authors deserve recognition for first observing the orthogonality of modularity and protection in operating system construction.

2.1.7 Spring

Spring [Hamilton & Kougiouris 93, Khalidi & Nelson 93a] is a distributed operating system that represents all system resources as objects, and supports a highly efficient IPC mechanism that supports object invocation. Like Kea, the system is microkernel based, with higher level services implementing functionality such as filesystems and networking. These services can be loaded into any address space, dependent on boot-time operating system settings. The Spring IPC mechanism has semantics very similar to that of Kea's, and a very efficient implementation. For small arguments (16 bytes or less of scalar data) it takes advantage of the SPARC calling mechanism, where parameters are passed in registers. For larger arguments (up to 5 Kb of data and capabilities), the "vanilla" path is used. A third, "bulk", path uses virtual memory remapping to transmit large data objects. The Kea IPC mechanism was largely based on the semantics of Spring's, but Kea provides only a single method of data transmission. While supporting a bulk data copying method within Kea IPC would considerably enhance the perfor-

mance for some applications, this possibility has currently been relegated to future work. The major limitation of Spring with respect to Kea is that Spring only supports a limited form of service co-location. While objects can be created in the same address space, the IPC mechanism is still used for invocation. While this does eliminate the cost of an address space switch, the full performance available from the automatic short-circuiting of calls is not available. This facility is vital to the Kea philosophy of giving the system designer the choice of safety or performance. Also, the Spring IPC mechanism does not directly support true procedure calls, but still relies upon argument marshalling, which incurs a further performance penalty. Finally, Spring doesn't support the migration and replacement of services, features which are necessary for dynamic system reconfiguration.

Spring also has support for system extensibility through object composition. This is discussed in section 2.2.7 on page 34.

2.1.8 Single Address Space Systems

Encouraged by the increasing availability of 64-bit processor architectures, a number of systems have been implemented that investigate the possibilities of supporting an operating system and its applications within a single address space [Chase et al. 93, Bartoli et al. 93, Heiser et al. 93]. While these systems promote code sharing and modularity, and consequently enhance the flexibility of the operating system itself, none of the existing systems have been using this for the investigation of application specificity. Also, these systems are, to date, all statically configured with regards to kernel/user boundaries and services.

An ancestor of the single address space systems, Psyche [Scott et al. 88] is similar to Kea in that it is composed of a set of modules. Psyche uses a single virtual address space to facilitate sharing between modules, trading protection for performance in much the same way that Kea does. However, Psyche is statically configured, and does not provide a means for arbitrary reconfiguration or module replacement.

2.1.9 Protected Shared Libraries

Protected shared libraries (PSLs) [Banerji et al. 97] have been proposed as a means of efficiently enhancing the modularity of operating systems. PSLs extend the notion of shared libraries [Gingell 89] to include protected state (i.e. information that can only be changed from within the library code, not from the library client) and data sharing across protection boundaries. When calling a library routine within a PSL, a "partial address space switch" is performed, which maps (or unmaps, as appropriate) data shared between the library and the caller. It is argued that by using PSLs, operating systems can be made more modular, while increasing system protection. While this may be true, there is no essential difference between PSLs and any other structuring mechanism that offers data protection (such as those described above), and no facility for co-location when the library code is trusted. The latter ability is important in order to obtain the performance demanded by system clients, especially since calls into a PSL are just as expensive as any other IPC mechanism that must change address spaces.

2.1.10 The Flux OSKit

The Flux OSKit [Ford et al. 97] is a project that is designed to enable researchers to easily *build* operating systems – it does not necessarily define the structure that the operating system has to take. The OSKit provides a number of OS components, such as file systems, device drivers and virtual memory systems that can be used in a "building block" fashion to build a complete OS. While not explicitly defining[1] an OS structure, the OSKit is relevant to Kea in that its components are analogous to Kea services, albeit statically configured. The OSKit, in parallel with Kea, is one of the first systems to explicitly explore OS construction as independent[2] subsystems, rather than dependent modules.

1. However, we would argue that a structure is implicitly defined by the nature of the components provided (all derived from Unix). Unfortunately, this is hard to avoid, and is partially true for Kea as well.

2. or as independent as possible. Whenever a service is used by another, there is a dependency.

2.1.11 Conclusions on Structure

All of the systems discussed above share some concepts with Kea. In particular, the necessity for the modular decomposition of operating systems has long been recognised, as has the trade-off between modularity and protection, which different systems have addressed in different fashions. While not unique in its construction as a set of cooperating services, Kea is the only system developed to date that fully optimises the IPC between services when they are co-located. Kea is also the only system that supports true procedural IPC and the transparent migration and replacement of services at run-time, features necessary for full extensibility. The final comparison to be made with other system structures is that, as far as possible, Kea explicitly avoids a set security policy, recognising that this is orthogonal to both the protection and modularity of the system. To summarise, while Kea does borrow heavily from the best ideas inherent in other systems, particularly Spring and Lipto, it also has additional functionality that makes it uniquely suited to the construction of a reconfigurable system.

2.2 Application Specific Extensibility

How best to build a system that explicitly allows applications to insert code into the kernel has been an active research topic in recent years, with several different methodologies explored. Equally important to the question of how code is inserted into the kernel, is the problem of assuring that the code can be trusted, and will not compromise either the system security or performance. During the development of Kea, it was decided that this question was being adequately solved by other researchers (the systems demonstrating this are discussed in sections 2.2.2 through 2.2.4). Consequently, a final decision on which method(s), were appropriate for Kea was left open. The remainder of this chapter examines the techniques developed for code injection, and some of the representative systems using these methods, and compares them to Kea. Kea security is discussed later in section 3.12 on page 60.

2.2.1 Single User Systems

Many personal computer operating systems, e.g. MS-DOS or Windows, run both applications and the operating system in a single address space, which provides good performance, and gives applications free reign to arbitrarily modify all system code and data [Schulman et al. 92]. There are many disadvantages to such systems. There is no well defined interface through which modifications can be made, the system is not constructed so as to make arbitrary replacement of a particular service obvious, and the system has absolutely no protection from a buggy or malicious application, and no way of testing for this before it is used.

2.2.2 SPIN

SPIN [Bershad et al. 95, Pardyak & Bershad 96, Sirer et al. 96, Hsieh et al. 96] has been developed to explicitly support application extensibility. SPIN allows applications to install low-level system services into the kernel, so that they can receive notification of, or take action on, certain kernel events. These events may be as disparate as a page fault, the reception of a network packet destined for the application, or context switches affecting the application. Kernel extensions are written in Modula-3 [Nelson 91], a type-safe, modular programming language. Due to these language properties, the compiler can enforce the safety of extensions at compile time. In particular, the compiler can guarantee that any extension will not reference any memory outside the module boundaries, except that for which it has been granted explicit access through an interface. By performing compilation at run-time, it also becomes possible to perform various optimisations on the code produced, further enhancing the efficiency of the system [Chambers et al. 96]. Modula-3 and compile-time checks enable the use of pointers as capabilities, which avoids expensive run-time security checks inherent in other capability systems, whether hardware [Carter et al. 94] or software [Wulf et al. 81, Black et al. 92] based. SPIN's developers have demonstrated that this architecture is both efficient and practical to use in a number of cases. A possible problem with the SPIN system is with the event system. There may be semantic difficulties when multiple handlers are installed on a single event. While SPIN has features that can control whether handlers execute asynchronously or synchronously,

which order they execute in, and which ones can return a result, it is certainly conceivable that multiple applications might install conflicting handlers.

There are several differences between the SPIN and Kea systems. Firstly, the approach to code installation is fundamentally different. Where Kea specialisation is based around the idea of interfaces, and uses interposition and/or composition on those interfaces for specialisation, SPIN effectively uses single procedure remappings (events can be associated with any procedure invocation). This makes SPIN extensibility even more finely grained than Kea, but may also make some extensions more difficult to develop, where the functionality of several parts of a single interface need to be managed as a single extension, although this will only be able to be evaluated after many different types of extensions have been developed. As an application extensible system, SPIN is definitely more mature, and arguably more functional that Kea. SPIN does not however have any functionality supporting dynamic system reconfiguration or global extensibility (although it is possible that the same technology could be extended in these directions).

2.2.3 Vino

The Vino operating system [Seltzer et al. 96] has also been explicitly designed to support application specific extensions. Vino is object-oriented, and allows applications to dynamically insert code that replaces the implementation of a method on any object. Alternatively, code can be installed to run when a specified kernel event (such as a network packet arrival) occurs. The granularity at which the system can be extended is dependent upon the objects and events exported by the system. Like other systems, Vino is concerned with the safety of the downloaded code, and takes two measures to prevent this code from damaging other applications or the system.

The first safety technique used is software fault isolation [Wahbe et al. 93, Small & Seltzer 96] or "sandboxing". Software fault isolation is a method whereby binary code is checked, and possibly modified, to ensure that any memory references generated by that code will be within cer-

tain boundaries (the sandbox). This can be done at either compile-time, or load-time, and ensures that an object can, at worst, only affect its own operation. The performance overhead associated with the generation of sandboxed code is relatively small, typically on the order of a few percent, although it is arguable that even this may be significant for some systems.

To prevent denial of service problems (caused, for example, by an extension that obtains a lock and doesn't release it), Vino also employs a transaction-like system of extension invocation. Effectively, all extensions are wrapped with stub code, which treats all invocations of that extension as a transaction, and can "abort" the extension if it performs dangerous actions. This requires an expensive invocation service (over two orders of magnitude more than a procedure call), and also mandates that all kernel functions which change kernel state must have an equivalent "undo" operation.

The complexity and performance overheads associated with the Vino transaction mechanism prevent it from being as efficient as either SPIN or Kea. Also, like SPIN, Vino does not include any features for either dynamic system reconfiguration or global extensibility.

2.2.4 Exokernels

Another approach to extensibility is taken by the Exokernel project [Engler et al. 95, Kaashoek et al. 97]. Exokernel design relies on the implementation of a very low level kernel, which does nothing except export the base abstractions provided by the underlying hardware [Engler & Kaashoek 95]. The abstractions provided by an exokernel permit applications running on the system to protect their hardware resources, and if necessary, share them with other applications. The remainder of the operating system functionality is implemented within libraries (libOS's), which are linked into an applications address space, and which provide and manage higher level abstractions such as filesystems. Extensions to the system are made by applications changing libOS code. Exokernel designers have found it difficult to design systems that can safely multiplex physical resources, and, in the latest version of their designs, have found it necessary to provide a means by which small pieces of code, written in a

restricted language, can be downloaded into the kernel, in order to specify the types and protection attributes of disk blocks.

There are two major failings with the exokernel approach to extensibility. Firstly, the means by which applications specialise the system is static (extensions must be compiled into, or with, the appropriate libOS and/or application). Secondly, the extension mechanism is not well specified. In particular, there appear to be no well defined interfaces through which extensions may be carried out. If application designers do not wish to develop directly on the base exokernel interface (i.e. develop a new libOS), then they need full knowledge, and code for, the existing libOS. In contrast, Kea relies on explicitly exposing the service interfaces comprising the system, and making explicit the mechanisms through which they can be manipulated.

2.2.5 Meta-Object Systems

Several systems, e.g. Pi [Kulkarni 93] and Apertos [Yokote 92], have proposed the use of an object-oriented operating system that uses the principles of reflective metacomputation [Maes 87, Kiczales et al. 91, Kiczales et al. 92] to provide a means through which the objects composing the system can be modified. These systems are constructed from fine-grain objects, representing fundamental system resources, and use the composition of these objects to build larger services. By providing a metasystem for manipulation of objects by applications, the developers argue that they can build incrementally modifiable systems. While the goals, and ultimate result of these efforts may be quite similar to that of Kea, the methods by which they are achieved are quite different. Also, the systems as designed use a much coarser breakdown than the Kea design, and it is anticipated that the overhead for supporting metaobject specialisation at the kernel level will be substantial. Finally, these systems only support a limited form of extensibility, as only existing objects can be modified – to add entirely new objects, the systems must be entirely rebuilt.

2.2.6 Synthetix

The Synthetix [Cowan et al. 96] operating system, and its ancestor Synthesis [Pu et al. 88, [Massalin & Pu 89] provide enhanced application performance through run-time generation and optimisation of code which interfaces to operating system services. This is transparent to the applications using the system, and results in better performance, even if the application developer is unaware of this possibility. The techniques learnt from these systems have also been applied to a commercial operating system [Pu et al. 95]. In contrast to other extensible systems, Synthetix only makes extensions at the top layers of the operating system services, and does not provide any means by which applications may control their own resources. A further limitation is that the system for generating extensions can only optimise for those cases that the designers foresee – there is no way in which the future requirements of applications can be anticipated. A valuable experiment would be an evaluation of the Synthetix technology in the context of another extensible system, where the best of both explicit and automatic extensions could be applied.

2.2.7 Spring

The Spring system (previously discussed in section 2.1.7) is able to dynamically extend the system by layering new object servers upon existing ones, although this has only been explored in the context of file systems [Khalidi & Nelson 93b]. Kea extends this capability by not only letting services be interposed, but also replaced and migrated. Additionally, it is possible to perform these actions on an application specific basis, and at any layer in the service hierarchy.

2.2.8 Other Approaches and Systems

Other approaches, such as interpreted code, have been suggested for system extensibility, but have been found to be very costly in terms of performance [Small & Seltzer 96]. Many researchers have investigated various facets of extensibility in restricted domains, such as file systems [Rees et al. 86, Bershad & Pinkerton 88], virtual memory management [Lee et al. 94, McNamee & Armstrong 90, Appel & Li 91, Harty & Cheriton 91], disk buffer caching

[Cao et al. 94], scheduling [Anderson et al. 92], I/O subsystems [Fall & Pasquale 94], the user/kernel interface [Jones 93] and networking [Mogul et al. 87, McCanne & Jacobsen 93, Maeda & Bershad 93]. Each of these systems demonstrates the applicability of letting applications perform their own resource management. Kea represents an attempt to develop a system in which these ideas can be applied in a single coherent manner.

2.2.9 Conclusions on Application Specific Extensibility

Of the several other systems built for the explicit support of application specific extensions, some are arguably more functional, and certainly more mature, than Kea. The important differences are that, with Kea, the support of application specific extensibility is only one goal among many, and that the Kea architecture supports several other abilities that are unique. Also, Kea is the only implemented design that uses the explicit remapping of interfaces and an application specific IPC mechanism for the support of application specific extensibility. Further experimentation and evaluation of this architecture is necessary to validate (or repudiate) its suitability for this purpose.

2.3 Summary of Related Work

This chapter has examined many systems that are related to that of Kea in terms of their structure or function. Kea has borrowed concepts from these systems where appropriate, and extended them where necessary or desirable, in order to build a configurable system. It has been shown that Kea is the only system that offers completely optimised cross-address-space communication for service co-location. Kea also has a different model of extensibility than other systems. Most importantly, it is unique in its ability to make service reconfiguration a dynamic operation, rather than a static one.

CHAPTER 3

The Kea Architecture

This chapter describes the low level details of the Kea architecture. These features, particularly service manipulation, are the fundamental basis around which the systems reconfigurability is based, and form the core of the thesis work. Later chapters will build on this knowledge in order to evaluate the system features.

In Section 3.1, the chapter discusses the philosophy and organisation of the system. Sections 3.2 through 3.6 then detail the fundamental kernel services. Sections 3.7 through 3.10 cover the specification, creation and manipulation of services, while Sections 3.11 through 3.13 examines some of the problems associated with building a decomposed system, and the specific solutions used in Kea. The chapter concludes with a summary of the implementation effort in Section 3.14 and a discussion of the system's performance in Section 3.15.

3.1 Kernel As Services

As intimated in Chapter 1, the Kea kernel is not viewed as a single monolithic entity, but as a cooperating group of services. Each of these services has a well defined interface, which applications and other services use to access the resources provided by the service. Through a process of composition, more complex high level services are built up from primitive, low level services. The questions immediately raised by this structure relate to the definitions of the primitive services. What should they be? What level of abstraction should they provide? How many of them should there be? There are several possible answers to these questions. The thesis provides one possible set of answers, but this is by no means the only possible set, and is certainly capable of improvement. The thesis helps define some of the characteristics of this set, and improvements in some areas are suggested in the section on future work in Chapter 7.

Many researchers, particularly those involved in microkernel research, espouse the view that the core of any system should be a small nucleus that exports only the base abstractions provided by the underlying hardware [Cheriton & Duda 94, Engler & Kaashoek 95, Härtig et al. 97]. On typical systems, this essentially reduces to four subsystems. The first would be a facility for creating address spaces and installing virtual to physical memory mappings, the second a method for changing the processor context, the third a means of capturing interrupts and traps, and finally an IPC mechanism, so that services can communicate with each other. While these capabilities are sufficient for the implementation of high level services, we instead chose to implement services that offered more complex functionality. We reason that any meaningful operating system has to provide higher level instantiations of these services (e.g., unless the system is only ever going to execute a single process, there has to be some true scheduling support, using the low level context switching interface), and decided that, for the prototype, we would concentrate on providing core services that would be applicable to, and needed for, any operating system design. In each case, these services do in fact rely on low level interfaces, but we have chosen not to make these available as services to the rest of the system, believing that their functionality is subsumed by the services actually provided.

We have also implemented these systems to be policy free, or where this is impossible to achieve, carefully separated the modules implementing policy from those providing mechanisms, allowing easy replacement. The exception to these statements is the virtual memory system. It is certainly possible to provide a substantially different model of virtual memory behaviour, as has been demonstrated by the proponents of single address space systems [Chase et al. 93, Bartoli et al. 93]. However, developing different virtual memory systems would also require fundamentally different sets of higher level services and as we only wished to build one set of services, it was not deemed feasible to address this with the current research.

The majority of the remaining sections contain descriptions of each of the lowest level services currently provided by the Kea system, with special emphasis on the facilities provided for service manipulation (the "service service"). Unless otherwise specified, full details on the interface to each of these services can be found in Appendix A.

3.2 Domains

Domains are the virtual address space abstraction provided by Kea. A domain defines the set of virtual addresses which are valid for threads executing within the domain. The physical memory pages mapped by domains can be arbitrarily shared, mapped as copy-on-write, or copied between domains. The various parts of a domain's memory can have different protection attributes, such as read only or execute. Non-protection related functionality includes the ability to lock parts of the address space in memory. For device driver support, memory can be mapped at a specified physical address. Kea has not innovated in this area, except to capture the interface to virtual memory as an interface. There are two separate interfaces which implement the domain functionality. The domain interface itself allows for the creation and manipulation of domains, with the exception of the functionality for manipulating the virtual memory of an individual domain – this is done through the "vm" interface. Complete details on these interfaces can be found in Appendices A.1 and A.2 respectively.

Domains are the key security abstraction within Kea – all other resources are "owned" by a given domain. This is a consequence of the service-centric viewpoint taken by Kea, where services can execute in their own domains, and the design of the IPC system (discussed in Section 3.6). The implications and management of security are discussed in Section 3.12.

3.3 Events

The *events* abstraction defines the means by which asynchrony is controlled. Under Kea, a domain can register to receive notification of events that are significant to that domain, or that it has an interest in. Events subsume the interrupt mechanism, as all interrupts are turned into events. This permits the implementation of device drivers as separate services, and frees them from any restrictions on their placement (in particular, they need not be in the kernel). Another use of events is the notification of processor generated traps such as page faults and illegal instruction faults. Certain predefined events are also provided by other parts of the system, such as the domain management service, which signals the death of a domain whenever this occurs. Complete details on the event interface can be found in Appendix A.3.

3.4 Names

For convenience, Kea includes a simple name system. This system lets arbitrary integer identifiers be attached to names. The identifiers can be used to represent other system objects, particularly domains, threads (see Section 3.5) and services (Section 3.7). Thus, for instance, instead of having to know the name of a service, clients can look it up under an appropriate name, such as "/system/service/file". The name service also allows other services implementing a name interface to attach themselves to any point in the name hierarchy. Naming in operating systems is a very complicated issue [Radia 89], and is made more interesting in Kea by the system's configurability. While it would be interesting to explore naming to a greater extent than has been possible, the simple system provided has proved to be adequate to the basic needs of system evaluation. The name interface is described in Appendix A.4.

3.5 Threads

Threads in Kea are superficially similar to the concept of threads in other operating systems: from the applications point of view, they are simply a context in which code execution takes place, or a "sequential flow of control" [Birrell 89]. In most operating systems, a thread is more than just a flow of control, but also encompasses the notion of schedulable entity, the "thing" to which CPU time is allocated. Kea separates these notions, dividing a thread into two entities. The first, which we continue to refer to as thread, is the construct to which the scheduler allocates CPU time. The second, referred to as an *activation*, contains all the thread state, in particular the current domain, register contents, user and kernel stacks. One thread can have multiple activations associated with it, organised as a stack, of which only the activation at the top of the stack is active. This thread organisation, and the thread semantics described in the remainder of the thesis, closely follows that of Spring [Hamilton & Kougiouris 93] and the "migrating threads" developed for Mach 3.0 [Ford & Lepreau 94] (from which, in the interests of reducing confusion and the growth of new jargon, the activation term is borrowed).

The primary reason for structuring threads in this manner is that it facilitates the design of the Kea IPC mechanism. This mechanism, including the details of its implementation and relationship to the thread model, is discussed in the following section. A description of the thread interface can be found in Appendix A.5.

3.6 Inter-Domain Calls

Conventionally, decomposed systems use message passing to communicate. As described in the introduction, we believe that this is fundamentally the wrong paradigm, and that direct operating system support for procedure calls between address spaces is superior. We refer to such a procedure invocation as an *inter-domain call*, or IDC. Other systems, such as LRPC [Bershad et al. 89] and Spring have claimed to support this paradigm, but still rely on the marshalling of procedural arguments into a single contiguous buffer, usually accomplished within

an automatically generated stub procedure. The IDC mechanism instead relies on the natural argument layout, usually on the stack and/or in registers, of the architecture on which it is running. There are several facets to the implementation of IDC. The primary entity representing a potential IDC is the *portal*, while the IDC itself is accomplished by a *portal invocation.*

3.6.1 Portals

To perform an IDC, a service client needs to have some means by which to refer to procedures within a service. This is accomplished through the use of portals. A portal appears to the client as an integer identifier. In order to make an IDC, the client uses the system call `portalInvoke()`, which takes the portal identifier as its first argument. This call is interesting for several reasons. It is the only system call in Kea – every other service, including the low-level kernel provided ones, is accessed through this call. It has the following prototype[1]:

```
int portalInvoke(int portal, int *ret, void *args);
```

Where `ret` is a pointer to the return value (if any) from the procedure to be called and `args` is a pointer to the location of a buffer containing the call arguments. It is the `args` variable that is machine dependant in it's meaning. On an Intel x86 machine, it will be a pointer to the stack location holding the arguments. On a SPARC based machine, the first six arguments are passed in registers, and `args` will point to the remainder (if any) on the stack.

During a portal invocation, the following actions occur:

1. The portal identifier is used as an index into a system table to obtain the destination domain and entry point.
2. A new activation is created.

1. It should be noted that, for clarity, some of the prototypes and data structures described have been simplified slightly. By far the most common change is to the type of some variables or parameters, to avoid the need to include the appropriate typedefs. These changes do not affect the semantics at all. An example is the replacement of the Kea vaddr (virtual address) type with "void *". The definitive code is shown in the appendices.

3. The new activation is pushed onto the current thread's activation stack.
4. Execution continues in the target domain, at the designated entry point.

The portal invocation process is illustrated in Figure 3.1.

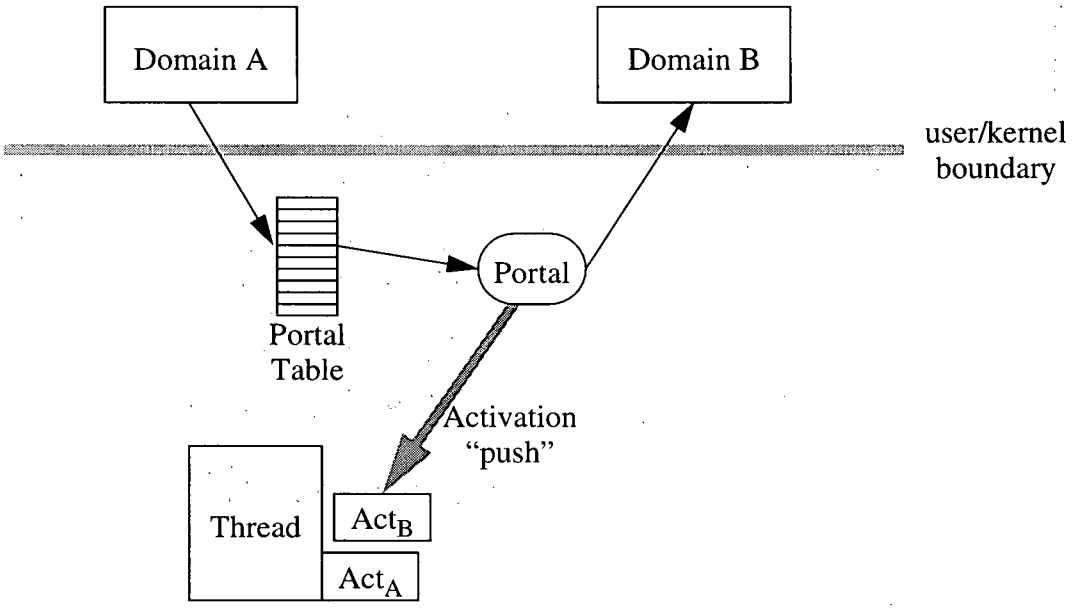

Figure 3.1: Portal Invocation
Domain A calls Domain B

Returning from an invocation is accomplished by placing a special marker value at the top of the called domain's stack. When the called procedure returns, this value causes a page fault at the specified address. This fault is interpreted by the kernel to mean that the current activation has finished, and the internal portal return code is called. This code "pops" the current activation off the thread's activation stack, copies any return arguments from the called domain to the caller, and returns control to the calling domain.

When portals are created, the creator must specify the signature of the procedure that will be invoked by the portal. The signature is the number and type of arguments that the procedure expects. The argument types are limited to simple scalar data (integers, characters and floating

point numbers), pointers to single structures or scalars, strings (pointers to zero terminated character strings) and pointers to arrays of structures or scalars. In the latter case, the argument after the pointer must be an integer, which the caller must initialise to be the number of elements in the array. Portals are created with the `portalCreate()` call, the prototype for which is:

```
int portalCreate(
        struct domain *domain,
        void *entry,
        struct portal_signature *signature,
        unsigned pssize,
        unsigned nargs
);
```

The arguments to this call are the domain in which the entry point exists, the entry point itself, an array of descriptors representing the signature of the underlying procedure, the number of signature entries, and the total number of arguments that the procedure has. The `portal_signature` structure has the following definition:

```
struct portal_signature {
     int ps_arg:4;        /* argument index */
     int ps_type:2;       /* argument type */
     int ps_modifier:2;   /* modifier flags */
     int ps_length:24;    /* number of bytes */
};
```

In this structure, `ps_type` is used for one of the three pointer types described above (pointer to single value/structure, pointer to array or string), `ps_modifier` determines whether the argument is to be copied to the callee (in), from the caller (out) or both (in/out), `ps_arg` determines which argument in the procedure is referred to and `ps_length` describes the size of the element(s) to be copied. The definitions used for these values are:

```
/*
 * values of ps_type
 */
#define PT_PTR_ABS  0x0  /* absolute pointer */
#define PT_PTR_MULT 0x1  /* array pointer    */
#define PT_STRING   0x2  /* null terminated string */

/*
 * bits in ps_modifier
 */
#define PM_IN  0x1       /* copy in */
#define PM_OUT 0x2       /* copy out */
```

It should be noted that the signature entries do not include any scalar arguments, but only specify the types of the pointer arguements (if any) for the procedure.

The use of signatures results in a slightly more restrictive viewpoint of procedure calls, in that arrays of greater than one dimension or data structures containing pointers (such as lists) cannot be passed by the IDC mechanism. In practice, we have found that these capabilities aren't necessary. We also believe that the addition of such a capability would result in a considerable performance degradation, due to the necessity for parsing any argument description in the kernel, and might also result in services that were too interdependent.

As an example, consider the following function prototype:

```
void foo(int bar, char *str, int *stuff, int nstuff);
```

Where `str` is only needed to be passed to the callee and `stuff` is presumed to be a pointer to `nstuff` integers, and contains information that is read and changed by the callee. The signature array describing this would be:

```
struct portal_signature fooSignature[] = {
     { 1, PT_STRING, PM_IN, 0 },
     { 2, PT_PTR_MULT, PM_IN | PM_OUT, sizeof(int) }
};
```

For each portal, the kernel keeps the descriptor containing information on the number and types of arguments used by the procedure, and uses this information in the invocation code, in order to copy arguments between the two address spaces. Thus, the kernel code itself effectively marshals arguments, instead of this being done by a user-level stub, as is the case in almost all other IPC/RPC mechanisms. In many instances, this can reduce the number of memory copies and results in a corresponding increase in efficiency.

3.6.2 Portal Remapping

Kea supports reconfiguration and extension through the remapping of portals. Remapping refers to the ability of IDCs using the same portal to be directed to different destinations. There are two types of remapping, *simple* and *domain specific*. Simple remapping is the simplest, and is almost trivial in its implementation. It merely changes the internal mapping between a portal and its destination. It is used when global reconfigurations are made, i.e. every client using the portal must now use a different service (for instance, if the original has been replaced).

Domain specific remapping is more complex. It is used for application specific modifications to a service, and results in portal invocations causing control to transfer to one of two (or more) different destinations, based on the domain owning the thread making the call. This is shown in Figure 3.1. In this figure, two applications, A and B, are using a service, S, through portal P_1. In turn, this service uses another service, S_P, through portal P_2, to make some policy decision. If A wants a different policy to be used, it can perform a domain specific remapping on P_2, and arrange for all calls that originate in A to be re-routed to a new policy service S_A. The two call paths are shown by the arrowed lines – solid lines for application B, dotted lines for application A. By using separate services for mechanism and policy (e.g. maintaining virtual memory mappings and page replacement strategies) the system becomes configurable in a large number of ways.

Domain specific remappings are implemented by giving each portal a table containing alternate portal identifiers. When a portal invocation is made, this table is checked, using the domain of

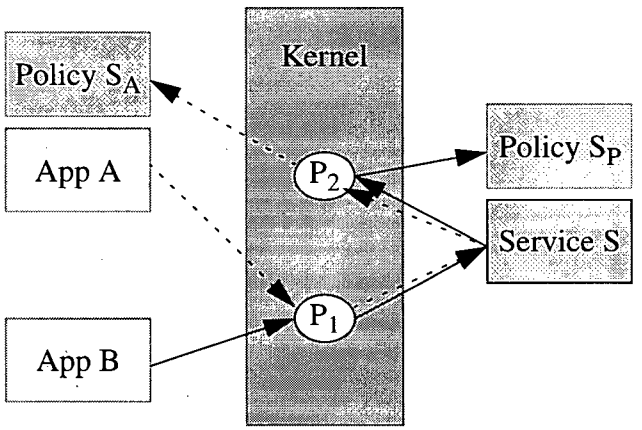

Figure 3.2: Domain Specific Remapping

the original caller (i.e. the activation at the root of the call stack) as the key. If an entry is found, then that portal is used instead.

3.7 Services

While portals provide a means through which individual service calls can be made, they do not provide the service structuring required by the Kea design. The kernel provides a service (the "service service") that is used for the description of services. This is accomplished by aggregating portals together: from the kernel's viewpoint, a service is essentially a set of portals. These portals are manipulated as a group, ensuring the consistency of the service – the programmer never has to be concerned about the underlying portal representation. In fact, portals are not visible to the user level programmer at all, except for the `portalInvoke()` call (which is normally "hidden" inside automatically generated stubs). In particular, all of the portal manipulation functions are only available from within the kernel (they are not exported as a service), forcing programmers to think in terms of services.

To create a new service, the service implementor must provide two things. The first is a set of source files that implement the service procedures. The second is an interface description file,

which describes these procedures, and in particular, the types and numbers of arguments each one takes. From the interface description file, it is relatively simple to generate the client side stubs[1] needed by other services and applications which access the service. When a service is compiled, the result is not an executable. Instead, all of the object files comprising the service are linked into a single object file, leaving all external references unresolved. This step is taken so that when the service is loaded, it can easily be linked against any other services or libraries already present in the domain.

To create a new service, the `serviceLoad()` call must be used. This call has the prototype:

```
int serviceLoad(
          int domain,
          char *name,
          void *code,
          unsigned codeSize,
          struct function_map *fnMap,
          int servSize
);
```

The arguments to `serviceLoad()` denote the domain in which the service code currently resides, the name to be applied to the service, a pointer to the service code and the code size (loaded into the domain from a service file), a descriptor for each of the procedures in the service, and the number of procedures in the service. The `function_map` structure used in `serviceLoad()` has the definition:

```
struct function_map {
     void *fm_entry;
     char fm_name[MAX_FN_LEN];
     unsigned fm_stacksize;
     struct portal_signature *fm_ps;
     unsigned fm_psSize;
     unsigned fm_nargs;
};
```

1. The stub compiler for the Kea system is not yet functional. This is mostly attributable to the ease of manual stub generation, and a desire to proceed with more interesting research issues.

The `function_map` specifies the entry point, name[1] and argument descriptors for each procedure. The name is used for dynamic linking and co-location purposes. Also included is an indicator of the stack space needed for the procedure to execute. It is intended that this parameter will eventually be removed, and that stacks will be dynamically growable, but this has not yet been implemented.

When a service is loaded, several different actions take place. Firstly, the service symbol table is read and verified. Then the service code and data segments are created, using the code pointer provided. The service code is then linked against any other library code resident in the domain. Typically, this code is the result of other services or applications that have already used the domain. A symbol table is maintained for each domain to facilitate this. External symbols available to the service are restricted to those loaded from libraries. Where a symbol cannot be resolved, it is located and loaded from the libraries available to the domain, and the new symbol added to the domain's symbol table. If the symbol cannot be resolved, the service load will fail. In the final stages of the link phase, operations such as text and data relocation take place. From the `function_map` information provided, the kernel next generates each of the portals that will be used by clients of the service. Associated with the internal service table, the kernel keeps separate pointers to the code and data segments, the symbol information for the service itself (as opposed to the domain's symbol table) and any associated relocation information. This ensures that the service can be efficiently relocated into another domain if necessary. The penultimate stage of loading a service is the creation of a new thread that will run the service initialisation routine, allowing the service to set up its internal state. The service initialisation routine has the name formed by appending the string "ServiceInit" to the service name, e.g. the "foo" service would have the function "fooServiceInit" called, if it existed. Once the initialisation function has completed, the final stage in service loading is the insertion of the service structure into the kernel service table, making it available to other users of the system.

1. One of the entry point and name could be eliminated as, given one, the other can be deduced from the services symbol table. This is currently not done, as it simplifies some code, and also provides a check against badly specified service definitions.

An important point in the loading of services is that all of the memory used by the service is mapped by the kernel in such a way that it cannot be deallocated, even by the owner of the domain. This applies to the service symbol table, code and data, and prevents other services, or applications, with which the service may be co-located, from interfering with the service. For similar reasons, the service code and symbol table are also made read-only.

3.8 Service Acquisition

Before using a service, a client must first acquire it, but before discussing the details of how this is accomplished, it is necessary to examine the client's view of a service. The only part of the service normally visible to the client developer is a function pointer for every procedure in the service. Function pointers have the advantage of being able to be used in a syntactically identical manner to function calls, which enables the usage of these pointers to be hidden from the client developer. More importantly, the kernel can change these pointers to refer to a different entry point, which is a necessary prerequisite to supporting service co-location, migration and replacement. Hidden from the developer is an array of integers, which is used to represent each of the service portal identifiers, and two stubs for each of the service procedures. The function pointers can refer to one of the two stubs, which are automatically generated by the stub generator. One stub (the portal stub) is used when the client is in a separate domain from the service, the second (the co-location stub) when it is co-located. The first of these stubs is very short, essentially containing only the minimum overhead necessary for a portal invocation. For example, the portal stub for the `foo` function described earlier might, for an architecture in which all parameters are passed on the stack, have the following code:

```
void fooStubPortal(int bar, ...)[1] {
     portalInvoke(portals[FOO_PORTAL], NULL, &bar);
}
```

Co-location stubs are slightly more complex. The essential task that has to be accomplished is a call to the co-located code, but this is complicated by the need to support service migration

1. For clarity, additional parameters are denoted only by ellipses.

and replacement. To explain this in the appropriate context, discussion of the structure of this stub is deferred to Section 3.10.1.

Services are acquired by using the `serviceAcquire()` call. This call has several purposes. It allows the kernel to keep a record of clients, which is important for reconfiguration. It also allows the security service to check whether the domain trying to obtain the service is allowed to access the resources managed by that service. For the client, it provides a means through which it can obtain the portal identifiers needed to invoke the procedures making up the service. In the currently existing system, calls to `serviceAcquire()` are contained in code automatically generated with the client side stubs from the interface description file. The prototype for `serviceAcquire()` is:

```
int serviceAcquire(
                char *name,
                int domain,
                int *portals,
                struct serviceStub *stubs,
                unsigned nstubs,
                struct serviceCount *count
);
```

Given a service name, the domain which is acquiring the service, a pointer to portal identifiers and an array of stub descriptors, the `serviceAcquire()` function determines whether the domain is allowed to acquire the service, and if so, initialises the portals to those needed for access to that service. The `serviceStub` structure is used to pass information on the function pointer used to access the client stubs, and the addresses of the stubs themselves. It has the definition:

```
struct serviceStub {
     void *ss_function;
     unsigned ss_stubPortal;
     unsigned ss_stubColocate;
     void *ss_colocateFn;
};
```

The `ss_function` pointer is used to pass the address of the function pointer used to access the service procedure. It will hold one of the two values passed in `ss_stubPortal` (the address of the portal stub) or `ss_stubColocate` (the address of the co-location stub). The `ss_colocateFn` field is used to support service co-location, migration and replacement, as is the `count` parameter to `serviceAcquire()`. The need for, and use of, each of these will be described in Section 3.10.

When a service is acquired, the kernel also checks to see if the service domain is the same as that of the domain acquiring the service. Based on the result of this test, it adjusts the client function pointers to refer to the appropriate stub.

3.9 Internal Service Structure

From the previous sections, it should be clear that the kernel keeps detailed information on each service, the relationships between services, and the domains containing and using services. This information is summarised in Figure 3.3. The internal service table records, for each domain, a table of the globally visible symbols, a list of the client domains for that service (this table also includes the addresses of the function pointers used to access this service), and the portal identifiers for the service. Not shown is some incidental information kept, such as the locations of the code and data segments in the service domain. In the figure, the expansion of this information for service B shows that both service A and the application are clients of the service. The necessity for, and use of, this information is described in the following section.

3.10 Service Manipulation

There are several ways in which service can be manipulated, contributing to the configurability and extensibility of the system. The primary three are service migration, replacement and interposition. The latter two operations can also be performed on an application specific basis. The remainder of this section discusses the implementation of each of these facilities. Experimental

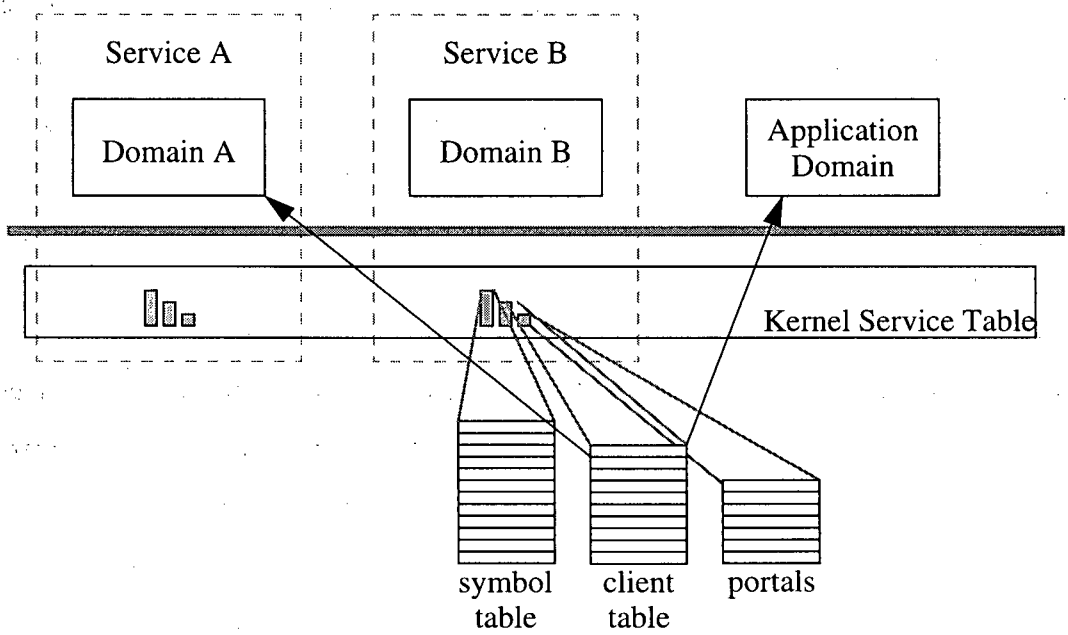

Figure 3.3: Service and Domain Relationships

evaluations and measurements of various manipulations on a variety of services are reported in chapters 5 and 6.

3.10.1 Service Migration

As a system is used, it may be desirable to move services into other domains. The primary reason to do this is performance. If a service can be migrated into the domain of the most frequent client of that service, then the time needed for each IDC can be reduced by the time taken for changing address spaces, which is a potentially expensive operation. Alternatively, IDCs into the kernel are far cheaper than those into another domain, as hardware architectures are optimised for such transfers of control. Thus, for trusted services such as device drivers, file systems and network protocols, it is desirable that these be moved into the kernel once they have been debugged. This offers several advantages to the service developer, particularly if they are developing kernel services. The primary advantage in this case is that of convenience. It is not uncommon for newly developed kernel services to cause a system crash, necessitating a tedious program/crash/reboot development cycle. Within Kea, services can initially be developed

within a private domain, which restricts the crash (if any) or consequences of a programming error, to that domain. Additionally, since the service is running in a separate domain, it can be easily debugged using standard tools.

The migration process is, in many respects, almost identical to that of the service load process. Memory in the destination domain is allocated for the service code, data, and symbol information, which is then copied over. The same linking and initialisation functions described in Section 3.7 are then carried out. Also, each of the service clients is checked to see if it was, or is now, co-located with the service, and the stub pointers are adjusted accordingly. These tasks are all easily accomplished due to the kernel's knowledge about the service structure. The process is however complicated by two extra factors not present when a service is first loaded. The first of these is that the service has already been initialised and running for some time, and will probably have some internal state that it is desirable to have migrated with the service. The second is the maintenance of service for clients that are executing within the service when it is migrated.

Service State Transfer

A large number of services will maintain some internal state (e.g. active file descriptors in a file system). The simplistic service migration described above will only migrate the text and statically allocated data of the service and not this state, which will be necessary for the continued, and correct, functioning of the service. To ensure the correct migration of services, Kea provides a means through which the service designer can arrange to have this state transferred with the service. When the service is migrated, the kernel checks for the existence of a state packaging function (the name of this function is composed by appending "MigrateState" to the name of the service). If it exists, the kernel makes an upcall to this function. The function should package the service state which needs migration into a single memory buffer, which is returned as a result of the function. This data is then copied to the destination domain with the service code and data, and a pointer to it is then passed as an argument to the service initialisation function, allowing the service to recover the state. Where there is no migration function,

or when the service is first loaded, the service initialisation function is passed a null pointer, enabling it to detect this. We believe that passive state (i.e. data allocated by the service, as opposed to active state, such as threads created by the service) could be automatically transferred using new techniques developed for process migration [Smith 97], and plan to pursue this possibility in future work.

Migration and Service Clients

The second major concern with service migration is the problem of clients who are currently using the service. These clients cannot have their calls terminated, and must continue to get the correct results. To solve the problem of calls made after the migration has started, associated with each portal is a variable that indicates whether the service backing the portal is currently being migrated. A check of this value is made during portal invocation, and if migration is taking place the calling thread blocks until the migration is complete. Additionally, any co-located clients have their stub pointers remapped back to the portal stub.

The problem of client calls already extant in the service code is more problematic. The state cannot be deemed to be consistent, or able to be gathered, until all clients of the service have completed their execution within that service. The solution to detecting current clients of the service is twofold. Firstly, each service has a counter associated with it, which records the number of portal invocations that have been made for that service. This counter is incremented immediately after the check for migration described in the previous paragraph and decremented in the portal return path. Thus, the counter maintains an exact count of the clients that have entered, or are about to enter, the service code. Before calling the service's state acquisition function, the kernel first checks to see if the service invocation count is zero. If not, it relinquishes the CPU, checking the state again when it is next rescheduled, an action which is repeated until the count is zero. This need to wait for all service clients to finish executing within the service is the major drawback of the migration mechanism, but it is the only feasible solution. In practice, we find that almost all of the kernel services finish execution within a few milliseconds, the exception being the low level disk drivers, which could potentially take hun-

dreds of milliseconds for long disk queues, although we have not observed this. This delay probably makes migration (and replacement) unsuitable for use in real-time systems, at least for some clients, but general timesharing systems should have few problems.

The second half of the solution concerns those clients that are co-located with the service. As these calls do not go through the portal invocation mechanism, they are not recorded by the service invocation count. The solution to recording co-located clients involves the use of the `ss_function` field of the `serviceStub` structure and the `serviceCount` structure passed as a parameter to `serviceAcquire()`. This structure has the definition:

```
struct serviceCount {
    bool sc_colocated;
    unsigned sc_count;
};
```

The stubs used when the service is co-located make use of this structure to record the number of times they have been invoked. Stubs generated in this manner have, in general, the following structure:

```
static struct serviceCount sc;
static (*fooColocatePtr)();

void fooStubColocate(...) {
    sc.sc_count++;
    if (!sc.sc_colocated) {
        sc.sc_count--;
        fooStubPortal(...);
    }
    else {
        fooColocatePtr(...);
        sc.sc_count--;
    }
}
```

These structures and stubs interact to make co-location and migration possible. The `ss_colocateFn` parameter for each stub is set to the address of the internal function pointer

for the co-location stub (this should not be confused with the stub function pointer, which refers to either the portal or co-location stub). This pointer is adjusted when services are co-located, in order to point directly to the entry point in the co-located service. When the migration function is first initiated, one of the first steps is to change the stub function pointer to the portal stub, and then to set the co-location boolean to false. Clients that call the co-location stub before the function pointer is changed will always increment the counter. Those that are pre-empted after incrementing the counter but before checking the co-location variable will quickly decrement the counter and be blocked in the standard portal invocation path, while others are guaranteed to eventually enter the service via the internal stub function pointer. As in the portal invocation case, the kernel checks the value of the service counter, and waits until it becomes zero before calling the state acquisition function.

Migration Prototype

The prototype of the service migration function is:

```
int serviceMigrate(int service, int domain);
```

where `service` is the identifier of the service to be migrated, and `domain` is the domain the service is to be migrated to.

3.10.2 Service Replacement

There are two types of service replacement, global and local (or application specific). Global service replacement is similar to migration, and is performed for similar reasons. The replacement may offer greater performance, or may be more reliable, or offer increased functionality. In this case, it is desirable that the system administrator be able to transparently replace the old service, ensuring minimal disruption to the users of the system. This is particularly useful if the machine is intended to be highly reliable, available or fault tolerant, as it ensures that all services offered by the machine suffer only a very small interruption in service.

The global replacement function is essentially a combination of service loading and service migration. The old service is suspended, and its state acquired, exactly as for migration. The new service is then loaded normally, except that it is given the state buffer from the old service as an initial argument, and instead of the creation of new portals, those from the old service are recycled.

Local Replacement

Local service replacement is conceptually similar to the global case, in that one service appears to replace another. The difference is that the replacement is done in such a way that the change is visible to only one domain – other domains continue to see the original service layout. Part (c) of Figure 3.4[1] shows a local replacement. In the figure, domain B has done a local replacement of S_2 with S_5, so that all calls originating in B now use S_5, while calls originating in all other domains continue to use S_2. Local replacement is used by applications in order to install services that offer them increased performance for their needs, while not affecting other applications.

Local replacement is relatively simply implemented by a process of installing a domain-specific mapping on each of the portals of the service being replaced. The only complication is for clients that are co-located, which cannot use optimised calls, as they do not go through the portal invocation mechanism. In this case, the clients are forced back to the portal invocation stub. This creates a potential performance problem, due to the additional overhead of portal invocation, as opposed to a direct call (although, if the services are co-located in the kernel, as is the normal case, this overhead is only a few fractions of a microsecond).

1. This is a copy of Figure 1.3, reproduced here for convenience.

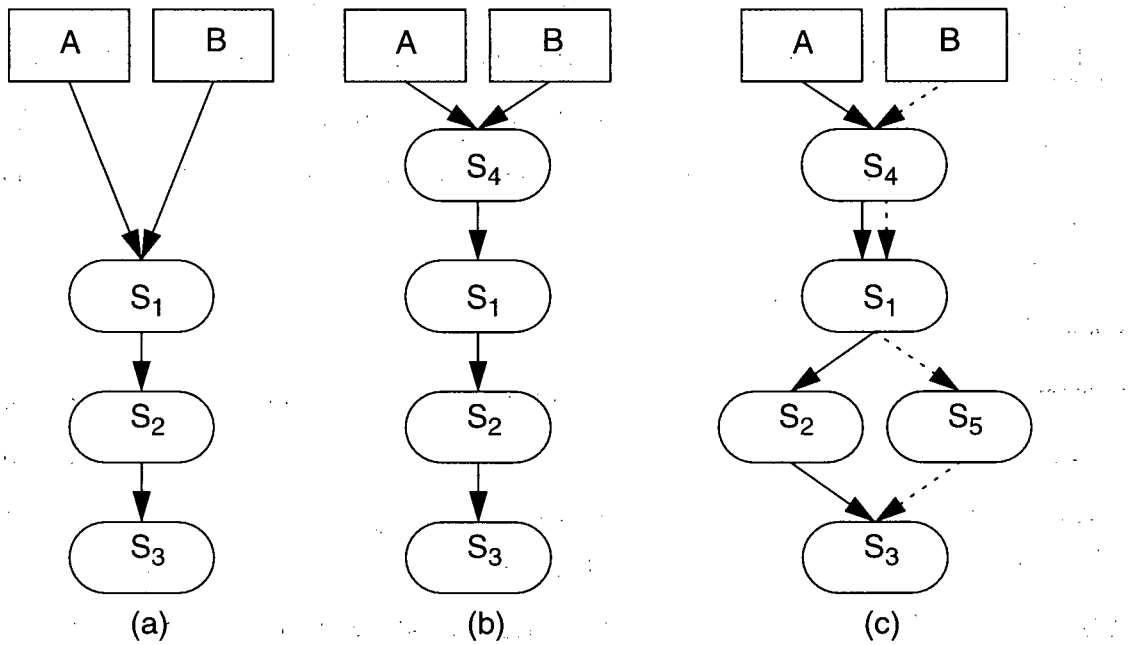

Figure 3.4: Service Reconfigurations

Replacement Prototype

The service replacement prototype is:

```
int serviceReplace(
          int service,
          bool global,
          int domain,
          char *name,
          void *code,
          unsigned codeSize,
          struct function_map *fnMap,
          int servSize
);
```

This function functions exactly like `serviceLoad()`, with the addition of two extra arguments, denoting the service to be replaced and whether the replacement is to be global (`global = true`) or local (`global = false`).

3.10.3 Service Interposition

The final method by which services can be manipulated is interposition. Interposition is illustrated in parts (a) and (b) of Figure 3.4, where a new service (S_4) is interposed above another (S_1). To perform a service interposition, we assume that the interposing service has already been loaded, and acquired any lower level services required[1]. To perform an interposition, the kernel must examine each of the clients of the service being interposed on, and remap them appropriately. If the client is the service being interposed on, then nothing needs to be done. If not, then the portals in the client domain are remapped to the interposing service. The kernel also records in the kernel service structure for the interposed-upon service that it is interposed on, and by whom, so that future clients who try to acquire the service can be remapped to the interposer. Service interposition, like replacement, can also be done on a global or local basis.

The prototype for the service interposition function is:

```
int serviceInterpose(
          int interposer,
          int interposee,
          bool global
);
```

where the first two arguments denote the service identifiers for the two services concerned, and the third serves the same purpose as that of `serviceReplace()`. In fact, once the service has been loaded, `serviceReplace()` uses the service interposition function internally for local replacements – both perform the same logical operation, namely doing a domain-specific remapping for each of the service's clients.

1. Service interposition is, strictly speaking, a misnomer for the operation being described. As services are expected to acquire their own lower level services, what we call interposition really only performs one half of the operation, namely the remapping of the original service clients to the interposer.

3.11 User/Kernel Unification

On of the interesting requirements of the Kea design is that exactly the same programming interface and semantics be provided to all users of the system, regardless of the address space (user or kernel) in which the code is ultimately run. Fully supporting this capability requires that the kernel be able to dynamically link against libraries required by code loaded into the kernel, that code running at both kernel and user level have the same system call semantics, and that kernel stacks be dynamically growable. With the exception of the last point, all of these capabilities are provided by the Kea system. Currently each thread has a fixed size kernel stack (typically 8 Kb). Although techniques do exist for growable kernel stacks [Draves & Cutshall 97], we have found that, in practice, this facility is not needed. It may however be advantageous to develop it for future versions, as it would allow for the allocation of large automatic variables and recursive code.

User/kernel unification offers significant software engineering benefits. It removes many of the problems normally associated with developing kernel level code. Some of these advantages are that standard debugging, profiling etc. tools can be used, there is no need for any special forms of synchronization (e.g. Kea has the same mutex and semaphore operations available to user and kernel threads) and developers only have to be aware of one programming environment (which also has an impact on the size of technical documentation required). These advantages, if they can be achieved with small overhead, make sense in any operating system.

3.12 Protection & Security

There are several issues of security interest with the Kea design. As well as the standard security issues that any operating system must deal with, special problems are raised by the decomposed system design, and the reconfigurability of the system, both general and application specific.

3.12.1 General Security

For the scope of this thesis, general security refers to those security issues that all operating systems must be concerned with. These can generally be constrained to the protection of resources, whether those be a process's memory space or a user's files, from access by users who are not privileged to access those resources. As Kea is designed to be a decomposed system with replaceable components, it tries to isolate the security aspects of the system from the other basic services provided. It is impossible to totally separate security policies from other interfaces, as it is fundamental that security checks take place before many of their operations. With this in mind, it was decided to place all of the security related information in a single structure, which is then made part of the per-domain kernel structure. In the current system, this structure holds only two integer variables, the user identifier (UID) and group identifier (GID) which denote the user and group who own the domain. These values are used to implement simple Unix-like security semantics. Matching the domain structure (which determines what entity is making an action), each thread, event and service has a corresponding structure[1], which determines which users and groups are allowed to manipulate or acquire it.

For each aspect of kernel functionality where it was decided that a security check was needed, the security service exports a number of boolean procedures which take domain identifiers as arguments and determine whether the operation specified is allowed. For example, only a user with UID 0 or with the same UID is allowed to map memory in another domain. The function used to check this has the following code:

```
bool secAllowVmMap(int domain1, int domain2) {
     int user1 = getSecInfo(domain1)->si_user;
     int user2 = getSecInfo(domain2)->si_user;
     return ((user1 == 0) || (user1 == user2));
}
```

1. Each structure holds a 32-bit value for each of the UID and GID, with bits that correspond to allowing or denying various operations on that entity.

By encapsulating the security information in a single structure (specified in its own header file), and keeping all the security functionality in a single source file, Kea modularises, as far as is possible, the security information for the system. Unfortunately, it is impossible to provide all the potential security structures and checks that may be desired in the future. For instance, if at some time it was judged that the ability to protect sections of a domain's virtual memory space with different per-user protections, this information would need to be added to the internal domain structure, and the `secAllowMap()` function expanded to take address ranges as arguments. Thus, while the security encapsulation provided by Kea goes some way towards making the implementation of other security methods easier, it does not totally eliminate the work required.

One complication caused by making domains the holder of security information is that of portal invocation, and the consequent need for threads to change domains. This raises the question of which domain should be seen as performing any action, the one at the root of the activation stack, or the one in which the thread is currently executing? Examining the simple choices implied by the question revealed that neither was feasible. Consider the first option, that of making the original domain responsible. If the thread is executing in another service, this service may wish to perform actions on its own behalf, rather than on its caller's. If every operation is interpreted as being made by the original domain, this becomes very difficult, although not impossible – the service could create separate threads based in its own domain to carry out such operations, but this would be extremely awkward. The obvious alternate, making the domain in which the thread is running responsible, is also infeasible, due to the system's reconfigurability, particularly in the face of service interposition. Consider a service (A) using another service (B), where B maintains a small list of domains which are allowed to perform various operations, and presume that A is on this list. If a service (C), which is unknown to B, is now interposed between A and B, then B will refuse to process any calls from C, even through they originate in A.

To solve these problems, Kea introduces the concept of *effective domain*, which is essentially a combination of both the methods described above. Each activation contains a value, the effective domain, which identifies a domain. The effective domain value for the root activation is set to the domain in which the thread is created, and each successive activation copies this value from the previous domain (this is done during portal invocation). Any service can change the effective domain, but only to one of two values, that of the domain in which the service is loaded (using the `setEffectiveDomain()` call), or to the default value, that of the calling domain (using the `resetEffectiveDomain()` call). Services that wish to check domains can use the `getEffectiveDomain()` function to obtain the current effective domain, and determine which of the domains in the threads' call stack wishes to be responsible for the current call. This system combines the default solution of making the original domain responsible, while allowing individual services to make calls in which they specify that the operation is to be carried out on their behalf, rather than that of the original caller. In several respects, this is similar to the Unix setuid mechanism [Ritchie 79], in which applications with the appropriate file protection bits set can run as if they were executed by the file owner, as opposed to the person who executes the file. Important differences are that the effective domain is changed on a thread, rather than a process basis, and is always temporary (it is reset when the thread returns through the portal), and that it is based on address space, rather than user identity (although this is only one level of indirection removed from the domain).

3.12.2 Protection in a Decomposed System

In a decomposed system like Kea, it is desirable to have some means by which services can protect themselves from interference by other services and applications. This is provided by letting services run in separate address spaces, which are efficiently supported by the system hardware. As described earlier in the thesis, the major disadvantage of this means of protection is the need to change address spaces when making inter-domain calls, which is a very expensive operation. As a consequence, Kea is designed with the presumption that there are only a

limited number of circumstances in which the system designer will wish to run services in separate domains:

- Service debugging – when services are being debugged, it is easier to control the service interactions, and monitor its operations, when it is running in an isolated domain.
- Service testing/verification – even if not actively debugging a service, it is desirable to restrict it to a single address space when it is newly developed or installed, in order to verify that it behaves correctly, and to restrict the damage if it does not.
- Very reliable systems – in systems where reliability and fault tolerance are extremely important, it may be desirable to increase these by sacrificing some performance.
- User provided service – if a user has provided a service, it is unlikely that it should be treated as trusted.

Each of these points are related by the theme of service reliability. For the bulk of systems, we believe that it is both unnecessary and undesirable to have services exist in independent address spaces once they have been debugged. In almost all cases, system administrators will choose to configure their systems in such a way as to give the system clients (applications) the greatest performance, which will only be possible if services are co-located in the kernel[1]. Kea has been designed with this in mind, but provides the means through which administrators and developers can choose for themselves where the trade-off between system modularity, safety and performance should be made.

The above discussion does not directly address all of the security issues associated with the last point on the list (user provided services), as this section focuses on protection only (which can be provided by address spaces). A complete discussion of the issues for user services is given in Section 3.12.4 on local reconfiguration.

1. It may be possible to get better performance for some services by migrating them to an application domain which makes heavy use of that service, but this is expected to be the exception, rather than the rule.

3.12.3 Global Reconfiguration and Security

Kea takes a simple viewpoint of global reconfigurations. It is assumed that system developers and administrators are competent, and understand the implications of system reconfiguration, that the services shipped with the system are safe to install, and are designed to work together. Given these conditions, there are no problems associated with global reconfigurations.

3.12.4 Local Reconfiguration and Security

Local reconfiguration is a much more complicated issue. The two primary problems are:

- Which services are "reconfiguration safe"? That is, of all the services making up the system, which ones can be interposed on or replaced on an application specific basis?
- How can the safety of code be guaranteed? That is, once code has been installed, how can the system administrator be reassured that it will not damage other system components?

The answer to the first of these questions is determined by the design and purpose of the services themselves. It is unlikely that a typical user should be allowed to install a service which interposed on, or replaced, the default disk driver. However, there is no reason why an application should not be able to interpose on any service which can be directly acquired by that application. In general, the answer to the question of which services are reconfiguration safe must be answered by the system architects (for general users) and the system administrators (who may choose to give certain applications greater freedom than others).

The second issue in local reconfiguration, determining the safety of code, is one of the principal research questions attacked by several other projects, notably SPIN [Bershad et al. 95] and Vino [Seltzer et al. 96]. Because these systems are answering these questions, it was decided that it would be more sensible to reuse any applicable methods developed, rather than expend resources on what was judged to be only one of the issues involved in the design of a reconfigurable system. As described in chapter 2, these systems use a variety of methods to accomplish

their goal. Of these, the most promising are software fault isolation [Wahbe et al. 93] and the use of a type-safe and modular language, such as Modula-3 [Nelson 91]. These solutions allow the system to guarantee that externally supplied code will not write to memory outside the service boundaries, and will access all internal memory as the correct type. Although Kea does not implement these solutions, there is no obvious reason why they could not be incorporated when needed.

With a Kea-like system, we believe that the future will bring about two distinct classes of extensions. Firstly, each system will be shipped with a large number of prebuilt services, many of which will be explicitly designed for the purpose of application specific changes to the system. These services will incorporate some method of digital signature [Chaum & van Antwerpen 90, Rivest 92, Microsoft 97] through which the system can guarantee their safety. As new application demands are made known, third-party services will be developed that provide solutions for these applications. These facilities will suffice for the majority of application demands. The remainder will be those applications that need to make a large number of, or highly sensitive, system changes, and also demand high performance. Examples of such applications might be database or file servers. We believe that these applications will normally be dedicated to specific machines, with minimal interference from other users, and will require enhanced privileges in order to run effectively. As these services will be confined to restricted environments, the issues of application interference will be much diminished. In fact, such systems will represent a blurring of the lines between operating system and application, with the machine dedicated for a single purpose.

While there are other solutions to the problems of service safety, such as transactions (as exemplified by Vino) or proof carrying code [Necula & Lee 96], these are either too costly in performance terms or too immature to be considered for use at this stage.

3.13 Scheduling

Like security, thread scheduling is an issue which is impossible to totally isolate from all other parts of the system. Too many users of the system rely on the knowledge and manipulation of scheduling attributes for it to be possible to realistically consider the possibility of a dynamically replaceable scheduler. We have however found that it is very possible to separate the scheduling behaviour from the rest of the system in such a way that it is trivial to build a kernel with a different scheduler. This can be contrasted with other systems which incorporate knowledge of the scheduler in many different parts of the system. For example, the FreeBSD Unix code has a large number of references to the scheduling information in many different kernel source files. The Kea system is unique in that it implements the scheduler entirely as a number of callbacks, made from the kernel whenever an event of interest to the scheduler occurs (such as a thread that finishes sleeping or a thread creation). All the low level code, such as the details of context switches, is kept in the body of the kernel, freeing the implementor of the scheduler to concentrate on the core algorithms. The scheduling information is defined in a single header file, and included into each thread's control block. Typically, the scheduler implementation is also contained in a single file. This system makes it relatively easy to implement schedulers, and several have been developed, including a fixed priority, round-robin scheduler, a real-time scheduler [Finkelstein et al. 95] and a Unix-style scheduler. Complete details of the scheduler interface are given in Appendix C.

3.14 Implementation Summary

Kea has been under development since July of 1994. Initial development was on a Sun SPARC IPC. The primary architecture changed to Intel i486 [Intel 90] and Pentium [Intel 94] based machines in January of 1995. It currently exists as a complete kernel, with all the services described in previous sections completely implemented, and many higher level services (described in the next chapter) providing device access, several file systems and complete networking stacks. All of this development has been the sole product of the author, except for the

low level code for the initial i486 port (done by Peter Smith), most of which has subsequently been reimplemented, the Sun 3 port (by Christian Vinther) the sockets interface to the TCP/IP service (by Davor Cubranic) and the PCI support (by Norm Hutchinson).

When compiled, the current version of the Intel kernel occupies 93 Kb (78 Kb code and 15 Kb data). The number of lines of code in each part of the kernel is shown in Table 3.1. Code lines were measured using "wc -l". The components of the kernel measured included a simple C library (libc), miscellaneous code (initialisation, assembler and other unclassified code) and each of the major kernel services discussed in the previous sections.

Subsystem	Lines
libc	9454
misc	4579
domain/VM	3938
thread	1592
event	628
name	665
service	3004
security	181
scheduler	472
Total	**24513**

Table 3.1: Number of Kea source code lines

3.15 Service Performance

The principal performance measurement made in the evaluation of any decomposed system is the time taken for a cross-domain call (although it has been argued that this and other related performance factors are becoming increasingly unimportant [Ousterhout 90, Anderson et al. 91, Bershad 92, Rosenblum et al. 95]). The traditional means of evaluating this factor is to measure the time required for a null procedure call (that is, a procedure call that has

no arguments and returns no value) between two user level domains. However, there are several serious weaknesses implicit in evaluating systems by this measure only:

- Null procedure call is a poor benchmark. While it captures the cost of transferring control to another domain, it does not measure the cost of transferring arguments and results. Due to the need to marshall and unmarshall arguments, and to then copy them between address spaces, these costs can be significant. Measuring only the null procedure call encourages implementors to ignore the potential expense of user level software stubs. Also, it is extremely atypical to have a call of this nature in code – there are almost always parameters and results to be managed.

- Relying on a single benchmark, especially one that does not accurately reflect the workload placed on the system, is bad practice. It is better to design a range of tests, and draw conclusions, or make decisions on optimisation, based on the aggregate test results.

- Null procedure call only measures user-user interactions. It is equally important to measure calls between user and kernel space (when services are co-located in the kernel, or the control transfers between a service in the kernel and one in user space) and between services that are co-located (either in the kernel or user space). These measurements are very important, as we believe that service co-location, rather than separation, will be the normal configuration of most systems.

By neglecting these weaknesses, and concentrating on the optimisation of null procedure call, OS developers may be being led into poor design decisions. To avoid falling into this trap, we propose a set of tests to measure the complete end-to-end performance of cross-domain calls, and measure these in a variety of different configurations, rather than concentrating solely on the user-user case. In the test suite developed, there are eight distinct types of test, many of which have several variations. The test types can be summarised as:

- *null* – The traditional null procedure call. This gives a first order measure of the overhead for cross-domain communication.

- *fixed argument* – null procedure call, with the addition of one or more simple parameters such as integers. The tests presented use from one to four arguments. This test, when compared with the null test, corrects for the cost of simple argument processing.

- *return* – null procedure call, but returning a value. This test allows an estimate to be made of the cost of returning a result.

- *array* – a procedure with two arguments, the first being a pointer to an array of integers, the second being the size of the array. This test measures the overhead from copying variable size arrays between domains. Several variations are possible. Firstly, the direction of argument transfer can be either *In* (from client to service), *Out* (from service to client) or *InOut* (copied in both directions). The size of the array is also varied, from 4 to 1024 entries.

- *structure* – a procedure with one argument, a pointer to a structure. This test evaluates the copying of relatively small, fixed size blocks of memory. Once again, several variations are possible. The size of the structure can be varied – in the test results shown, the small structure contains 12 bytes, the large structure has 80. While the direction of copying can also be varied (as for the array test), doing this reveals no more information than for the array test, so these results are not reported.

- *block* – moving blocks of anonymous data within the system is important for any application which performs I/O. The block test measures the time taken to copy large, fixed size blocks of data. Variations of the block size (1 Kb, 4 Kb and 8 Kb) were performed.

- *string* – many C-style procedures manipulate zero-terminated character strings. This test measures the time taken to copy such strings. Variations were done with a small string (5 characters) and a large string (60 characters).

- *combination* – This test combines various facets of each of the above. The first argument is an integer, the second a small string, the third a small structure, the fourth an array pointer, and the fifth the size of the array (16 integers).

Each of these tests measures different capabilities of any cross-domain call, and allows an evaluation of the total system capabilities to be made, as opposed to just those shown by the null RPC test.

3.15.1 Kea IDC Performance

Each of the tests described above were run for Kea. The experimental machine was a 100 Mhz Pentium with 256 Kb L2 cache and 64 Mb of RAM. Times were measured using the "rdtsc" (read timestamp counter) instruction, which returns the number of clock cycles executed by the processor. On a 100 Mhz machine, this enables times to be measured with a resolution of 10 nanoseconds. Results from these tests are shown in Table 3.2. These results were obtained by doing the test once (to warm caches) and then repeating each test 1000 times, measuring the aggregate time taken, and dividing to obtain a result. Repetitions of the tests showed minimal variance (typically on the order of 0.1%).

It should be noted that the times for kernel/kernel IDCs are not shown because they are constant at 0.6 µs (kernel/kernel IDC is very efficient because no traps or address space changes are performed, and the destination function can be called directly). The kernel/kernel times were still measured through the portal invocation mechanism, instead of being co-located, as if there are application-specific remappings present, services cannot call each other directly, but instead must use the portal invocation mechanism.

The results show that, as might be expected, simple calls take a relatively small amount of time. (approximately 25 µs) which gradually increases with the amount of data to be transferred. When the call is coming from kernel mode to user mode, the times are reduced somewhat by the elimination of one trap and slightly simpler argument processing. There are some small

Test	Time (μs)		
	user → user	user → kernel	kernel → user
null	25.7	3.1	21.4
fixed1	26.1	3.0	21.4
fixed2	25.8	3.0	21.4
fixed3	25.9	3.1	21.4
fixed4	26.0	3.2	21.4
return	25.5	2.9	21.2
arrayIn4	30.1	4.3	26.4
arrayIn16	31.5	4.6	27.0
arrayIn128	37.0	10.3	32.2
arrayIn1024	79.7	51.9	75.3
arrayOut4	31.6	4.6	27.3
arrayOut16	32.6	4.9	28.4
arrayOut128	37.0	8.7	31.2
arrayOut1024	67.1	31.7	60.9
arrayInOut4	33.8	4.8	31.0
arrayInOut16	35.2	5.0	32.2
arrayInOut128	40.8	11.7	37.2
arrayInOut1024	90.3	43.0	83.3
smallStructIn	31.3	4.2	27.3
largeStructIn	31.8	4.7	27.8
blockIn1K	45.7	15.7	38.7
blockIn4K	80.1	51.4	76.0
blockIn8K	144.5	105.7	137.9
blockOut1K	40.2	8.1	34.2
blockOut4K	64.9	31.2	59.1
blockOut8K	150.0	106.6	138.6
smallStringIn	31.5	4.6	27.3
largeStringIn	34.4	7.5	30.2
combination	39.4	7.6	35.5

Table 3.2: IDC Times.
Related tests are shown with similar shading

anomalies in the table that may need additional explanation. The time for the "return" test is marginally smaller than that for the "null" test. This is because of a branch misprediction in all tests that do not return a value. Adding a return value to the other tests reduces the time for each test by 0.2 µs. Also, the "out" tests typically execute faster than the "in" tests. This is entirely due to cache misses in the data buffers.

Another observation about the times in Table 3.2 is that they could potentially be much improved. Currently, the portal invocation code is written entirely in C. It is probable that by rewriting and hand-optimising this code in assembler, substantial performance gains could be realised. The current compiler used for the Kea source code is gcc 2.7.2, which only supports optimisation for i486 processors. Using an experimental version of gcc with support for Pentium optimisation, savings on the times reported of between 5 to 10% can be made. Unfortunately, the compiler is not quite stable enough to use for production purposes at this time. Also, for the Pentium, memory copies can possibly be done much faster by using the floating point registers to copy memory in 64 bit chunks, rather than the 32 bit copies currently used, although this technique would make context switches more expensive due to the need to save floating point state. Finally, only a relatively small amount of effort has gone into profiling and optimising the code at this time. The primary goal of the system was to develop something that was "fast enough" to enable the reconfiguration experiments, and the current code meets this goal. As will be shown in the following section, Kea compares well with other systems in any case.

3.15.2 Comparisons With Other Systems

The results in Table 3.2 should be compared to other systems. Unfortunately, there is no easy way to do this, as the results reported in the literature are typically for user/user null procedure calls only. For this single test however, a number of results are available and are shown in Table 3.3. This table is based on one in [Liedtke et al. 95] and shows times extracted from Liedtke et al. 97, Liedtke et al. 93, Hildebrand 92, Ford et al. 96, Schroeder & Burroughs 89, Draves et al. 91, Bershad et al. 95 and van Renesse et al. 88.

System	CPU, Mhz	Time (µs)	cycles
L4	Pentium, 133	3.12	416
L3	486, 50	10	500
QNX	486, 33	76	2508
Kea	Pentium, 100	25.7	2570
Fluke	Pentium Pro, 200	14.9	2980
Mach	R2000, 16.7	190	3173
SRC RPC	CVAX, 12.5	464	5800
Mach	486, 50	230	11500
SPIN	Alpha 21064, 133	89	11837
Amoeba	68020, 15	800	12000
Mach	Alpha 21064, 133	104	13832

Table 3.3: Null Procedure Call Times

The important result shown by examining the times in this table is how well Kea IDC compares to other systems. Although three systems (L4, L3 and QNX) appear to perform better than Kea, there are several additional factors that must be taken into account. Firstly, because the times in the table are for null procedure calls only, much of the real costs of handling parameters and data copies are ignored, which is not the case with Kea IDC, which implicitly includes this cost. Secondly, the L4 and L3[1] systems incorporate significant optimisations not made in, or not available for, Kea. Both systems are written exclusively in assembler, and have had many man-months of effort put into the optimisation of the execution path. Kea on the other hand is written almost completely in C, and has not had the optimisation effort invested in other systems. L4 also uses Pentium segment switching to accomplish address space switches. This technique is much faster than using conventional page table switching (saving a minimum of 3 µs in IPC times [Liedtke et al. 95]), but is only applicable to small address spaces of less than 4 Mb. All three systems are extremely small, and fit into the L1 cache of the systems on which they are run, which also substantially decreases their IPC times. In particular, for the L4 time reported, fully half the IPCs have all code and data resident in L1 cache, while the other half have all code and data entirely in the L1 or L2 cache. It is perhaps much more accurate to com-

1. L3 is an ancestor of L4.

pare with the times for the Fluke system [Ford et al. 96], a microkernel of similar complexity to Kea. An interesting observation from the results shown is the predominance of Intel i486 and Pentium processors. These architectures have significantly higher costs for both address space switching and trap handling than all other modern processors. Their dominance in the table probably reflects more on their availability and the implementation effort invested for these systems than any other factor.

As calls between user and kernel address spaces are analogous to system calls we compared the "return" test for Kea with the "getpid" system call on FreeBSD 2.2.2-RELEASE on an identical machine. The Kea time of 2.9 µs compares vary favourably with the FreeBSD time of 3.3 µs, showing that Kea is faster than a well developed monolithic system on at least one equivalent microbenchmark.

It is difficult to compare upcall and kernel/kernel times, as these results are not often published. Perhaps the best comparison to be made is with other extensible systems, as they, like Kea, must include support for call redirection in the kernel. The Vino authors report times of between 67 and 130 µs for the "null graft" case in several of their tests [Seltzer et al. 96]. While this includes support for the Vino transaction mechanism, this is still more than an order of magnitude greater than the Kea times. The SPIN system requires only 0.13 µs for a kernel to kernel call, but this is for services that have been dynamically linked (the Kea overhead for co-location is similar, about 0.3 µs, including the co-location stub overhead), rather than going through a redirection mechanism as in Kea.

3.16 Architecture Summary

This chapter has presented a detailed study of the Kea architecture, with particular emphasis on the support for services. The design of services allows for the dynamic reconfiguration of the system structure through service migration, replacement and interposition. By using domain specific portal remapping each of these actions can also be performed on an application

specific basis, enhancing the system flexibility. It has been demonstrated with microbenchmarks that Kea's IDC performance, vital for the support of these features, is comparable with, or better than, IPC mechanisms in similar systems.

The critical design techniques necessary for building a reconfigurable and extensible system that have been described are:

- Structuring the kernel as a collection of services. Services are the fundamental entities around which reconfiguration and extensions are constructed.
- Low level kernel support for address spaces, threads and asynchrony are necessary for the implementation of services. The interfaces for these services are the base ones upon which all other services are based.
- Procedural IPC is implemented using inter-domain calls, which transfer a thread's flow of control directly between address spaces. IDCs provide for efficient cross-domain data copies, simplify stub generation, and make possible service co-location through direct procedure calls (as the natural system paradigm supports the processor's natural call mechanism).
- Portals are used by the kernel to represent an IDC entry point. By recording portal addresses and imposing a layer of indirection in the portal invocation mechanism, the portal remapping mechanism can be used to arbitrarily redirect portal invocations, on both a global and application specific basis, independently of the application using the portal.
- The stub structure and function pointers make service co-location simple. Both stub and portal invocation mechanisms provide for reference counting on service usage, enabling safe migration.
- Representing services as object files lets them be easily linked into any domain.

- User/kernel unification provides significant software engineering advantages, as total system complexity (as seen by the programmer) is reduced, and also enables services to be run transparently in either kernel or user address spaces.

Each of these features operate together synergistically to reinforce the others, and provide a system that can be reconfigured and extended in a number of ways. The following chapters introduce some high level services built on the system, and then show how these, combined with the service features described in this chapter, can be used to provide development, administration and performance advantages.

CHAPTER 4

High Level Services

The previous chapter described the design and implementation of the lower level services and capabilities of the Kea operating system. This chapter examines the higher level services that comprise the bulk of the system's user visible functionality. These services primarily support file systems and networking. Most of the sections deal with the design of the individual services, while the remainder describe how these services are composed into functional units.

Unless either the operation of a service is significantly different in some way from the equivalent normally found in a standard operating system, or some facet of its operation is important to the experiments described in subsequent chapters, its design will not be discussed in any detail. The key point of this chapter is that it is possible to develop these services independently and then compose them into a functional whole – there are certainly many other possible combinations and decompositions available, and it is not claimed that the ones presented are the best possible. Some of the interfaces for these services are detailed in Appendix B.

4.1 Disk Driver Service

As the primary platform to which Kea is targeted is the IBM PC compatible architecture, one of the first services developed (other than for simple testing) was a service to allow reading and writing of IDE disk drives. This service is important for two reasons. Firstly, it forms the base layer for various file system services and secondly, it illustrates the development of a Kea device driver. Device drivers in Kea are especially interesting as they can be developed entirely independently of the kernel – there is no inherent difference between a device driver service and any other form of service. Each can be run in an arbitrary address space, makes use of the same programming interfaces, and can be transparently relocated. This can be contrasted with other systems, in which device drivers require special interfaces, only available to certain privileged processes, or must be developed and executed exclusively in the kernel environment. Full details on the IDE interface can be found in Appendix B.1.

4.2 Buffer Cache Service

The buffer cache ("bcache") service provides buffering of disk data, typically for filesystems. The buffer cache service controls the reading of blocks of physical disks, and indexes each block under a client supplied file identifier, as well as the physical device location. Blocks can be assigned priorities, with separate LRU lists for each priority level. This allows clients of the buffer cache service to control the order of block recycling. By interposing services upon the buffer cache service, applications can also control the caching of their blocks (this is described further in Section 6.2.2). Full details on the bcache interface can be found in Appendix B.2.

4.3 Filesystem Services

The current Kea system includes three filesystems. The first implements a MS-DOS FAT filesystem, the second a BSD fast filesystem (FFS) [McKusick 82] and the third a memory (as opposed to disk) based filesystem. Each of these file systems implements a simple file interface

(detailed in Appendix B.3) that is very similar to the POSIX standard [IEEE 90]. The FAT and memory based filesystems are fully functional, while the FFS filesystem does not currently have support for file creation or extension (i.e. writes cannot occur past the end of a file) or for opening directories. While these restrictions limit the utility of the filesystem, they are not significant for assessing the reconfiguration capabilities of the system as a whole, which was the primary purpose of its development.

4.4 File and Mount Service

Each of the filesystems can be used as a stand-alone system, and when so configured, uses file names relative to the root of that filesystem. To be of use to the majority of applications however, there needs to be some means by which different file systems can be coalesced into a single namespace. This is accomplished by two more services, *mount* and *file*.

4.4.1 The *Mount* Service

The mount service simply associates a set of name prefixes with a service identifier. It provides a procedure that, given a fully specified file name, returns the service identifier of the file system that handles that particular file, together with the length of the prefix. The mount interface is shown in Appendix B.4.

4.4.2 The *File* Service

The file service effectively groups all of the other file services into a homogeneous whole, by multiplexing each under a single namespace. When a file is opened, it uses the mount service to determine the relationship between files and underlying services. If it has not already acquired the appropriate service, it does so. The mount point prefix is stripped from the file name, which is then passed to the underlying service. Subsequent operations pass directly through the service.

4.5 File Service Composition

The IDE, bcache, mount and file services can be composed to provide a fully functional file service to clients. The typical composition of these services is shown in Figure 4.1.

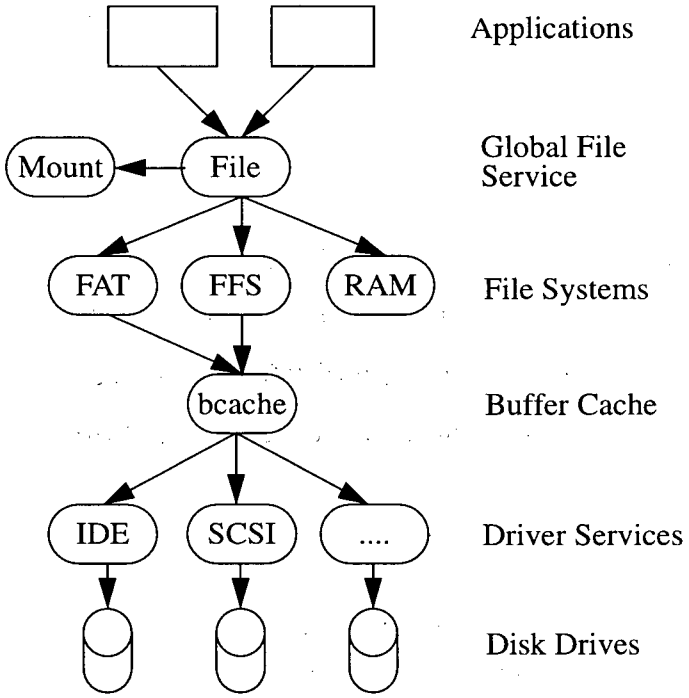

Figure 4.1: File Service Composition

Because the set of file services are composed together in a reasonably complex hierarchy, they are ideal for experimentation with system reconfiguration. Chapter 5 describes a number of such experiments. Chapter 6 uses them in order to investigate some number of some application specific system extensions.

4.6 Compressed Filesystem Service

One other file system service has been developed, the compressed file system (CFS). This is a service designed to be interposed on another filesystem. As such, it relies on the underlying file-

system for storage. As the name implies, the compressed file system compresses (and uncompresses) file contents. This operation saves disk space, and can decrease the time required for file operations, trading CPU time against disk accesses. The compressed filesystem stores files in one of two ways. Where the file is written consecutively, it manages the file as a sequence of compressed blocks, each block preceded by a descriptor giving its length, both compressed and uncompressed. Whenever a write to anywhere but the end of a file occurs, the filesystem falls back to the standard file viewpoint, and only compresses the file when the file is closed. The compressed file system is used in experiments in Chapters 5 and 6.

4.7 Networking Services

The second major group of services in Kea support networking. A simple ethernet service allows its client to send and receive ethernet packets. The major client of this service is a port of the *x*-kernel [Hutchinson & Peterson 88, Peterson et al. 90], which provides a complete set of network protocols. The *x*-kernel version of Kea contains a complete implementation of Berkeley sockets [McKusick et al. 96], through which user level clients access the network services.

As the *x*-kernel is implemented as a single service, and relies only on the ethernet service, there is limited scope for reconfigurability experimentation, although some experiments are described in Chapter 5. Several proposed projects involve the decomposition of the *x*-kernel into a number of independent protocol services, which is more in keeping with the Kea philosophy. These proposals are discussed further in Section 7.2.1.

4.8 Other Services

Several other services have been developed that add to the functionality of the system. Because these services perform only minor operations, or were not used in the experiments described in subsequent chapters, they will only be described briefly.

4.8.1 Syslog Service

The syslog service performs approximately the same task as the Unix daemon of the same name. It provides an interface through which services can register themselves, and then send messages of various priority levels to be printed on the console. Almost all of the other services described use the syslog service for reporting various configuration and error messages.

4.8.2 Unique Identifier Service

Originally each filesystem maintained its own set of identifiers, and the buffer cache combined the filesystem domain and file identifier in order to lookup file blocks, while the global file service remapped each file service identifier into a globally unique identifier for client applications. However, when application specific system filesystem extensions were added, it was realised that having different file identifiers used at different layers of the file system hierarchy made the development of extensions far more difficult. To solve this problem, a service was developed that generated unique identifiers. When opening a file, the low-level filesystems use this service to obtain a file identifier, which is then used by all other services. This had the pleasant side effect of making both the buffer cache and global file services slightly simpler.

4.8.3 Console Service

A service that is able to write to the console is provided in Kea. It typically maps the physical memory block used by the hardware for the screen, and directly inserts characters into this space. The only clients that use this service directly are the standard I/O routines in the C library.

4.8.4 Keyboard Service

Kea includes a simple service that enables its clients to receive keyboard events. It registers itself for the keyboard interrupt, processing each to generate characters for its clients. Like the

console service, the only clients that use this service directly are the standard I/O routines in the C library.

4.9 High Level Service Summary

A summary of the size of the most important services is shown in Table 4.1. For each service, the table shows the number of lines in the service (measured with "wc - l"), and the size of each of the services compiled data and text segments in Kb. All the services, with the exceptions of the ethernet and *x*-kernel, include migration support.

Service	Line count	Text Size	Data Size	Total Size
IDE	1440	5.7	0.1	5.8
bcache	583	2.4	2.2	4.6
FAT filesystem	3533	14.9	0.5	15.4
FFS filesystem	1196	4.3	0.0	4.3
mount	210	1.0	0.0	1.0
file	280	1.2	0.1	1.3
compressed fs	2761	6.5	193.1	199.6
ethernet	1115	4.8	0.0	4.8
x-kernel	26739	178.1	48.8	226.9
Total	**37857**	**218.9**	**244.8**	**463.7**

Table 4.1: Service Size Summary

This chapter has briefly described each of the high level services currently developed for Kea. For the purposes of demonstrating the thesis statement, is not important how these service are built, only that they can be built and composed together to make a complete system. Once the system is composed of a collection of services, various experiments into their efficiency, reconfigurability and extensibility can be undertaken. Such experiments and their results are shown in Chapters 5 and 6.

CHAPTER 5

Performance and Reconfigurability

This chapter describes the results of several experiments which were performed to evaluate the performance of the system. These experiments were designed to measure the performance of the system as a whole (i.e. the aggregate system throughput, using the high level services described in chapter 4) as well as that of the reconfiguration primitives, particularly global service migration, replacement and interposition (application-specific reconfiguration is investigated in chapter 6). In particular, the experimental results demonstrate the following points:

- That services can effectively be run in either user or kernel address spaces.
- That the performance of the system is comparable to other, standard operating systems.
- That services can be dynamically and transparently migrated between address spaces.
- That service migration is efficient.
- That services can be efficiently interposed.

5.1 Experimental Overview

The majority of the experiments described use the filesystem hierarchy described in chapter 4 and illustrated in Figure 4.1 on page 81. These services (ide, bcache, FFS/FAT, file and mount) provide a sufficient base to evaluate the performance of structuring a system as a collection of such services, and more importantly, allow an evaluation of how the system performance can be changed by system reconfiguration. In addition, the service stack allows for experimentation with service interposition, for which the compressed file system service is used.

The experiments can be roughly categorised into four areas:

- System performance with various service configurations. By configuring services into different address spaces, the performance/protection trade-off can be evaluated. These experiments also demonstrate that services can be run in arbitrary address spaces.
- Performance comparisons, Kea vs. FreeBSD. This allows a comparison of a system configured as a set of co-operating services with that of a monolithic system.
- Performance of service migration. Measuring the overhead of service migration provides a measure of its utility.
- Service interposition. By interposing services, and measuring either (or both of) the changes in performance or functionality provided, an assessment of the utility of system extensibility can be determined.

5.1.1 Experimental Hardware

Each of the experiments were performed on the same machine, a 100 Mhz Pentium, with 64 Mb RAM, 256 Kb L2 cache and a Western Digital Caviar 2850 EIDE disk drive (the properties of this drive are shown in Table 5.1). Where applicable, Kea's performance was compared with that of FreeBSD (version 2.2-RELEASE), running on the same machine.

cache size	64 Kb
spindle speed	4,500 RPM
average seek	11 ms

Table 5.1: WD2850 Parameters

5.2 FFS Performance

As stated, each of the Kea services can be developed and loaded into a user address space. These services can also be loaded into the kernel without changing any part of the service, either at the source level, or in the final compiled object. Reconfiguring the system by running these services (that are normally trusted parts of the kernel in other systems) within the kernel should result in performance benefits from shorter IDC times (for user/kernel, as opposed to user/user control transfers), and also from the optimisation of some IDC's to procedure calls due to co-location. Performance should be further enhanced by far less cache and TLB misses, due to a reduced number of address space crossings. The measured performance must also be comparable to other systems in order to demonstrate that there is no significant performance impact attributable to the decomposed nature of the design.

5.2.1 Reconfiguration and Performance

To verify that reconfiguring the system to run services in the kernel increases performance, we measured the time (in microseconds) required for basic file operations in the FFS filesystem hierarchy with a number of permutations in the location of its services. The results are shown in Table 5.2. The table shows the time required for each of the file open, read, write and close operations. The read and write operations used a block size of 8 Kb. Each of the experiments was performed with the system in one of two states – "cold", when the system had just been booted and "warm", immediately after the "cold" measurements were recorded. Making the measurements in each state enables some estimate to be made of the effects of warm caches and cached buffer blocks. The operations were timed with the system in a variety of different configurations. The left-most column shows the times with each service in a separate domain,

while successive columns show the impact of co-locating increasing numbers of the services into the kernel, with the penultimate column showing all services in the kernel[1]. The final column shows the times with all the services co-located into a single domain, roughly equivalent to the "single-server" model used with many microkernels. The "cold" times shown were heavily variable, due to differences in disk rotational position – depending on the disk head position when the trial began, successive I/O operations can vary by over 13 ms (the rotational latency of the disk). This was especially obvious for the open times, where three disk operations are required (the root inode, a directory block and the file inode). To remove this influence, the times shown were determined by measuring each of the operations five times, discarding the lowest and highest times, and taking the average of the remaining three. Also, for each operation, the actual I/O time[2] was measured for each trial. This was then averaged over all the trials for the same operation, and the times normalised by using this average instead of the measured I/O time in each case. This provides an accurate indication of operation times, independent of variations in disk seek time and rotational delay.

	Operation	Separate domains	IDE in kernel	+bcache in kernel	+FFS in kernel	All in kernel	Single domain
"cold"	Open	31650	31573	31110	30641	30103	30561
	Read 8K	20960	19882	18136	17693	17288	17256
	Write 8K	1745	1562	1369	929	471	852
	Close	279	273	267	197	63	120
"warm"	Open	617	609	361	252	124	176
	Read 8K	791	782	734	572	362	424
	Write 8K	781	756	723	534	352	394
	Close	115	112	112	83	53	70

Table 5.2: FFS Filesystem Operation Times

1. The mount and file services are moved together into the kernel. The mount service is only used on open, and was deemed unimportant enough that doing this does not effect the points that the results illustrate. The mount and file service are loaded in independent domains in other result columns.

2. This is the time from the initiation of the disk controller to the reception of the final disk interrupt.

There are several important observations that can be made from the results shown in Table 5.2. Firstly, it can be observed that moving services into the kernel can have significant performance advantages (e.g. the decreases in CPU time for the "warm" read/write cases are over 50%, and more than five-fold for open, which requires a larger number of operations utilising all services). It is also important to note however, that even in the case where each service is run in a separate address space, the performance is still somewhat acceptable, which implies that it is practical to consider developing systems as groups of separate services, and then combining these at a later time. This combination will almost certainly occur, due to a desire for enhanced performance, despite the increased protection implied by running services in separate domains. What is important is that developers and administrators be given the choice, and the tools with which to make this choice.

5.2.2 FreeBSD Comparison

To prove the thesis statement, the system's performance must meet that of a traditional system, at least when the services are co-located in the kernel. Table 5.3 shows the comparison between FreeBSD and Kea for the FFS file operations. In order to make the results comparable, the FreeBSD operations have had their disk I/O times normalised relative to those of Kea. The table shows that, except for open, the times are almost equivalent. In the "warm" case, Kea reads take more time, but this is partially compensated for by faster write times. The "cold" open and write times appear to be major anomalies. The former is actually a result of FreeBSD having cached the root inode and directory blocks when the filesystem is mounted. Where Kea must do three disk operations to open the file, FreeBSD does none, resulting in a far faster time. The "warm" open times are therefore a better indication of the systems' performance, and show that the systems are more equally matched, although FreeBSD is still more efficient. This may be partially due to the caching of name/inode translations in the FreeBSD filesystem, an operation which the Kea version does not yet support. The "cold" write time is due to the Kea buffer cache service optimising away disk I/O when the block to be written is the same size as that requested by the filesystem. In general, the results are very promising, showing that the Kea

times, with a relatively simple implementation of the filesystem hierarchy, and no tuning or optimisations made to any part of that hierarchy, can come close to that of FreeBSD, which has been tuned over a number of years.

	Operation	FreeBSD	Kea
"cold"	Open	202	30103
	Read 8K	17388	17288
	Write 8K	7801	471
	Close	55	63
"warm"	Open	90	124
	Read 8K	327	362
	Write 8K	354	352
	Close	54	53

Table 5.3: FreeBSD FFS Filesystem Operation Times (μs)

A second comparison that can be made between FreeBSD and Kea is that of aggregate system throughput. This is determined by measuring the times required to read or write 1 Mb of data, using both 1 Kb and 8 Kb data blocks. The results are shown in Table 5.4. There are a number of interesting observations that can be made based on these results. For the "cold" numbers, the read times are very close. For writes however, FreeBSD is much slower with a 1 Kb block size, and much faster with an 8 Kb block size. There are two factors contributing to this result. Firstly, the Kea filesystem always performs disk I/O based on the natural disk block size used by the file (8 Kb for the file in question), rather than on the block size used in the read or write operation. It appears that the FreeBSD filesystem does the opposite, resulting in many more disk operations for the smaller block size, and consequently a much slower overall time. The second factor is that for writes, the Kea filesystem always reads the block of disk first, even when it is going to be totally overwritten by the write operation. The FreeBSD filesystem foregoes the read when it realises the block will be totally overwritten, resulting in far faster times for an 8 Kb block size.

	Operation, block size	Time (s)	
		FreeBSD	Kea
"cold"	Read, 1K	0.497	0.447
	Write, 1K	1.099	0.388
	Read, 8K	0.459	0.446
	Write, 8K	0.110	0.388
"warm"	Read, 1K	0.038	0.050
	Write, 1K	0.110	0.045
	Read, 8K	0.029	0.043
	Write, 8K	0.039	0.036

Table 5.4: FFS Aggregate Read/Write Performance

The "warm" times show that, in general, FreeBSD is slightly faster than Kea. In the "Write, 1K" category however, Kea is over twice as fast. This is again attributable to the handling of smaller blocks in the FreeBSD filesystem not being as efficient as Kea's.

Overall, the FreeBSD/Kea comparisons show that the design of the components, and the strategies they use (e.g. block size choices) is probably more important than how they are structured within the final system. While the Kea approach may involve slightly more overhead due to an increased number of cross-component calls (approximately 0.2 µs per call/layer in the architecture under test), this is trivial compared to the cost of a single I/O operation. Even when the two systems are performing identical tasks, the individual variances in design decisions make deciding how much of any differences observed are due to the systems structure, as opposed to those attributable to the individual service designs, very difficult. The single most important observation is that is possible to build a system such as Kea that performs favourably when compared to a monolithic kernel.

5.3 FAT Performance

To confirm the results discussed in the previous section, the experiments were repeated for the FAT filesystem. The results for the system reconfiguration are shown in Table 5.5. The numbers produced are comparable to the FFS results and show exactly the same general trends.

	Operation	Separate domains	IDE in kernel	+bcache in kernel	+FAT in kernel	All in kernel	Single domain
"cold"	Open	1294	1297	1292	1053	305	630
	Read 8K	26546	25671	25082	25007	24750	25291
	Write 8K	4164	4081	3514	3228	2751	3077
	Close	425	426	321	187	173	190
"warm"	Open	320	309	292	244	143	166
	Read 8K	727	727	708	644	368	426
	Write 8K	728	728	623	577	344	403
	Close	179	174	150	96	73	99

Table 5.5: FAT Filesystem Operation Times (µs)

Comparing the FAT filesystems performance to FreeBSD (Table 5.6) is more interesting. When "cold", FreeBSD takes substantially longer for the open and read operations, and less for the write. The open time is a reversal of the FFS case, as the Kea filesystem reads and verifies the root directory of the filesystem when it initialises itself, as opposed to the FreeBSD filesystem, which only verifies the superblock. The FreeBSD FAT filesystem also performs several disk reads in order to translate the file name, resulting in a longer open time. The read/write times are explained by examining the FreeBSD I/O pattern. For the initial read, it reads ahead on the disk, getting more blocks into the cache (even though they may be overwritten in a subsequent write), slowing the initial read time slightly, at the cost of greater performance for future operations.

The "warm" times are generally similar, with FreeBSD having a slight edge in most operations, except for write, where it appears to be substantially slower. These results reinforce the earlier

	Operation	FreeBSD	Kea
"cold"	Open	27733	305
	Read 8K	28124	24750
	Write 8K	518	2751
	Close	331	173
"warm"	Open	132	143
	Read 8K	317	368
	Write 8K	483	344
	Close	55	73

Table 5.6: FreeBSD FAT Filesystem Operation Times (µs)

observation that the design decisions made when building filesystems are of more importance to the overall system throughput than the architecture used to combine system services.

One puzzling result not shown in the table concerns the FreeBSD "warm" close time. With the system in single user mode, this operation would take either 55 µs (as shown) or about 600 µs, with the latter being the most frequent by a factor of approximately five. In multi-user mode, the time was consistently 55 or 56 µs. Unfortunately, without detailed traces of system behaviour, it was impossible to determine exactly where the single-user delay was coming from.

The aggregate throughput (reading and writing 1 Mb of file data) for both FAT filesystems is shown in Table 5.7. Once again, times are generally comparable, with two exceptions. For small blocks, with cold caches, the FreeBSD filesystem is over five times slower – it does disk operations in the smallest possible unit (1 Kb), rather than the natural filesystem block size (8 Kb) used by the Kea system, resulting in a huge I/O overhead (this also results in a slower times in the "warm" case, although not to the same marked degree). The second exception is the "warm" read time, which is twice as fast on FreeBSD. This is due to the FreeBSD filesystem detecting consecutive reads to the same disk location. In this case the filesystem performs far larger disk reads. The savings observed are apparently due to the need for less buffer management. This does not affect the time for the "cold" reads overly much, as the same number of "real" I/O operations (i.e. those that go to the disk) still need to be performed. It would be

entirely possible to implement this same optimisation in the Kea filesystem, although this has not been done to date.

	Operation, block size	Time (s)	
		FreeBSD	Kea
"cold"	Read, 1K	0.565	0.583
	Write, 1K	3.457	0.625
	Read, 8K	0.578	0.570
	Write, 8K	0.625	0.625
"warm"	Read, 1K	0.037	0.045
	Write, 1K	0.051	0.038
	Read, 8K	0.022	0.044
	Write, 8K	0.036	0.037

Table 5.7: FAT Aggregate Read/Write Performance

5.4 Service Migration and Replacement Costs

The next set of experiments were performed to demonstrate that services can be efficiently migrated between user and kernel spaces. The time taken to migrate a service depends on several factors – the amount of executable code comprising the service, the time needed to link this code into the new domain, the amount of service state to be transferred, the time taken by the services initialisation function to execute, the amount of portal remapping that has to be done due to the change in address spaces, and the cost of modifying any clients of that service in the destination domain, in order to use direct procedure calls rather than portal invocations (or vice versa, in the case of clients in the original domain).

To assess the times attributable to each component of the migration process, several measurements were made on each of the principal filesystem services. The results for the ide and bcache services are shown in Table 5.8. These services are shown first, as the state to be transferred does not depend on the number of open files (other services had more extensive experiments performed, with correspondingly more complex results). The first column shows the service

name, the second the size of the service text, the third and fourth the copy and link times for that text, the fifth and sixth the time to save and restore the state, while the final column shows the total time taken. In the case of the IDE service, the "restore state" column has two numbers. The first is the actual time for the restore operation, while the second is the latency introduced by a disk I/O, done to read the partition table off the disk. The major result shown is that the link time is the dominant component of the migration cost. This is generally true for all services, and making linking more efficient is a goal for future research. In the case if the IDE service, it is possible to remove the disk I/O, at the cost of increasing the complexity of the save/restore state functions. If the migration time for this service proved to be a problem – which it has not been to date – then this could be decreased through this modification.

service	size (Kb)	copy time	link time	save state	restore state	total time
ide	10.6	0.2	13.2	1.2	1.2+31.4	47.7
bcache	6.0	0.2	9.3	1.4	1.2	12.4

Table 5.8: ide and bcache Migration Times (ms)

One important point to note about the bcache service is that the state transferred does not include all of the disk blocks in the cache – only those that are currently referenced by a filesystem are copied over. This is typically an extremely small number, as a filesystem only holds a reference for the short time needed to copy data from the block to a user buffer. Copying all the cached blocks is not practical, as temporarily at least, twice the amount of memory in the cache would be needed, since the state buffer must be allocated before the cache blocks could be copied to it.

The second set of migration results – those for the filesystem services – are shown in Table 5.9. This table shows the same general results as for the ide and bcache services, with the addition of information describing the overhead required for state transfer with varying numbers of open files. The table shows that while the cost of state acquisition and transfer does rise proportionally to the number of open files, it does not, compared to the other times, impose much of an

overhead. As for the other services, the major overhead is the link time, and any disk I/O required (the FAT filesystem reads root directories).

		FFS	FAT	file
size (Kb)		4.3	23.5	2.9
copy time		0.1	0.4	0.1
link time		20.1	48.8	10.3
0 open	save state	1.3	1.4	1.2
	restore	1.4	1.4+71.2	1.2
	total	23.2	125.7	13.2
1 open	save state	1.3	1.4	1.3
	restore	1.4	1.4+71.2	1.4
	total	23.2	125.7	13.5
10 open	save state	1.5	1.6	1.2
	restore	1.7	1.5+71.2	1.5
	total	23.8	126.0	13.6
100 open	save state	3.5	3.2	1.4
	restore	3.6	3.3+71.2	2.0
	total	27.7	129.3	14.3

Table 5.9: File Service Migration Times (ms)

Both Tables 5.8 and 5.9 show that the various components of the time required to migrate a service can vary substantially between services. While the copy time is small for each (and linear, based on the size of the service, as would be expected), the relative times required to link the service can be highly variable, depending on the number of external library modules needing to be loaded and the amount of text and data relocation required.

The most important conclusion that can be made is that migrating services can be done quickly enough that there should be little, or no, effects on the time perceived by a user for a service operation, although more data on larger services would be desirable. It is important to point out that the service is only unavailable to clients while the state transfer and initialisation are carried out – the copy and link phases are done before service access is blocked. This means that

the perceived migration time for service clients is much less than the total migration time, e.g. for the FAT filesystem, the perceived migration time is between 73 and 77 ms, which is effectively only the cost of a small number of disk operations.

The time taken to replace a service is also the same as the time needed to migrate that service, as exactly the same set of actions have to be carried out for this operation. The only possible variations are in the size of the code (which, if the service is being replaced for the purpose of bug fixes, should be small), and in (possibly) the creation of a new domain for the service. In the first case, the size of the code has very little influence on the cost of replacement, only affecting the copy time. There may be some effect on the link time, although for most services this should be negligible. In the case where a domain has to be created, this also has no effect on the time the service is unavailable, as this operation is performed before portal invocations are blocked from entering the service. It can therefore be concluded that service replacement is also a viable operation.

5.5 Service Interposition

To demonstrate and test global service interposition, some of the measurements described in Section 5.2 (FFS filesystem performance) were repeated using the compressed file system. The results are shown in Table 5.10. The four columns of this table show the operation performed, the case where the compressed file system is interposed above and below the global file service, and for comparison, the original results (from Table 5.2). The results shown are for all services co-located in the kernel – other service configurations have been measured, but they show the same pattern as the results shown.

The results from the compressed file system experiments show that opening a file for the first time is very expensive, while the subsequent operation (read) is much cheaper. This is because the compressed file system preprocesses parts of the file when it is opened, requiring expensive disk I/O, while the subsequent read finds that the information required is already in the buffer

	operation	above "file"	below "file"	original (no cfs)
"cold"	Open	35674	35654	30103
	Read 8K	1409	1411	17288
	Write 8K	7865	7870	471
	Close	126	126	63
"warm"	Open	142	141	124
	Read 8K	1258	1259	362
	Write 8K	2963	2967	352
	Close	94	92	53

Table 5.10: Compressed File System Operation Times

cache[1]. The "warm" times show that each of the open, read and write operations are much more expensive, due to the extra processing and CPU time involved in compression, but the write times should be offset by the (eventual) reduced disk I/O times when the blocks are written to disk. This is a common feature of compressed file systems, where the CPU time for compression must be offset against the frequency of I/O operations that need to go to disk, and the requirement for less disk space usage. This is important, as for some file systems, or applications (particularly text-based), compression may be beneficial, while for others it might have lower performance. By enabling the interposition of the compression service in different configurations, the system administrator or application can decide on the most effective placement. The trade-off depends entirely on the balance and type of disk activity experienced by the file system. To demonstrate this, the test where 1 Mb of data is written to a file in 8 Kb[2] blocks was repeated using the compressed file system. The results are shown in Table 5.11, together with the original results (from Table 5.4). The results show, that for a cold cache, reading is faster, due to the reduced number of I/O operations required, while every other operation is more expensive, due to the extra CPU overhead of compression. The algorithm used is also far more

1. The open time shown should be higher (closer to the sum of the open and read times for the original case). The reason it is not is because a different test file is used, with different disk block allocations (and hence seek times). This is one more reason why such comparisons should be examined carefully.

2. The first 8 Kb of text from this chapter were used for all tests in this section. This compressed to 4.7 Kb, equivalent to a 41% compression rate. Blocks are compressed independently.

efficient at decompression than compression, resulting in elevated times for write operations, even for a cold buffer cache.

	Operation	Time (s)	
		cfs	Original
"cold"	Read	0.367	0.446
	Write	0.563	0.388
"warm"	Read	0.162	0.043
	Write	0.377	0.036

Table 5.11: Aggregate cfs Read/Write Performance

5.6 Conclusions on Reconfiguration Performance

This chapter has presented the results from a number of experiments designed to measure the performance of the system in a number of areas related to system reconfiguration. Using a file-system service hierarchy, it has been shown that co-location of services through service migration can result in greatly improved performance, and that Kea's performance is comparable to another, mature, operating system. The absolute overhead for service migration has been shown to be small, and generally dominated by the time required for service initialisation, particularly if those services have to perform time consuming operations such as disk I/O. The final experimental section showed that service interposition can be used at different points in the service hierarchy, and that this can be done transparently to the other services involved.

In general, the results given in this chapter have demonstrated that global system reconfiguration using service migration, replacement and interposition is not only possible, but able to be accomplished with reasonable performance. It has been shown a reconfigurable system can be implemented that does not impose an undue performance penalty. Possibly the most important conclusion if that the system architecture seems to play less of a role in determining the ultimate system performance than the design of the individual components making up the system.

CHAPTER 6

Application Specificity Evaluation

The previous chapter evaluated the performance of global reconfiguration in Kea. This chapter describes some studies made to evaluate the usefulness of the system's application specific features. As in the previous chapter, the primary focus is on the filesystem hierarchy, but with an emphasis on techniques for increasing the performance of individual applications.

6.1 In-Kernel Applications

Because Kea has a unified kernel/user programming environment, with facilities for migrating code between these environments, it is ideal for the development of in-kernel applications. An in-kernel application is defined as a service that performs actions that are normally viewed as the province of an "ordinary" application, but that, when executed within the kernel, can realise a substantial performance gain. The domain of such applications is normally restricted to applications that carry out trusted tasks, and for which performance is very important. The best examples of such applications are those that are substantially restricted by the I/O bandwidth

of the machine, especially those that offer file services. Particular examples might include Web servers, multimedia file servers and "network appliance" type machines. Each of these has specialised demands, in that they must respond to client requests quickly (often within a fixed time limit), and may need to specialise the caching behaviour of the system. Currently, only very few systems (notably Rialto and SPIN) offer this functionality. Rialto has been used for the implementation of a commercial video file server [Bolosky et al. 96], while SPIN has a small in-kernel http server [Bershad et al. 95]. Both of these applications have demonstrated that moving applications into the kernel is both possible and desirable. Rialto, like Kea, also has the added advantage of being able to first implement the application at user level, easing the development process.

Although there are not yet any large examples of in-kernel applications for Kea, several micro-benchmarks have been implemented which demonstrate the system's utility in this area. One of these is shown in Table 6.1. The test was to write a server that, given a file name, can open, read, and close that file. This simulates a typical task performed by most file servers. In particular, the file used in the test was a relatively short text file[1] (2702 bytes), as would be typical for many of the files read by an http server. The test was run with the service in-kernel (and directly accessing the FFS service, instead of the file service) and out of kernel, for both the warm and cold buffer cache cases[2]. The results show the times with the disk I/O factored out.

	Location	Time (µs)
"cold"	In-Kernel	344
	Out-Kernel	537
"warm"	In-Kernel	252
	Out-Kernel	419

Table 6.1: In-Kernel vs. Out-Kernel File Times

1. The BSD /etc/inetd.conf file.

2. The "cold" times were measured with the cache "warmed" with the directory blocks. This was done to ensure that only the time relevant to file access was measured.

The results clearly show that moving an application into the kernel can have significant performance benefits. The "cold" time shows a 36% reduction, while that for the "warm" time is 40%. If these results scale to other aspects of system behaviour, such as network access, then it will be possible to realise substantially greater throughput than conventional servers running in user mode.

6.2 Application-Specific Extensions

Kea also allows applications to interpose services for their own use. This can allow applications to specialise the system in order to increase either their own performance or the total system throughput. One example of a service that can be interposed on an application specific basis is the compressed file service discussed in Chapter 5. By interposing this on the standard file service, any application can arrange to have the files used by that application transparently compressed. While this service does illustrate one example of application-specific remapping, it is primarily a service that is used to reduce the storage required by an application, as opposed to increasing it's performance. To experiment with this facet of application specificity, several other services were developed. These services were designed to be used at various levels of the file system hierarchy, and illustrate some of the trade-offs involved in supporting application specific service interposition. These services are discussed in the following sections.

6.2.1 The Read-Ahead Service

A common file access pattern is that of sequential reads. Many filesystems (including the FreeBSD filesystems used for comparisons in Chapter 5) detect this access pattern, and arrange to read ahead in the file. This saves time when the next read request is received, as either the disk I/O will be at least partially complete, or the requested block will already be in the buffer cache. For applications that perform sequential file accesses, with some processing time required after each read, it might therefore be possible for them to gain increased performance from the Kea filesystem by allowing them to read ahead. To accomplish this, a service designed

to be interposed on any other file service was designed. For every disk read operation, this service makes another read immediately following the requested one. This read is carried out in a thread specific to the read-ahead service, and completes asynchronously with respect to its client.

To measure the effectiveness of the read-ahead service, an application that sequentially reads 1 Mb of data in 8 Kb blocks was developed. The time required for each read was measured, and combined to get an aggregate read time. The application was also configured with a CPU bound loop after each read in order to simulate the per-block processing overhead present in any real application. Just running the application by itself is not sufficient to demonstrate the applicability of the read-ahead service, as the underlying filesystem (FFS) lays out files on contiguous disk blocks. In the absence of other I/O operations, this results in very small seek times, and total transfer times per block of under 3 ms. To mitigate this factor, and ensure that blocks were driven out of the disk controllers cache, a second application was introduced. This application continuously made 8 Kb reads from random locations within the disk partition, introducing a constant background "noise" into the disk (as would be expected in a typical timesharing system). The test (reading) application was then tested[1], with different loop delays, from 0 to 100 ms. The aggregate I/O time for the application, as plotted against the delay, is shown in Figure 6.1. The curve shown in the figure shows several interesting properties and requires some careful analysis. The most important aspect is that for the first 17 ms of CPU delay, the time required to complete the I/O's decreases as a linear, 1:1 function – that is, for every millisecond of delay, the average I/O completes a millisecond faster, or 128 milliseconds in aggregate over all the I/O's. This is the most important feature of the curve, particularly as 17 ms is far longer than would normally be expected for the processing of a single 8 Kb data block. Past 17 ms, the curve exhibits several distinct peaks and troughs, before finally converging to a fraction of a second (where all reads are satisfied by the buffer cache). The oscillations in the curve are caused by a complex set of interactions between the three active threads in the system

1. In the test described, the read-ahead service was configured as an application (user) level service, with all other parts of the filesystem hierarchy configured into the kernel.

(reader thread, read-ahead thread and the background thread), the average times they take for an I/O operation (about 17 ms for the read-ahead thread, and 20 ms for the background thread, due to its greater range of seek head movement) and the scheduler quantum (40 ms). Detailing the exact relationships between these factors that result in the curve shown is beyond the scope of the thesis – the important point is that read-ahead can result in an apparent decrease in read latency (as seen by the application installing the service).

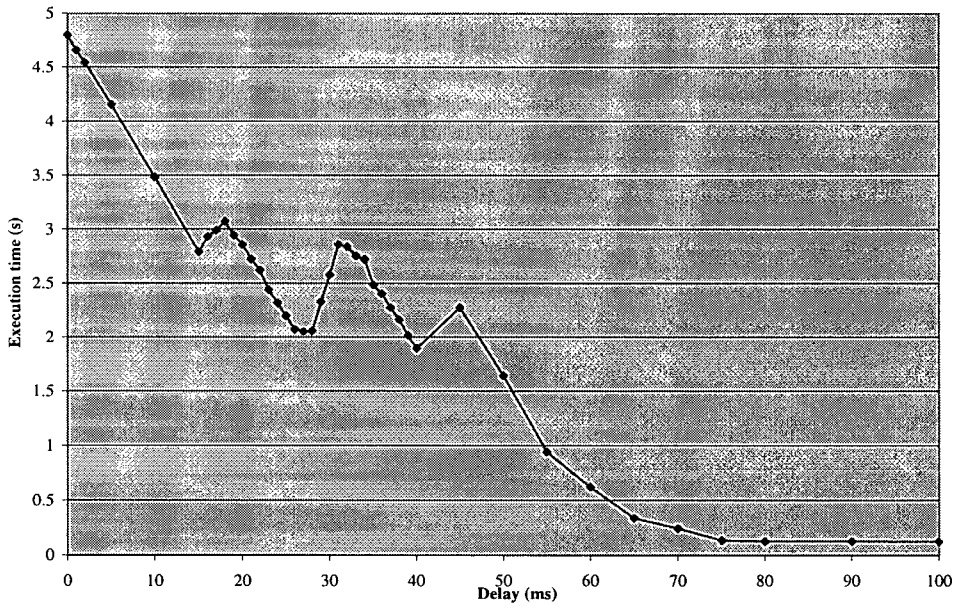

Figure 6.1: Aggregate I/O Time vs. Delay

6.2.2 Buffer Cache Management

Although the read ahead service allows an application with specific behaviour properties to specialise the system in order to obtain better performance, it is still a service that is interposed at a relatively high level in the filesystem hierarchy, and does not truly demonstrate that it is possible to efficiently manipulate lower level services in an application specific manner. To

demonstrate this feature of the Kea design, the buffer cache ("bcache") service was used. It has been shown that giving an application control over the replacement of blocks for files it owns in the buffer cache allows that application to increase its performance on several tasks [Lee et al. 94]. Instead of repeating one of these previously developed tests, a new type of service was developed to illustrate a novel viewpoint – that of not increasing a specific applications performance, but allowing that application to specialise the system in order to minimise its resource requirements, and ensure greater performance for other users of the system. This is accomplished by the "throw-away" service.

The "throw-away" Service

The majority of applications in any computer system read or write files sequentially. A reasonable proportion of these files are only utilised once, and are then not used again for a long (in computer terms) period of time. Examples of such applications are tar, cpio and compress, each of which are often used to manipulate large archive files. When these applications are used, a typical system will read (or write) the appropriate file, with the result that blocks from that file will be entered into the buffer cache. Unfortunately, the file is usually not manipulated again for some time, and these blocks force other, potentially more useful, blocks out of the cache. The "throw-away" service is designed to prevent this by being interposed on the buffer cache service, and modifying buffer cache requests in order to arrange that the blocks requested be thrown out of the cache before other blocks that are already resident. This is accomplished very simply by modifying the "hint" parameter to each of the buffer cache[1] request calls. The "hint" parameter determines how likely the given block is to be reused. For each hint value, the buffer cache contains a list in LRU order. The filesystems developed currently use either `HINT_KEEP` (the block is likely to be reused) or `HINT_DELETE` (the block is to be deleted from the cache). The "throw-away" service merely changes every `HINT_KEEP` value to `HINT_DISCARD` (remove the block from the cache sooner than any other block). Note that this does not result in the block being immediately removed from the cache, it is instead placed

1. The buffer cache interface is described in Section 4.2 and Appendix B.2.

at the end of the LRU list that is first considered for replacement when the cache has used up the available physical memory reserves.

To test the efficacy of the throw-away service, the memory available to the buffer cache was first artificially limited to one megabyte. Two applications were then run. The first application did 1000 random 8 Kb reads (that is, a read, followed by a seek to a random file location) within a 1 Mb file, while the second sequentially reads a 10 Mb file, also using 8 Kb blocks. The observed (wall clock) time required for each application to complete, and the total number of disk operations performed, were then recorded in two configurations. The first configuration was with the standard, unmodified system, while the second had the sequential reader interpose the "throw-away" service above the buffer cache service. The results are shown in Table 6.2.

	Time (s)		Disk I/O count	
Test	random	sequential	random	sequential
without throw-away	13.43	15.99	420	1280
with throw-away	4.08	7.53	128	1280

Table 6.2: Throw-away Service Results

Table 6.2 shows that the total number of disk operations and the perceived read latency for both applications drops substantially when the throw-away service is used, demonstrating that a very simple application specific modification can have a significant effect on the performance of the system as a whole.

6.3 Conclusions on Application Specificity

This chapter has presented some experiments in which Kea's ability to install application-specific extensions have been tested. While Kea has yet to be applied to the development of fully fledged applications that take advantage of these features, we believe that the test results shown clearly demonstrate Keas' potential in this area. However, there are several areas in which further investigation is needed before the Kea approach to application extensibility can be vali-

dated. In particular, there are many circumstances in which it will not be appropriate. The major issues to consider are those of performance, security, and the necessity for an equitable global policy.

6.3.1 Performance Issues

There is one vital performance issue associated with Kea's style of application specificity, which appears when services are co-located. Normally, such services call each other through the co-location stub, which has only marginally more overhead than an ordinary procedure call. However, since portal invocation is the point at which application specific remapping takes place on an interface, any services which are remapped in this fashion must instead use the portal invocation stub[1]. In the case where the services are kernel co-located, this adds only 0.2 μs to the invocation time for each call, a trivial amount. In the case where the services are not kernel co-located, the overhead may be several 10's of microseconds. While this overhead is relatively small, it is unrealistic to allow application specific remappings that will not, at least potentially, recoup this loss in performance. This is the case with the examples discussed in this chapter, for two separate reasons. In the case of the read-ahead service, it is designed to be used directly above the existing file service. In this circumstance, the application-specific remapping is on the file service, which resides at the top of the service hierarchy. Because of this location, any clients of the file service will already be using the portal invocation path, essentially negating any overhead due to the remapping. The throw-away service also meets the performance requirement, as its use has the potential to dramatically reduce the number of disk operations performed, each of which is many orders of magnitude more expensive than the remapping overhead. Also, as explained in Section 3.12.4, such services will probably be provided by systems vendors or other trusted entities, and be able to be kernel co-located, which reduces the overhead to a trivial amount.

1. This process is described in Section 3.10.2.

A second performance impact is that of naive or malicious applications using application specific services which are not suited to their requirements. Consider the use of the throw-away service for an application that instead does cyclical reads on a single file. In this case, the application could well cause the buffer cache service to continually read the disk blocks comprising the file, reducing the system throughput. Unfortunately, there is little that can be done about such behaviour. However, it should be noted that this is no worse than any other "standard" operating system, as an application can read large amounts of information from various files in the system, with similar affects, as this forces more legitimate blocks out of the cache. Also, the behaviour described only affects the system when the buffer cache has a need to recycle or release memory blocks, a behaviour which is somewhat independent of any individual application.

6.3.2 Security Issues

As discussed in Section 3.12.4, there are a number of security issues associated with application specific extensions. The most important of these are whether to allow the co-location of interposed services, and determining which services can be interposed on, and in which manner. The answer to the first of these questions depends entirely upon the implementor of the service – it is likely that services shipped by the operating system vendor or a major software manufacturer can be trusted, while those produced by ordinary system users cannot be. Decisions in this area can only be made by the system administrator.

The second issue, that of which services are appropriate for interposition, and how, is more complicated, as the answers depend entirely on the tasks undertaken by the service, the privileges it requires to accomplish the task, and the implications of any functionality changes on its clients. The multifaceted nature of this problem requires careful analysis, and is, for the most part, beyond the immediate scope of the thesis. The suggested solution involves the ability to tag every service with a set of attributes, which determine how trustworthy the service is (in particular, which address spaces it can be trusted to be co-located within) and which services it

can be interposed on, with the possibility of the latter being further specialised by categories of users. It is likely that recently developed technologies for code signing will be appropriate for this use. Deciding on and evaluating an appropriate set of these attributes is an important component of the future development of Kea and similar systems.

6.3.3 Policy Issues

Another important consideration when designing services is the policies they implement, and how these may, or should be, affected by application specific extensions. Many kernel services implement a global policy that ensures fairness for all applications using the system. An example is the buffer cache, which ensures that the most recently accessed file blocks are kept, and does not prefer the blocks of any one application over those of others. Generally speaking, it is important that users be prevented from installing or replacing services that will cause such policies to favour their applications. What may be acceptable is giving applications control over local policy, i.e. those decisions that only affect the application. An example might be letting an application specify an alternate cache block to be replaced, instead of one selected (for that application) by the global policy. The separation of services into global and local policies, where appropriate, is another open research issue. Current extensible systems address the mechanics of installing extensions, rather than the control, and appropriateness of, any policies implemented by that extension. It is likely that the answers to this issue will depend heavily on the structure and functionality of the system.

6.3.4 Application Specific Summary

The primary promise of Kea is in its support for in-kernel applications. This facility enables those selected applications that absolutely require high performance and/or direct access to low-level parts of the system to be easily developed. Often, these applications will be the only significant application running on the machine (examples might be dedicated file, WWW or database servers), for which the security and policy concerns discussed in the previous sections will not apply. Secondly, it has been demonstrated that for other, more general purpose appli-

cations, the judicious use of simple services can dramatically increase the system's performance. We believe that future configurable systems will allow both of these styles of extension, with the latter being restricted to specialised services either shipped with the system or made available by the system administrator. The utility of other application-specific extensions, and the degree to which they can be used, remains an open research issue.

CHAPTER 7

Conclusions and Future Work

7.1 Conclusions

The Kea operating system has shown that it is possible to design and implement a fully dynamically configurable and extensible operating system, and that this can be done without sacrificing performance. Several concepts were vital to the accomplishment of this task:

- System as a collection of services. A prerequisite to being configurable is having something that can be configured. The Kea system is viewed as being composed from a collection of services, each of which implements one part of the complete operating system. Each of these services can then be configured into various address spaces, composed in different fashions, or replaced by equivalent services. The service concept also includes that of isolating the interface of each service from its implementation.

- Procedure call oriented IPC. Making communications between services (IDCs, or Inter-Domain Calls, in Kea parlance) appear as procedure calls instead of message passing

considerably simplifies both the engineering of service composition and the generation of service stubs. While an undesirable side affect may be extra cost for simple procedure calls, this is offset by increased performance for more complex calls, particularly those that pass memory buffers.

- Unification of kernel and user programming environments. Kea offers developers an identical programming environment at both user and kernel levels. This ensures that services can be easily located in any address space, making reconfiguration possible. It also ensures that system extensions in the form of in-kernel applications are able to be easily and transparently developed.

- Full co-location of services. As performance is one of the most important considerations for operating system acceptance, Kea optimises the IDC's to (almost) direct procedure calls when services are co-located within an address space. This is managed by the kernel, and is totally transparent to the services. This facility is only made simple by a combination of the previous points.

- Portal remapping. The kernel level view of an IDC entry point is the portal. The kernel service manipulation routines modify portal identifiers and service stubs in order to transparently reconfigure the inter-service relationships. Portals also permit application-specific remappings, which allows portal invocations to be re-routed to other services, dependant upon the application at the root of the call chain, making various application-specific extensions possible.

Through a combination of these points, Kea accomplishes the thesis goals of a configurable and extensible operating system. By careful design, features such as service migration and interposition allow the system to be configured in many different ways.

Kea offers many advantages, particularly to system developers and administrators. Services can be developed and debugged in a user address space, and then transparently relocated into the kernel for performance purposes. Upgrading the system becomes a simple matter of replac-

ing a service, which can be done dynamically, without the need to recompile, shutdown and reboot the machine on which the system is currently running. In summary, by giving increased control of service location and configuration to the system developers and administrators, and letting services be co-located when safe and appropriate, the Kea system can give the best of all worlds – safety, modularity and performance, depending upon the system requirements.

Reviewing the thesis, Chapter 3 provides a detailed design of the base Kea abstractions, concentrating on the algorithms and data structures developed for service manipulation, and briefly covered the low level services inherent to Kea. Chapter 4 describes the higher level services that make up the current Kea system, primarily those required for filesystem support. Chapter 5 then uses these services in several experiments designed to measure the overhead of system reconfigurability. The primary conclusion made in this chapter is that the Kea design may have some minor performance degradation when compared to a standard monolithic system, but that overall, the design of individual components has a far greater impact. Chapter 6 concentrates on application-specific extensions to the system, with results showing that some simple services can dramatically increase performance for selected application behaviours. Overall, the conclusion is drawn that configurable and extensible systems such as Kea have much to offer, both in terms of convenience and performance.

7.2 Future Work

During the design and implementation of Kea, many interesting issues arose, several of which there was not the time to pursue. The following sections examine some of the work that remains to be done in order to make Kea fully functional, and describes some of the research directions arising from the thesis work.

7.2.1 Further development

Although Kea offers many of the services of a traditional operating system, several of these are still in a raw or undeveloped stage. In particular, the networking services of the *x*-kernel need

to be converted to real services. Simply using the *x*-kernel as a single enormous service would be a necessary first step, but the splitting of this into separately configurable services would be required in order to fully evaluate the necessity for, and performance of, such a system. Although the *x*-kernel is already statically configurable, and includes a relatively clean inter-protocol messaging mechanism, substantial work would be required in order to convert many of the components into services, due to various assumptions made about the sharing of memory buffers and synchronisation made by the system.

There are several other obvious areas of functionality where Kea could be improved. Currently only a small number of devices are supported, and increasing this number would be an important step in making the system more functional. Also, many of the services that are provided are very simple. The only terminal (keyboard and screen) handling that is provided is very raw, and is an obvious area where a stack of services could be used. On the application side, it would be interesting to provide libraries that offer POSIX [IEEE 90] compatibility[1], to more easily enable the porting of other programs to Kea.

Another obvious area of improvement is in the portal invocation code. Although it was shown in Section 3.15 that portal invocation is relatively fast, it is still not as efficient as is theoretically possible. In order to achieve the best performance, it will probably be necessary to rewrite this part of the kernel in customised assembly language, and re-evaluate the trap-handling and argument processing functionality. This will make it possible to take advantage of various machine specific optimisations that are only available to assembly programs.

As is true of any system of Kea's functionality and size, there are also still bugs and errors in some parts of the system. Although these are fixed as they are found, it would be useful to design a set of tests for each part of the system, in order to identify and remove as many of these as possible.

1. Some progress in this area has been achieved in the area of filesystems – it is currently possible to run GNU diff on the system.

7.2.2 Finer Grain Decomposition

One of the premises underlying reconfigurability is the presence of services that can be reconfigured. The granularity of these services, i.e. how small the functionality of the system is split, is a major factor in determining the number of possible useful service configurations. Although Kea demonstrates that filesystem reconfiguration is possible, each of the filesystems is a large monolithic system, with fixed policies on behaviour such as file layout and access characteristics. While some services have been presented that can modify some of these policies, it would enhance the reconfigurability of this aspect of system behaviour if the filesystem could be composed from smaller services. With recent research on this topic, the Hurricane file system [Krieger & Stumm 97] has demonstrated that it is possible to construct a highly flexible and customisable filesystem. Hurricane composes filesystems from primitive building blocks in a manner that is very similar to service composition in Kea. Similar research has also been performed with network protocols [Hutchinson et al. 89, Bhatti & Schlichting 95], demonstrating that the fine-grain decomposition of services is possible in more than one domain. It would be a valuable exercise to use this technology in conjunction with Kea's reconfigurable design to further demonstrate the utility and applicability of reconfigurable systems.

7.2.3 Improved State Migration

When migrated, services must package any state they wish to be retained between domains into a contiguous memory region, which is then passed to the new service instantiation as an argument for unpackaging. Generally speaking, a service has two "types" of state that need to be transferred. The first, "static" state, is the contents of various local variables and allocated memory. The second, "active" state, is information about any threads that have been created by the service (such as the interrupt thread created by device drivers, or the read-ahead threads in the read-ahead service), namely their existence and current CPU state. It should be possible to combine the technologies developed for process migration in other operating systems [Nuttall 94] with new techniques for heterogeneous process migration [Smith 97] and remove all, or at least a substantial part of, the requirement for service developers to write the code for

service state migration. This would considerably ease the development of future services, while also increasing Kea's functionality.

7.2.4 Dynamic Code Generation

Systems such as Synthesis [Pu et al. 88] and Synthetix [Cowan et al. 96], as well as recent techniques for dynamic code generation [Auslander et al. 96] have demonstrated that it is possibly to efficiently generate code to specialise system operations. An interesting application of this technology would be to the portal invocation path. Currently, the kernel executes a general code path, that must be capable of handling all possible procedure signatures, copying any parameters and data between domains as appropriate. Instead, it should be possible to generate specialised code for each portal. This would result in major efficiencies within the portal invocation code, and presumably enhance the performance of the system considerably.

7.2.5 IO Buffer Manipulation

One of the major performance problems with configuring services in different address spaces is the need to copy procedure parameters. While this can be accomplished relatively quickly for small arguments, copying larger areas of memory (typically those involved in I/O, such as file data) is more time consuming, and also results in double buffering of data, as multiple blocks of memory are allocated to hold essentially identical information. In order to reduce this overhead, it would be desirable to implement a system in which special areas of memory could be set aside as I/O buffers, and arrange for the portal invocation code to map this memory into each of the domains in the invocation path. This would eliminate copying, as each domain would be able to directly access the memory provided. Two systems, fbufs [Druschel & Peterson 93] and container shipping [Pasquale et al. 94] make use of similar ideas for arranging copy-free I/O paths in standard operating systems, while the Scout operating system [Montz et al. 95] uses similar technology to support "paths" between producers and consumers of information. Implementing this extension would require the development of data structures describing the domains for each IO buffer, and an extension to the procedure signa-

ture types (I/O buffers would be another argument type, extending the current pointer and string types), together with the appropriate application support to make effective use of these features.

7.2.6 Service Format

Currently, service files are represented on disk in the native object file format of the destination architecture, and internally by data structures describing the text and data segments, relocations inside those segments, and a symbol table. Currently, one of the major costs in migrating service between address spaces is the relinking stage. It would be interesting to investigate alternative file formats and data structures in order to increase the speed of the relinking process.

7.2.7 Application Specific Extensions

One of the more interesting applications of Keas configurability is in the area of application specific system extensions. While the experiments described in Chapter 6 showed that it is possible to construct application specific services that can result in increased performance and system throughput, it would be valuable to construct more of these services, and evaluate their effect with real application loads. Particularly interesting is the development of "network appliance" type systems, in which the operating system is configured to support only one major application, providing services to other machines on the network. With its high configurability and ability to easily run applications within the kernel, Kea should be ideal for development of this style of system.

Related to the network appliance style of system, it may be interesting to redesign the low level service currently provided by the kernel. These were designed in their current form in order to more easily support the development of other, high-level, services. It should be possible to redesign the low level service closer to the hardware, and provide services such as virtual memory as high-level services. This would allow the configuration of systems with fundamentally different modes of operation. An example might be embedded systems, which seldom require

full virtual memory support, but are designed to run in a fixed address space, or systems that require address spaces, but not paging.

Another related area is the determination of security guidelines for application specific extensions. As described in Chapter 6, there is a need for a means to specify which services are trustworthy, and where and how they can be installed. What these properties should be, and how they are determined, is one of the most challenging research issues remaining for extensible systems.

Bibliography

Anderson *et al.* 91 Thomas E. Anderson, Henry M. Levy, Brian N. Bershad and Edward D. Lazowska. The Interaction of Architecture and Operating System Design. In *Proceedings of the 1991 International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, April 1991, pp. 108-120

Anderson *et al.* 92 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), February 1992, pp. 53-79

Appel & Li 91 Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991, pp. 96-107

Auslander *et al.* 96 J. Auslander, M. Philipose, C. Chambers, S.J. Eggers and B.N. Bershad. Fast, Effective Dynamic Compilation. In *Proceedings of the Conference on Programming Language Design and Implementation*, May 1996

Banâtre *et al.* 91 Michel Banâtre, Pack Heng, Gilles Muller and Bruno Rochat. How to Design Reliable Servers Using Fault Tolerant Micro-Kernel Mechanisms. In *Proceedings of the USENIX Mach Symposium*, November 1991, pp. 223-232

Banerji *et al.* 97 Arindam Banerji, John M. Tracey and David L. Cohn. Protected Shared Libraries – A New Approach to Modularity and Sharing. In *Proceedings of the USENIX 1997 Annual Technical Conference*, January 1997, pp. 59-75

Bartoli *et al.* 93 Alberto Bartoli, Sape J. Mullender and Martijn van der Valk. Wide-Address Spaces – Exploring the Design Space. *ACM Operating Systems Review*, 27(1), January 1993, pp. 11-17

Bershad & Pinkerton 88 Brian N. Bershad and C. Brian Pinkerton. Watchdogs – Extending the UNIX File System. *Computing Systems*, 1(2), Spring 1988, pp. 169-188

Bershad *et al.* 89 Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska and Henry M. Levy. Lightweight Remote Procedure Call. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles*, December 1989, pp. 102-113

Bershad 92 Brian N. Bershad. The Increasing Irrelevance of IPC Performance for Microkernel-Based Operating Systems. In *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, April 1992, pp. 205-212

Bershad *et al.* 95 Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995, pp. 267-284

Bhatti & Schlichting 95 Nina T. Bhatti and Richard D. Schlichting. A System for Constructing Configurable High-Level Protocols. In *Proceedings of SIGCOMM '95*, August 1995, pp. 138-150

Birrell 89 Andrew D. Birrell. *An Introduction to Programming With Threads*. Technical Report SRC-35, DEC Systems Research Center, January 1989

Black *et al.* 92 David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan and David Bohman. Microkernel Operating System Architecture and Mach, In *USENIX Workshop on Microkernels and Other Kernel Architectures*, April 1992, pp. 11-30

Bolosky *et al.* 96 W.J. Bolosky, J.S. Barrera III, Richard P. Draves, R.P. Fitzgerald, G.A. Gibson, Michael B. Jones, S.P. Levi, N.P. Myhrvold and Richard F. Rashid. The Tiger Video Fileserver. In *Proceedings of the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video*, April 1996

Cao *et al.* 94 Pei Cao, Edward W. Felten and Kai Li. Implementation and Performance os Application-Controlled File Caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994, pp. 165-177

Carter *et al.* 94 N.P. Carter, S.W. Keckler and W.J. Dally. Hardware Support for Fast Capability-Based Addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, October 1994, pp. 319-327

Chambers *et al.* 96 Craig C. Chambers, Susan J. Eggers, J. Auslander, M. Philipose, M. Mock and Przemyslaw Pardyak. Automatic Dynamic Compilation Support for Event Dispatching in Extensible Systems. In the *Workshop on Compiler Support for System Software*, February 1996

Chase *et al.* 93 Jeffrey S. Chase, Valérie Issarny and Henry M. Levy. Distribution in a Single Address Space Operating System. *ACM Operating Systems Review*, 27(2), April 1993, pp. 61-65

Chaum & van Antwerpen 90 D. Chaum and H. van Antwerpen. Undeniable Signatures. *Advances in Cryptology – CRYPTO '89 Proceedings*, Springer-Verlag, 1990, pp. 212-216

Cheriton & Duda 94 David R. Cheriton and Kenneth J. Duda. A Caching Model of Operating System Kernel Functionality. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994, pp. 179-193

Clark 85 David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, December 1985, pp. 171-180

Condict *et al.* 94 Michael Condict, Don Bolinger, Dave Mitchell and Eamonn McManus. Microkernel Modularity with Integrated Kernel Performance. Presented at the *Mach-Chorus Workshop at the First Symposium on Operating Systems Design and Implementation*, November 1994. Available at `http://www.cs.utah.edu/~lepreau/osdi94/condict/abstract.html`

Corbato *et al.* 72 F.J. Corbato, J. H. Saltzer and C.T. Clingen. Multics – The First Seven Years. In *Proceedings of the American Federation of Information Processing Societies Spring Joint Computer Conference*, 1972, pp. 571-583. Reprinted in P. Freeman, *Software Systems Principles*, Science Research Associates, 1975.

Cowan *et al.* 96 Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu and Jonathon Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In *Proceedings of the International Conference on Configurable Distributed Systems*, May 1996

Draves *et al.* 91 Richard P. Draves, Brian N. Bershad, Richard F. Rashid and Randall W. Dean. Using Continuations to Implement Thread Management and Communication In Operating Systems. In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles*, December 1991, pp. 122-136

Draves & Cutshall 97 Richard P. Draves and Scott M. Cutshall. *Unifying the User and Kernel Environments*. Microsoft Research Technical Report MSR-TR-97-10

Druschel *et al.* 91 Peter Druschel, Larry L. Peterson and Norman C. Hutchinson. Service Composition in Lipto. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, October 1991, pp. 108-111

Druschel *et al.* 92a Peter Druschel, Larry L. Peterson and Norman C. Hutchinson. Beyond Microkernel Design: Decoupling Modularity and Protection in Lipto. In *Proceedings of the Twelfth International Conference on Distributed Computing Systems*, June 1992, pp. 512-520

Druschel *et al.* 92b Peter Druschel, Larry L. Peterson and Norman C. Hutchinson. Modularity and Protection Should be Decoupled. In *Proceedings of the Third Workshop on Workstation Operating Systems*, April 1992, pp. 95-97

Druschel 93 Peter Druschel. Efficient Support for Incremental Customization of OS Services. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*, December 1993, pp. 168-190

Druschel & Peterson 93 Peter Druschel and Larry L. Peterson. Fbufs: A High-Bandwidth Cross-Domain Transfer Utility. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 189-202

Engler *et al.* 95 Dawson R. Engler, M. Frans Kaashoek and James O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995, pp. 251-266

Engler & Kaashoek 95 Dawson R. Engler and M. Frans Kaashoek. Exterminate All Operating System Abstractions. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, May 1995, pp. 78-83

Fall & Pasquale 94 K. Fall and J. Pasquale. Improving Continuous-Media Playback Performance with In-Kernel Data Paths. In *Proceedings of the First IEEE International Conference on Multimedia Computing and Systems*, May 1994, pp. 100-109

Finkelstein *et al.* 95 David Finkelstein, Norman C. Hutchinson, Dwight J. Makaroff, Roland Mechler and Gerald W. Neufeld. *Real Time Threads Interface*. Computer Science, University of British Columbia Technical Report TR-95-07, March 1995

Ford & Lepreau 94 Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a Migrating Thread Model. In *Proceedings of the 1994 Winter USENIX Conference*, January 1994, pp. 97-114

Ford *et al.* 96 Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back and Stephen Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, October 1996, pp. 137-152

Ford *et al.* 97 Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997, pp. 38-51

Gingell 89 Robert A. Gingell. Shared Libraries. *Unix Review*, 7(8), August 1989, pp. 56-66

Golub *et al.* 90 David Golub, Randall W. Dean, Alessandro Forin and Richard F. Rashid, Unix as an Application Program. In *Proceedings of the 1990 Summer USENIX Conference*, June 1990, pp. 87-95

Guillemont *et al.* 91 Marc Guillemont, Jim Lipkis, Douglas Orr and Marc Rozier. A Second Generation Micro-Kernel Based Unix; Lessons in Performance and Compatibility. In *Proceedings of the Winter 1991 USENIX Conference*, 1991, pp. 13-22

Hamilton & Kougiouris 93 Graham Hamilton and Panos Kougiouris. The Spring Nucleus: A Microkernel for Objects. In *Proceedings of the 1993 Summer USENIX Conference*, June 1993, pp. 147-159

Härtig *et al.* 97 Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg and Jean Wolter. The Performance of µ-Kernel-Based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, October 1997, pp. 66-77

Harty & Cheriton 91 K. Harty and D.R. Cheriton. Application-Controlled Physical Memory using External Page-Cache Management. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, April 1991, pp. 187-197

Heiser *et al.* 93 Gernot Heiser, Kevin Elphinstone, Stephen Russell and Jerry Vochteloo. Mungi: A Microkernel for Objects. In P*roceedings of the 1993 Summer USENIX Conference*, June 1993, pp. 147-159

Hildebrand 92 Dan Hildebrand. An Architectural Overview of QNX. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992

Hsieh *et al.* 96 Wilson Hsieh, Marc Fiuczynski, Charles Garrett, Stefan Savage, David Becker and Brian Bershad. Language Support for Extensible Operating Systems. In the *Workshop on Compiler Support for System Software*, February 1996

Hutchinson & Peterson 88 Norman C. Hutchinson and Larry L. Peterson. Design of the *x*-Kernel. In *Proceedings of the SIGCOMM 1988 Symposium*, August 1888, pp. 65-75

Hutchinson *et al.* 89 Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott and Sean O'Malley. RPC in the *x*-Kernel: Evaluating New Design Techniques. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989, pp. 91-101

IEEE 90 *IEEE Std. 1003.1-1990 Standard for Information Technology – Portable Operating System Interface (POSIX) – PART 1: System Application Programming Interface (API) [C Language]*. IEEE, Piscataway, NJ

Intel 90 Intel Corporation. *i486 Microprocessor Programmers Reference Manual.* Osborne McGraw-Hill, 1990

Intel 94 Intel Corporation. *Pentium Microprocessor Programmers Reference Manual.* Osborne McGraw-Hill, 1993

Jones 93 Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 80-93

Kaashoek *et al.* 97 M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinkney, Robert Grimm, John Jannotti and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, October 1997, pp. 52-65

Khalidi & Nelson 93a Yousef A. Khalidi and Michael N. Nelson. An Implementation of UNIX on an Object-oriented Operating System. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993, pp. 469-479

Khalidi & Nelson 93b Yousef A. Khalidi and Michael N. Nelson. Extensible File Systems in Spring. *Operating Systems Review*, 27(5), December 1993, pp. 1-13

Kiczales *et al.* 91 Gregor Kiczales, Jim des Riviéres and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991

Kiczales *et al.* 92 Gregor Kiczales, M. Theimer and Brent Welch. A New Model of Abstraction for Operating System Design. In *Proceedings of the International Workshop on Object-Oriented Operating Systems*, 1992, pp. 346-350

Krieger & Stumm 97 Orran Krieger and Michael Stumm. HFS: A Performance-Oriented Flexible File System Based on Building-Block Compositions. *ACM Transactions on Computer Systems*, 15(3), August 1997, pp. 286-321

Kulkarni 93 Dinesh C. Kulkarni. *Pi: A New Approach to Operating System Structuring for Flexibility.* Technical Report 93-4, Department of Computer Science and Engineering, University of Notre Dame, April 1993

Lee *et al.* 94 Chao-Hsien Lee, Meng Chang Chen and Ruei-Chuan Chang. HiPEC: High Performance External Virtual Memory Caching. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, November 1994, pp. 153-164

Lepreau *et al.* 93 Jay Lepreau, Mike Hibler, Bryan Ford, Jeffrey Law and Douglas Orr. In-Kernel Servers in Mach 3.0: Implementation and Performance. In *Proceedings of the USENIX Mach III Symposium*, April 1993, pp. 39-55

Levin *et al.* 75 R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/Mechanism Separation in Hydra. In P*roceedings of the Fifth ACM Symposium on Operating Systems Principles*, November 1975, pp. 132-140

Liedtke *et al.* 93 Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 175-188

Liedtke *et al.* 95 Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, December 1995, pp. 237-250

Liedtke *et al.* 97 Jochen Liedtke, Kevin Elphinstone, Sebastian Schönberg, Hermann Härtig, Gernot Heiser, Nayeem Islam and Trent Jaeger. Achieved IPC Performance (Still the Foundation for Extensibility), In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, May 1997, pp. 28-31

Maeda & Bershad 93 C. Maeda and Brian N. Bershad. Protocol Service Decomposition for High-Performance Networking. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 244-255

Maes 87 P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages and Applications*, 1987, pp. 147-155

Malan *et al.* 91 Gerald Malan, Richard Rashid, David Golub and Robert Baron. DOS as a Mach 3.0 Application. In *Proceedings of the USENIX Mach Symposium*, November 1991, pp. 27-40

Massalin & Pu 89 Henry Massalin and Calton Pu. Threads and Input/Output in the Synthesis Kernel. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989, pp. 191-201

McCanne & Jacobsen 93 S. McCanne and Van Jacobsen. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993

McKusick 82 Kirk McKusick. 4.2BSD File System. In *Proceedings of the USENIX Tools Users Group Joint Conference*, July 1982, pp. 31-45

McKusick *et al.* 96 Marshall K. McKusick, Keith Bostic, Michael J. Karels and John S. Quaterman. *The Design and Implementation of the 4.4 BSD Operating System.* Addison Wesley, 1996

McNamee & Armstrong 90 Dylan McNamee and Katherine Armstrong. Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies. In *Proceedings of the USENIX Mach Workshop*, October 1990, pp. 17-29

Microsoft 97 Microsoft Corporation. *How Software Publishers Can Use Authenticode Technology.* `http://www.microsoft.com/intdev/signcode`

Mogul *et al.* 87 Jeff Mogul, Richard Rashid and M.J. Accetta. The Packet Filter: An Efficient Mechanism for User-level Network Code. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987, pp. 39-52

Montz *et al.* 95 Allen B. Montz, David Mosberger, Sean W. O'Malley, Larry L. Peterson and Todd A. Proebsting. Scout: A Communications-Oriented Operating System. In *Proceedings of the 4th Workshop on Hot Topics in Operating Systems (HotOS-IV)*, May 1995

Mukherjee & Schwan 93 Bodhisattwa Mukherjee and Karsten Schwan. Experimentation with a Reconfigurable Micro-Kernel. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, September 1993, pp. 45-60

Necula & Lee 96 George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, October 1996, pp. 229-243

Nelson 91 G. Nelson (editor). *System Programming in Modula-3.* Prentice-Hall, 1991

Nuttall 94 Mark Nuttall. A Brief Survey of Systems Providing Process or Object Migration Facilities. *Operating Systems Review*, October 1994

Organick 72 Elliot I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, 1972.

Orr *et al.* 93 Douglas B. Orr, John Bonn, Jay Lepreau and Robert Mecklenburg. Fast and Flexible Shared Libraries. In *Proceedings of the 1990 USENIX Summer Conference*, June 1993, pp. 237-251

Ousterhout 90 John K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the USENIX Summer Conference*, June 1990, pp. 247-256

Pardyak & Bershad 96 Przemyslaw Pardyak and Brian Bershad. Dynamic Binding for an Extensible System. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, October 1996, pp. 201-212

Pasquale *et al.* 94 Joseph Pasquale, Eric Anderson and P. Keith Muller. Container Shipping: Operating System Support for I/O-Intensive Applications. *Computer*, 27(3), March 1994, pp. 84-93

Peterson *et al.* 90 Larry L. Peterson, Norman C. Hutchinson, Sean W. O'Malley and Herman C. Rao, The *x*-kernel: A Platform for Accessing Internet Resources. *Computer*, 23(5), May 1990, pp. 23-33

Pu *et al.* 88 Calton Pu, Henry Massalin and J. Ioannidis. The Synthesis Kernel. *Computing Systems*, 1(1), Winter 1988, pp. 11-32

Pu *et al.* 95 Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathon Walpole and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, December 1995, pp. 314-324

Radia 89 Sanjay Radia. *Names, Contexts and Closure Mechanisms in Distributed Computing Environments*. Ph.D. Thesis, University of Waterloo, Department of Computer Science, 1989

Rees *et al.* 86 Jim Rees, Paul H. Levine, Nathaniel Mishkin and Paul J. Leach. An Extensible I/O System. In *Proceedings of the 1986 Summer USENIX Conference*, June 1986, pp. 114-125

Ritchie 79 Dennis M. Ritchie. *Protection of Data File Contents*, United States Patent (4,135,240), United States Patent Office (January 16, 1979). Assignee: Bell Telephone Laboratories, Inc., Murray Hill, NJ, Appl. No: 377,591, Filed: July 9, 1973

Rivest 92 R. Rivest. The MD5 Message-Digest Algorithm. *Network Working Group RFC 1321*, April 1992.

Rosenblum *et al.* 95 Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmett Witchel and Anoop Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995

Rozier *et al.* 92 Marc Rozier, Vadim Abrossimov, François Armand, I. Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, C. Kaiser, S. Langlois, P. Léonard and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992, pp. 39-70

Sabatella 90 Marc Sabatella. Issues in Shared Library Design. In *Proceedings of the Summer 1990 USENIX Conference*, June 1990, pp. 11-24

Schroeder & Burroughs 89 M.D. Schroeder and M. Burroughs. Performance of the Firefly RPC. In *Proceedings of the Twelfth ACM Symposium On Operating Systems Principles*, December 1989, pp. 83-90

Schulman *et al.* 92 A. Schulman, D. Maxey and M. Pietrek. *Undocumented Windows*, Addison-Wesley, 1992.

Scott *et al.* 88 Michael L. Scott, Thomas J. LeBlanc and Brian D. Marsh. Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System. In *Proceedings of the 1988 International Conference on Parallel Processing*, 1988, pp. 255-262

Seltzer *et al.* 96 Margo I. Seltzer, Yasuhiro Endo, Christopher Small and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, October 1996, pp. 213-227

Sirer *et al.* 96 Emin Gün Sirer, Marc Fiuczynski, Przemyslaw Pardyak and Brian Bershad. Safe Dynamic Linking in an Extensible Operating System. In *Proceedings of the Workshop on Compiler Support for System Software*, February 1996

Small & Seltzer 96 Christopher Small and Margo Seltzer. A Comparison of OS Extension Technologies. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996, pp. 41-54

Smith 97 Peter W. Smith. *The Possibilities and Limitations of Heterogeneous Process Migration*. Ph.D. Thesis, Computer Science, University of British Columbia, October 1997

Temple 94 Philip Temple. The Feisty Parrot. *New Zealand Geographic Magazine*, October 1994. Also available at `http://www.kiwihome.co.nz/magazine/NZGeographic/Kea-The_feisty_parrot.html`.

van Renesse *et al.* 88 R. van Renesse, H. van Staveren and Andrew S. Tanenbaum. Performance of the World's Fastest Distributed Operating System. *Operating Systems Review*, 22(4), October 1988, pp. 25-34

Veitch 96 Alistair C. Veitch and Norman C. Hutchinson. Kea – A Dynamically Extensible and Configurable Operating System Kernel. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, May 1996, pp. 236-242

Veitch 98 Alistair C. Veitch and Norman C. Hutchinson. Dynamic Service Reconfiguration and Migration in the Kea Kernel. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, May 1998, pp. 156-163

Wahbe *et al.* 93 Robert Wahbe, Steven Lucco, Thomas E. Anderson and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, December 1993, pp. 175-188

Walpole *et al.* 92 Jonathon Walpole, Jon Inouye and Ravindranath Konoru. Modularity and Interfaces in Micro-Kernel Design and Implementation: A Case Study of Chorus on the HP PA-RISC. In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, April 1992, pp. 71-82

Wulf *et al.* 74 William Wulf, E. Cohen, W. Corwin, A. Jones, Roy Levin, C. Pierson and F. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6), June 1974, pp. 337-345

Wulf *et al.* 81 William Wulf, Roy Levin and Samuel P. Harbison. *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, 1981

Yokote 92 Yasuhiko Yokote. The Apertos Reflective Operating System: The Concept and its Implementation. In *Proceedings of the Seventh Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, October 1992, pp. 414-434

APPENDIX A

Kernel Service Interfaces

This appendix contains details of the interface to each of the services provided by the Kea kernel, with the exception of the service interface, which is discussed in detail in chapter 3.

A.1 Domain Interface

Domains are virtual address spaces. A new domain is created with the `domainCreate` call:

```
int domainCreate();
```

This function creates a new domain, and returns its identifier. Domains are destroyed with the `domainTerminate` call, which takes the identifier of the domain to be destroyed as its argument:

```
int domainTerminate(int id);
```

All the threads in a domain can be suspended and resumed using the `domainSuspend` and `domainResume` calls:

```
int domainSuspend(int id);
int domainResume(int id);
```

The user identifier of the domain owner can be set and retrieved with the `domainSetOwner` and `domainGetOwner` calls:

```
int domainSetOwner(int id, unsigned owner);
int domainGetOwner(int id);
```

Finally, the identity of the current domain can be found by calling the `domainID` function:

```
int domainID();
```

A.2 VM Interface

The "vm" interface allows for the manipulation of a domain's virtual memory. New memory is allocated with the `vmAllocate` procedure:

```
int vmAllocate(int domain, vaddr *address, unsigned size,
          vaddr exact);
```

In this procedure, `domain` is the domain in which memory is to be allocated, `size` is the size of the desired memory region in bytes (this will be rounded up to the nearest number of whole pages), and `address` is used to return the address of the allocated memory. If `exact` is not zero, then the vm service will attempt to allocate the memory at the specified address.

Virtual memory can also be allocated, and backed by a specified physical address range. This functionality is used by device drivers that need to access physical I/O locations mapped into the processors physical address space. The `vmPhysAllocate` function is used for this purpose:

```
int vmPhysAllocate(int domain, paddr paddress,
                   vaddr *vaddress, unsigned npg);
```

In this function, `domain` specifies the target domain for the allocation, `paddress` is the physical address, `vaddress` the (returned) virtual address to which the physical address is mapped and `npg` the number of pages to be allocated.

Virtual memory is freed with the `vmFree` function:

```
int vmFree(int domain, vaddr address, unsigned size);
```

where `domain` specifies the domain, `address` the virtual address within the domain (this address should be within a range returned from `vmAllocate` or `vmPhysAllocate`, and will be rounded down to the nearest page boundary) and `size` the size (number of bytes) of memory to be freed.

A region of memory can be made read-only or read-write (the default) with the `vmProtect` procedure:

```
int vmProtect(int domain, vaddr address, unsigned size,
              bool read);
```

The domain and memory region is specified as in the other procedures, while the `read` parameter determines the protection of the memory region – true for read-only, false for read-write.

Memory can be "pinned" or made unpageable with the `vmLock` procedure, and unpinned with the `vmUnlock` procedure[1].

```
vmLock(int domain, vaddr address, unsigned size);
vmUnlock(int domain, vaddr address, unsigned size);
```

1. Kea does not yet have paging support, so these procedures remain largely unimplemented.

The final vm function is vmMap, used to map various memory regions from one domain to another. The prototype for this procedure is:

```
int vmMap(int srcDomain, vaddr srcAddress,
          unsigned size, int dstDomain,
          vaddr *dstAddress, bool exact,
          bool dstProt, int mapping);
```

The first three parameters determine the domain and virtual region from which the memory is to be mapped. The next three parameters determine the destination domain and region, in a manner similar to that of vmAllocate. The dstProt parameter acts to determine the protection of this memory (as for vmProtect). The final parameter, mapping, determines the type of mapping to be made. Currently, three different values are supported:

- VM_MAP_COPY – The memory is copied to the target domain. Any references in either domain do not affect the other. For efficiency, the copy is lazily evaluated using copy-on-write.
- VM_MAP_SHARE – the underlying physical pages are shared between domains. Any update done by one will also be seen by the other.
- VM_MAP_MOVE – the memory range is moved into the target domain. This is equivalent to a vmMap with VM_MAP_SHARE, followed by a vmFree in the source domain.

A.3 Event Interface

Events provide support for the asynchronous delivery of significant system events, such as page faults, interrupts and domain termination. A domain registers itself for an event by calling eventRegister:

```
int eventRegister(int domain, unsigned event,
                  void *handler);
```

eventRegister specifies the domain to receive the event, an event identifier, and the address of a function that will be called when the event occurs. The `eventDeregister` function is used to remove notifications of an event.

```
int eventDeregister(int domain, unsigned event);
```

Finally, the `eventWait` call is used by a specific thread to wait for an event. The function takes a pointer to an argument that will be filled in by the kernel on event reception (all event handlers also get an argument as a parameter when invoked).

```
int eventWait(unsigned event, int *arg);
```

A.4 Name Interface

The name interface provides a very simple hierarchical naming structure for the system. New names are registered with the `nameRegister` call:

```
int nameRegister(const char *name, bool dir, int id);
```

`nameRegister` takes as parameters a string (the name to be created) a flag describing whether the name to be created is a directory (container for other names) or a plain name, and an identifier to be associated with the name. The identifier can represent entities such as domains, threads or services, depending upon the application's purposes. Names are removed from the system with `nameDeregister`:

```
int nameDeregister(const char *name);
```

Applications can determine whether a name exists with the `nameExists` call:

```
int nameExists(const char *name);
```

Sometimes, applications may need to wait until a name is defined before they can continue. The `nameWait` call fulfills this need:

```
int nameWait(const char *name);
```

The identifier associated with a name can be determined by the `nameGetID` call:

```
int nameGetID(const char *name);
```

For directory names, the count of names in the directory can be found using `nameDirentries`, while the name with a given index can be found using `nameGetDirentry`:

```
int nameDirentries(const char *dir);
int nameGetDirentry(const char *dir, unsigned index,
                    char *name);
```

Finally, a new naming service can be "attached" to an existing name by using the `nameAttach` call. This will result in all function calls on names that begin with the `name` prefix (i.e. the first argument to `nameAttach`) being passed to the service identified for completion.

```
int nameAttach(const char *name, int service);
```

A.5 Thread Interface

The thread service supports threads, the Kea context of execution. New threads are created using the `threadCreate` call:

```
int threadCreate(int domain, vaddr entry,
                 unsigned maxstack, void *argp,
                 unsigned nargs);
```

This function creates a new thread in the specified domain, at the entry point `entry`. The stack size allocated to the thread is determined by the `maxstack` argument. Finally, `argp` is a

pointer to a memory buffer containing arguments to be passed to the thread, while `nargs` is a count of the number of arguments. Threads are created in a suspended state, and do not run until `threadResume` is called. Threads can be terminated by the `threadTerminate` call:

```
int threadTerminate(int id);
```

An application can check for the existence of a thread with the `threadExists` call:

```
bool threadExists(int id);
```

Any thread can find out what its own identifier is by calling `threadID`:

```
int threadID();
```

Threads can be indefinitely suspended until resumed at a later time by the following calls:

```
int threadSuspend(int id);
int threadResume(int id);
```

A thread can sleep for a variable time period by using the `threadSleep` call:

```
void threadSleep(struct time *);
```

The scheduling attributes for a thread can be set and retrieved by the following two functions. The definition of the `sched_info` structure is dependant upon the scheduler being used.

```
int threadSetSched(int id, struct sched_info *);
int threadGetSched(int id, struct sched_info *);
```

Finally, the following three calls are used to set, reset and retrieve the thread's effective domain. The use of these functions is described in Section 3.12.1.

```
void threadSetEdomain();
void threadResetEdomain();
int threadGetEdomain();
```

APPENDIX B

High-Level Service Interfaces

This appendix contains interface details for some of the high level services discussed in chapter 4.

B.1 IDE Interface

The interface presented by the IDE service is very simple. There are only three procedures making up the interface:

```
int ideRead(int partitionID, unsigned offset,
                void *data, unsigned sectors);
int ideRead(int partitionID, unsigned offset,
                void *data, unsigned sectors);
int ideSize(unsigned partitionID);
```

The first two procedures deal with reading and writing data. They each take a partition identifier, offset (in 512 byte sectors) into the partition, a pointer to the data to be transferred, and the

number of sectors to be transferred. The third function returns the number of sections in a specified partition. Clients of the IDE service obtain partition identifiers from the name service, where they are associated with names such as "/device/ide0/bsd0"[1] by the IDE service when it is initialised.

B.2 Bcache Interface

The bcache service provides buffering of disk blocks for file services. New blocks are read from and written to the disk with the `bcacheRead` and `bcacheWrite` procedures. Each of these has identical paramters. The prototype of `bcacheRead` is shown below:

```
int bcacheRead(int fid, unsigned physOffset,
               unsigned size, bcacheHint hint,
               unsigned bufOffset, void *buffer,
               unsigned bufSize);
```

Each of the read and write procedures have seven arguments – a unique file identifier, the physical offset of the block, the size of the block, a hint (used to control how quickly blocks are recycled from the cache), and three arguments describing a memory buffer into (or from) which the cache is to copy the data. The last two of these point to the buffer and its size (i.e. how many bytes are to be copied) while the first determines an offset into the physical block. This allows filesystems to control how much of the block they currently wish to access. On successful completion, these procedures return the number of bytes copied, or a negative value if a failure occurs. Note that there is no device parameter to this function. This is because the current version of the Kea system only contains a single IDE disk. It is planned that future versions will have a parameter identifying the device.

The `hint` parameter is used to control how quickly the block is removed from the cache. Currently four values are supported:

1. This can be interpreted as the first partition with a BSD filesystem on the first IDE device.

- `HINT_KEEP` – probable block will be used again.
- `HINT_MAYBE` – possible block will be used again.
- `HINT_DISCARD` – not expected to be used again.
- `HINT_DELETE` – block is invalid: don't need to write, even if dirty.

Three other procedures are provided that allow clients to force the writing of dirty blocks to disk. These procedures allow cache flushing on a either a file, block or global (all blocks in the cache) basis:

```
void bcacheSyncFile(int fid);
void bcacheSyncBlock(int cacheID);
void bcacheSyncAll();
```

B.3 File Interface

All of the Kea filesystems conform to a single interface, that is very similar to the POSIX file procedures:

```
int fileOpen(char *name, int oflags);
int fileClose(int handle);
int fileRead(int handle, void *buffer, unsigned bytes);
int fileWrite(int handle, void *buffer, unsigned bytes);
int fileCreate(int handle, char *name,
               struct fileStatus *buf);
int fileRemove(int handle, const char *name);
int fileStat(int handle, struct fileStatus *buf);
int fileWstat(int handle, const struct fileStatus *buf);
int fileSeek(int handle, int offset, seekType whence);
```

The `fileOpen` procedure is used to open a file for writing, returning a positive file handle on success. Each of the other procedures use this handle for successive operations. The `fileStatus` structure is analogous to the Unix "stat" structure, and is used to store information about the file, such as its size, owner and last access time, and is read and written by the

`fileStat` and `fileWstat` calls respectively. The `fileCreate` and `fileRemove` procedures require a handle to the directory in which the file is to be created or removed, together with the name of the file (which must not contain any pathname separators, such as ".." or "/").

B.4 Mount Interface

The mount service provides a mapping between name prefixes and service identifiers. A new mapping is added with the `mount` call, and removed with the dismount call:

```
int mount(char *path, int servID);
int dismount(char *path);
```

Given a file name, the `mountQuery` procedure searches for the longest path which matches the file prefix, and returns the service identifier of the backing service:

```
int mountQuery(const char *name, unsigned *plen);
```

The length of the prefix matched is returned using the `plen` parameter.

APPENDIX C

Scheduler Interface

This appendix contains details of the callback scheduler interface provided by Kea. The unique properties of this interface are described in Section 3.13. This interface provides a full separation of scheduler policy (when and which threads are run) from the operating system mechanisms required to support those operations (clock ticks, context switching, etc.) Following is a list of function prototypes in the scheduler interface ("schedInt") and their purposes. There is one special global variable also associated with the scheduler, `doPreempt`. This variable is of type bool (boolean) and can be set to true by any of the scheduler functions. When set, the system guarantees to preempt the current thread, and call `schedIntSelect`, immediately upon return.

```
void schedIntInit()
```

This function is called once at system start-up.

```
void schedIntAdd(struct thread *t)
```

This function is called when a new thread is created

```
void schedIntRemove(struct thread *t)
```

This function is called when a thread is destroyed.

```
void schedIntSetInfo(struct thread *t,
                     struct sched_info *s)
```

Called when thread scheduling properties are changed.

```
void schedIntRunnable(struct thread *t)
```

Called when a thread is made runnable.

```
void schedIntNonRunnable(struct thread *t)
```

Called when a thread is made non runnable (i.e. the thread is going to sleep, or has been suspended).

```
struct thread *schedIntSelect()
```

Called when a new thread should be selected to be run. Returns the thread selected, or NULL if no thread is currently runnable.

```
void schedIntTick()
```

Called on every clock tick

```
void schedIntEnqueue(struct thread *t, mutex *m)
```

Called when a thread is going to sleep on a mutex/semaphore.

```
void schedIntDequeue(mutex *m)
```

Called when a thread acquires a mutex/semaphore. These last two functions enable a scheduler to implement custom handling of semaphore queues.