List of Figures

Chapter 1 Introduction

1.1 Traditional Computer Systems and their Applications

When the first computer systems were deployed some forty years ago, the applications they supported were limited to the domains of science and engineering. This is not surprising because computer systems and their software are designed and built by scientists and engineers.

In these early days, it was the cost and relatively limited computational power of the technology that restricted its availability to the laboratory environment. But, as technology evolved from the valve to the transistor and finally the microprocessor, the cost of systems dropped while the available power rose dramatically. With the convergence of these two factors it became economically feasible to deploy computer systems in a commercial environment, initially with mainframes and minicomputers and latterly with the personal computer. The guiding force behind this commercial success was the realisation that computers were not only suited to number crunching activities but also every other domain that requires some form of activity that can be expressed algorithmically. Influential domains include; education, the office, digital media processing, medicine and finance. Notably, many domains are explicitly enabled through the integration of peripheral components alongside the ability to process, e.g. graphical terminals enable desktop publishing, CAD / CAM and video game applications, while magnetic and optical disk based storage devices enable database and digital multimedia server applications.

Unfortunately, by the time these new application domains were beginning to emerge, considerable research and development effort had been expended in the development of the complex system software and support tools that the original scientific and engineering applications had required in order to operate. The general assumption was that system software had become stable and correct for its intended purpose and that emerging application domains should be supported within the same context.

1.2 The Lack of Support for Emerging Application Domains

Most of the currently important commercial operating systems (e.g. Microsoft's Windows NT/9x/3.x, Apple's MacOS, Sun's SunOS etc.) are based around a programming model devised some twenty five years ago in the original Unix operating system. This programming model subsumes notions of unique protected address spaces, concurrent execution over one or more multi-programmed processors and a fixed Application Programming Interface (API) exported to application programs from a privileged operating system kernel. Although applicable in many situations, emerging application domains are highlighting the limitations of this model. For instance, the Unix programming model implicitly embeds a number of *policies* regarding the way in which the resources it manages are to be utilised. In general, the overriding principle is the fair sharing of resources between applications. Examples of this strategy include round robin scheduling of the processor, order-of-reception processing of network packets and the working set approach to various caching algorithms. While these schemes are appropriate for batch and throughput oriented scientific and engineering applications, they are not necessarily correct in other domains.

In addition, having been designed for scientific and engineering applications, the original versions of Unix only attempted to manipulate a limited set of media types, namely numeric and textual data. When technology enabled the manipulation of more complex media types (e.g. graphical data), applications and libraries (e.g. X11 [Poundian,89]) were quickly devised to make use of these new forms of information. Unix itself was extended to enable the integration of the new media types, each of which required its own peculiar form of storage and processing services. Typically, each new form of media type enabled the emergence of a new application domain to make use of it. However, continually extending an operating system to support new media types has proven to be a time consuming, error prone and on-going task thus reducing the system's effectiveness in supporting the new application domains that make use of them.

The following sub sections demonstrate how particular domains that have emerged after the inception of the traditional operating system have suffered particularly badly with the aforementioned fixed computational model and policies.

1.2.1 Multimedia Applications

The types of multimedia applications that cause particular concern are those that manipulate *continuous media* [Anderson,90]. This type of data set places unusual demands on the processing facilities of a computer system. For instance, the traditional notion of a computational result is extended to include the timeliness of the results as well as their correctness. In order for timing constraints to be met, the system must process data in *unfair* patterns [Nieh,93], e.g. network packets and flows of control must be processed in terms of deadlines, rather than using the fair share strategies of throughput oriented environments. In addition, multimedia data is itself a new type of media consisting of many sub-types including bit-map, vector and 3D graphics and raw and formatted audio. Each of these sub types require special processing including caching and storage algorithms in order to be handled efficiently.

1.2.2 Mobile Applications

Due to the continued miniaturisation of computer terminals, e.g. laptops, notebooks, palmtops and personal data assistants, and the invention of wireless networking technologies, e.g. GSM and WaveLAN, there is now a demand for users to be able to roam freely with portable machines while maintaining network connectivity whenever possible. Conventional networking protocols, e.g. the TCP/IP suite [Postel,81], are relied upon by a myriad of applications, e.g. Telnet [Postel,83], FTP [Postel,85] etc., but assume fixed hosts and assign network addresses based on these static locations. New protocols that enable host machines to roam, e.g. IPv6 [Deering,98], are being integrated with existing protocols but this clearly complicates the maintenance of the networking stack. In addition, changes in connectivity (e.g. disconnection from a fixed network followed by connection to a wireless network) must be handled by the operating system thus requiring special support for dynamic adaptation [Blair,97b] between the available protocols and network mediums. This support is currently not found in traditional operating systems.

1.2.3 Transactional Applications

Typical database systems take the storage of media specific data types to the limits of computational capacity. Enterprise level corporations' databases have capacities in the terabyte region and are simultaneously accessed by hundreds of thousands of users. This media must be stored as efficiently as possible and many algorithms have emerged from related research (e.g. [Date,95]) that make the task of searching for data within such enormous repositories tractable. Unfortunately, due to the general purpose nature of operating systems such as Unix, domain specific storage strategies must be sacrificed in favour of generality in order to cater evenly for all application domains.

1.2.4 Embedded Applications

The heavy computational resource requirements of Unix based operating systems are at odds with embedded systems that only have meagre resources available to them. Although embedded applications are typically modest in their purpose, they demand maximal efficiency from the available resources through minimal operating system resource overhead. New types of embedded system, e.g. palmtops and video game consoles, exacerbate these demands by attempting to support heavyweight applications on machines that are typically many times less powerful than those they were originally developed for.

1.3 Thesis Aims

The aim of this thesis is to develop an operating system that can meet the demands of both current and future application domains without the drawbacks currently experienced by traditional operating systems. This is to be achieved through a thorough evaluation of existing operating system architectures against a set of generic requirements formulated to simulate the demands of emerging application domains.

The thesis will attempt to show that current operating system architectures either cannot meet these requirements or have not yet been explored thoroughly enough to determine their suitability for doing so. In particular, it will be contended that operating systems must be adaptable as no one system configuration can simultaneously hope to meet the requirements of all future application domains without compromising their maintainability. In addition, it will also be conjectured that portability in the operating system is not necessarily a desirable requirement and can be made orthogonal to maintainability through the re-organisation of services into components.

The operating system described in this thesis is intended to facilitate the arbitrary configuration of its services in order to support unforeseen application domains. It is *not* the aim of this thesis to criticise or mandate the redesign of existing operating system services. Instead, these and future services should simply be developed independently of the core operating system and integrated when necessary to support target application domains.

Finally, this thesis will attempt to show that configuring *fundamental* system services, i.e. memory model, concurrency model and inter-process communication mechanisms, has yet to be considered by the research community and offers scope for improving operating system support for custom purpose and emerging application domains.

1.4 Thesis Structure

The structure of this thesis is as follows: Chapter 2 begins by analysing prominent operating system architectures and highlighting their strengths and weaknesses. Next, a comprehensive set of requirements are formulated by which to compare these architectures. Finally, through a process of evaluation, one architecture is selected for further examination based on its ability to support emerging application domains.

Chapter 3 presents an assessment of the current state of the art in the field of operating systems based on the architecture selected in chapter 2. Projecting from the state of the art, chapter 4 details the design of an operating system that incorporates the requirements of chapter 2. Chapter 5 describes two implementations of this design and expresses the benefits of exploiting hardware dependent functionality from the application level. In addition, this chapter details how operating system services can be implemented as discrete components. Chapter 6 then describes how fundamental system services can be factored into components like any other service. This chapter also details how the potential problem of having applications depend on platform dependent functionality can be overcome through the provision of standard execution environments. Chapter 7 evaluates the design through both a quantitative performance analysis of one implementation's salient features and a qualitative assessment of how the system can be configured to provide support for varied application domains. Finally, chapter 7 draws some general conclusions and highlights a number of areas in which further work is required.

Chapter 2 Operating System Requirements for Supporting Emerging Application Domains

2.1 Introduction

This chapter begins with an analysis of the three most prominent general purpose operating system architectures. Next, the demands of emerging application domains are established and used to formulate a comprehensive set of requirements by which to evaluate the suitability of the aforementioned operating system architectures in terms of their support for new application domains.

2.2 Operating System Architectures

2.2.1 General Purpose Versus Custom Operating Systems

The purpose of this chapter is to establish the most suitable general purpose operating system architecture for supporting emerging application domains. However, it should be noted that custom built operating systems are also capable of supporting emerging application domains because they can be designed from scratch with a particular domain in mind. Unfortunately, this domain is typically the only domain for which the custom system is suitable for; if a new domain is to be targeted then a new custom operating system must be constructed. Several custom operating systems have achieved commercial success including Chorus [Rozier,92], in the field of telecommunications, QNX [Hildebrand,92], in the field of real time and embedded systems and BeOS [Be,00], in the field of multimedia applications. In addition, a myriad of research based custom operating systems (including Serpent [Waddington,99], Roadrunner [Miller,96], Pegasus / Nemesis [Mullender,94], SUMO [Coulson,93], x-kernel [Hutchinson,91], Scout [Montz,94] and RT-Mach [Tokuda,90]) have demonstrated superior support for particular application domains compared to conventional operating systems such as Unix.

Custom operating systems can be constructed in any of the three architectural forms described below, but they are of limited research interest compared to general purpose operating systems because the challenges of supporting a single application domain after it has emerged are trivial compared to optimally supporting multiple and diverse domains that have emerged after the operating system itself.

2.2.2 The Monolithic Architecture

The monolithic operating system architecture simply amalgamates all of a systems' services and abstractions into a single, inseparable code image. Although logically structured in terms of source files, monolithic systems lose all notion of structure when they are compiled and linked[1]. Figure 2.1 illustrates the logical separation of services within the formless structure of the monolithic kernel.

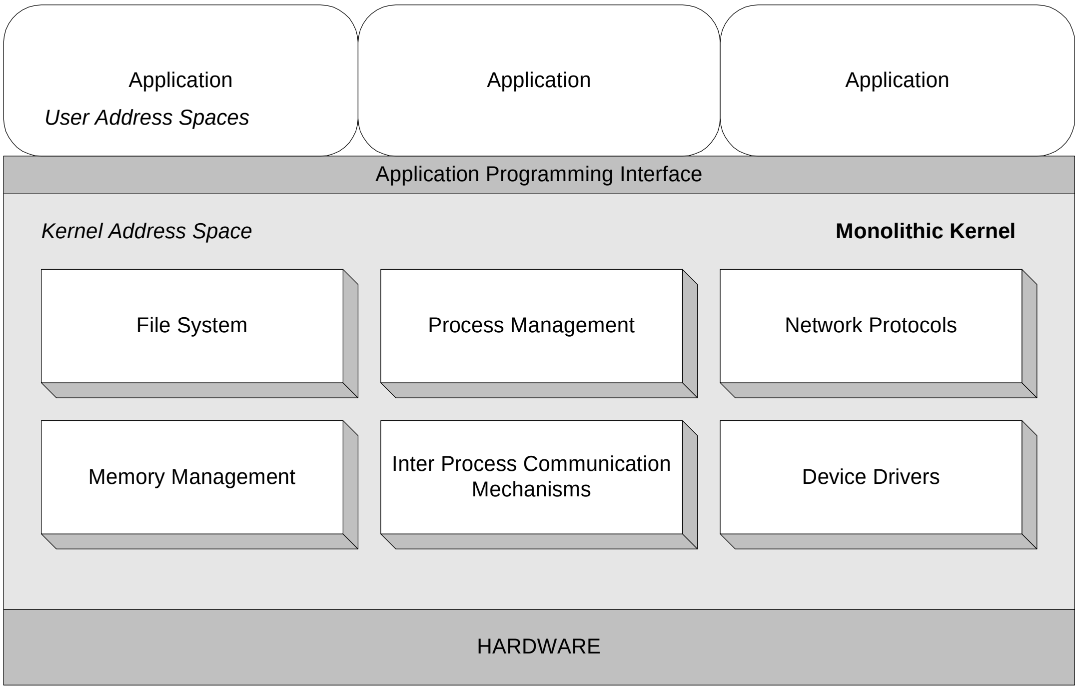

Figure 2.1: The Monolithic Operating System Architecture

A major advantage of the monolithic system is the performance that they exhibit. This is a direct consequence of their architecture, i.e. a discrete code image loaded into a single address space with logical subsystems communicating by direct procedure calls and able to access logically distinct data structures by virtue of collocation. Conversely, such close coupling combined with a lack of internal interface definition leads to a system that is extremely difficult to maintain and extend. Often, changes or bug fixes in one logical subsystem will have unforeseen effects on some other supposedly unrelated subsystem.

---

[1] Even monolithic systems built from object oriented languages lose their object oriented structure after compile time because they must employ languages (e.g. C++) that lose all notions of hierarchy, inheritance, membership etc. in deference to performance when they are compiled.

2.2.3 The Microkernel Architecture

Microkernel based operating systems define a primitive kernel that exports a minimal set of privileged services and abstractions to unprivileged servers operating at the user level and collectively defining a complete suite of operating system functionality as illustrated in figure 2.2. The servers are said to implement an operating system *personality*[2] [Albinson,91] and make direct requests of the underlying microkernel programming interface for privileged services such as register level device access and the creation of execution contexts and control flows. These base services allow the servers to implement more traditional operating system abstractions such as processes and files. These are then exported to conventional applications through the servers own programming interfaces. Figure 2.2 illustrates this split level structure and programming architecture.

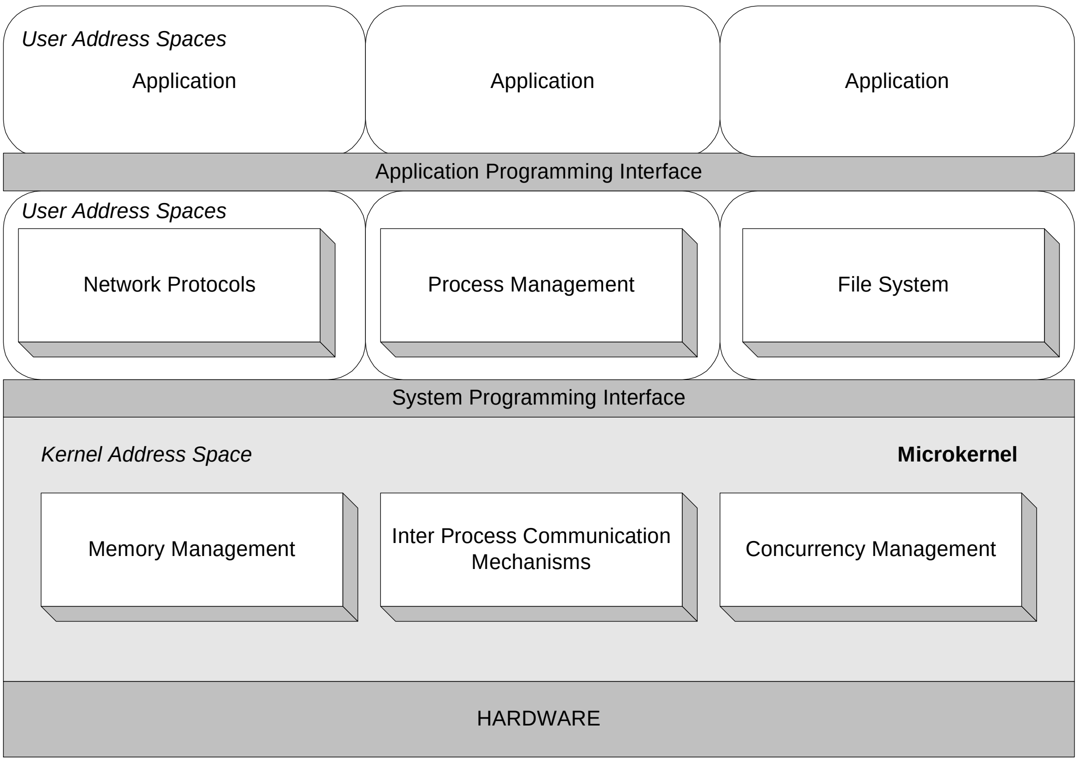

Figure 2.2: The Microkernel Operating System Architecture

Although a large amount of research has been expended on the design of the microkernel itself (focussing mainly on the nature of the services and abstractions it should implement), much less work has been carried out on the design of the personality servers. The majority of code within a microkernel based operating system

[2] A personality may be a new operating system or an emulation of an existing one redesigned for the microkernel architecture.

exists within these servers and simply creating them as monolithic entities leads to almost as many maintenance problems as encountered with a purely monolithic operating system.

Note that a microkernel based operating system should not be physically smaller (in terms of lines of code) than a monolithic operating system; the difference lies in the fact that the privileged kernel portion is far smaller because it does not embed policies over how its base abstractions are to be used. For example, a microkernel does not define a file system itself because file systems can be implemented in a number of different ways by personality servers. Making the privileged portion of an operating system smaller greatly increases its reliability and portability. This is because it takes far longer to develop kernel level services than user level services. For instance, less tool support is available (e.g. for symbolic debugging) at the kernel level.

2.2.4 The Extensible Architecture

Extensible operating systems, in their purest form, define a minimal executive with the sole purpose of integrating traditional operating system services and abstractions encapsulated as both logically and physically discrete entities. This integration process may take place at the behest of either applications or the system itself. The basic organisation is shown in figure 2.3.

Extensible systems depend on well defined frameworks for the construction of their externalised system services and also on open interface definitions in order to accommodate these services once they have been constructed. Both of these features are essential if the development of services by third parties (including application developers) is to be encouraged. Without third party services, extensible systems are only as useful as the services that the system vendor creates, i.e. they may not be any more applicable to emerging application domains than those provided in traditional systems.

Generally, extensible systems allow the integration of external services within the supervisor address space. This raises issues of protection (against maliciously designed services), authorisation (over who may introduce services) and contention (when services compete for the same resources). All these issues must be addressed for the system to be classed as extensible rather than just a system that supports ad-hoc service integration, as in the case of Linux [Beck,96] and SunOS [Sun,91] (with loadable kernel modules) and Chorus [Rozier,92] (with supervisor actors).

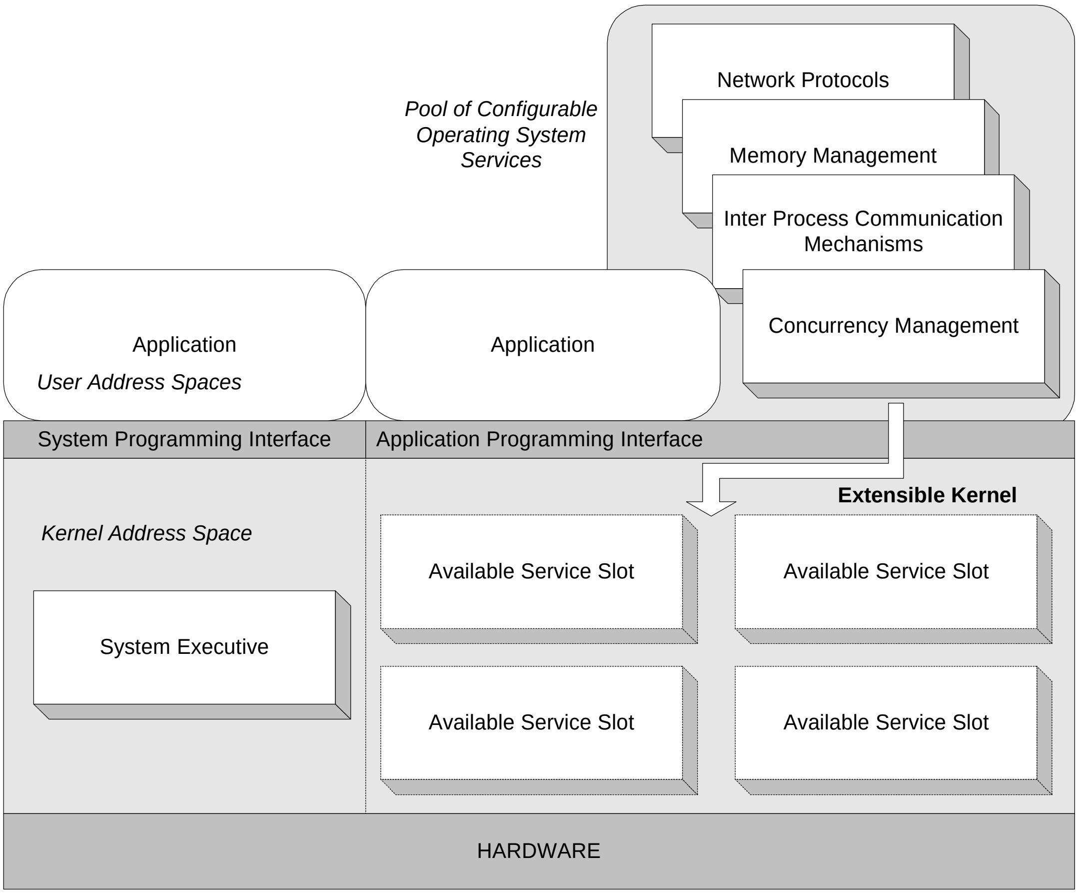

Figure 2.3: The Extensible Operating System Architecture

Requiring an application developer to specify the system services their application requires may appear to place a heavy burden upon the developer. However, it can be contended that a developer wishing to specialise the default services provided by an operating system does so because they are aware of the shortcomings of those services for their particular application domain. Having established these deficiencies, it should be possible to construct or select (from a pool of appropriate replacements) appropriate services.

2.3 Operating System Requirements

An operating system manages and exposes resources to applications via the services defined in their programming interface. Programmers must employ these services in order to carry out hardware related tasks within their applications, e.g. opening disk based files, writing to the display adapter and acquiring physical memory.

Supporting emerging application domains requires that system services be readily adapted to the demands of new domains. For instance, virtual memory subsystems should expose their cache eviction policy as a discrete entity thus allowing the policy to be changed according to an applications demands. In addition to these *adaptable services' requirements*, an operating system must also exhibit a number of *non-functional* characteristics that indirectly enhance their ability to support emerging application domains.

2.3.1 Adaptable Service Requirements

2.3.1.1 Timely Service Provision

An operating system must provide enhanced services for emerging application domains in a timely manner, i.e. almost immediately after they emerge. Without this requirement, an operating system cannot provide the same first class status to emerging domains as is afforded to existing application domains.

2.3.1.2 On-the-Fly Service Reconfiguration

The ability to integrate upgraded or modified versions of services at runtime satisfies the above requirement for timely service provision and, additionally, allows applications to dynamically adapt their behaviour in response to changes in their environment. For instance, in the distributed mobile multimedia application domain [Blair,97a], the varying levels of bandwidth experienced may mandate the dynamic reconfiguration of an application, e.g. causing the instantiation of bandwidth reducing media filters [Yeadon,96] during periods of attenuated connectivity.

2.3.1.3 Encapsulation of Services

Encapsulating services as discrete entities with well-defined interfaces allows operating systems to be composed into configurations suitable for particular application domains simply by connecting the services in appropriate patterns.

Components [Szyperski,98], are a rapidly emerging technique for the encapsulation of both system [Blair,98] and application [Pryce,98] activities. Like objects, components encapsulate state and export the ability to manipulate that state via well defined interfaces. Components also exhibit a number of additional features that increase their usefulness, including;

- the notion of *required* interfaces that mandate the presence and binding of other components before a component can become operational,

- deployment in binary form so that they are available for dynamic linking at runtime and are independent of their source language, and
- adherence to a design framework, cf. the inheritance of the `IUnknown` virtual interface in Microsoft's COM [COM,95] compared to the freeform design of objects, e.g. in Sun's Java [Gosling,96] programming language. A framework provides guidance for developers and facilitates the control of components through standard functionality (manipulated by applications and the system runtime) that the component must implement in order to comply to the framework.

In addition, structuring a system out of reusable components enables the continued visualisation of this structure after compile time (c.f. the loss of structure in a conventional operating system built out of source files after they have been compiled and linked into a single object image). Visualisation of structure allows the expression of service interdependencies and indicates whether services can be safely replaced at build or runtime. In order to restructure the services inside a conventional operating system, a designer must delve into the source code and deal with the undocumented inter-relationships that exist at that level.

2.3.1.4 Extend Adaptability to Fundamental Services

Current general purpose operating systems implement three fundamental services, i.e. virtual memory, concurrency and inter process communication (IPC), in order to support existing application domains. Emerging domains have no choice but to use them despite losing out on optimisations they could benefit from. For instance, implementing full process semantics in a system to be deployed for supporting embedded applications or in a system that is supporting a sole, resource heavy, application (e.g. a Video on Demand server) will detrimentally affect the performance of those applications because there is no need to pay for the protection and complexity that such abstractions entail. Therefore, it is contended that even fundamental services should be open for adaptation.

2.3.1.5 Evaluation

In a monolithic system, all service modifications are carried out by the system vendor according to their agenda and timing constraints. Often, it can take months or years for new release containing support for emerging application domains to become available. The overall strategy for supporting emerging application domains involves continuously introducing more and more services (for each domain to be supported) with each release. Such a strategy not only reduces the maintainability of the system but necessarily reduces overall efficiency because, for much of the time, many services are not going to be used thus leading to code bloat and unnecessary consumption of resources.

Microkernels already exhibit a degree of service encapsulation thus leading to a partial satisfaction of this requirement. Their user level servers can be regarded as very crude components that can, in some cases [Abrossimov,96b], be dynamically started, stopped and replaced in order to furnish applications with some modified level of functionality. For instance, the Mach and Chorus microkernels both define an externalised paging service that can implement diverse page replacement strategies and distributed virtual memory. However, such a server must have a well defined and public interface for it to be re-implemented. Additionally, there are no rules constraining when they can be replaced, who can replace them or for limiting the potentially dangerous effects that a malicious implementation may exhibit to other applications. Finally, the whole microkernel philosophy is centred around the fixing of the three fundamental abstractions described above. Not only is it not possible to alter these services, it is also against the microkernel edict.

Generally, microkernel operating systems are split into a small number of user level servers that rarely communicate. This is because the overhead of communication between separate protection domains (including between the servers and the microkernel itself) has been shown to be too high for frequent use. Consequently, adding services in new servers is an inherently inefficient process. This performance penalty means that microkernel based systems rely on a similar approach as used by monolithic systems for introducing new services, i.e. the vendor must continually modify the source code of their monolithic servers (incurring release delays and maintainability problems).

Extensible operating systems ideally satisfy the adaptable service requirements described above. When an application domain emerges, the application developer is at liberty to either create or acquire (possibly from a third party) new services and integrate them within the system thus removing the dependency on the system vendor. In addition, extensible systems lend themselves readily to the fine grain encapsulation of services because they can be safely collocated within the kernel address space thus incurring little communication overhead. Finally, the ability to componentise services means that only those that are required by a given application need ever be included in the system thus avoiding code bloat and unnecessary resource consumption. It is as easy to apply these principles to fundamental services as it is to non-fundamental services.

Table 2.1 summarises the suitability of the three architectures against the adaptable service requirements just outlined.

	Monolithic	Microkernel	Extensible
Timely Service Provision	✕	partial	✓
On-the-Fly Service Reconfiguration	✕	partial	✓
Encapsulation of Services	✕	partial	✓
Adaptable Fundamental Services	✓	✕	✓

Table 2.1: Summary of Operating System Architecture Against Adaptable Services Requirements

2.3.2 Non Functional Service Requirements

2.3.2.1 Pragmatic Service Portability

Traditionally, operating systems have been designed to minimise their machine level dependencies and to isolate them as far as is possible in well established locations (usually in a particular branch of the source tree). The reason for doing this is to enhance the portability of the system between different machine level architectures based on the software engineering argument that machine dependent code is the most difficult to alter and maintain [Gien,90], [Gien,91].

Portability concerns have also pervaded the services that operating systems offer, i.e. they are based on reference hardware architectures. This means that the operating system expects a minimum level of hardware support from any underlying platform that it is to be ported to. For instance, many general purpose operating systems expect the presence of hardware support for virtual memory in order to implement a secure process model. The problem with affording this much importance to portability is that traits and optimisations offered by a particular platform are overlooked and ignored [Liedtke,95] in favour of maintaining the hardware reference model. This means that many optimisations that emerging application domains could benefit from are never actually exploited. It is contended that services themselves should contain machine specific dependencies in addition to the system core.

2.3.2.2 Recording Service Evolution

There is a risk that by allowing the rapid evolution of services to support new domains, a situation will arise whereby operating systems consist of a morass of similar but incompatible services all being used simultaneously by different applications. This situation would increase the footprint and resource utilisation characteristics of the system and create inconsistencies in the programming interface making it more difficult to create applications. To combat this problem, a methodology should be employed that clearly identifies the nature and capability of each service including any enhancements that are made to them and their status regarding backwards compatibility.

2.3.2.3 Evaluation

Producing services that are machine dependent should not be a factor related to the architecture of the host operating system because any piece of code with kernel privilege can autonomously make dependencies upon the raw hardware. However, this assumes that the operating systems service will operate with kernel privilege. This is true for monolithic and extensible systems (though their services can also be run at user level), but not the case for microkernel based systems where operating system services exist as user space servers and can only make use of machine dependent services indirectly through the microkernel programming interface. This interface is fixed and portable between different machine architectures making it impossible for personality services to exploit facilities offered by one hardware platform but not by another.

Versioning is an effective way of tracking the evolution of all types of software and is particularly suitable for code that is maintained discretely (c.f. objects and components). Both extensible and microkernel based systems can make use of versioning after compile time in order to configure services because these services continue to retain versioning information due to their inherently discrete nature. This information can be used to configure systems using more friendly and flexible configuration tools (e.g. graphical tools) outside of the compilation tool chain. Conversely, monolithic systems only retain versioning information within their source file structure and this information is consequently lost at compile time. This means that versioning information can only be utilised within the context of inconvenient conditional compilations. Table 2.2 summarises the suitability of the three architectures against the non functional service requirements just outlined.

	Monolithic	Microkernel	Extensible
Pragmatic Service Portability	✓	✕	✓
Recording of Service Evolution	✕	✓	✓

Table 2.2: Summary of Operating System Architecture Against Non Functional Service Requirements

2.4 Summary

Having previously established that current commercial operating systems are not supporting emerging application domains in an effective manner, this chapter began by analysing the features of three prominent operating system architectures used in both commercial and research based systems. Based on the assumption that an operating system is simply the sum of all the services that it offers, each architecture was then evaluated against a number of service requirements appropriate for any system wishing to support emerging application domains. From this evaluation, the extensible architecture clearly emerged as the one most suited to this task.

Chapter 3 State of the Art in Extensible Operating Systems

3.1 Introduction

This chapter assesses the state of the art in extensible operating systems. The systems reviewed have been categorised into a *continuum* depending on two key characteristics:

- *When* does the system allow service integration take place ?
- *Who* decides which services are to be integrated ?

The chapter begins by highlighting the idiosyncrasies of each classification within the continuum and then evaluates each by reviewing related projects from the literature.

3.1.1 The Continuum of Extensibility

Extensible operating systems can be extended at three points in their lifecycle:

- *Build time extensible systems* can only accommodate new supervisor space system services while they are being built. Once the system is booted, no more changes can be made unless they are stopped, rebuilt and rebooted.
- *Link time extensible systems* (also known as library extensible operating systems) define unusually low level (and application neutral) functionality in their kernel APIs. This functionality simply wraps the underlying hardware in a thin layer of software based protection making it safe to expose to the user level. It is then the role of user level libraries (linked to by applications) to implement particular operating system personalities by utilising the system API. By selecting an appropriate library, applications can extend the system as their requirements demand. However, once a library personality has been linked to by an application, the system cannot be extended further from that application's point of view.

- *Runtime extensible systems* can accommodate new system services while they execute. Unlike library extensible systems, these new services can be downloaded directly into the supervisor address space. From here they can access kernel interfaces and the privileged subset of the machine's hardware. Naturally, protection is of vital importance in such systems; arbitrary application defined code should not be allowed to corrupt the systems integrity. Typically, runtime extensible services can be reconfigured (i.e. deleted or replaced) on an on-going basis.

Extensible operating systems can be extended by either applications or the system itself:

- *Application defined system extension* involves the co-operation of application developers. This manifests itself in two ways; i) the application developer must decide which services they wish to introduce (based on domain specific knowledge), and ii) in the case of runtime extensible systems, the application writer must define the timing of the introduction and later reconfiguration of each service.

- *System defined extension* involves having the system attempt to calculate (from general purpose heuristics) and respond to the resource usage patterns exhibited by their applications. Once such patterns have been isolated, the system itself introduces new services to offer superior performance for the target application.

The remainder of this chapter presents work from the literature classified in terms of the continuum of extensibility. At present there is only one example of a system that is self extending and this takes place at runtime. However, system defined extension at build and link time could still be realised. For instance, application control and data flows could be analysed off-line using some tool and the system could be configured with appropriate services based on the results.

3.2 System Defined Operating System Extension

3.2.1 Runtime Extensible Operating Systems

3.2.1.1 Synthetix

The Synthetix group at the Oregon Graduate Institute have performed extensive work in the field of self extending operating systems. The central tenet of their Synthetix operating system [Pu,94] is the ability to self monitor events within the system, e.g. how applications are using resources and then to *incrementally specialise* the system code (system calls) that interacts with those resources according to the usage patterns that have been established. Working closely with the COMPOSE group at INRIA / University of Rennes in France, the Synthetix group make use of *partial evaluation* [Jones,93] (now using the TEMPO tool [Consel,98b]) to provide the specialised code required to optimise application performance. With this technique, system developers identify static and quasi-invariant conditions within system control paths and according to the runtime status of these conditions, differing control paths can be selected (the process of specialisation) which exploit the optimisations that are possible when the conditions can be assumed to be invariant (process of partial evaluation). For example, the Unix *write()* system call can be optimised when a given file is not being shared; all the concurrency control code can be ignored. If, at a later stage, the file becomes write shared then the system must note this change of condition and replace the specialised control path with the original version that implements disk block locking strategies. Whilst the quasi invariant holds, significant performance increases can be achieved. However, great care must be taken when altering the control path to be followed in case a system call is already in progress; the Synthetix team define a *replugging* protocol [Pu,95] to handle the complications that can occur in this case.

3.3 Application Defined Operating System Extension

3.3.1 Build Time Extensible Operating Systems

3.3.1.1 Choices / μChoices

Choices [Campbell,93] from the University of Illinois at Urbana-Champaign exhibits both build and run time forms of extensibility. In this section, Choice's mechanisms for build time extensibility are discussed, while section 3.3.3.12 covers its mechanisms for dynamic extensibility.

Choices is an object oriented, monolithic operating system which endeavours to enable specialisation, code reuse and rapid prototyping via its object model. In terms of build time extensibility, its most important feature is the *abstract framework* which is defined as part of the system design. This framework details a set of abstract classes which completely describe (in a hierarchic fashion) the essential functionality of a Choices operating system. The framework is decomposed into top level component classes and subframeworks which add their own class components and recursively specialise those in the higher layers. Top level classes include *process*, *domain* (address space) and *memory object* (regions backing the domain).

A considered approach to the design of the abstract framework was taken which involved the use of entity relationship, data flow and control flow diagrams. This has resulted in the modelling of unusual design features, such as control flow, as well as more obvious class hierarchy relationships within the framework. By statically implementing the classes in the Choices framework, developers can produce a Choices operating system tailored to particular hardware platform characteristics and have complete freedom over the semantics of the methods implementing functionality that may influence the performance of target applications, e.g. choice of page replacement algorithm in the virtual memory subframework.

μChoices [Tan,95], [Campbell,95] is the microkernel oriented redesign of the original Choices operating system. The key goal was to realise the portability benefits offered by the microkernel approach. μChoices extends the microkernel paradigm by defining an even lower level *nano-kernel* on which to support the more traditional microkernel. The nano-kernel simply offers wrapped access to the raw machine hardware. The microkernel uses these services to define abstractions such as virtual memory and concurrency. Finally, user level servers implement traditional operating system services such as processes and files. The original Choices abstract framework is maintained but is split between the three architectural layers in order to remove hardware dependencies from the user level servers which implement the bulk of a complete operating system.

3.3.1.2 Chorus ClassiX

Chorus [Rozier,92] was originally a distributed systems research project at INRIA in France during the early 1980's. From this work emerged a microkernel which was subsequently spun off into a commercial product marketed by Chorus Systems. At this time, the microkernel was in its fourth revision and had matured considerably since its inception. Like Choices, Chorus demonstrates both build time (covered here) and run time (covered in section 3.3.3.10) extensibility.

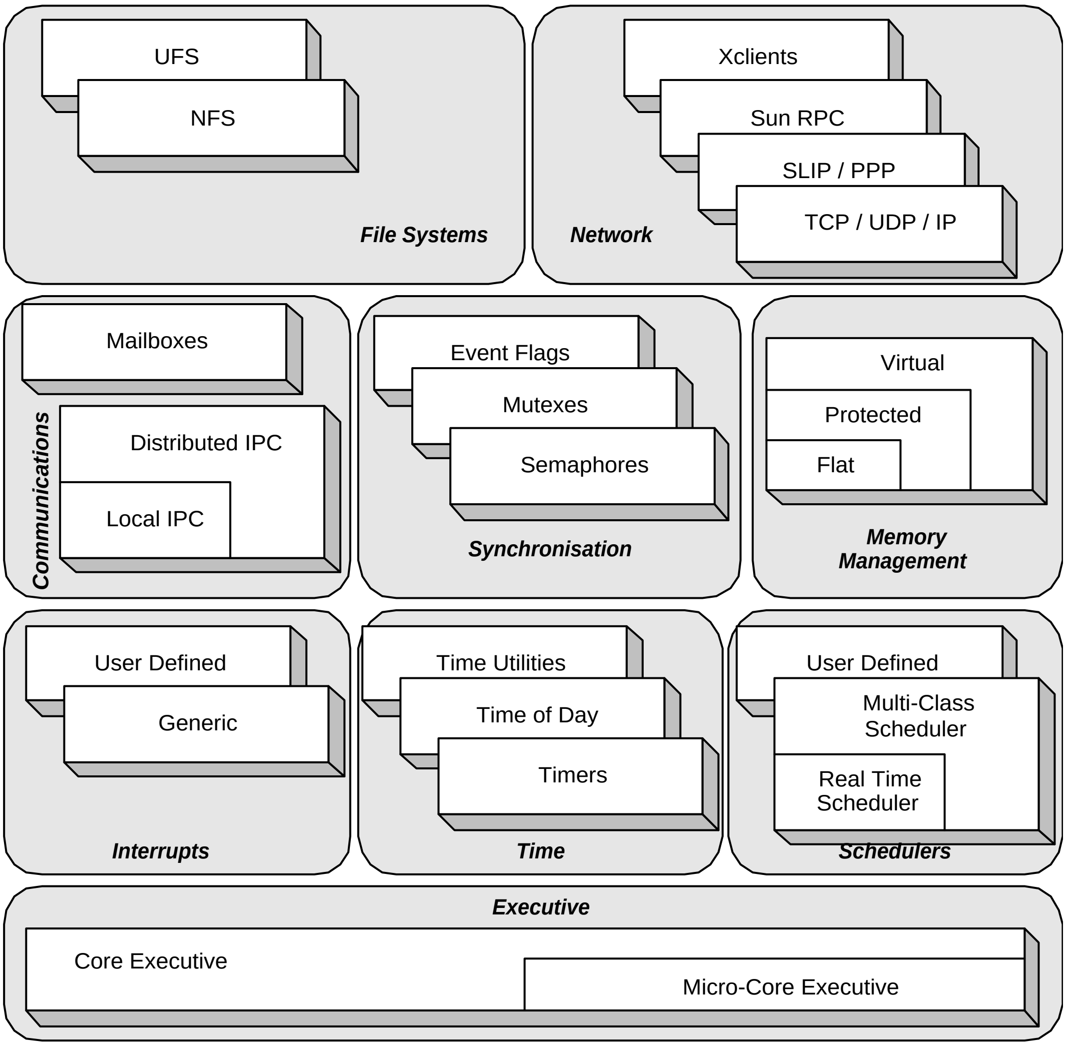

Figure 3.1: Selectable Components of the Chorus ClassiX Microkernel

The latest revision of the Chorus microkernel (v3r6) has been developed under the auspices of the STREAM (Scaleable Technology for the Real-time Application Marketplace) project and is known as the STREAM kernel [Abrossimov,96a]. This project is attempting to develop a family of scaleable, real-time, distributed operating systems which can support embedded applications ranging from line cards to telephony PABX switches in an open systems environment. The STREAM kernel empowers a system builder to configure a kernel in terms of its modularised services at build time. Only the modules required for a specific (application) purpose need be included in a given STREAM instance. The configuration process is driven by a text based scripting system in a cross development environment. To guide the process, specific rules are enforced which make conditions over the dependencies between modules available in the module pool. For instance, there are two executives currently defined; Core and MicroCore, only one may be selected at a time. Depending on which is chosen, a range of modules implementing memory and scheduling services may be selected. Other service classes including communication, synchronisation, timing and interrupts define alternative modules which allow the configuration of a complete microkernel instance. Chorus ClassiX r3 [Guillemont,96] is a complete, build-time configurable operating system which incorporates the STREAM kernel and supplements it with modules to provide networking, file system and other device access services. Although it is theoretically possible for developers to construct and link their own modules into a STREAM environment, the necessary interface specifications which define a module (i.e. kernel internal interfaces, monitoring interface, hardware abstraction interface etc.) are not publicly available.

3.3.1.3 Scout

Scout [Montz,94] is a communications oriented operating system developed at Arizona University. It aims to provide efficient support for *network appliances* ranging from PDA's to video on demand storage servers. Scout centres around the tenet that the *communication path* between network appliances should be a fundamental end to end abstraction whose sole purpose is to enable the optimal flow of data along its length. The path begins with an end system IO source, spans the network and terminates at an end system IO sink. The end system CPUs provide a virtual link from the network interface to the IO sources and sinks, while the physical network medium links the network interfaces. To realise this tenet, the end to end path is implemented as a first class abstraction within the operating system. It becomes the unit of scheduling and memory allocation rather than the process. Depending on the network appliances involved in the end to end communication, each component of a path can be tailored during application development time (i.e. effectively extending the path notion) to optimise the flow of data. For instance, on the CPU link component of a path, annotations can be made to the path to indicate where scheduling can take place without inopportunely harming the characteristics of the data flow such as its CPU cache utilisation and message dispatch points (e.g. there is no point switching to a message receiving component if a message has not been sent). Protection domains are enforced using compiler based software techniques such as Software Fault Isolation (SFI) [Wahbe,93] so that heavyweight domain switches are not required when components communicate.

In keeping with the scalability demands placed on Scout, each instance is built with exactly the correct network protocol units required for the specific application that the system is being built, i.e. build time extensibility is used to ensure that instances do not have to pay a resource penalty for functionality it does not use. The designers take the approach that if the target appliance is known a-priori then there is no need to enable the function to be dynamically extensible because the purpose of the node is necessarily static, e.g. a palmtop computer with no disk could never be used for video storage. The configurable protocol work leverages an earlier project from Arizona university, namely the x-kernel [Hutchinson,91]. In this system (and therefore in Scout) it is possible to configure a kernel at build time with a number of protocol units and have applications dynamically compose them to form communication sessions as required. Protocol selection is specified via textual or graphical scripting tools in terms of protocol graphs which are then automatically built into a target system. Scout extends the x-kernel work by extending communication to cover end system issues and encapsulate the entire end to end communication path as one abstraction. In addition, Scout provides real time resource mechanisms (such as scheduling and *fast buffering* [Druschel,93a]) which allow the path abstraction to support varying network appliance requirements, such as Quality of Service (QoS) [Aurrecoechea,98].

3.3.1.4 Evaluation

Both Chorus and Scout take a black box approach to extensibility by defining services as discrete building blocks that can be configured together in a scripting environment. This approach relies heavily on the publication of private kernel interfaces and the clear, concise and complete specification of each services' semantics (including external requirements). Without such details, it is virtually impossible for third party service providers to ensure that their services will operate correctly with the kernel or other services. Neither project provides definitions of interface or service semantics (which would ideally be in a custom formal language).

Scouts notion of path annotation can be seen as the exposing of module tuneable parameters (cf. Unix) to the scripting environment. This is a useful technique for tailoring module execution to suit individual application resource usage patterns but is not a form of extensibility within itself. This is because the *policies* of such modules (e.g. page replacement strategy of a virtual memory system) remain unchanged and are unavailable for replacement.

Choices also defines discrete modules (i.e. classes) but improves on the semantics of how they should interconnect by employing sound design methodologies within an object oriented context. However, the problem here is that the abstract framework is nothing much more than a plan of how an operating system should look. The system implementer is provided with guidelines on what they must implement, but these guidelines are generally too rigid. For instance, the framework implies the provision of virtual memory but the underlying platform may not provide such facilities.

3.3.2 Link Time (Library) Extensible Operating Systems

3.3.2.1 Lipto

Like Scout, Lipto [Druschel,93b] is a derivative project of the x-kernel at Arizona University. The project was started in order to evaluate the concepts of modularity and runtime specialisation in a more general context than the network domain of the x-kernel. Many of the results of the work have been fed back into the x-kernel project in order to facilitate the integration of the x-kernel as an extensible subsystem (rather than as a standalone operating system) within other kernels (in particular the Mach microkernel [Rashid,89]).

As with the other library extensible systems, the work is based around the premise that most of an operating systems code can execute at the user level and that only a small subset of functionality (that deals with physical device and resource access) requires supervisor privileges. Having made this assumption, it is proposed that the bulk of an operating systems code is provided in a shared library linked with each application and thus collocated in the same address space. This reduces the number of context switches to the kernel (because only hardware dependent kernel services must be invoked in the kernel domain) and reduces the amount of locking required (because the library is not being concurrently shared[3]). The traditional kernel now consists of low level and policy neutral functionality implementing services protected from concurrent access and exported to the library based operating systems.

It is possible for applications to link to different library operating systems in order to access optimised functionality, but additionally it is recognised that *incremental specialisation* should also be supported. This means that the effort required to

[3] The *text* of the library is being shared, but there is one instance of the library per process, i.e. the libraries *data* is a per process item. Shared data (between libraries) need only be provided and protected at the point where the libraries are multiplexed , i.e. in the reduced kernel.

implement a customised service should be in proportion to the increase in functionality that is achieved. For instance, to alter the block caching policy of a file system it should be enough to provide a new function rather than an entire file system. To achieve incremental specialisation, it is proposed that the operating system library is actually a large number of smaller libraries implementing separate service objects. These objects are safely encapsulated by their library and export well-defined interfaces. By *composing* objects together it is possible for applications to modify the operating system's structure on a per-process basis. New objects can be inserted and old objects can be replaced by observing and mimicking their existing interface definitions.

3.3.2.2 Exokernel

The Exokernel system [Engler,95c] from M.I.T. is based around the provision of a very low level kernel interface (encapsulated inside a so-called *Exokernel*) which has the single role of exporting protected access to the available physical resources of the target platform. These resources include the privileged portion of the machines instruction set, board level hardware devices including DMA and interrupt controllers, peripheral hardware devices (via memory mapped and port controlled IO) including disks and network interfaces, and CPU controlled resources such as memory, TLB and processing time. By exposing the hardware at this level instead of providing a virtual machine with semantically rich but static services it is argued that the application developer can implement their own services directly over the securely multiplexed hardware rather than having to build them, in contrived fashion, out of a virtual machines own existing services.

The architecture for deploying an Exokernel suggests a two level operating system structure with the privileged Exokernel at the bottom and a user level library (which applications select and link to at compile time) implementing traditional operating system semantics on top of the Exokernel's base services. By developing a number of library operating systems (and allowing developers to modify or create them at the source level if required) it can be assumed that emerging application domains will be suited to one or another without too much further development effort. In the worst case of having to develop a specialist library operating system from scratch, the most complicated aspects (controlling access to resources) are already provided in a policy neutral manner by the Exokernel. To date, a number of Exokernel instances have been built (Aegis and XOK) along with a number of library operating systems (ExOS, PhOS and OO-OS) [Kaashoek,97].

An Exokernel is premised on the ability to provide protection between competing library operating systems by separating the authorisation to access a resource from the actions taken once authorisation is permitted. Secure bindings to resources (which effectively provide a capability to use the resource) provide this protection mechanism. When a resource is no longer needed, well behaved library operating systems can relinquish their secure binding on that resource via a revocation protocol. In addition, the Exokernel can itself request that a library should release a given secure binding if the resource is need elsewhere (e.g. by another library). Finally, an abort protocol allows an Exokernel to forcibly remove a libraries capability to access a resource if it does not do so voluntarily within a given time frame.

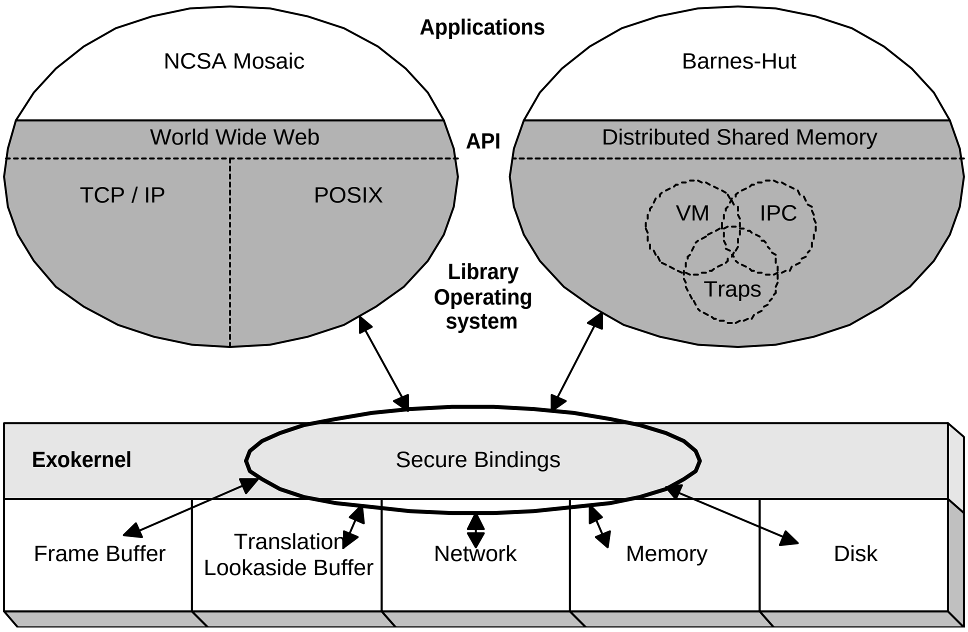

Figure 3.2: Architecture of the Exokernel

3.3.2.3 V++

V++ [Cheriton,94] from Stanford University builds on previous work on the V distributed operating system which dates back to the early 1980's. V++, like the other library extensible operating systems covered, proposes a split level operating system structure consisting of a minimal kernel (in this case, the Cache Kernel – which is fully multiprocessor compliant) and user level *application kernels* which support conformant applications with traditional operating system functionality. Application kernels can either be linked to directly (for supporting a single application) or can exist in separate address spaces to protect themselves from their applications in a multi-programmed environment. Whichever option is chosen, each application can only access the services defined by their selected surrogate application kernel.

The Cache Kernel implements a very basic level of operating system functionality, namely; simple thread structures (consisting of registers and stack information), address spaces (consisting of virtual to physical page mappings) and application kernels (consisting of an address space, trap and exception handlers, and the resources that have been assigned to the application kernel). These three entities are encapsulated as objects and assigned descriptors which are used by application kernels in order to identify them. It is the role of application kernels to share out the resources they have been allocated (e.g. physical memory, processor time etc.) between the thread and address space objects that they own. In addition, functionality which is specific to a given application kernel is used to extend the basic thread and address space objects in order to represent a more complete abstraction. For instance, a Unix emulator application kernel must extend the thread and address space abstractions with the notion of a process, i.e. a combined address space and thread, which also contains the Unix specific notion of a process identifier. Only the Unix application kernel is aware of the need to handle process identifiers and must maintain this state itself.

As well as providing core abstractions, the Cache Kernel provides a number of mechanisms in order to implement a basic computational environment which application kernels can extend. These facilities include shared memory messaging for IPC, on top of which any type of communication abstraction can be implemented (by application kernels), a fixed priority based, pre-emptive scheduling strategy and the provision of interrupt and exception redirectors (via a signal mechanism) to allow application kernels to handle IO (although the Cache Kernel still implements device drivers for the processing of the lowest layers of device interrupts), page faults and other asynchronous events.

The main crux of the Cache Kernels approach to application level resource management is the *caching* of the thread, address space and kernel objects that it exports; before an object can be considered for use (e.g. before a thread can be made ready) it must be cached within the Cache Kernel. The Cache Kernel primarily exports an interface to application kernels which consists of i) functions to load and unload these objects into the Cache Kernel's object cache, ii) signals to indicate to an application kernel that an object is missing from the cache, and iii) facilities to allow application kernels to store object state once it is written back from the Cache Kernel. By manipulating the way in which objects are considered for loading into the cache, an application kernel implements policy over the applications (consisting of these objects) that are associated with it. For instance, the Unix application kernel can have a high priority thread operating at clock intervals which adjusts the priority of its application threads according to any chosen algorithm and only loads leading candidates to the Cache Kernel thread cache for actual execution. The Cache Kernel allows the locking of a small number of objects into physical memory to ensure correct operation, e.g. the Unix priority adjustment thread just mentioned must be locked down otherwise it may never be able to execute and change the priority of its application threads. Another example is the locking of certain address space page mappings which are required for the execution of exception handlers that cannot tolerate taking a page miss, e.g. the page fault handler.

When the Cache Kernel is booted, a special application kernel named the System Resource Manager (SRM) is loaded which enables the introduction of other competing application kernels[4] in a safe and controlled manner by assigning each of them physical resources for given units of time. Each application kernel is allowed to divide up resource between their applications as they desire. Policing mechanisms are in place to ensure that resources are not abused, for example memory access arrays prevent kernels from using memory they are not entitled to and thread accounting stops kernels having their threads executed for longer than was agreed.

It is argued that the caching approach to operating system design greatly simplifies the supervisor component of the system because it deals solely with caching operations. User level kernels now implement traditional operating system functionality but have domain specific freedom to define services and allocate resources as required for their applications. One interesting observation of the caching approach to application level resource management is that a hard limit on the number of object descriptors never has to be defined (c.f. maximum process number on a Unix system) because only cached objects consume physical computational resources (CPU, memory and device bandwidth) whilst uncached objects are stored in backing memory until required.

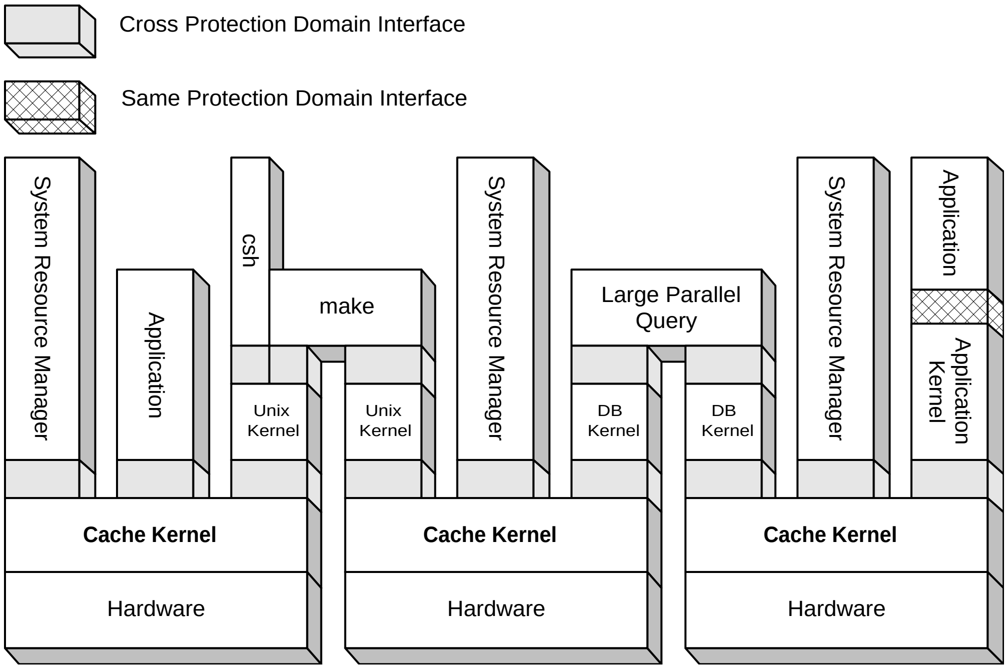

Figure 3.3: Architecture of the V++ Operating System

3.3.2.4 Evaluation

Exokernel and Lipto are relatively similar but differ in several key areas. Lipto has considered the implications of specialising library operating systems through its notion of incremental and modularised specialisation thus potentially reducing the amount of developer effort required to produce an application optimised library. Conversely, the Exokernel project has expended its efforts on the design of the low level kernel itself; library operating systems are provided and expected to be used off-the-shelf. Here, any amendments required to optimise a library for a particular application domain must be made in a similar manner to monolithic operating systems and microkernel personality servers, i.e. at the source level. This requires extensive knowledge of not only the service to be altered but also those implicitly coupled with it. However, it should be noted that incremental specialisation is only as useful as the initial decomposition of the service involved. For instance, if a file system is provided as a single component then it cannot be composed with alternative file system policies and cannot be reused in the design of other types of file systems.

---

[4] The SRM implements the loading and unloading of application kernel objects into the Cache Kernel cache.

The extra kernel development effort of the Exokernel manifests itself over Lipto in the area of resource management, specifically the allocation and protection of resources. Lipto simply embeds its resource management mechanisms within its kernel. The general principle is that the kernel should share resources fairly between competing library operating system instances. Consequently, although libraries can employ different policies over the resources they individually hold, they cannot guarantee that, under load, they can maintain the resources that their applications require. Clearly, this is inappropriate for providing differing levels of QoS between applications. Conversely, the Exokernel cleanly separates protection from allocation (though there is a common mechanism for both concerns, i.e. secure binding capabilities). By providing a revocation protocol, it is possible for different libraries to maintain different levels of resource utilisation with protection from starvation provided by the kernel's abort protocol. The Cache Kernel allows a similar type of flexible resource allocation through its SRM, i.e. resources are granted to libraries (application kernels) on a per time unit basis. Libraries can guarantee differing amounts of resources to their applications for the duration of these periods.

The Cache Kernel's notion of uniform resource allocation through cache eviction policy compares well against the Exokernel and Lipto's effectively ad-hoc resource policy decisions, i.e. implemented differently for each resource type that they manage. However, it is unclear whether cache eviction strategy is a good paradigm for all types of resource allocation, or whether it is fine grained enough for specifying specialised application domain behaviour. For instance, the allocation of non CPU related resources through caching (e.g. a network or disk) have not yet been considered. Additionally, the penalties of having to reload cache-missed objects from backing store are unspecified but can be expected to be large.

3.3.3 Runtime Extensible Operating Systems

Allowing the execution of user defined code within the supervisor address space clearly presents integrity issues. In the following sections, it can be seen that some projects have worked on resolving these issues using various techniques while other projects simply expect to employ these techniques to secure their own services and concentrate on other extensibility issues, e.g. the design and reuse of extensible services.

Service design in runtime extensible systems can be divided into two categories, *ad-hoc* and *reflective*. Ad-hoc designs are typically functionally oriented and are characterised by the use of different design methods at different points of extension within the kernel. Additionally, there are no clear guidelines as to what a service should look like (i.e. they lack a service design *framework*). Reflective service designs are object oriented and employ the principles of reflection [Kiczales,91] to standardise the design of services. Reflection allows the reification of a service's internals and the subsequent adaptation of those internals to a specific need. This is achieved through the definition of a Meta Object Protocol (MOP), i.e. a management interface, as a fundamental characteristic of every extensible service. Unless otherwise stated, the following projects employ an ad-hoc approach to service design.

3.3.3.1 SPIN

SPIN [Bershad,95c], from the University of Washington had its original white paper produced in 1994 (reworked in [Bershad,95a]) thus making it one of the most mature examples of this type of system.

In SPIN, application defined system services are decomposed into three sub units;

- *SPIN Dynamically Linked Extensions* (SPINDLEs), that are dynamically downloaded into the supervisor address space,
- *libraries*, that exist in the same address space as the application and may execute autonomously or contact kernel SPINDLEs via system calls, and
- *user level servers*, that maintains long lived state about the extensible service, i.e. state that outlives the thread or application that instantiated the service.

The SPIN operating system kernel itself is structured as a set of embedded resource controllers (providing low level abstractions and mechanisms over physical and logical resources) and a set of default SPINDLEs. Using default SPINDLEs allows SPIN to function as a fully featured operating system for applications that do not wish to define their own services. This is important because it promotes incremental specialisation thus reducing the programming effort required from application developers to make use of the system.

Applications dynamically augment the default set of SPINDLEs when they introduce new services. All SPINDLEs are attached to *event lists* and are invoked when the corresponding event occurs. Typical events include system calls, processor exceptions (e.g. page fault) and context switches. The role of a SPINDLE is to define policies over the resource usage abstractions implemented by the resource controllers. Once a SPINDLE has been inserted into the kernel, its installing application can benefit from the superior performance (reduced latency in responding to events compared to user level servers) and fine grained hardware access capabilities enjoyed by kernel level code.

In order to protect its kernel's integrity, SPIN mandates that all SPINDLEs must be written in Modula-3[5] [Harbison,92] (an object oriented, type-safe programming language). To demonstrate that Modula-3 is not unduly restrictive for writing system code, the SPIN team have implemented much of SPIN itself using Modula-3. SPIN encapsulates kernel data structures in secure and well defined objects and the Modula-3 compiler verifies (by type checking pointers) that SPINDLEs do not access interfaces that they do not have permissions for. Other useful safety properties that Modula-3 exhibits include dynamic bound checking on arrays and automatic storage management (e.g. to prevent memory returned to the heap from being subsequently reused as a different type).

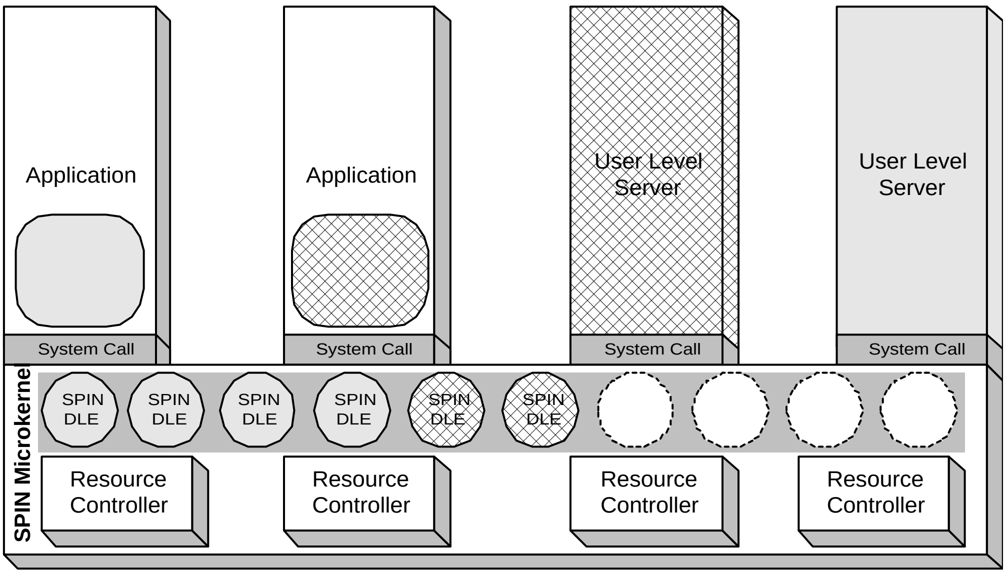

Figure 3.4: Extensible Service Design in the SPIN Operating System

[5] Ensuring that SPINDLEs are produced in Modula-3 and, further, that they have not been tampered with when they are loaded is the responsibility of a separate encryption algorithm built into each SPIN compatible Module-3 compiler. The SPIN kernel is responsible for verifying the authenticity of the cipher embedded in each SPINDLE (by the compiler) when they are loaded.

3.3.3.2 VINO

VINO [Endo,94], [Seltzer,94] from Harvard University is of equal stature and maturity as SPIN. It introduces the notion of *grafts* as the unit of kernel extension. Grafts are written in C++ [Stroustrup,94] and dynamically linked into the VINO kernel address space. From here they have access to certain kernel level interfaces in order to enhance the performance of the installing application. Generally, grafts are installed on a per-application basis. For instance, an application may choose to replace the default read-ahead caching policy it is using for accessing open files. This replacement applies only to this application; all other applications will continue to use the default read-ahead cache policy. However, it is also possible for applications to replace global policies, e.g. the kernel thread scheduling algorithm, but this facility is limited to designated authorities (c.f. the Unix superuser). Grafts are linked into the VINO kernel at designated graft points of which two types are supported. The first is the *method overriding* graft point that simply replaces a method. The second, *event handler* graft points, allow the insertion of application defined handlers for kernel events.

The VINO kernel is structured as a set of objects encapsulating resources (exporting methods for grafts to invoke) and a protected inner core of functionality implementing fundamental services such as the virtual memory page mappings and the global scheduling algorithm.

The VINO project has gone to great lengths to identify potentially unsafe behaviour in its grafts. Two broad classes of integrity violation have been identified and VINO defines explicit protection techniques to combat both of them.

The obvious integrity hole with any form of in-kernel extension is the ability to access code and data that, if misused, can cause corruption within the system and force the abnormal exit of applications belonging to other users. This is an issue of memory protection and is prevented using Software Fault Isolation (SFI) [Wahbe,93] techniques that *sandbox* grafts into dedicated logical regions within the kernel address space from which the code cannot arbitrarily escape (escapes are only possible through the invocation of exported VINO kernel interfaces acting like system calls for grafts but with reduced call overhead). The VINO team have constructed a compiler tool (MiSFIT) [Small,97] for sandboxing arbitrary graft code at the assembly level.

The code produced is secured from external tampering by encryption techniques.

A more subtle form of integrity problem is the hoarding of the resources that grafts acquire [Seltzer,96]. Malicious grafts can indirectly impede the progress of other applications via resource starvation. Two forms of resource hoarding are identified; *time constrained*, involving the holding of a resource for too long, e.g. a lock, and *quantity constrained* where a graft attempts to hold enough resources to starve out other applications. In the former case, timeouts are instantiated when a resource becomes contended; the holder must release the resource before the timer expires. In the latter case, experimentally determined physical limits are placed on the amount of resources that can be held by a graft. To protect the system when such limits are disregarded, each graft invocation is executed within the context of a kernel maintained transaction log recording each grafts actions. A transaction rollback protocol can subsequently be executed that deletes the graft and undoes its actions, thus restoring the system to a stable state.

Though VINOs' techniques for avoiding integrity violation are correct, they introduce their own performance overheads. However, as long as the benefits of application defined extension outweigh these overheads then the approach is valid. Unfortunately, determining when extension will bring an overall performance gain relies on experimental reasoning and it would be possible to develop a graft only to find that its benefits are outweighed by the overhead of protection.

3.3.3.3 Fox

The Fox project [Harper,98] at Carnegie Mellon University is applying advanced language techniques to system software in a variety of fields. The main application for this work is in the field of active networks, where it must be ensured that active packets injected into network routers do not affect the integrity of the router. The protection techniques that have been created in the Fox project are also applicable to the field of extensible operating systems, i.e. application defined code that is injected into the kernel address space must not affect the integrity of the kernel. The Fox project is exploring a number of language based techniques for securing the actions of user defined code but the approach most applicable to extensible operating system architectures is that of Proof Carrying Code (PCC) [Necula,96]. Note that the Fox team have not deployed PCC in their own operating system but simply offer it as a generic way of securing arbitrary code execution in extensible operating systems.

A system (e.g. a kernel) that employs PCC defines a *safety policy* that application defined code must adhere to (through the provision of an attached proof that conforms to the safety policy) before being allowed to execute within the system. The system derives a logical predicate for the code by scanning it (at runtime) and verifying that this predicate is satisfied by the attached proof. The beauty of this system is that it does not rely on encryption; if the proof is tampered with it cannot satisfy the predicate, if the code is tampered with, the predicate generated will not conform to the proof. In addition, apart from the one off verification of the code's proof, there is no overhead from PCC itself. This is because it is based on rigorous analytical formalisms and static type checks that do not involve dynamic modification or interpretation of the extension code.

3.3.3.4 Kea

Kea [Veitch,96] from the University of British Columbia, defines the *service* (encapsulated as an object) as the central unit of reconfiguration and extensibility. The service is supported by three fundamental system abstractions; *threads* (implemented using scheduler activations [Anderson,91]), *domains* (virtual address spaces) and *portals* (communication end-points). Client threads perform Inter Domain Calls (IDCs) on portals causing their execution context to be transferred to that of the server at the portal's end-point. Once here, they execute the desired server method.

A portal is represented as a proxy in a client's domain and maps onto a system data structure that links the proxy to a service method entry point in another domain. In this way, portals act like agents of indirection for a service. By altering the destination of a portal, client IDCs can be transparently remapped to *replacement* services. Furthermore, by making this process available during application runtime it is possible to implement extensible services. The Kea researchers are careful to observe that service replacement should be possible in two different contexts, i.e. on a per client basis and on a global basis. In the former case, only the client requesting the replacement of a service has its portal for the service remapped to the new service. In the latter case, all clients for a service that is to be replaced have their portals for that service remapped to the new service. Kea also supports the dynamic migration of services into the kernel address space for efficiency, but defers protection issues to techniques developed in other projects.

3.3.3.5 Kernel Tool Kit

KTK [Mukherjee,95] from the Georgia Institute of Technology, is an effort in enabling runtime system configuration in a multiprocessor environment. The basic premise is that application resource usage patterns change over time and therefore the way in which these resources operate must also be adapted over time in order to obtain maximum efficiency from them.

KTK is an object oriented library implemented over the Cthreads portable user level thread package. Applications link to the KTK runtime and have access to its encapsulated primitives representing virtual memory regions, threads and synchronisation primitives. KTK has also been implemented as a native operating system kernel on one multiprocessor machine.

The adaptation and extensibility mechanism defined by KTK involves the specification of programmer defined, per-object *attributes* associated with state variables, methods, instances and classes. These attributes are interpreted at runtime by system or programmer defined *policy* objects monitoring and adapting the behaviour of the base objects to which the attributes have been attached. A policy module has the ability to control non-functional aspects of an object without altering the objects external interface. For instance, an attribute can be defined which specifies that a method should be invoked synchronously or asynchronously; a policy object then checks and acts on this attribute each time the method is invoked.

In terms of parallel processing applications, significant advantages can be gained by monitoring and adapting the behaviour of locks protecting shared resources such as critical sections. An example is the monitoring of contention patterns on a spinlock and subsequently altering the number of attempts an application can make to acquire the lock before it is blocked. Trading off thread latency against bus utilisation in this way can return large efficiency gains over fixed locking algorithms. A further example of application defined extensibility is the fine grained resource management that can be achieved by defining policy objects associated with attributes that are themselves associated with KTK’s resource abstractions, i.e. threads, memory and synchronisation. A set of such attributes and policies (known as CHAOS-arc) have been defined for optimising real time parallel applications by monitoring deadline attributes associated with thread objects.

3.3.3.6 RT-Mach Dynamic Kernel Modules

Real Time Mach from Carnegie Mellon University [Tokuda,90] is an effort to extend the original Mach microkernel [Rashid,89] with distributed real time capabilities. This involves the analysis of components at all levels of the Mach operating system in order to temporally isolate their execution characteristics from one another. Components currently isolated in this way include; scheduling, synchronisation and IPC at the microkernel level, and file systems, devices, network stacks and windowing systems at the personality level. Having achieved temporal isolation, it is possible to reason about the real time execution characteristics of an application in order to ensure that it meets its deadlines or conversely to determine that those deadlines cannot be met. In the latter case, notification of a missed deadline can be passed onto the application to allow it to take corrective action, e.g. to adapt to the lack of a resource.

RT-Mach has recently been augmented with the ability to download extension code directly into the microkernels address space [Oikawa,96]. This is made possible through the Dynamic Kernel Module (DKM) which is an extension of the Unix Loadable Kernel Module (LKM) used for dynamically introducing trusted code into supervisor address spaces and deployed, for instance, in SunOS [Sun,91]. Unlike the LKM, the DKM is implemented around an object oriented framework augmented with state preservation features across successive instantiations and the ability to secure the actions of downloaded code by incorporating external techniques such as safe languages and SFI.

DKM management is implemented by a DKM server running at the user level. This server is both *active* and *programmable* and is responsible for the preservation of DKM state between successive instantiations. The server is active in the sense that it is capable of periodically executing application specified *probes* that can analyse the state of the system, e.g. to ascertain the battery life in a mobile system. The server also supports a passive mode of state notification via events generated from within the kernel itself. DKMs can be introduced into the system in one of two ways; by direct application request to the DKM server or in a response (previously programmed into the server) to some system event (detected actively or passively).

DKMs are specified in terms of *policies* and *mechanisms*. Simple DKMs implement both, while more complex DKMs may consist of a mechanism and multiple, separate policies. The DKM server is programmed to attach a default policy to a mechanism if no policy is specified by the DKM instantiator. Note that a policy may be dynamically changed at a later stage if system events dictate, e.g. battery life preservation policies may be deployed in a mobile system rather than the default policy which may exhibit superior performance but relies on algorithms that excessively drain the battery.

3.3.3.7 Paramecium

Paramecium [Doorn,95] is an object oriented operating system developed at Vrije University. The system is being used to support the Orca parallel programming language with adaptable operating system services for the production of high performance parallel applications.

The system consists of a fundamental core that is statically configured and consists of processor event management, memory management, a directory service and a certification service. All other services (and applications) are implemented as objects that are introduced on demand and, with a small amount of alteration, can be configured to execute at either the user or kernel level.

Objects are expected to be relatively coarse grained, e.g. representing a network protocol, device driver or scheduler. All objects are named and positioned in a hierarchical name space that allows them to conveniently resolve each others identity for purposes of communication. Non local communication between objects is carried out using a proxy mechanism that satisfies the linking of an object but then forces a page fault trap upon invocation. This is subsequently handled by a system routine that copies the callers arguments and switches context to the remote object.

Paramecium solves the integrity concerns raised by downloading objects into the kernel address space by issuing *certificates*. Certificates are only granted to those objects known to be trusted to execute safely within this context. No single technique for issuing a certificate to an object is prescribed, instead any validation technique can be incorporated within the certification system including; hand certification by an experimenter or system administrator, automated program proofing and the use of specialised compilation tools for securing the execution of arbitrary code sequences.

Encryption is employed to verify that a certificate has been generated from a valid source and the certificate incorporates a digest of the associated object to ensure that the object is not altered after the certificate is issued. This approach to kernel security unifies existing safety mechanisms under one framework and benefits from the ability to define fall-through paths to different certification methods if one or more other methods cannot validate the intentions of an object.

A major drawback of this system is the use of heavyweight and relatively inefficient public key cryptography that must be tediously managed on a per node basis. In addition, several other important concerns for a runtime extensible operating system are ignored or inadequately resolved. For instance, the potential contention that can arise between kernel objects wishing to use the same resource is pushed into the design of *shared use* objects that must explicitly encode as many ways to safely access the resource as required (implying that they will undergo constant modification as more access patterns are discovered).

3.3.3.8 SPACE

The SPACE (Systems Programming using Address spaces and Capabilities for Extensibility) project [Probert,95], from the University of California at Santa Barbara is an investigation into fundamentally extensible operating system architectures. Unlike in conventional operating systems, there is no notion of a single shared kernel mapped into each application address space. Instead, different applications instantiate coexisting kernel instances with each one offering a different range of application specific services. Any application with a capability to access a kernel instance may utilise that kernels services.

In order to make the system fundamentally extensible, the SPACE team have factored out the traditionally immutable kernel level abstractions of threads, virtual memory and inter-process communication. In their place are three new abstractions supposedly capable of supporting arbitrary implementations of the traditional abstractions;

- the *space* maps processor addresses to bus addresses (i.e. virtual memory mappings),
- *portals* enable processor execution to switch domains in response to interrupts

(i.e. portals abstract over interrupt and trap vectors), and

- *domains* are discrete, contiguous regions within spaces protected from random access by capabilities.

Domains can be used to host both applications and kernel services on a per-space basis, but due to the capability based protection mechanism, applications must use portals to access collocated kernel services. The authors demonstrate how these new abstractions can implement the traditional Unix-like notions of process and IPC. However, it is not clear whether these abstractions can be used in any other way, e.g. to implement a non Unix like execution model. In effect, they appear to be just functionally reduced equivalents. In addition, the use of *processor control blocks,* scheduling and clock portals implies that concurrent threading models must be implemented by application tailored kernels, thus reducing the authors claims of achieving fundamental extensibility.

3.3.3.9 JavaOS

JavaOS [Sun,97] is Sun Microsystem's original Java operating system and is designed to execute Java [Gosling,96] applications on a Java Virtual Machine (JVM) implemented directly on top of native hardware platforms rather than requiring the services of some other underlying host operating system. By eliminating this level of generic indirection, Java applications are able to execute more efficiently in terms of speed and resource requirements. A typical Java operating system is split into platform specific code (written in C and assembly) and platform independent code (written in Java). The former includes the Java kernel (incorporating boot code, interrupt handling, threading etc.) and the JVM, while the latter includes windowing, networking and device driver code.

JavaOS defines a single address space in which to execute Java threads and therefore applications. Additionally, this address space subsumes the JavaOS itself making it potentially accessible to application programs (the system can be configured so that system calls are merely extended method calls). It is therefore feasible for Java applications to dynamically extend the system using the languages in-built dynamic class loading capability, however this potential has not yet been explored.

In order to prevent the types of memory violation that may de-stabilise an operating system with a single shared address space (or, alternatively, a protected kernel accepting downloaded code into its privileged address space), JavaOS utilises the protection features of the Java programming language itself. Java is an object oriented, type safe language that has eliminated unsafe language features, such as C's pointers, thus making it impossible to access objects for which permission has not been granted (by the JVM or the typing system). This ensures that Java applications sharing a single address space cannot interfere with each other. Unlike Modula-3 (the safe language used by SPIN), Java is an interpreted language requiring an interpreter (present in the JVM) to convert intermediary bytecode into machine level code at runtime. Although this is a very slow process compared to executing compiled code, the advantage is that compiler and encryption support are not required. If the JVM notices a dangerous runtime state that was not detected by the Java compiler (perhaps because the bytecode has been tampered with), it simply refuses to execute the code and deletes the hosting application.

JavaOS employs a simple strategy for resource control. Remotely downloaded applications are restricted from accessing local file storage or the ability to use the network. Conversely, locally sourced applications have the freedom to acquire as many resources as they wish whatever the detriment to other applications.

Several other Java operating system projects have now emerged that are attempting to improve upon JavaOS particularly in the area of resource management. These include; GVM and Alta, both from the University of Utah [Back,98], J-Kernel [Hawblitzel,98] from Cornell and Joust [Hartman,97] from Arizona, which uses resource management strategies originally devised in their Scout operating system.

3.3.3.10 Chorus ClassiX

Previously mentioned as an example of a build time extensible operating system architecture (see section 3.3.1.2), the Chorus ClassiX operating system also exhibits a function unique among its commercial competitors in that it is able to dynamically replace functional code units without losing the state of the original. These code units include server based processes and portions of the STREAM kernel at the heart of ClassiX. The mechanism involved is known as the *hot-restart* [Abrossimov,96b] and was developed at Philips Research Laboratories [Hauptmann,96] (a STREAM partner).

Hot-restart was originally developed to enable fault tolerance in high availability devices such as PABXs, but it can also be used to upgrade and extend any other hot-restart enabled services. Enabling a service with a hot-restart capability involves the semi-automatic check pointing of source code to insert *exchanges* where;

- the state of a component can be fully determined, and

- the component can be completely stopped (by having its threads blocked).

Whenever on-the-fly replacement is required, a check point must be reached before the replacement can take place. At this point all the state of the running service is transferred to the replacement which updates itself using a specialised initialisation routine and commences execution.

The most obvious problem with this type of extensibility is the intrusive nature of the source code editing (it cannot be completely automated). As this is achieved in-line with existing code it presents large code maintenance and understanding problems as well as introducing some inefficiency. In addition, replaceable components are currently of a very coarse grain nature in order to facilitate a reasonable chance of meeting the requirements for establishing an exchange.

3.3.3.11 Fluke

The Flux Research Group (FRG) at the University of Utah are extremely active in the field of general operating system research. Their projects include; Java operating systems (see section 3.3.3.9), the OSKit [Ford,97a], [Ford,97b], Fluke [Ford,96] and Flask. The OSKit is an effort to remove the drudgery from developing native operating systems on behalf of the operating system research community by collating and documenting source code from various free software projects (plus Utah's own contributions, particularly in bootstrap and loading) into an organised distribution. Already several teams from other research establishments (including Off++ [Ballesteros,97]) are using OSKit to help their projects bypass the uninteresting fundamental development tasks (bootstrap, device drivers, file systems etc.) and move quickly onto their particular research interests. The Flux group have also used OSKit to develop an implementation for their Flux microkernel (Fluke) specification.

Finally, Fluke OS is a complete operating system environment based around the Fluke implementation and augmented with user level servers providing services such as process management and file systems.

Fluke represents the FRG's major contribution to the field of runtime extensible operating systems. The specification (which has an implementation of the same name) is based around the concept of *virtual machines* originally introduced in IBM's VM/370 system [IBM,73]. However, rather than present an exact replica of the underlying machine architecture at each layer, Fluke defines its own virtualised software architecture that may be extended at each layer by the addition of *nesters* (virtual machine monitors). Nesters are supported by the infrastructure that is implemented by parent nesters in a hierarchical fashion. Applications (at the leaves of the hierarchy) make use of each nesters functionality.

Fluke exports three interfaces that collectively define the overall virtualised architecture. These interfaces are:

- the basic instruction set of the machine (privileged operations are protected by the microkernel),
- a primitive microkernel API exporting kernel objects including policy free address spaces, capabilities, IPC channels and threads, and
- the *common protocols* API that enables nesters to be inserted into the nester hierarchy and subsequently layer policies and mechanisms over the kernel objects. For instance, a process management nester can take thread and address space objects to produce a Unix like process model.

One of the problems of a layered virtual machine architecture is the performance overhead of communicating between layers. The designers attempt to resolve this issue by intelligently dividing resource management responsibilities between nesters and the microkernel. Only relatively infrequently used aspects of the system can be implemented by nesters and there is a short circuit mechanism to allow nesters at higher levels to directly interact with those at lower layers. Nesters that choose to extend rather than replace a parent nesters interface can allow direct access from its children to its parent thus improving efficiency. The restrictions placed on the initial level of resource manipulation (i.e. what must be present in the microkernel for efficiency reasons) are of more concern because they limit the scope for fundamental extensibility and, from the interfaces and kernel objects described, the system seems limited to the creation of Unix like personalities (albeit each potentially optimised for different application domains) rather than radical new architectures. A final concern is that the granularity of nesters is very coarse (akin to user level servers in traditional microkernel based systems) indicating that implementing a nester is a significant task not suited to application developers.

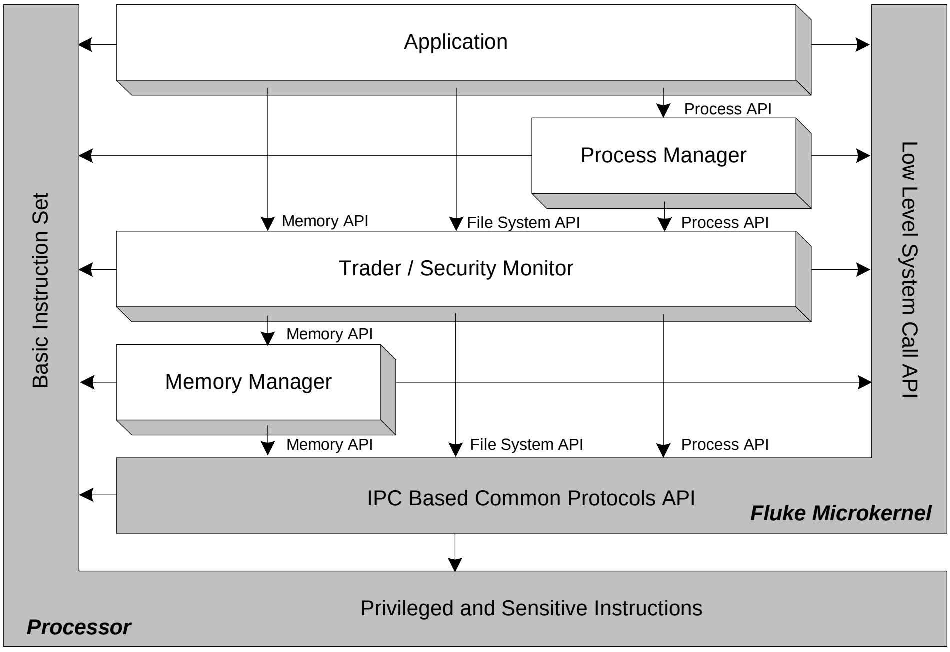

Figure 3.5: Architecture of the Fluke Operating System

3.3.3.12 Choices / µChoices

Choices / µChoices has already been introduced in section 3.3.1.1, where its notion of build time extensibility via sub classing of an abstract framework was presented. Choices also demonstrates a high degree of runtime extensibility through dynamic code insertion.

Although the system is implemented in C++ (for performance reasons), the authors of the work are quick to point out the deficiencies of this language in terms of genuine object oriented functionality and detail a number of extensions intended to overcome these deficiencies. In particular, a need is expressed for runtime facilities such as garbage collection, first class classes and dynamic class loading. These needs have been satisfied through three new features;

- alteration of the C++ runtime for object creation,
- a library of base classes to use, and
- a number of programmer conventions (patterns) to be followed.

Runtime extensibility is implemented through the on-demand dynamic loading of classes (from their object files) into the system framework. The classes in this framework are afforded first class status in that they are implemented by objects that are built into an actual Choices system. They maintain information such as the class hierarchy (that is lost by C++ after compile time) and the ability to dynamically create instances of the classes when required.

When a new object is to be created within the system framework, its class object's constructor is called to dynamically load its real class code. The class object then calls a well known, separate constructor compiled as a static function outside the class. This constructor calls the real class constructor to dynamically create the object and returns a pointer to it. The object's creator can now access the new system object by pointer to member function calls. By only instantiating appropriate system objects, the system can be dynamically extended to suit the demands of target application domains.

3.3.3.13 SLIC

The SLIC [Ghormley,98] (Secure Loadable Interposition Code) project from the University of California at Berkeley is derived from work relating to the Network Of Workstations (NOW) project also at Berkeley. NOW is concerned with combining clustered commodity workstations (using commodity operating systems and interconnected by high speed, low latency networks) into a single virtual supercomputer image. Central to this work is the development of a portable distributed operating system layer (named GLUnix [Vahdat,94]) that leverages support from the underlying commodity operating systems in order to present a unified machine image across the network. In order to unify diverse native operating systems, GLUnix requires the ability to *interpose* its own functional code units over those systems. Interposition is the insertion of trusted binary code onto an existing kernel interface, i.e. within the kernel address space. The interposed code enhances the facilities of the interface while exporting the same method signatures so that application binaries are unaffected by the interposition. SLIC is being developed to ascertain the effectiveness of interposition over existing commodity operating systems with the goal of feeding back results into GLUnix.

As a research project in its own right, SLIC takes a pragmatic view of operating system extensibility. Central to this view is the fact that existing commodity operating systems (Unix in the case of SLIC) will not disappear overnight to be replaced by new types of extensible operating system architectures. Furthermore, due to the complexity and unstructured nature of existing systems, it is not feasible to introduce many of the esoteric extensibility mechanisms proposed in the literature. SLIC maintains that existing operating systems can only be leveraged through interposition on existing interfaces. Some advantages of interposition include; the ability to stack arbitrary numbers of extensions on the same interface and an incremental nature, i.e. extensions only have to be written for the service that is to be extended.

The mechanisms involved in implementing interposition include;

- *dispatchers* that intercept events crossing kernel interfaces, e.g. system calls and signals, and selectively (on a per process, per event basis) redirect these events to extensions,

- *extensions* that implement new functionality over existing kernel interfaces (and may completely disable an underlying function, e.g. when eliminating a system bug) and,

- *utility functions* that provide extensions with an abstract view of system internal interfaces by hiding platform dependent details.

SLIC demonstrates a good degree of efficiency but suffers from a lack of flexibility which the authors candidly admit. First, SLIC must assume well defined interfaces onto which it can interpose; this is not always the case. Second, extensions can only build on the existing underlying kernel infrastructures for a given service. For example, a file system can only be modified to support alternative pre-fetching algorithms if pre-fetching is already a supported mechanism. The former problem has led to the establishment of a number of principles that the SLIC team would like commodity operating system vendors to observe in order to make interposition more effective, e.g. the correct decoupling of policy and mechanism in system functions.

3.3.3.14 MMLite

Recent work by Microsoft Research has realised the MMLite operating system [Helander,98]. This runtime extensible system was originally developed and deployed as a board level operating system for supporting the control of peripheral processors, e.g. those present on modern graphics cards. These processors have become extremely powerful and have direct access to large amounts of board level memory. MMLite attempts to offload CPU computation onto these peripheral processors in order to increase system performance.

The remit of MMLite has since been expanded to address issues of extensibility in a general purpose computing environment, i.e. within the execution context of standard CPUs rather than just with peripheral processors. The system goes significantly further in opening up more of the operating system to application initiated reconfiguration than other extensibility projects in this survey. For instance, IPC mechanisms can be introduced dynamically and the system can function either with or without virtual memory.

MMLite is based around the Component Object Model (COM) [COM,95] in that its services are encapsulated as COM objects and bound together using the COM binary interconnection standard, i.e. the C++ vtable structure. Using a technique called *mutation* (that effectively overwrites the pointers in a services vtable), services can be dynamically replaced while they are still in use. A variant of the holding tank algorithm (described in [Cowan,96]) is used to avoid potential race conditions while replacement takes place.

3.3.3.15 Apertos

Apertos [Yokote,92] (formerly known as Muse [Yokote,88] and now known as Aperios) from the Sony Computer Science Laboratories in Japan is the most prominent exponent of reflective techniques in the field of runtime extensible operating systems.

Apertos introduces the notion of a meta-object for implementing the non-functional aspects (including persistence, execution environment, replication, message handling, invocation semantics etc.) of its base objects. This separation of concerns allows base object programmers to concentrate on supplying the semantics of their objects without the need to integrate the listed orthogonal behaviours. Each base object is associated with a group of meta-objects providing a *meta-space* for the execution of the base object. As meta-objects are themselves objects, they are also associated with meta-objects in a potentially infinite tower of recursion. Similarly, each base object may exist in the context of several meta spaces. At the base of this tower is *metaCore,* which is a microkernel-like primitive providing fundamental services available to each meta-object. Base objects communicate with meta objects via methods that they export in their Meta Object Protocols (MOPs) [Kiczales,91].

Apertos defines *reflectors* to represent a group of meta objects (i.e. a meta space) implementing system services with which base objects can associate themselves. In Apertos there is an extensible hierarchy of these reflectors consisting of a number of immutable and predefined reflectors providing essential system services to other reflectors. Note that these services are not part of metaCore because the latter is intended to be as free of policy as possible (for instance it provides context objects for scheduling but allows a reflector to control the actual task of object selection and dispatch). The most notable reflectors include; m^{Common}, at the root of the hierarchy defining a number of core meta object properties for other reflectors, $m^{\mathrm{ZeroObject}}$ which provides services to enable the creation of new reflectors, and $m^{\mathrm{BaseObject}}$, which defines a concurrent, object oriented programming environment. Meta objects can be created using $m^{\mathrm{BaseObject}}$, *migrated* to $m^{\mathrm{ZeroObject}}$ and form their own reflector that can then be associated with base objects.

The process of meta space migration is the central abstraction for extensibility within Apertos; any object (base or meta) can request to have its meta space descriptor updated by transferring or replacing its current meta object environment with that of a new meta space offering different non functional services. This process can also be initiated arbitrarily by the system in the face of dynamically occurring events. In order to migrate to a new meta space, the source and target spaces must agree on their compatibility levels for the base object to succeed in its request. The compatibility negotiation is performed using a base service inherited from m^{Common}.

Although Apertos is largely implemented in C++ it is well known to suffer from large performance overheads [Itoh,95]. This is due mainly to the immense amount of management information required by the system runtime in order to support the meta space abstraction and all its involved processes and indirection. In addition, each object is associated with its own address space context thus implying a high inter-object communication overhead.

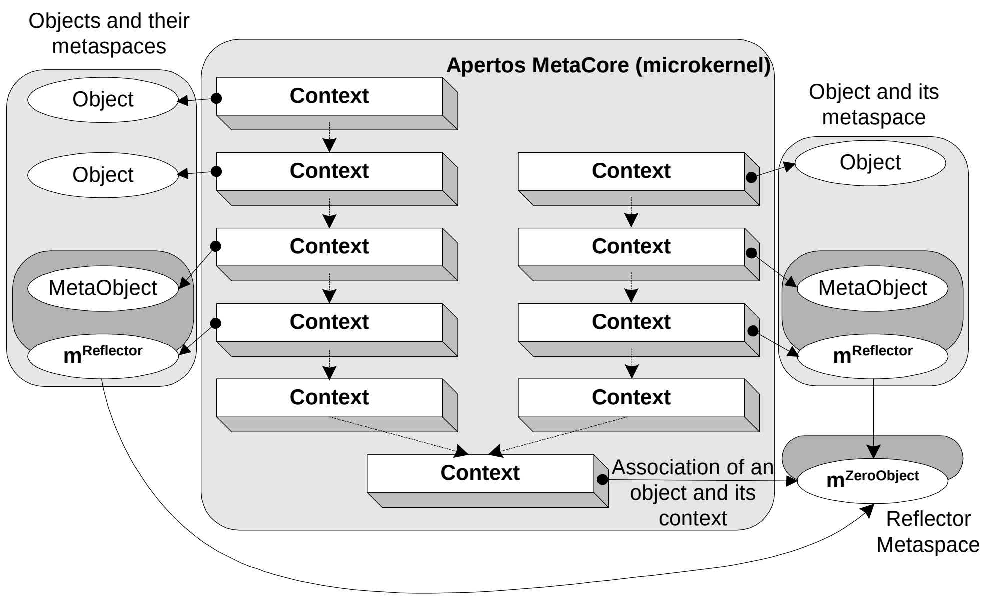

Figure 3.6: Architecture of the Apertos Operating System

3.3.3.16 MetaOS

The Panda group at the University of Victoria in Canada have strong links with Sony Computer Science Laboratories and have been allowed to derive their own reflective operating based on the work of the Aperios group. MetaOS [Horie,98] attempts to adhere to the principles of meta level programming whilst enhancing performance and proposing a way of assigning authority over who may introduce extensions (through meta-space migration to new reflectors) to the system. In particular, MetaOS limits the tower of recursion to just two levels arguing that further levels are inefficient and rarely required. This two level hierarchy is split into local meta spaces (the meta level) and global meta spaces (the meta-meta level). Local meta spaces deal with services that apply to individual objects such as file read-ahead policy, whereas global meta-spaces pertain to services which, if altered, would affect all objects in the system, e.g. the thread scheduling policy. Objects can generally only manipulate meta objects at the local level (thus only impacting themselves). In order to manipulate global policies, objects must be present on administrator defined *access* *control lists* thus shifting the responsibility of reconfiguration to trusted objects.

3.3.3.17 DECADE

DECADE [Chiba,97] by Chiba et al. at the University of Tokyo, Japan is a recent example of a reflective, object oriented operating system. The system follows the usual base and meta level separation with communication via a MOP. Drawing on experience with reflective programming languages, DECADE allows application programmers to define meta objects in a C like language that is interpreted by an in-kernel interpreter thus ensuring protection from malicious meta-objects.

As a second phase of the project, it is proposed that to make the system efficient an alternative protection mechanism to in-kernel interpretation must be found. The DECADE team propose that *weak protection* can be used to support the coexistence of meta-objects in the same address space without the need to interpret their code. Weak protection techniques include;

- encapsulating meta objects in user level libraries exporting method entry points and communicating with the kernel via shared memory, and

- using 64 bit address spaces to separate meta objects in the knowledge that the address space is so sparsely backed by physical memory that erroneous memory accesses will result in traps that can take action on the offending meta object.

Clearly, weak protection does not stop malicious programmers from damaging the system but it is argued that typically only trusted users can instantiate meta objects anyway and that they are not likely to wilfully damage the systems integrity. In this regard, weak protection can statistically protect against accidental, mistaken memory access. Other kernel level extensibility issues such as resource starvation have not yet been considered.

3.3.3.18 CHEOPS

CHEOPS [Schubert,97] (CHEmnitz OPerating System), by Schubert at the Chemnitz University of Technology, Germany, is a reflective, object oriented operating system that attempts to bridge the gap between language level objects and their system level equivalents. The bridging technology used is the C++ programming language suitably modified to enable the reification of runtime properties thus allowing the system to be dynamically extended. The benefits of using C++ are that it is both efficient (as it is a compiled language compared to the interpreted languages used by many other languages implementing reflection) and it is well known at the application level, i.e. will be familiar to developers wishing to provide new CHEOPS services.

Reflective systems rely on the presence of management information detailing the relationships between entities (typically objects) within the system. Contrary to this, C++ is purely a compile time object oriented language; all class related information (that might be used to reify a programs runtime structure) is lost when code is compiled. In CHEOPS, the designers have identified this problem and augmented C++ with *class object classes* (derived automatically from a program's source) that are instantiated as objects at runtime and exist solely to maintain the explicit class relationships between any objects that are created (c.f. Choices). Each class object (an instance of a class object class) maintains meta level information for all the base objects that are instantiated from it. In order to obtain this information, the C++ object creation operator is modified to create structures within the context of the associated class object and to allow the class object to anchor base objects into the runtime class hierarchy. Note that this means objects must be created via the dynamic instantiation operator or else they will not managed by their designated class object (c.f. stack based local objects). There are also several other restrictions placed on C++ that must be observed to avoid clashes with the CHEOPS infrastructure. Currently these restrictions are not actually enforced on programmers (through modified C++ compilers) because there is a desire to use standard tool chains.

By enabling the manipulation of object creation, class objects are able to interpose on the virtual method tables associated with their base objects and perform pre and post method processing on each base object invocation. In this way reflective computation that makes use of the management information maintained in the class object hierarchy (including information about their base objects, e.g. location) can be performed without the knowledge of the base objects, i.e. a true separation of functional and non functional processing is achieved.

3.3.3.19 2K

2K [Kon,98] represents the latest effort by the Systems Research Group (SRG) at the University of Illinois at Urbana Champaign to manage dynamic adaptation within an operating system environment. 2K is a reflective system that encompasses a wide area of runtime extensibility research including; operating systems, active networking and middleware in order to present a distributed system that treats users and resources as networked entities not restricted to residing at any particular node within the environment.

The key to dynamic adaptation in 2K is architectural awareness, i.e. the reification of structure, state and behaviour in order to enable the controlled introduction of new system services. Adapting to changes in the environment is the responsibility of the middleware as implemented by a reflective Object Request Broker (ORB). The ORB admits changes to the system structure either with or without the knowledge of its applications and is responsible for the initial configuration of the services required by an application when it is launched. An important part of the reification of 2K is the explicit statement of interdependencies between objects thus allowing 2K to ascertain the best services required to support a given application.

The Off++ microkernel [Ballesteros,97] is being used in the implementation of a pilot 2K system. It has been explicitly designed with reflective properties in mind to allow the 2K middleware to observe and alter its internal state according to dynamic changes in the environment.

3.3.3.20 Evaluation

Runtime extensible operating systems can be evaluated according to how well they provision for the following criteria:

- *Locality of service extension*; some systems only allow the introduction of *global* service extensions that affect the operation of all applications. An example would be the replacement of a scheduling policy. In contrast, *local* service extensions apply only to the application that initiated its introduction. For example, an application may specify a different read-ahead strategy for its open files than is being used by other applications. Clearly, there are situations where both types of extension are useful and systems should support both.

- *Protection against malicious behaviour*; the major problem with runtime extensible operating systems is the damage to system integrity that can be caused by having erroneous application defined code downloaded into the supervisor address space. Specifically, there are three actions that such code can perform that may damage system integrity; i) corrupting memory, ii) executing privileged CPU instructions, and iii) consuming enough system resources to starve other activities from continuing to function. Prevention of the first two actions has largely been resolved through the use of compiler techniques that have already been discussed, e.g. safe languages, SFI etc. However, resource hogging has been ignored by most related projects.

- *Authority to initiate extension*; most runtime extensible systems assume that if kernel downloadable extensions are safe (usually in the context of memory safe rather than resource safe) then any application may introduce them. Clearly, this issue should be related to locality of service extension, i.e. any application may extend their own local services, but the question of who may extend a global service is still an open issue (such changes may detrimentally affect unrelated applications despite not harming the system). Some systems defer this authority to the superuser or Access Control Lists (ACLs) but this merely avoids the problem by moving it out of the system and into the hands of an operator. A better solution may be to automate the authority process by having the system evaluate the effects of a global extension before allowing it to be introduced.

- *Mode of extension*; generally, extensible services either replace existing services or add new ones. Interposition can only replace existing services with replacements that *must* mirror the replaced services exactly in terms of signature and semantic (though the service will be enhanced in some way). This is because interposition is used to replace existing service interfaces that legacy applications still depend upon. Adding new services at runtime or altering the semantics (usually policy) of an existing service in both local and global locality contexts offers more flexibility and can subsume interposition.

Of the non reflective systems, only SPIN, VINO, Kea, Fluke, KTK and Choices make specific mention of local extension mechanisms. The remaining systems only allow global extension. Enabling both local and global locality extension need only be a matter of providing an indirection mechanism between applications and their services. In this way, if an application extends a service locally, only its indirect link to the new service need be updated. If a global service extension is performed, then every applications indirect links to that service are updated. This approach is exemplified by Kea. Reflective systems have implicit support for extension locality with their hierarchical meta space architectures. MetaOS, in particular, highlights how changes to the meta space of an object only affect that object, while changes in the meta-meta space affect all objects. For instance, each object may have user level threads with their own scheduling policies at the meta level, while there is a single global scheduling policy that exists at the meta meta level for the kernel threads they are multiplexed on.

All runtime extensible operating systems recognise the importance of protection against malicious extensions. For instance, SLIC only allows the downloading of code that is deemed to be *trusted*, i.e. it is assumed that all extensions are safe. Clearly though, this approach cannot make any real guarantees on the integrity of its extensions as even trusted extensions may be bugged. Likewise, KTK systems are premised on the presence of only one active parallel application, therefore there is no need to define mechanisms for protecting the system from malicious extensions or trust relationships over who may introduce those extensions. Some other systems have either defined their own protection techniques (e.g. SPIN, VINO and JavaOS) or, more usually, just cite these protection techniques as ways of securing their kernel extensions (e.g. Kea, Paramecium and RT-Mach). Of the reflective systems covered, only DECADE attempts to improve on the simple notion of trust for securing the execution of its extensions. It defines weak protection for enhancing performance but cannot guarantee safety against malicious kernel extensions.

The majority of the systems reviewed have only considered protection in the context of memory and privileged instruction safety. FLUKE and VINO also attempt to prevent extensions from starving other applications of resources. FLUKE achieves this through the equal division of resources amongst its nesters (which further allocate these resources to their applications) and VINO through resource monitoring using the transaction rollbacks if resource quotas are ignored.

Currently, the state of the art in assigning authority over who may initiate system extensions is the ACL. This is clearly only one step away from the inflexible notion of the Unix superuser. Expecting system administrators to shoulder the burden of deciding which applications may introduce extensions is unrealistic in that they cannot foresee the effects of each and every extension that is created. This issue is particularly important in the field of active networking operating systems as they accept code for execution from remote sources whose identify may not be verifiable. The extensibility community must address the issue of automated authority designation if the technology is to become commercially acceptable.

Finally, only SLIC defines interposition as its sole method of extensibility though this is for pragmatic reasons, i.e. to allow legacy applications to function unaltered in existing commodity operating systems. All the other systems allow new services to be added and existing services to be augmented or replaced.

3.4 Overall Evaluation

3.4.1 Evaluation of the Continuum

This section begins by evaluating each category in the continuum of *application defined* extensible operating systems and then makes some comments on *system defined* extensible operating systems.

3.4.1.1 Build Time Extensibility

Build time extensible systems are able to support application specific functionality. However, selecting services at build time requires that the target application domains be known a-priori. This makes them unsuitable for hosting general purpose (workstation) computing workloads where a users choice of applications cannot be predicted. In addition, applications that wish to dynamically alter the services that they are using (in order to adapt to changes in their runtime environment) cannot be supported.

3.4.1.2 Link Time Extensibility

Library extensible systems basically suffer from the same problems as build time extensible systems. Although the system can now be extended at runtime (by having different applications select different libraries) and can therefore support general purpose workloads, applications still cannot dynamically adapt to changes in their environment (as they cannot be unlinked from the library that they selected). More importantly, library operating systems can only implement services in terms of the underlying kernel's interface. The sole purpose of this interface should be to provide wrapped access to hardware resources. However, there is currently no consensus on the nature of these wrappings. If too much wrapping is applied, then the system runs the risk of providing too much generality (thus meaning that libraries cannot implement highly optimised, application specific services) and can also be accused of influencing the design choices available to their library operating systems. For example, all three systems in this category provide a notion of concurrency control within their wrappings. This implies that all library operating systems assume a multithreaded environment. Clearly this would not necessarily be the case for an embedded system's library. Consequently the embedded system would pay the price of unnecessary concurrency management overhead.

One positive point to make about link time extensibility is that the libraries execute at user level and are therefore implicitly protected from any malicious behaviour by any other library. In addition, the overhead issue of having operating system services making cross protection domain calls to a kernel interface has largely been resolved by virtue of the fact that most of the operating system code executes within the library and only rarely needs to invoke the kernel's services. Furthermore, the Exokernel in particular has made impressive optimisations of this communication channel to offset the system call overhead [Engler,95a].

3.4.1.3 Runtime Extensibility

Runtime extensible systems are able to support both dynamically changing environments and dynamically changing workloads. In addition, the protection issues over downloaded kernel code have largely been resolved by a number of software based techniques that eliminate dangerous artefacts that such code may contain.

The main difference between reflective and ad-hoc systems is that the former provide a uniform environment for extensibility, i.e. the notions of meta space and MOP define a clear and orthogonal separation between applications and the ability of those applications to extend the system. In addition, there is typically only one unit of extensibility, i.e. the object, thus allowing services to be extended in the same manner. In contrast, ad-hoc systems are based on unstructured approaches to extensibility meaning that only those services explicitly identified as being available for extension (by the systems' designers) may be extended from the applications point of view. Furthermore, there is no consistent methodology over the way in which these extensions are designed, i.e. the way in which the system is extended at one point is not the same way in which it will be extended at some other. This makes it difficult for application developers to create their own extensions.

Unfortunately, the well documented performance inadequacies of reflective systems are a fundamental consequence of their meta level facilities, i.e. the overhead of maintaining enough management information to enable reflection causes a concomitant degradation in runtime performance. In contrast, ad-hoc runtime extensible systems can implement their extension techniques with little or no performance overhead. Note that both types of system must pay an additional performance overhead if they wish to use a software protection technique to protect the integrity of the system in the face of extensions that are loaded into the supervisor's address space.

3.4.1.4 System Defined Extensibility

Runtime based systems are the only representative of system defined extensibility. Furthermore, only the Synthetix project has been active in this area. The major advantage of this type of system is that the application developer is completely insulated from the need to explicitly encode extensibility as the system transparently extends itself on their behalf to optimise performance. However, it can be argued that an application developer already knows the types of service they require from an operating system, therefore specifying them (which does not require actually creating them) is not an inherently difficult process. Furthermore, it is clear that no matter how good a system is at transparently extending its services, it will never be able to perform as well as hand crafted extensions that have been designed according to application domain specific knowledge (not available to self extending systems).

3.4.2 A Critique of the State of the Art

The most serious deficiency in the state of the art is the lack of *fundamental* extensibility exhibited by the systems reviewed. By fundamental, it is meant that the entire operating system should be open to extension and not just a limited set of services (e.g. file system, device drivers, process model) that have already been factored out of traditional monolithic operating systems by various microkernel projects. In particular, the concurrency, memory management and Inter-Process Communication (IPC) mechanisms should be available for extension. These facilities offer some of the best opportunities for application defined optimisation. Currently, the only other extensible systems to identify this requirement are the Chorus STREAM kernel, SPACE and MMLite. However, the Chorus kernel is only extensible at build time, and, of the remaining two runtime extensible systems, the former replaces these fundamental mechanisms by virtual equivalents, while the latter operates within a limited operational context, i.e. it is not yet a fully fledged general purpose operating system. Every other system defines some notion of these fundamental mechanisms (usually virtual memory, threading and default IPC).

Finally, an additional criticism of the state of the art is the lack of support for visualising a system's extension points and installed extensions (particularly at the global level of locality). This means that applications must rely on ad-hoc methods of ascertaining where they may install their extensions, and further, they cannot take the pre-existence of other extensions into account, which may limit the effectiveness of their own extensions.

3.5 Summary

In this chapter, the state of the art in the field of extensible operating systems has been reviewed and critiqued against the requirements laid out in chapter 2. The literature has been divided into a continuum that is categorised according to;

- *When* each system allows service integration take place, and
- *Who* decides which services are to be integrated.

Based on the evaluation, it is argued that application initiated, runtime extensible systems implemented with non-reflective techniques best satisfy the aforementioned requirements. However, current research systems of this type exhibit a number of deficiencies that need to addressed if extensible operating systems are to deliver on their promises of improved performance and convenience.

Chapter 4 An Architecture for Runtime Extensible Operating Systems

4.1 Introduction

Chapter 1 of this thesis argued that current operating systems are not providing adequate support for emerging application domains. Chapter 2 began by proposing a number of requirements that must be satisfied if this situation is to be rectified and then used them to evaluate the three most prominent operating system architectures. This analysis concluded that extensible systems offered the most appropriate solution for meeting the needs of future application domains. In chapter 3, the literature regarding such systems was categorised and evaluated according to the points in time at which they allow system extensions to be integrated, i.e. at build-time, link-time and runtime. This examination has lead to the recommendation of the following proposals that will be used to guide the design of a new extensible operating system that extends the state of the art in terms of support for emerging application domains:

- *Extensibility should be application defined and should take place at runtime*; applications implicitly embody domain specific knowledge which only they can utilise to improve their performance. Therefore, it should be up to the application programmer to define the system extensions that their applications will depend on. This process should occur at runtime to allow applications to take advantage of dynamically changing environments, e.g. distributed multimedia applications may need to adapt their processing algorithms according to changing levels of Quality of Service (QoS) and connectivity in the network.

- *Extensibility should be possible at fundamental levels*; current extensible systems only allow a limited amount of alteration to their fundamental abstractions (consisting of memory model, concurrency model and IPC model). For example, scheduling algorithms can be adjusted or the page eviction policy of the virtual memory subsystem can be modified. However, the actual wholesale replacement of a fundamental abstraction is not possible. Similarly, diverse implementations cannot simultaneously coexist. It is argued that such levels of flexibility can provide applications with leverage to extract far more benefits than the minor variations that are currently possible.

- *Extensions should be introduced through a conventional API programming style*; it was argued in chapter 3 that the overhead of supporting reflective techniques in extensible operating systems outweigh their advantages of openness and uniformity. However, systems that use a conventional API programming style to introduce extensions tend to suffer from not being open enough, i.e. their designers only define functions to extend the system at locations where they anticipate extension to be useful. By designing systems in terms of componentised services (that are introduced at runtime) it is possible to offer a neutral API which simply offers services to load, configure and reconfigure these components without prescribing any ordering, locality or timing constraints upon them.

- *There should be a uniform model for designing extensions*; ad-hoc extensible systems tend to support different designs of extension at different points of extension. This discourages developers from designing extensions because they are presented with a non-uniform programming model. Consistency should be provided through the use of a single unit of extensibility, cf. the object, as used in many reflectively extensible systems.

The rest of this chapter describes the design of an extensible operating system named DEIMOS (Dynamically Extensible Incrementally Modularised Operating System) which has been derived according to the above recommendations. Section 4.2 covers the overall design of DEIMOS including its split-level architecture, base services and user defined services, while section 4.3 details the provision of integrity conditions that can be used to constrain how applications may extend the system. Finally, section 4.4 offers a summary of the chapter.

4.2 Architecture

DEIMOS is structured in terms of two architectural elements;

- a pool of dynamically loadable components [Syzperski,98] representing both application and operating system functionality (e.g. algorithms, windowing code, synchronisation primitives, file systems, device drivers, memory models etc.), and

- a boot archive (known as the *core*) that provides a minimally prescriptive computational environment for these components to be loaded into through an API dedicated solely to enabling their introduction and reconfiguration.

This conforms with the classic architecture of an extensible operating system as presented in section 2.2.4.

Application programmers ascertain the operating system requirements of their applications and dynamically request that the DEIMOS boot archive load and integrate appropriate services in order to form an optimal environment for them to execute within. Such an *execution environment* consists of at least a memory model, Inter-Process Communication (IPC) services, concurrency model and loader for loading environment specific component images into the contexts defined by the environment.

In DEIMOS, an application can be defined in one of three ways depending on how DEIMOS aware its programmers are:

- *Fully aware applications* are constructed from components that orchestrate the complete integration of their execution environments and interact directly with low level components implementing services such as memory models and loaders. Such applications would be expected to routinely dynamically reconfigure the components that they load in order to adapt to a changing environment.

- *Semi-aware applications* are constructed from components that load and interact with intermediary components (known as *execution environment components* and defined fully in section 6.2) in order to indirectly access low level system services. These intermediary components sub-load their own set of system components to form an execution environment for their applications to execute within. The competent semi-aware application programmer may reconfigure their applications environment, but this type of operation would not be performed frequently.

- *Unaware applications* are not actual DEIMOS components at all. They are typically legacy applications designed for other operating systems that can be executed on a DEIMOS configuration only if a suitable environment is set up for them, e.g. by some initial component that loads further components to emulate such an environment. These applications will clearly not attempt to reconfigure the components supporting their execution as they are unaware of their existence.

4.2.1 Component Model

Structuring a system in terms of discretely loadable components lends itself well to an object oriented design methodology[6]. In DEIMOS, components are encapsulated entities which communicate with other components in their environment through strongly typed *interfaces* defined in a programming language independent Interface Definition Language (IDL) [OMG,99]. Before a component may communicate with a peer, it must explicitly establish an association, or *binding*, between them. The establishment of this binding is subject to the evaluation of *protection constraints* associated with the two parties, e.g. it may be specified that calls on an interface are re-entrant or that a component may only allow one binding at a time. If the conditions are not satisfied then the binding request is rejected.

DEIMOS supports only operational bindings [Blair,97a] and accordingly defines a specialised mode of interaction between components that consists of a notion of a source and a sink entity. The sink provides an interface (with each interface having its own set of protection constraints), while the source provides a *receptacle* [OMG,98] that hosts a reference to an interface once a binding has been established with one. In addition, a component defines its own set of protection constraints (separate to those associated with its interfaces) that are evaluated whenever the component is named in a local binding request, i.e. when one of its receptacles acts as a source or one of its interfaces acts as a sink.

Note that receptacles may hold more than one instance of an interface reference (but each reference is to an interface of the same type as the receptacle). This

[6] The component model employed by DEIMOS is based around an enhanced version of the computational model of the Reference Model for Open Distributed Processing (RM-ODP) [ITU-T,95]. Although this model was originally designed for open distributed computing environments, i.e. middleware, it is equally suited to representing application and system activity and interaction within DEIMOS.

extension allows the modelling of event style interactions where a component may solicit the binding of multiple instances of an interface type with each implementing a call back method with the same signature. The component then propagates event messages to all of its bound interfaces through the receptacle. In addition, it is sometimes necessary to distinguish between different instances of an interface pointer within a receptacle rather than treat them all as homogenous (e.g. as with event sinks). To accomplish this, a receptacle will define index positions that correspond to specific implementations of an interface. By checking documentation on a component it is possible for a user to request that a particular interface reference is fed into a specific index within the receptacle. For example, consider a component selecting between different types of network interface each represented by other components all implementing the same interface, e.g. `INetwork`. There may be Ethernet, GSM and ATM implementations of the `INetwork` interface (see figure 4.1). By binding each component's implementation of `INetwork` into the `INetwork` receptacle at the appropriate index, the component is able to call the correct interface depending on the type of network currently available.

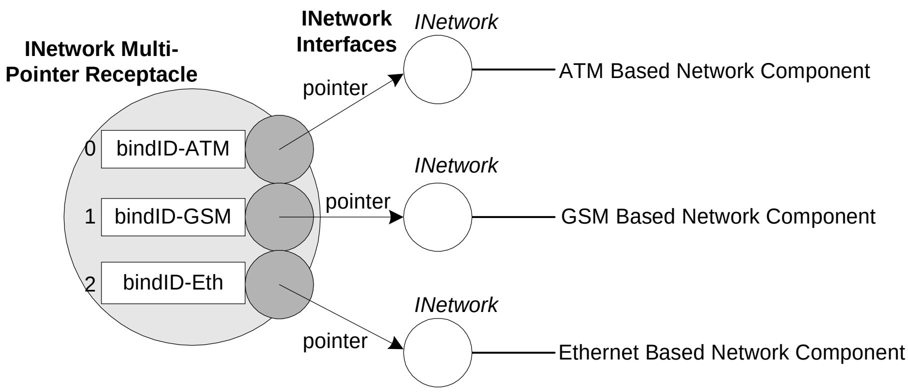

Figure 4.1: A Multi-Pointer Receptacle

The bindings discussed so far have been examples of *local bindings* that interconnect receptacles and interfaces that exist within the same address space and are usually implemented as a local procedure call. Local bindings do not have to be created directly by components involved in the binding but may instead be created by third parties that obtain references to interfaces owned by sink components and pass them to receptacles within source components. Binding non-collocated receptacles and interfaces requires the use of specialised binding components that present interfaces and receptacles corresponding to the receptacles and interfaces of the actual components to be bound. The components must locally bind to the binding component (which deploys proxies in both address spaces) in order to establish the communication channel. Figure 4.2 highlights the general organisation of components in terms of interfaces, receptacles and bindings.

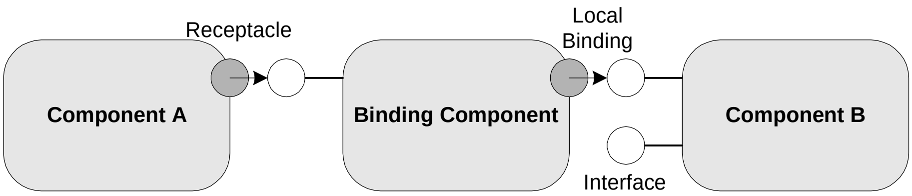

Figure 4.2: The General Organisation of Components within DEIMOS

4.2.1.1 Components

Components are used to encapsulate both system and application level activity in a uniform manner. This promotes a consistent development framework that assists application developers in designing their own system related components if a suitable one does not exist in the current pool. In addition, two particular component specialisations are required to fulfil specific roles within DEIMOS:

- *Binding components*, discussed above, provide communication services between the receptacles and interfaces of other components when they are not collocated. Different communication mechanisms are encapsulated in different binding components.

- *Wrapper components* give access to low level hardware resources in a similar way to link time extensible operating systems. It should be noted that wrapper components do not define policies over how the hardware is used; i.e. once components have successfully bound to a wrapper's interface, they may control the hardware as they wish. Wrapper components provide one of three types of service; *device access wrappers* offer a low level interface to the port and memory mapped IO address space of a device, *privileged instruction wrappers* allow applications to issue privileged instructions such as those for IO and halting the CPU, and *neutral service wrappers* encapsulate access to critical hardware resources. These are primarily the memory management hardware (managed by the Neutral Memory Manager - NMM, see section 6.3.1) and the interrupt vectors (managed by the Neutral Interrupt Vector Manager - NIVM, see section 6.5.1).

Granularity is an important issue in the design of components. Some related systems (e.g. [Oikawa,96]) provide special support for disjoint *policy functions* defined over their components to enable fine grain configuration. For example a large grained component may be configurable by specifying alternative fine grained policies such as scheduling functions in a component implementing a concurrency model or page replacement functions in a component implementing a virtual memory model. DEIMOS relies on the generality of its component model to efficiently support both large and fine grained components without distinction. In the preceding examples, the policies would be realised as 'small' DEIMOS components locally bound to the main scheduler and virtual memory components (see figure 4.3). Encapsulating policies in this way makes them subject to dynamic reconfiguration like any other component.

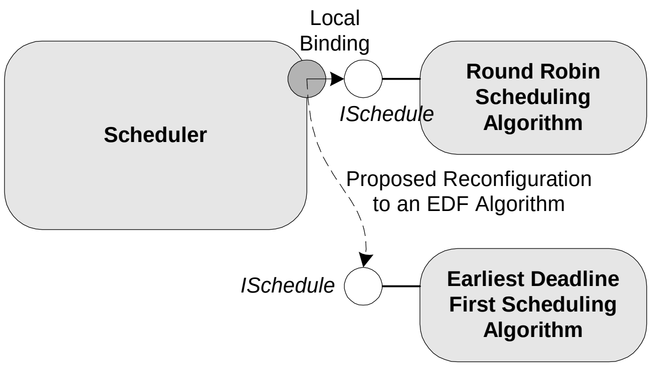

Figure 4.3: Implementing Efficient and Uniform Dynamically Selectable Policies Using Local Binding and 'Small' DEIMOS Components

4.2.2 Component Framework

DEIMOS is based around an abstract model that espouses encapsulation, interfaces and explicit binding in the form of components. In addition, these components should adhere to a strictly defined concrete framework that mandates certain attributes, features and functionality that a compliant instance should exhibit. The component framework should specify not only the design of a component, but also the way in which components interoperate. There are a number of reasons why a formal framework for component design is superior to an ad-hoc approach:

- *Frameworks provide guidelines for component designers*; systems that do not mandate a framework for the design of their components are often difficult to work with. Designers must rely on sample code and may be left unsure as to how implementations may deviate from these examples. Conversely, a framework provides a specification for component design that can be interpreted freely so long as the semantics are obeyed.

- *Frameworks allow a system to control dynamic component behaviour*; by requiring that components divulge information to the system in well-known ways (e.g. by implementing abstract interfaces), it is possible for the system to arbitrarily access this information and use it to control dynamic operations such as component creation and binding.

- *Frameworks specify a consistent pattern for component interoperability*; without a framework, components would interoperate in ad-hoc ways rendering those designed by one organisation incompatible with those from another.

4.2.2.1 Framework Background

There are many examples of component frameworks that already exist within the literature, e.g. Java Beans [Sun,97], COM [COM,95], XPCOM [Mozilla,99] and CORBA components [OMG,98]. Rather than deploy yet another, DEIMOS uses a suitably adapted version of one these frameworks, namely Microsoft's Component Object Model (COM). In addition, influences from a variant of the COM framework known as XPCOM (from the Mozilla organisation) are also taken into account. These frameworks are discussed in appendix A of this thesis.

At its most fundamental level, COM is a simple but highly efficient means of interconnecting language independent components and lends itself well to the performance intensive operating system domain. In addition, using an adapted version of a well-known component framework immediately opens up the development of third party components to a large pool of developers. Furthermore, it provides credibility for the project outside of the research community.

4.2.2.2 Deploying the Component Object Model within DEIMOS

Similarities between COM and the DEIMOS Framework

DEIMOS components support first class interfaces, the `IUnknown` interface (i.e. reference counting and interface querying), vtable based interface binding, multiple language component implementations (so long as the language can support pointers to functions) and dynamic component instantiation.

As with object oriented systems in general, DEIMOS supports the notion of templates defining the characteristics that instances created from the template will inherit. *Component classes* represent templates and *component instances* are created from a creation method defined within the template (see section 5.4 for more on the implementation of a component class). Component classes are loaded into memory via a loader and instances are created from them, i.e. once a class has been loaded into an address space it does not have to be loaded again (unless it is deleted). The loader has the responsibility of checking whether a component class has already been loaded into a specified address space before creating a component instance. If an instance is the first to be created or the last to be deleted then the corresponding component class will also be loaded and deleted respectively.

DEIMOS Specific Limitations over COM

Like XPCOM, DEIMOS does not implement the full COM framework. The reasons for this are as follows:

- *COM is continually growing to cover ever more diverse and esoteric application scenarios*; it would be impractical to implement a faithful port.

- *Only the core of COM functionality is necessary to produce a component framework*; the basic model used by COM (essentially what XPCOM uses) is sufficient to provide a complete component framework. Excess functionality complicates the model and reduces efficiency.

- *Certain COM programming idioms would complicate the component programming model*; DEIMOS specifies that certain techniques should not be used in developing components. For instance, aggregation and containment can lead to *split personality* components, i.e. components with multiple `IUnknown` pointers (though each one describes the component in the same way but through a different implementation). In DEIMOS, an `IUnknown` pointer (in conjunction with an address space) should uniquely identify a component at runtime, therefore containment and aggregation are prohibited.

Consequently, DEIMOS avoids aggregation, containment, dynamic dispatch and multiple instances of the same interface on a single component.

DEIMOS Specific Enhancements over COM

The main enhancements that DEIMOS offers over its COM based framework pertain to the need to dynamically reconfigure components after they have been instantiated, i.e. at runtime. Microsoft have already begun to consider such scenarios, e.g. their DirectShow API [Chaterjee,97] offers specialised component templates capable of handling dynamic reconfiguration. However, this only applies to typically stateless multimedia components, e.g. filters and compressors. DEIMOS borrows from this work and attempts to extend it into general purpose components including applications and system services.

The list of enhancements and alterations to the standard COM framework include:

- *Naming*; COM uses globally unique 128-bit identifiers for naming its interfaces and components. In DEIMOS, it is sufficient to use string names within a hierarchical naming space. This has the advantage that a registry is not needed and name uniqueness can still be assured by using stringified versions of unique identifiers (e.g. COM's GUIDs). Alternatively, name uniqueness can be assured by assigning namespace hierarchy branches through some central and trusted authority. By convention, interfaces are preceded by the letter I, so an interface X would be identified by the string "`IX`". A component class encodes the names of all its implemented interfaces within its implementation. Components are located by a platform dependent string (this usually translates to some hierarchically located, disk based file and requires that the instantiating party knows the components implementation location), but their actual class name is also encoded within their implementation. By convention, the component's object file location relates to its class name (i.e. a component with a class name of "`myClass`" would be located in a file at `/.../myClass.obj`). Additionally, when a component is instantiated it is also statically assigned an instance name so that it can be easily located later on,

- *Local binding*; COM uses simple variables to hold pointers (references) to collocated interfaces. Though this successfully implements local binding, the

COM runtime does not maintain information about the storage location of interface pointers when they are resolved (through `QueryInterface()`). Consequently, such *implicit* local bindings cannot be reconfigured later. DEIMOS uses the notion of a receptacle (a data structure that includes a simple pointer variable plus extra contextual information) to store information about a local binding so that the binding can be reconfigured later. This information is passed to the receptacle through an *explicit* local binding operation thus forcing the location of its stored interface pointers to be made known to the runtime. Receptacles are used to represent persistent bindings, i.e. they are declared (using special macros) within the member data area of the class implementing the component. This type of binding is used frequently (e.g. to an interface that the component must be bound before it can be used – a so-called *required* interface) and it would be inefficient to continually rebind the receptacle to the interface. In addition, DEIMOS uses the standard COM interface pointer to represent transient bindings, i.e. those used infrequently, typically within method implementations where they are implemented by local variables; each time the method is called, the variable is bound to a new or pre-existing instance of the required interface. The significance here is that receptacle based (explicit) bindings must be recorded by the runtime in case reconfiguration is required later, while simple interface pointers use standard COM (implicit) binding and do not have to be recorded so long as reconfiguration is not allowed while a method call is in progress,

- *Support for dynamic reconfiguration*; DEIMOS mandates that every component must implement two additional abstract interfaces; `IReceptacle` and `ILifeCycle` (see sections 5.4.1 and 5.4.2 for their definition). These interfaces contain methods that the runtime will invoke whenever a component is involved in a dynamic reconfiguration. `IReceptacle` contains methods (`connect()` and `disconnect()`) to be called whenever an explicit local binding is reconfigured, while `ILifeCycle` contains methods (`startup()` and `shutdown()`) to be called whenever a component is created or deleted. In the event of a dynamic reconfiguration, only the component programmer is capable of knowing what actions their component must take in order to satisfy any outstanding dependencies that may be affected by the reconfiguration. As an example of appropriate binding reconfiguration actions implemented by `IReceptacle`, consider a source component that wishes to deliver invocations to a sink via a thread. When the binding is established, the thread should be started and when the binding is deleted, the thread should be stopped from making further invocations. In addition to these binding specific actions, `IReceptacle` methods are also responsible for passing and deleting interface pointers from the appropriate receptacles within a component implementation. An example of a lifecycle action that a component may take (when being deleted) is the writing of persistent state to permanent storage. Other components may acquire this state in order to replace the original without disrupting the rest of the system. In addition, some components (e.g. factories) may create non-component objects (e.g. representing resources) that cannot be tracked by the runtime (as they do not adhere to the receptacles and explicit local binding model). If these types of object contain state that must be preserved during a reconfiguration then it is the responsibility of the programmer to ensure that this happens in their `ILifeCycle` implementations.

4.2.3 The DEIMOS Core

The core is comprised of a number of integral, platform dependent elements that allow it to fulfil its role as an enabler for the dynamic loading of componentised operating system services. A key principle in developing the core has been the minimisation of the amount of hardware initialisation that it performs compared to traditional operating systems. This allows more scope for the fundamental extensibility called for in the introduction to this chapter. The initialised core should leave the target platform in its most minimally prescriptive state in terms of execution environment; all further execution semantics, e.g. memory model, concurrency model and IPC should be dynamically introduced as componentised services. The following sub-sections describe the elements that comprise the core. Note that the order in which they are presented also represents the order in which they are initialised.

Several core elements are themselves components[7] (while the remainder are standard functional code sequences) and export interfaces for use by other members of the core and user defined components located outside of the core (all component interconnections with core interfaces must be established using explicit local binding for reasons discussed in section 5.3.2). These interfaces represent part of the design of the component (in terms of their signature, capabilities and platform dependencies) and are presented in Microsoft's dialect of the Interface Definition Language (MIDL) [COM,95]. In accordance with the requirement for operating systems to allow applications to exploit the peculiarities of specific platforms at the lowest level, the methods provided by the various core interfaces can vary from system to system. For this reason, each interface method is categorised depending on its portability between platforms:

- *Portable* services will appear consistently across platforms in terms of name, parameter signature and semantic.

- *Semi-portable* services will appear consistently across platforms in terms of name, parameter signature and semantic so long as the platform offers an underlying mechanism to support the services. Otherwise, the service will not be supported. For instance, many memory management methods regarding virtual memory are semi-portable; they require the target platform to implement virtual memory management hardware.

- *Non-portable* services are specific to a particular hardware platform and will not appear outside the context of that platform. They make use of facilities that are only implemented on the given hardware so that competent component designers can take advantage of them.

This chapter only details methods belonging to the portable category. Chapter 5 extends some of these interface definitions with semi-portable and platform specific methods over a reference implementation.

4.2.3.1 Bootstrap Sequence

Depending on the capabilities of the target platform, the bootstrap sequence is responsible for initialising the memory management unit and interrupt vectors and establishing a protected region of memory (assuming the hardware supports such protection) into which the remaining elements of the core can be loaded. The latter

---

[7] Typically though, they are statically selected when the core is compiled and linked and cannot be dynamically reconfigured, e.g. because of platform based initialisation constraints.

task ensures that the integrity of the core is protected against the actions of subsequently loaded components. In particular, these components should not be able to gain direct access to the wrapper components controlling sensitive hardware functionality.

4.2.3.2 Wrapper Components

Each DEIMOS core is built with a default set of wrapper components (configured during the build process) appropriate for the target platform. The purpose of a wrapper component is to protect sensitive device hardware from uncontrolled access by user defined components, consequently they cannot be reconfigured. This is achieved by locating each wrapper component and any associated hardware specific memory or port mapped registers and buffers within the core's protected memory region. A component must explicitly establish a binding with the wrapper's interface in order to access the protected hardware. This process is subject to the conditions imposed by the protection constraints associated with the wrappers interface. From their secure location within the core, wrapper components are able to verify that invocations from user defined components are made on legitimately established binding channels (cf. system call argument checks in traditional operating systems).

Device Access Wrapper Components

There are two options for the implementation of a device access wrapper components' interfaces. These pertain to the way in which a bound component is granted access to manipulate the wrapped devices hardware (each device wrapper implements the same style of interface, the difference between instances is that they protect different regions of port, register or buffer memory depending on the device);

- *indirect component access through wrapper interfaces*, where bound components call interface methods that wrap calls to processor level IO instructions (this assumes that the caller does not have the privilege to execute such instructions directly while the wrapper does, as it is situated in the core), and

- *direct component access through IO priority boosting*, where a bound component has its privilege boosted to a level where it is able to call a processor's IO instructions directly.

The former allows the production of more secure systems because control can be exercised over each sensitive IO invocation meaning that less trust is required in the intentions of components attempting to bind to such an interface. The latter is more efficient because IO calls are made directly rather than indirectly through interface methods (as in the former case). However, in order to guarantee the production of a stable system, the options for arbitrary component reconfiguration must be reduced in order to avoid having non-trusted components taking control of a device and hanging the system through malicious or accidental usage.

Direct component access depends entirely on the capabilities of the target platform and its interface is therefore described in a reference implementation detailed in chapters 5 and 6. Indirect access is implemented as follows:

```
interface IWrapperDeviceIndirect : IUnknown {

   [portable] HRESULT read([in] DEIMOS_Addr addr, [out] byte *pValue,

   [portable] HRESULT write([in] DEIMOS_Addr addr, [in] byte value);

};
```

Figure 4.4: `IWrapperDeviceIndirect()` Interface

`Read()` and `write()` encapsulate CPU operations for memory mapped IO, i.e. using regions of system memory to access the registers and buffers of devices mapped into those areas. Consequently, such a mode of interaction can use standard data movement instructions to transfer information to and from the device. As all CPUs support such operations (e.g. `move`, `store`, `load` at the assembly level), these methods are portable. `Read()` takes an address and returns the byte value at that address, whereas `write()` outputs a byte value into a given address.

Privileged Instruction and Neutral Services Wrapper Components

Both of these types of wrapper component are implicitly platform dependent and their interfaces are described in the context of a reference implementation presented in chapters 5 and 6.

4.2.3.3 Default Disk and Network Drivers

Components are dynamically loadable entities and therefore must exist on media from which they can be accessed by the system loader (see section 4.2.3.4). This implies access to a locally or remotely located (i.e. across a network) disk thus requiring the presence of default device drivers utilising the interfaces of the core's disk and network device access wrapper components. Depending on which default drivers are included (which in turn depends on the devices available), the loader can locate a component's object file (a number of object code formats are supported) and load its image into memory.

The default drivers are themselves components, though they are included in the core at build time and cannot be reconfigured. The loader establishes bindings with their interfaces when it initialises. However, these bindings may be reconfigured to point to replacement drivers loaded later. It is the responsibility of the reconfiguring agent to ensure that the replacement drivers implement the semantics expected by the system loader.

4.2.3.4 Default System Loader

The default system loader is a build time integrated component that depends upon the services of the default disk and network drivers (to which it binds at initialisation) in order to locate components that have been requested for execution. It cannot be reconfigured, but it can coexist with subsequently loaded components that implement alternative loading semantics. The default loader must always be present in order to field requests from components that exist outside the context of any particular execution environment (e.g. those loaded when the system has just finished initialising). Once a complete execution environment has been configured, components wishing to execute within it are loaded through a supplementary loader component that forms part of that execution environment.

The system loader offers a single interface to allow components to sub-load further components:

```
interface IDefaultLoader : IUnknown {

  [portable] HRESULT createComponent([in] const char *location,

                                     [in] const char *name,

                                     [out] IUnknown **ppIUnknown);

  [portable] HRESULT deleteComponent([in] const char *name);

}
```

Figure 4.5: **`IDefaultLoader()`** ***Interface***

`CreateComponent()` specifies a platform dependent `location` parameter for locating the associated component's object code. Once the loader can locate the component, it is loaded into the initial memory environment (if it is not already present) and an instance is immediately created at a free address range independently resolved by a neutral services wrapper component known as the Neutral Memory Manager (NMM - see section 6.3.1) to which the loader is bound at initialisation time. If a suitable free address range cannot be provided, an error code is returned. The supplied `name` parameter uniquely identifies the instance within the system. If an attempt to instantiate a component with the same name is made then the creation fails. Once a component instance has been created, its pointer to `IUnknown` (`ppIUnknown`) is returned to the caller.

The default loader only loads components into the initial memory environment (a flat, shared contiguous region of addresses - see section 6.3) so there is no need for its methods to delineate a particular address space to operate on. In order to load an external component into another segment or region of virtual memory (that has been instantiated since initialisation was completed), an extra loader must be configured that will obtain one of these additional address spaces from the NMM. The interface methods of such external loaders must take the presence of multiple address spaces into account when creating and deleting components by including additional parameters to identify them.

Finally, the loader is responsible for notifying the Configuration Manager (CM - see section 4.2.3.5) that it has successfully created a new component instance by calling the CM's `ICM_LoadNotify::notifyComponentCreate()` method. The default loader establishes a binding on this interface when it initialises and additional loader components must do likewise. Only *trusted* external loader components (i.e. those known to notify the CM of successful component creation and deletion operations) are able to establish such a binding.

`DeleteComponent()` notifies the CM that a component instance is about to be deleted by calling its `ICM_LoadNotify::notifyComponentDelete()` method. In addition, it is the responsibility of the loader to maintain a table of the components that it instantiates along with the regions of memory that they occupy in order to notify the NMM when the allocated regions may be freed for reuse. When the last instance is deleted, the component class can also be deleted from the address space.

4.2.3.5 Configuration Manager (CM)

The CM is a component that is built into every DEIMOS instance and is responsible for two key activities which, taken together, support the monitoring and control of a DEIMOS system in terms of its dynamically evolving component and binding topology:

- *The implementation of explicit local binding*; the CM fields all component requests for explicit local binding operations. Using this condition, it is ensured that all reconfigurable communication within DEIMOS is channelled through CM arbitrated bindings (recall that even binding components must themselves be bound through local bindings).

- *The policing of contention*; the fact that all explicit local bindings are established through the CM leads to an efficient method of policing the many and varied types of contention that are possible in a system that (initially) allows arbitrary dynamic configuration to take place. This involves the definition of suitable protection constraints over the receptacles and interfaces of system related components making it is possible to eliminate contention by refusing to allow the system to evolve in ways that would lead to it. The CM incorporates a protection constraint interpreter that evaluates the constraints of a receptacle and interface whenever a local binding request is issued over them.

Although it is a key component in the architecture, the CM is *not* a kernel in the traditional sense of the term (i.e. it does not contain memory management, concurrency and IPC mechanisms). In addition, it is possible for the CM itself to be unloaded from the system (as it is a standard component) if desired (e.g. if memory is in short supply). This will, however, result in a system that cannot be further reconfigured at run time.

During initialisation, the CM establishes the topology of the system in terms of components and bindings. This information is dynamically maintained using an *object graph* [Fitzpatrick,98] that is updated whenever a component is loaded, deleted or has a binding (re)configured. On the system graph, nodes represent components (including binding components), while arcs represent local bindings. The graph can be traversed programmatically by selecting a component and following its bindings to other components. This allows components to gain useful insights into the system structure before they request the loading, binding and reconfiguration of other components. Although component loading operations are implemented by loaders, a protocol exists to allow them to notify the CM of each successful load operation; any component implementing a loader must agree to implement this protocol by establishing a binding with the CM for this purpose.

Note that the system graph is a flat data structure and does not directly support the nesting of components and bindings into higher level composites. This stipulation is necessary to avoid having the CM lose track of global visibility through having components hidden within composite components. Disallowing composite components in the system graph does not, however, preclude the use of virtual hierarchical composition from the application point of view. To achieve virtual hierarchical composition, a logical grouping of components is represented as a single group component. The group component performs two functions: i) it requests the loading and binding of the components comprising the logical grouping, and ii) it provides a unified interface to the services provided by the logical grouping. With this arrangement, applications can load and interact with the logical group as if it were a single component while the CM retains global visibility of all components and bindings in the system.

The CM exports three interfaces, two of which are available to all components, while the third is only available to loader components. The general interface (described below) is available to everyone and allows components to find out each others instance names, class names and `IUnknown` pointer references:

```
interface ICMGeneral : IUnknown {
  [portable] HRESULT getComponentClassName([in] IUnknown *pIUnknown,
                                           [in] DEIMOS_AddressSpace addrID
                                                                  = UPS,
                                           [out] char *ppClassName[]);
  [portable] HRESULT getComponentName([in] IUnknown *pIUnknown,
                                      [in] DEIMOS_AddressSpace addrID = UPS,
```

```
                                  [out] char *ppName[]);
 [portable] HRESULT getComponentPIUnknown([in] const char *pName,
                                          [out] DEIMOS_AddressSpace*pAddrID,
                                          [out] IUnknown **ppIUnknown);
}
```

Figure 4.6: `ICMGeneral()` Interface

`GetComponentClassName()` returns the class name of a component instance based on its pointer to `IUnknown` and address space identifier (`addrID`). Initially, components only exist within the default memory environment (identified by the address space identifier `UPS` - see section 6.3). However, memory model components may instantiate new address spaces into which components may be loaded. As it is then possible that two components in different address spaces may have the same pointer to `IUnknown`, the identifier of the address space must also be supplied (or returned) to uniquely identify an instance when querying for information about it.

`GetComponentName()` returns a component's instance name based on its address space identifier (`addrID`) and pointer to `IUnknown`.

`GetComponentPIUnknown()` returns a component's address space identifier (`pAddrID`) and pointer to `IUnknown` given its instance name.

All components are bound, by default, to the CM's explicit local binding interface, thus making its services available to components from the moment they are created. This effectively provides bootstrapping for the system; if components did not have access to the binding interface, they could not bind to other core interfaces and, in particular, the inability to use the default system loader would render the system useless.

```
interface ICMLocalBinding : IUnknown {
  [portable] HRESULT createBinding([in] const char *source,
                                   [in] const char *sink,
                                   [in] const char *iid,
                                   [in] DEIMOS_ReceptacleID recpID = NULL);
                                   [out] DEIMOS_BindID *pBindID,
  [portable] HRESULT deleteBinding([in] DEIMOS_BindID bindID);
```

```
  [portable] HRESULT getBindingInfo([in] DEIMOS_BindID bindID,
                                    [out] DEIMOS_BindInfo **ppBindInfo);
  [portable] HRESULT freeBindingInfo([in] DEIMOS_BindInfo *pBindInfo);
  [portable] HRESULT enumBoundRecps([in] const char *sink,
                                    [in] const char *iid,
                                    [out] IEnumBoundRecps **ppIEnumBRecps);
  [portable] HRESULT enumBoundIntfs([in] const char *source,
                                    [out] IEnumBoundIntfs **ppIEnumBIntfs);
}
```

Figure 4.7: `ICMLocalBinding()` Interface

`CreateBinding()` allows a component to request the creation of a local binding between a receptacle on the source component of the same type as the indicated interface on the sink component. Either of these may be the requesting component itself. If the CM successfully verifies the protection constraints of the two parties then the source is notified that a binding has been established (through a CM initiated call to its `IReceptacle::connect()` method) and an identifier for the binding (`pBindID`) is returned to the caller. The optional `receptacleID` specifies an index within a multi-pointer receptacle where the source component should store its interface pointer upon establishment of a local binding. Finally, the system graph is updated to record the creation of the binding.

`DeleteBinding()` deletes the local binding specified by the `bindID`. The source is notified of this event (through a CM initiated call to its `IReceptacle::disconnect()` method) and has a chance to take appropriate loss of connection actions. The receptacle holding the interface pointer has it `NULL`ed in order to prevent erroneous invocations before the component instance (and possibly the component class) is actually freed from memory. Finally, the system graph is updated to record the deletion of the binding.

`GetBindingInfo()` allows a component to retrieve information about a local binding from its `bindID`. This information includes; the source and sink names, interface name and receptacle index identifier (for a multi-pointer receptacle, or `NULL` otherwise).

FreeBindingInfo() frees a binding information structure returned from a call to getBindingInfo().

EnumBoundRecps() returns a COM-style enumeration interface pointer (ppIEnumBRecps) representing all the bindIDs for local bindings that have been established on the named interface (iid) of the sink component. Enumeration interfaces have methods previous(), skip(), reset() and next() to navigate through the bindIDs in the item list.

EnumBoundIntfs() returns an enumeration interface pointer (ppIEnumBIntfs) representing all the bindID's for local bindings that the source component is currently providing receptacles for.

Using the last two methods, it is possible to navigate around the particular sub-graph in which a named component is located. In effect, these operations can be used as iterators for moving around the graph to dynamically establish the topology of the system. Reconfiguration requests can then be made based on this information.

The final CM interface enables components implementing loaders (including the default system loader) and concurrency models, to pass information to the CM regarding aspects of component execution that are outside the control of the CM, i.e. memory allocation and flows of execution. Loaders and concurrency models must be trusted to generate these notifications or else a stable DEIMOS system configuration cannot be guaranteed:

```
interface ICMNotification : IUnknown {
  [portable] HRESULT notifyComponentCreate([in] const char *loader,
                                           [in] const char *name,
                                           [in] const char *location
                                           [in] DEIMOS_AddressSpace addrID
                                                                 = UPS,
                                           [in] IUnknown *pIUnknown);
  [portable] HRESULT notifyComponentDelete([in] const char *loader,
                                           [in] const char *name);
  [portable] HRESULT notifyFlowCreate([in] const char *concurrency,
                                      [in] const char *name,
```

```
                                   [in] DEIMOS_FlowId flowID = NULL);
}
```

Figure 4.8: `ICMNotification()` Interface

`NotifyComponentCreate()` is called by a loader in order to let the CM know that a component has been created. The CM requires this information because it must perform two actions whenever a component is created;

- the component's *instance specific* and *class specific* protection constraints (see section 4.3.2.3) must be loaded before it can be allowed to execute. These are located using the `location` and `name` parameters passed to the loader during a `createComponent()` call,
- the system graph must be updated to represent the presence of the new component.

Note that the CM does not itself invoke a newly created component's `ILifeCycle::startup()` method. This is the responsibility of the component that created it and allows the new component to take startup actions appropriate for a dynamically changing environment (e.g. to pick up and absorb the state of a predecessor).

The components instance `name`, object code `location`, address space identifier (`addrID`) and pointer to `IUnknown` are passed to the CM. In addition, the loader component passes its own instance name (`loader`). The CM requires this in order to ensure that the component is deleted by the loader that created it; only this loader knows the precise region of memory (within the given address space) that the component image was created in. Also, there may be occasions when the CM must request the deletion of a component directly (e.g. if its protection constraints cannot be located), in which case the CM must know the name of the loader that created the component.

`NotifyComponentDelete()` passes notification to the CM that the `named` component is about to be deleted by the given `loader`. Assuming that the loader is the one that created the component (otherwise a panic is asserted), the CM then performs three actions based on this notification;