Trends in Operating System Design: Towards a Customisable Persistent Micro-Kernel

David Hulse and Alan Dearle

Department of Computing Science and Mathematics
University of Stirling
Stirling, FK9 4LA, Scotland

{dave, al}@cs.stir.ac.uk

Abstract

Monolithic- and micro-kernel-based operating systems such as Unix have failed to provide application developers with sufficient flexibility. They provide a host of inefficient and often inappropriate abstractions that prevent applications from accessing the hardware to exploit efficiency gains. These problems motivated the Grasshopper project to build a new operating system designed to explicitly support orthogonal persistence. Five years on, Grasshopper has demonstrated the feasibility of such an operating system although several problems have been identified. In light of this, we decided to redesign our kernel using modern techniques. This paper examines the trends in operating system design over the last few years and describes our plans to produce a new persistent micro-kernel.

Keywords: operating systems, micro-kernels, exo-kernels, persistence.

1. Introduction

During the past five years we have played a key role in the design and construction of the Grasshopper persistent operating system [17, 41]. As the implementation matured, we began to gain valuable experience in the use of the basic abstractions by porting various application systems [15, 18] that could benefit from orthogonal persistence. At the same time, development work was proceeding on the operating system kernel to experiment with various techniques for managing persistent data created by the application systems and by the kernel itself. This work revealed a number of problems with the current implementation of the Grasshopper system. These problems result in unnecessary inefficiency, duplication of system data structures, and unwanted complexity for application developers. In addition, the monolithic nature of the kernel has made it very difficult to maintain.

In light of the design and maintenance problems and inspired by recent research in the field of operating system architecture, we decided to redesign the Grasshopper kernel to take advantage of recent techniques. The main objective is to obtain a significant increase in performance while solving the maintenance problems through better modular decomposition. The goal is to build a flexible micro-kernel architecture that supports orthogonal persistence as well as general-purpose application systems.

In the remainder of this paper we briefly explain the rationale behind persistent operating systems and describe the original design of Grasshopper and the problems that we encountered. This is followed in Section 4 by a review of a number of approaches to operating system kernel design. In Section 5 we present our conclusions on kernel design and discuss our current position on the design of our new nano-kernel named *Charm*†.

† Originally, the working title of our new kernel was *quark*, a name that is also used to denote a class of six subatomic particles that, along with the electron, form the basic building blocks of matter. However, since the name *quark* has already been used by a company and several of its products, we elected to name the layers of the kernel architecture after the more interestingly named quarks. An excellent source of information about quarks and other subatomic particles can be found on the Web on the *Particle Adventure Home Page* at http://pdg.lbl.gov/cpep/adventure_home.html.

2. Persistent Operating Systems

A *persistent operating system* [19], is defined as an operating system designed expressly for the support of orthogonal persistence. The four principle requirements of such an operating system are as follows:

1. Support for persistent objects as a basic abstraction.
2. Persistent objects must be both stable and resilient.
3. Processes must be integrated with the object space in such a way that process state is contained within persistent objects.
4. Some form of protection must be provided to restrict access to persistent objects.

The need for persistent operating systems has arisen because the construction of persistent application systems on top of conventional operating systems such as Unix, Mach, or Windows NT is prone to inefficiency [19]. This inefficiency is due to a *semantic gap* between the fundamental abstractions required to support orthogonal persistence and the abstractions provided by the underlying operating system. To overcome this mismatch of abstractions, it is common for the semantic gap to be bridged through the use of an abstract machine that maps the abstractions required by the persistent application system onto those provided by the operating system. Naturally, the introduction of an extra layer of software in this manner can only have an adverse effect on performance.

There are numerous examples of operating systems that support persistence in one form or another. Examples include *Multics* [5, 13, 37], *MONADS* [26, 39, 40], *Eumul/L3* [28], *Clouds* [14], *Choices* [9], *KeyKOS* [8, 24], and *Grasshopper* [17, 41]. For the past five years, we have been involved in the design and implementation of the Grasshopper system, an overview of which is presented below.

3. Grasshopper

Grasshopper is an operating system explicitly designed to support orthogonal persistence. To this end, it provides three abstractions, *containers*, *loci* and *capabilities*, all of which are inherently persistent. *Containers* are the only storage abstraction provided by Grasshopper. They are conceptually very large address spaces capable of holding persistent data and are therefore suitable as coarse-grained persistent storage repositories. The data stored within a container may be directly referenced by a computation using an address relative to the start of the container. In contrast to address spaces in conventional operating systems, which are inextricably part of the *process* abstraction, containers can exist independently of computation. Thus, at certain times a container may be active and support multiple concurrent computations, while at other times it may be purely passive and void of activity.

The *locus* is the only abstraction over execution. Loci are scheduled by the kernel and execute within separate address spaces. The address space of a locus is composed from views of contiguous regions of various containers called *mappings*. In the simplest case, the address space of a locus contains a single mapping allowing it to access the contents of a particular container known as its *host*. Although each locus executes within a separate address space, a single container may host many loci allowing them to share the code and data stored within. Just as containers are not bound to any particular computation, the dual of this is true for loci. In other words, a locus is not bound to any particular container and may move to a new host container via the *invocation* mechanism [17, 41]. Thus, containers and loci are completely orthogonal abstractions.

Figure 1 shows three loci and three containers. Each container stores code and data accessible to the loci executing within it. *Locus 1* and *Locus 2* are executing within *Container 1*, which is their current host. Hence, their address spaces map directly onto that of *Container 1*. Similarly, *Container 3* is hosting the execution of *Locus 3*. In contrast, *Container 2* is not currently hosting any loci and is completely passive.

The Grasshopper kernel must be able to control access to abstractions such as containers and loci. Furthermore, a naming mechanism is required to identify a particular container or locus to the kernel when performing system calls. Grasshopper provides the *capability* abstraction for this purpose [16].

A capability is conceptually a typed reference to an instance of a kernel abstraction combined with a set of access rights. The rights determine the operations that the holder of the capability may perform on the referend. Capabilities may be held by both containers and loci and are stored in a list associated with the holder. Capability lists are maintained within the kernel to prevent forgery.

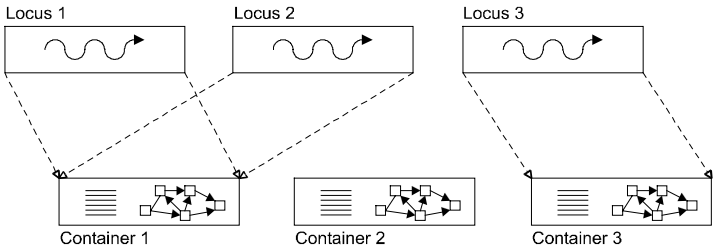

Figure 1: Pictorial representation of containers and loci.

3.1 Mapping

Mapping is a viewing mechanism used to compose address spaces from regions of other address spaces. In particular, mapping allows a contiguous region from one address space to be viewed from within a region of the same size in another address space. Changes made to data visible in either region are immediately reflected in both regions. Grasshopper provides two forms of mapping, *container mapping* and *locus mapping*. Container mapping allows the address space of a container to be composed from the address spaces of other containers. Since the address space of a locus is composed of regions mapped from the address spaces of various containers, any mappings affecting these containers will also be visible to the locus. For this reason, the effect of container mappings is said to be *global*. In contrast, locus mapping allows loci to install *private* mappings to container regions within their address spaces. These are typically used to provide access to per-locus data structures such as stacks. A more detailed account of the mapping mechanism and its possible uses is provided in [32].

3.2 Managing Persistent Data

From an external view of the Grasshopper system, it appears as if all data is stored within containers. In reality however, the actual data is maintained by *managers* [31]. A manager is a distinguished container holding code and data to support the transparent movement of data between primary and secondary storage. They are the only component of Grasshopper in which the distinction between long and short-lived data is apparent. To support the implementation of various different store architectures, managers have control over the virtual memory system in a similar manner to external pagers in micro-kernel operating systems such as Mach [1] and Chorus [12].

3.3 Problems with the Current Implementation

From our experience with the initial version of the Grasshopper system, we feel that we have made significant progress towards the goals we set out to achieve. However, through our porting efforts, a number of problem areas have been identified. Sections 3.3.1 through 3.3.5 describe the main problems and their origins.

3.3.1 Overhead of Page Fault Resolution

From past experience of building persistent object stores on top of Mach using the external pager interface [47, 48], it was decided that Grasshopper should support a similar mechanism to enable various store architectures to be tested without changing the kernel. Thus, the concept of a *manager* was developed to provide user-level control over the storage of persistent data. Since each container can have its own manager, many different store architectures can co-exist within the same system allowing each application to manage its persistent data in the most appropriate manner.

The problem with managers, and with external pager mechanisms in general, is that moving control over the virtual memory subsystem out of the kernel and into a user-level server adds to the

latency with which page faults can be serviced [38]. To illustrate this, consider the sequence of events required to service a page fault within a Grasshopper container.

1. Locus references a non-resident page, thereby inducing a trap into the kernel.
2. Kernel examines data structures to determine the container in which the fault occurred and hence the manager to notify.
3. Kernel upcalls into the manager.
4. Manager issues a system call to allocate a physical page.
5. Manager consults data structures to find the required page and issues a system call to read the page from disk into physical memory.
6. Manager issues a system call to bind the physical page into the address space of the faulting locus.
7. Manager returns control to the kernel.
8. Kernel resumes the faulting locus.

In the best case when the required page is already resident, it is necessary to call and return across the user-level/kernel boundary *three* times. Each time this occurs, certain registers must be saved and restored and capabilities must be checked to enforce protection. In the worst case when the page must be retrieved from disk, *five* boundary crosses are required. Although the cost of the disk I/O far outweighs the cost of the boundary crosses, it represents a significant overhead that could be better spent running other loci. In contrast, if page faults were serviced entirely within the kernel, the only boundary crosses would result from the initial traps, and since Grasshopper has a multi-threaded kernel, other loci could be scheduled during any necessary disk I/O.

3.3.2 Duplication of Information

Another source of inefficiency is the duplication of information within managers and the kernel. The most notable example of this involves the virtual address translation tables. Within the current implementation of the Grasshopper kernel, this information is effectively stored in three separate locations. Firstly, the information is held within the three-level hierarchical page tables (termed *contexts*) used by the memory management hardware. These contexts are typically sparsely populated making them expensive to maintain on a permanent basis. Therefore, a fixed-size pool of memory is used to cache the most recently used contexts.

Since contexts merely cache recently used address translations, a permanent record is maintained by the kernel using *local container descriptors* (LCDs) [30, 33]. An LCD represents a mapping from container addresses to physical addresses and contains an entry for every resident page of the with which it is associated. Each LCD stores address translations very concisely within an extensible hash table. During the resolution of page faults, managers enter address translations into the appropriate LCDs via a system call and the kernel uses this information to create corresponding context entries when required. It was originally intended that the presence of an LCD entry for a particular page would stop the kernel from notifying the manager to resolve future faults on the same page. However, this policy prevents managers from using page faults to implement transactions. Therefore, the current implementation passes all faults on a page through to the manager. In light of this, managers must now keep track of the current set of address translations. Although they can obtain this information using a system call to query the appropriate LCDs, it is more time-efficient if a separate hash table is maintained within the manager. This is crucial for implementing many of the common page replacement policies that operate by scanning the set of resident pages and examining their *dirty/referenced* status.

3.3.3 Context Switching Overhead

One of the fundamental aspects of Grasshopper is that a single container may host many loci simultaneously. When this occurs, the host container forms the basis for the address space of each

locus. However, since a locus can privately map regions from other containers into its address space, it is necessary for loci to have separate address spaces. Restating this in another way, Grasshopper does not provide a way for two loci to share the same address space other than by arranging for them to share the same set of mappings. Thus, when context switching from one locus to another, the virtual address space must also be switched to that of the new locus. This is unfortunate since loci were intended to be extremely lightweight processes somewhat akin to threads, although in this implementation they are forced to use a heavyweight context switch much like conventional processes.

3.3.4 Suitability of Abstractions

The basic abstractions supported by Grasshopper were designed to bridge the semantic gap between the abstractions offered by conventional operating systems and those required by persistent application systems. Judging from our experience in porting applications to Grasshopper, we have largely succeeded in this goal. However, the chosen abstractions are not entirely appropriate in all areas. For example, the interface to disk devices uses physical memory to buffer I/O data. Therefore, any application wishing to work with disks must also deal with the complexity of managing physical memory. On the positive side, this makes disk I/O relatively efficient since the device driver can simply perform DMA using the physical buffers thereby eliminating the need to copy data to or from the appropriate virtual address. The downside is that applications making use of disks pay for the efficiency through added management complexity.

Another area in which the Grasshopper abstractions are inappropriate is the management of virtual memory as touched upon in Section 3.3.2. The duplication of information occurs partly because of the inefficiency of accessing the required information using the kernel interfaces provided. By providing a more direct interface to the hardware and kernel data structures, much of the duplication could be eliminated.

3.3.5 Complexity of the Kernel

Despite our best intentions, the current implementation of the Grasshopper kernel exhibits a monolithic architecture. The only system components residing outwith the kernel are the managers responsible for handling persistent data. The current implementation of the kernel contains code for each of the basic abstractions, all of the device drivers, the network protocols, and the checkpoint and recovery mechanisms for the internal meta-data. On top of this, the kernel is also multi-threaded and must consequently observe strict mutual exclusion conditions as well as protect itself against deadlock. All of this amounts to some 54,000 lines of C code. Big projects such as this are extremely difficult to maintain. Every time the kernel is altered there is a significant risk of introducing new errors, most of which are extremely subtle and invariably involve race conditions. This creates a disincentive to extend the kernel for fear of corrupting a working system.

In building a monolithic kernel, we are, in effect, guilty of creating a system similar to those that motivated the Grasshopper project in the first place. A key criticism of those systems was their inflexibility which stemmed from forcing applications to use the abstractions provided. We have improved a little on this through the introduction of managers to allow user-level control over persistent data. However, the remaining abstractions such as *loci*, *containers*, and *capabilities* are all fixed and may not be the most appropriate building blocks for all classes of application.

4. Approaches to Operating System Design

Over the past few decades, a variety of approaches have been used in the construction of operating system kernels. The modern trend is towards smaller kernels in which the application programmer is provided with ever-increasing levels of flexibility. This has been brought about by the separation of mechanism and policy wherever possible and by providing simple abstractions which mimic the functionality of the hardware rather than high-level abstractions that are overly general. In Sections 4.1 through 4.4 we examine a number of kernel architectures and discuss their relative merits. From this review, we have drawn a number of conclusions on which we have based the design of our new kernel architecture. These conclusions and the resulting design are presented in Section 5.

4.1 Monolithic Architectures

Many regard the *monolithic* kernel design, characteristic of early Unix systems, to be the progenitor of modern operating system architecture. Its basic premise is to abstract over the machine hardware to provide high-level abstractions such as *processes* and *file systems* that enable application developers to build systems both efficiently and portably. Although high-level abstractions like this are suitable for many systems, there are also systems for which they are not appropriate [4, 6, 25, 43, 46]. This would be acceptable if the developers of these systems could simply ignore the abstractions that are unsuitable and create their own from scratch. However, since the operating system defines a high-level virtual machine, applications are effectively forced to use the abstractions provided.

The problem with high-level abstractions is that they are often inefficient and overly restrictive for many purposes. For example, the only way to create a new thread of control in most Unix systems is to create a new *process* using the *fork* system call. However, in addition to a new thread of control, a new address space is also created regardless of whether or not this is actually required. Therefore, using processes to obtain fine-grained parallel execution is often prohibitively expensive due to the overhead of context switching. To work around this particular problem, many developers have implemented thread-libraries on top of processes. Unfortunately, this does not entirely solve the problem since the operating system does not understand the semantics of threads. For example, a thread performing blocking I/O will cause the entire process to block whereas it would be more appropriate to schedule one of the other threads.

From the point of view of the kernel developer, a monolithic architecture can be very difficult to maintain due to numerous complex interactions between internal modules. Ideally, each module should present a well-defined interface through which all interactions with other modules occur. However, in practice this is extremely difficult to achieve making it all too easy for bugs to creep in unnoticed as the system evolves. Since all the kernel modules typically reside within the same address space, a bug in one of the modules can bring the entire system down.

4.2 Micro-kernel Architectures

The *micro-kernel* architecture was devised to solve some of the problems inherent with monolithic systems. The rationale behind its design is to separate mechanism from policy allowing many policy decisions to be made at user level. This goal is typically realised by building a kernel that supports only a small number of abstractions such as threads, address spaces, and inter-process communication. These abstractions can be composed to build user-level *servers* that implement common services such as file systems, processes, and virtual memory. Examples of micro-kernel architectures include Mach [1], Chorus [12], Amoeba [35, 45], V [10], Choices [9], Psyche [42], L3/L4 [28, 29], and Arena [34].

The most notable effect of the micro-kernel design is to separate the implementation of the system into smaller, more manageable pieces in the form of the kernel and the user-level servers. This arrangement greatly improves the modularity of the system and allows individual modules to be replaced if necessary. Because of the modular design, the reliability of the system is increased since faults are isolated to individual modules. For example, a file system crash would not bring down the entire system as it would in a monolithic architecture. Instead, the file system server can simply be restarted.

Micro-kernels are undoubtedly a step in the right direction towards addressing the problems with monolithic architectures. However, in solving some problems they have introduced others and have yet to demonstrate compelling advantages over monolithic designs. For instance, the introduction of user-level servers to implement high-level policy does not offer significantly more flexibility than monolithic kernels. This is because the server processes are typically crucial to the operation of the system and cannot be replaced without the necessary privileges. In addition, many micro-kernels do not give user-level servers ultimate control of the hardware. For example, in Mach, significant aspects of the virtual memory system are controlled within the kernel. At user-level it is possible to build external pagers which are capable of exploiting application specific knowledge. However, certain important decisions such as which pages to replace are made within the kernel. From our experience in designing the *manager* abstraction in Grasshopper, there is a trade off to be made when designing mechanisms such as external pagers. The design needs to provide sufficient flexibility at user-level

without compromising the security of the system. Therefore, systems such as Mach have tended to err on the side of security rather than allowing untrusted user-level code to make decisions that affect the entire system. Thus, we see many micro-kernels in which a significant amount of policy is retained within the kernel for security reasons.

Performance has also proven to be a problem in many micro-kernels due to the increased modularity. Since much of the functionality of systems such as Mach resides outwith the kernel, there is an increase in inter-process communication and interactions with the kernel compared to a monolithic design. As Anderson notes [3], crossing from one protection domain to another is a relatively expensive operation that is not getting faster as the speed of machines increase. Therefore, IPC and system call performance can be a serious bottleneck in many micro-kernels. By way of opposition, Liedtke claims that the inefficiency of micro-kernels and their abstractions is largely to do with poor implementation [27, 29]. His own micro-kernels, L3 and L4 in particular, have demonstrated that it is possible to build extremely efficient thread and IPC mechanisms through clever use of the hardware provided.

Overall, micro-kernels are still quite large and provide relatively high-level abstractions when compared to the hardware. These abstractions are often inefficient yet applications are forced to use them. Unfortunately, this prevents applications from accessing the hardware to exploit efficiency gains. Sometimes it is possible to implement something very simply and efficiently at the hardware level but is overly complex if built on top of higher-level abstractions. Therefore, providing applications with more control over the hardware is the key to flexibility. Techniques such as *scheduler activations* [2] have made significant inroads here although only in isolated areas.

4.3 Library Operating Systems

Due to the inadequacy of conventional operating systems, including those based on micro-kernels, a new style of operating system has recently come to light. The goal of these new systems is to allow all policy decisions to be made at user-level. Kernels should contain only those mechanisms that must be implemented securely. The majority of traditional operating system functionality is implemented at user-level and can be changed on a per-application basis. Applications that do not require custom made facilities are simply linked with standard libraries implementing common operating system interfaces such as POSIX. Tailoring the operating system to exploit application-specific knowledge is simply a matter of linking against a different library. The result is a system in which each application may potentially be running a completely different operating system. Since all of the typical services and abstractions are implemented within user-level libraries, the term *library operating system* has emerged to describe systems of this nature. Two examples of systems supporting the library operating system concept are *Aegis* [21-23] and the *Cache Kernel* [11]. The approaches taken by these systems are quite different and will be described separately.

4.3.1 The Exo-Kernel Approach

The philosophy behind *exo-kernels*, as typified by Aegis, is that abstracting over the hardware resources is the wrong approach for an operating system to take. No matter what abstractions an operating system provides, they will always be inappropriate for some class of applications. Therefore, rather than provide a host of abstractions, an exo-kernel aims to export the hardware resources directly, its only responsibility being the secure multiplexing of resources. For example, the resources exported by Aegis include physical memory (divided into pages), the CPU (divided into time slices), disk storage (divided into blocks), TLB entries, and interrupt/trap events. Thus, in contrast to monolithic and micro-kernels, the interface to an exo-kernel is non-portable. However, this is not seen as a problem since portability can be achieved by providing a standard set of libraries that implement portable interfaces such as POSIX. The only difference between this arrangement and that of a monolithic design is that the portability boundary no longer coincides with the kernel interface. The advantage of this is that specialised applications such as database systems, language run-time modules, and garbage collectors can replace the standard interface and work directly with the hardware to achieve optimal performance. These applications will not be portable although this is to be expected once hardware optimisations are exploited.

Allowing the functionality of the operating system to be tailored on a per-application basis, is exactly what developers have been asking for. The only issue is how to expose the hardware resources in a secure and efficient manner to make the provision of such flexibility feasible. In Aegis, security is controlled using *secure bindings* implemented by associating each resource with a password capability [35] that determines the rights of its owner. Every time a resource is used, Aegis checks the rights encoded within the associated capability. Previously allocated resources are reclaimed using a *visible revocation* protocol in which applications are notified of a resource shortfall and may make their own decisions as to which resources to relinquish. To protect itself against recalcitrant applications that either refuse or take too long to comply with requests, Aegis also employs an *abort* protocol in which resources are forcibly reclaimed.

One of the effects of providing such a low-level kernel interface is that the frequency of system calls is increased. Thus, implementing higher level abstractions such as threads or address spaces will incur the overhead of several system calls per higher level operation. This overhead is not present in monolithic- or micro-kernel-based systems since all of the abstractions requiring access to the hardware are implemented within the kernel. As noted in [23], one way to overcome this problem is to allow code implementing higher-level operations to be downloaded into the kernel. Using this technique, performance critical operations such as thread scheduling or interrupt handling can execute in supervisor mode and avoid the overhead of system calls. However, it must be possible to prevent downloaded code from corrupting the operating system. More is said about this subject in Section 4.4 below.

4.3.2 The Cache Kernel Approach

Rather than exporting an interface to the hardware as do exo-kernels, the Cache Kernel supports a small number of primitive abstractions for which all policy decisions are implemented by library operating systems at user-level. The interface to the Cache Kernel supports three types of object: *address spaces*, *threads*, and *kernels*. Each object is represented by a descriptor that is managed by the application (library operating system) that created it. Descriptors contain the state or meta-data required to realise an object. For example, a thread descriptor includes the saved register values and a stack pointer for use when handling exceptions. Applications may load the descriptor for an object into the kernel where it is cached. During this time, the kernel maintains the descriptor and executes the performance-critical actions supported by the objects. Like most caches, the insertion of a new entry into the cache may displace an existing entry. Displaced object descriptors are returned to the applications that own them.

To illustrate the operation of the Cache Kernel consider how an application might control the scheduling policy of its threads. Applications load thread descriptors into the kernel when their respective scheduling policies determine that the threads are eligible to run. The Cache Kernel will schedule and dispatch the set of cached thread objects in round robin fashion, effectively sharing the CPU resources among the active threads. When a cached thread blocks, its descriptor is unloaded from the kernel and returned to the application that owns it.

According to Cheriton [11], the approach taken by the Cache Kernel offers three benefits. First, the low-level caching interface allows applications to control resource management according to any desired policy. Second, the caching approach prevents applications from exhausting the supply of object descriptors as may occur in conventional operating systems. Finally, the cache model greatly reduces the complexity of the kernel compared to previous micro-kernel designs.

4.4 Customisable Kernels

A competing approach to the library operating system paradigm is the idea of providing a minimal kernel that may be dynamically specialised to safely meet the performance and functionality requirements of applications. Examples of systems which typify this approach are SPIN [6, 7] and Aegis [23]. The motivation for these *customisable* or *extensible* kernels is the same as for library operating systems. In fact, the two approaches overlap to some extent as illustrated by Aegis.

The SPIN system consists of a set of core system services such as memory management and scheduling, and a set of extension services that enable developers to customise the behaviour of the

operating system. Customisation is achieved by loading extensions into the kernel where they are integrated with the existing infrastructure. The security of loaded extensions is guaranteed by requiring them to be written in Modula-3 [36]. The strict modularity enforced by the language prevents extensions from accessing memory directly or executing privileged instructions unless explicit access is provided through a module interface. Loaded modules execute in response to system events such as exceptions or system calls. All events are declared within Modula-3 interfaces and can be dispatched with the overhead of a procedure call.

Whereas SPIN relies on the use of a type-safe, modular language to protect the operating system from untrusted extensions, Aegis takes a different approach. Potentially unsafe extensions are downloaded into the kernel where they execute within separate logical protection domains. Downloaded code is inspected [20] to ensure that it does not contain any privileged instructions that could compromise the safety of the protection domain boundaries. The boundaries themselves are simulated using *sandboxing* [49] to restrict the domain of indirect memory operations. Using these techniques, downloaded code is prevented from interfering with the operation of the kernel.

5. Towards a Persistent Micro-Kernel: The Nano-Kernel Layer

The failure of monolithic and micro-kernel operating systems to provide sufficient flexibility coupled with their poor performance was one of the motivating factors behind Grasshopper. However, in building a system to support orthogonal persistence we have followed a similar path to that of the systems that had originally failed us. Because of this and the maintenance problems with Grasshopper, we elected to design a new system based on the library operating system approach. These systems allow policy to be tailored to given hardware platforms to exploit efficiency gains. Furthermore, policies can be tailored on a per-application basis allowing radically different systems to coexist and make the best use of the hardware resources. Finally, the overhead of interrupts, TLB faults, traps, and context switches can be drastically reduced because of the simplicity and efficiency of the primitive operations [23].

The first step in the design of our new operating system has been to construct a minimal *nano-kernel* layer, called *Charm*, which exports an interface to the hardware. This has been written in C++ to encourage better code modularity and information hiding. The interface to the nano-kernel is currently expressed as a C++ class. It currently contains primitives for allocating physical memory, constructing address translation mappings, masking interrupts, reflecting exceptions and interrupts, and context switching. A prototype of Charm has been fully implemented on a Pentium/133 PC system and is approximately 35 KB in size. To test the interface, we have constructed a small micro-kernel layer that implements threads and address spaces on top of the primitives supplied by Charm. We have also implemented a number of device drivers using code from the freely available Linux distribution. The source code for Charm will be placed on our Website [44] for examination by interested parties.

One problem with our initial prototype of Charm is that hardware resources are not exposed as fully as they might otherwise be. The interface provides a number of low-level abstractions that are tailored to the Pentium processor. For example, context switching is supported by an abstraction called a *CPUContext* that enables multiplexing of the register set. Address translation is supported by the *MMU* abstraction which can be used to install address spaces represented by the *VMMap* abstraction. Abstractions of this nature are still extremely efficient since they correspond directly to the hardware resources available on the Pentium. However, it is necessary to modify the abstractions to allow higher level code to have more control over resource allocation. For example, the physical memory allocation primitives do not currently allow the allocation of specific pages. This would prevent systems with physically indexed direct-mapped caches from avoiding unnecessary cache conflicts.

Once we are satisfied with the Charm interface, our next step is to begin the design of the persistent micro-kernel layer, called *Strange*, which will be built on top of Charm. However, before embarking on this task, it is necessary to decide where the user-level/supervisor boundary will lie. One possibility is to implement a small protection layer on top of Charm and place the supervisor boundary at the Charm interface. This would result in an exo-kernel style system in which the micro-kernel layer would become a library operating system. Another possibility is to place the supervisor boundary at the interface to the micro-kernel layer and retain Charm in its existing form. This would produce a

more traditional micro-kernel style system that would not be as flexible as the previous alternative. Using our test micro-kernel, we plan to experiment with these approaches although it is likely that we will choose the exo-kernel route to maximise flexibility.

6. Conclusion

The Grasshopper project was started in 1992 to build an operating system with explicit support for orthogonal persistence. This work was motivated by the poor support for persistence offered by conventional operating systems. Five years later, the resulting system has demonstrated the feasibility of constructing such an operating system, although several problems have been identified with the current implementation. These problems have led us to begin work on the design of a new kernel.

Having followed the trends in operating system design over the years, it is apparent that more and more functionality, traditionally found within the kernel, is being migrated to user-level. At the same time, kernels are exporting increasingly lower-level interfaces. The rationale behind this is to provide developers with systems in which policy can be tailored on a per-application basis using library operating systems implemented at user-level.

Inspired by the recent developments in kernel design, we have begun work on a new kernel based on the library operating system approach. To date we have designed and implemented a minimal nano-kernel layer called *Charm*. We are in the process of designing a micro-kernel layer called *Strange* that will interface with Charm. Our goal is to build a library operating system that will support the development of persistent systems and replace the ageing Grasshopper kernel.

References

[1] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. "Mach: A New Kernel Foundation for Unix Development", in *Proceedings of the Summer Usenix Conference*, pp. 93-112, 1986.

[2] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *ACM Transactions on Computer Systems*, **10**(1), pp. 53-79, 1992.

[3] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. "The Interaction of Architecture and Operating System Design", in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 108-120, 1991.

[4] A.W. Appel and K. Li. "Virtual Memory Primitives for User Programs", in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, pp. 96-107, 1991.

[5] A. Bensoussan, C.T. Clingen, and R.C. Daley, "The Multics Virtual Memory: Concepts and Design", *Communications of the ACM*, **15**(5), pp. 308-318, 1972.

[6] B.N. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Sirer, "SPIN - An Extensible Micro-Kernel for Application-Specific Operating System Services", *Technical Report 94-03-03*, University of Washington, 1994.

[7] B.N. Bershad, S. Savage, P. Pardyak, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. "Extensibility, Safety, and Performance in the SPIN Operating System", in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, Colorado, 1995.

[8] A.C. Bomberger, N. Hardy, A.P. Frantz, C.R. Landau, W.S. Frantz, J.S. Shapiro, and A.C. Hardy. "The KeyKOS Nanokernel Architecture", in *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures*, 1992.

[9] R.H. Campbell, G.M. Johnston, and V.F. Russo, "CHOICES (Class Hierarchical Open Interface for Custom Embedded Systems)", *Operating Systems Review*, **21**(3), pp. 9-17, 1987.

[10] D.R. Cheriton, "The V Distributed System", *Communications of the ACM*, **31**(3), pp. 314-333, 1988.

[11] D.R. Cheriton and K.J. Duda. "A Caching Model of Operating System Kernel Functionality", in *Proceedings of the First Symposium on Operating System Design and Implementation*, Monterey, California, pp. 179-194, 1994.

[12] Chorus-Systems, "Overview of the CHORUS Distributed Operating Systems", *Computer Systems*, **2**(4), 1990.

[13] F.J. Corbató and V.A. Vyssotsky. "Introduction and Overview of the Multics System", in *Proceedings of AFIPS FJCC*, pp. 619-628, 1965.

[14] P. Dasgupta, R.C. Chen, *et al.*, "The Design and Implementation of the Clouds Distributed Operating System", *Computing Systems*, **3**(1), pp. 11-46, 1990.

[15] A. Dearle. "Persistent Servers + Ephemeral Clients = User Mobility", in *Proceedings of The Second International Workshop on Persistence and Java*, to appear, 1997.

[16] A. Dearle, R. di Bona, J. Farrow, F. Henskens, D. Hulse, A. Lindström, S. Norris, J. Rosenberg, and F. Vaughan. "Protection in the Grasshopper Operating System", in *Proceedings of the Sixth International Workshop on Persistent Object Systems*, Tarascon, France, pp. 60-78, 1994.

[17] A. Dearle, R. di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, and F. Vaughan, "Grasshopper: An Orthogonally Persistent Operating System", *Computer Systems*, **Summer**, pp. 289-312, 1994.

[18] A. Dearle, D. Hulse, and A. Farkas. "Java on Grasshopper", in *Proceedings of the First International Workshop on Persistence and Java*, Drymen, Scotland, 1996.

[19] A. Dearle, J. Rosenberg, F. Henskens, F. Vaughan, and K. Maciunas. "An Examination of Operating System Support for Persistent Object Systems", in *Proceedings of the 25th Hawaii International Conference on System Sciences*, Poipu Beach, Kauaii, pp. 779-789, 1992.

[20] P. Deutsch and C.A. Grant, "A Flexible Measurement Tool for Software Systems", *Information Processing*, **71**, 1971.

[21] D.R. Engler and M.F. Kaashoek. "Exterminate All Operating System Abstractions", in *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, 1995.

[22] D.R. Engler, M.F. Kaashoek, and J.W.O.T. Jr. "The Operating System as a Secure Programmable Machine", in *Proceedings of the Sixth SIGOPS European Workshop*, Germany, pp. 62-67, 1994.

[23] D.R. Engler, M.F. Kaashoek, and J.W.O.T. Jr. "Exokernel: An Operating System Architecture for Application-Level Resource Management", in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, Colorado, 1995.

[24] N. Hardy, "The KeyKOS Architecture", *Operating Systems Review*, (September), 1985.

[25] K. Harty and D.R. Cheriton. "Application-Controlled Physical Memory Using External Page-Cache Management", in *Proceedings of the Fifth Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 187-199, 1992.

[26] J.L. Keedy. "Support for Objects in the MONADS Architecture", in *Proceedings of the Third International Workshop on Persistent Object Systems*, Newcastle, Australia, pp. 392-406, 1989.

[27] J. Liedtke. "Improving IPC by Kernel Design", in *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Asheville, North Carolina, pp. 175-188, 1993.

[28] J. Liedtke. "A Persistent System in Real Use: Experiences of the First 13 Years", in *Proceedings of the Third International Workshop on Object-Orientation in Operating Systems*, Asheville, North Carolina, pp. 2-11, 1993.

[29] J. Liedtke. "On μ-Kernel Construction", in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, Copper Mountain, Colorado, 1995.

[30] A. Lindström, A. Dearle, R. di Bona, J. Farrow, F. Henskens, J. Rosenberg, and F. Vaughan, "A Model For User-Level Memory Management in a Distributed, Persistent Environment", *Technical Report GH-07*, Universities of Adelaide and Sydney, Australia, 1993.

[31] A. Lindström, A. Dearle, R. di Bona, J. Rosenberg, and F. Vaughan, "User-Level Management of Persistent Data in the Grasshopper Operating System", *Technical Report GH-08*, Universities of Sydney and Adelaide, Australia, 1994.

[32] A. Lindström, J. Rosenberg, and A. Dearle. "The Grand Unified Theory of Address Spaces", in *Proceedings of Hot Topics in Operating Systems (HOTOS-V)*, Orcas Island, USA, pp. 66-71, 1995.

[33] A.G. Lindström, "User-Level Memory Management and Kernel Persistence in the Grasshopper Operating System", *Ph.D. Thesis*, Basser Department of Computer Science, University of Sydney, Australia, 1996.

[34] K.R. Mayes, "Trends in Operating Systems Towards Dynamic User-level Policy Provision", *Technical Report UMCS-93-9-1*, Department of Computer Science, University of Manchester, Manchester, 1993.

[35] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba - A Distributed Operating System for the 1990s", *IEEE Computer*, **23**, pp. 44-53, 1990.

[36] G. Nelson, "System Programming in Modula-3", Prentice Hall, 1991.

[37] E.I. Organick, "The Multics System: An Examination of its Structure", MIT Press, 1972.

[38] D.K. Raila, S.-M. Tan, and R.H. Campbell, "Remote Procedure Call Implementations of Micro-Kernel Virtual Memory Services Degrade System Performance", *Technical Report* , Department of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[39] J. Rosenberg. "The MONADS Architecture: A Layered View", in *Proceedings of the Fourth International Workshop on ersistent Object Systems*, Martha's Vineyard, Massachusetts, pp. 215-225, 1990.

[40] J. Rosenberg and D. Abramson. "MONADS-PC - A Capability-Based Workstation to Support Software Engineering", in *Proceedings of the 18th Annual Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

[41] J. Rosenberg, A. Dearle, D. Hulse, A. Lindström, and S. Norris, "Operating System Support for Persistent and Recoverable Computations", *Communications of the ACM*, **39**(9), pp. 62-69, 1996.

[42] M.L. Scott, T.J. LeBlanc, and B.D. Marsh. "Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System", in *Proceedings of the 1988 International Conference on Parallel Processing*, pp. 255-262, 1988.

[43] M. Stonebraker, "Operating System Support for Database Management", *Communications of the ACM*, **24**(7), pp. 412-418, 1981.

[44] Systems Group, *http://www.cs.stir.ac.uk/os-research/* , University of Stirling, Stirling.

[45] A.S. Tanenbaum, R. van Renesse, H. van Staveren, G.J. Sharp, S.J. Mullender, J. Janson, and G. van Rossum, "Experiences with the AMOEBA Distributed Operating System", *Communications of the ACM*, **33**(12), pp. 46-63, 1990.

[46] C.A. Thekkath and H.M. Levy. "Hardware and Software Support for Efficient Exception Handling", in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 110-121, 1994.

[47] F. Vaughan, T.L. Basso, A. Dearle, C. Marlin, and C. Barter, "Casper: A Cached Architecture Supporting Persistence", *Computing Systems*, **5**(3), pp. 337-359, 1992.

[48] F. Vaughan, T. Schunke, B. Koch, A. Dearle, C. Marlin, and C. Barter. "A Persistent Distributed Architecture Supported by the Mach Operating System", in *Proceedings of the First USENIX Conference on the Mach Operating System*, pp. 123-140, 1990.

[49] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. "Efficient Software-Based Fault Isolation", in *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, Asheville, North Carolina, pp. 203-216, 1993.