To appear in the Proceedings of OSSurveyF98, UC San Diego, November 1998

Practice and Technique in Extensible Operating Systems

Lee Carver, Ying-Hung Chen, Theodore Reyes
leeca@pnambic.com, ying@ucsd.edu, treyes@qualcomm.com

Abstract

High performance applications require low overhead operating systems. Conventional operating systems can impose severe overhead on database managers, web servers, and other applications that use the system resources in unexpected manners. A major cause of this overhead is the operating system's abstract model of system resources.

Extensible operating systems allow applications to safely bypass these abstractions and avoid unnecessary overhead. Extensible operating systems use downloadable code (grafts) and user-mode libraries to grant applications fine-grained control over system resources. Applications can introduce new, application specific features into the operating system.

Operating systems are required to enforce the safe sharing of system resources. User-mode libraries are an inherently safe extension mechanism. Grafts use a number of protection schemes to ensure safety. This paper reviews the operating system extension techniques used by the SPIN, VINO, Fox, Kea, Exokernel, and Protected Shared Library projects.

Extensible operating systems have a clear advantage to server applications. Their intense resource usage makes it reasonable to spend time and effort improving performance. It is less clear that extensible operating systems are a good model for general purpose computers. Safe interaction among varied applications is very important in this environment. In addition, most applications are not a heavy resource drain on modern computers.

1 Introduction

Operating systems are a necessary evil. By their definition, they have a monopoly on system resources. Applications, the real users of system resources, must cope with the overhead of an operating system when they acquire system resources.

Since applications are judged on their performance, operating systems with low overhead are attractive. Current operating system design can impose severe overhead on applications that use the system resources in unexpected manners.

As computers become more pervasive and new applications are created, unexpected resource usage becomes more and more likely. Extensible operating systems allow new applications to safely bypass unnecessary overhead.

The necessary part of operating system is that system resources are shared. The disk, network card, memory, and processor are shared among the applications. Safe secure sharing of resources is a fundamental responsibility of the operating system.

2 Overview

An operating system defines how an application communicates with the hardware. Very often, the operating system uses sophisticated resource management techniques to provide general purpose, highly flexible, and highly available access to computer resources. This rich set of features makes it easy to develop many common applications.

However, this rich interface is inadequate or inappropriate for some important applications. Database managers and web servers have highly stylized patterns of resource usage. Ignoring these patterns of use can result in inefficient resource management [Sto91]. This is an important problem since these applications have significant, long-term demand for the system resources.

Achieving acceptable performance for these applications requires fine-grained control over system resources. The standard API cannot be used since it abstracts, or hides, information that is important for effective resource use. Extensible operating systems aim to improve resource management by allowing applications to introduce new, application specific features into the operating system. Improved resource management is especially important for new applications like web servers and multi-media.

2.1 Operating System Structure

Operating systems provide two distinct services to applications: an abstract model of the system resources

and the safe sharing of system resource among applications.

A common way to implement both the abstraction model and resource sharing is for each application to execute in a virtual machine. The operating system creates a virtual machine for each process and ensures that the virtual machines do not interfere with each other. Each application accesses a standardized system that typically includes processor sharing, memory sharing, disk sharing, synchronization, file system, and window management. Abstracting system resources has simplified many applications and made it possible to readily port applications to different computers.

In order to guarantee the safe sharing of resources, operating systems use hardware mechanisms to create multiple protection domains [Lam71]. Typically, all system resources are owned by the kernel, a protection domain that runs in a special system mode. Each virtual machine is a separate protection domain that runs in user mode. The kernel is responsible for all system resources and all interactions among protection domains.

There are two common models for kernel design. The traditional monolithic design exports the operating system API directly from the kernel. All of the operating system is implemented to run in the kernel. Micro-kernel operating systems are designed to place only the essential functions in the kernel [HHL97]. The rest of the virtual machine API is implemented in user mode "subsystems". Micro-kernels support flexibility in the operating system through the development of new subsystems.

In practice, the high cost of protection domain transfers has forced micro-kernels to offer only coarse-grained access to system resources. Early micro-kernel operating systems had protection domain transfers on the order of 2000 secs [BC94]. This overhead is unacceptable for frequent access to fine-grained resource management and has encouraged research into safe techniques for application management of resources without a protection domain transfer. Avoiding or reducing the time for domain transfers has been a focus of recent operating system research.

2.2 Extension Techniques

All operating systems provide some mechanisms to extend the built-in features. Installable device drivers are a form of downloading code into the operating system. Due to its inherent risks, downloading code is generally restricted.

An inherently safe way to extend the virtual machine is to place the extensions outside the kernel. The C runtime library is an example of this. Extension libraries are generally easy to develop. User mode programming is easier – it takes no special permissions, it uses conventional tools, and has less need to be bug free. In addition, there are lots of user mode programmers.

There are two problems with this approach. Some extensions require access to and control of information that is not available through the system interface. When this information is revealed through fine-grained APIs, the high cost of changing protection domains can be prohibitive.

An alternative approach is to place code for the extension within the protection domain of the kernel. The terminology for this is varied. We refer to the code as a "graft" that is "co-located" with the kernel. The process of installing the graft is called downloading.

Grafts avoid the need to change protection domains when application specific code is executed. The problem with grafts is that all code in the kernel protection domain has arbitrary access to the system resources. Downloaded code can misbehave in a variety of ways, violating the operating system's non-interference guarantees [Lam71, SES96]:

- Illegal data access
- Resource hoarding
- Unauthorized interface usage
- Antisocial behavior
- Denial of service

In order to prevent these misbehaviors, grafts must be tightly controlled. The simplest control is to limit permission for installing a graft. Other mechanisms to ensure that grafts are well behaved include type-safe compilation (Modula-3), code translation (interpreters, code generation), code inspection (software fault isolation, proof carrying code), and other limits to a graft's interaction with the kernel.

There are other approaches to improving the resource management performed by the operating system. Some operating systems provide special interfaces to allow an application to select among several resource management strategies or to provide hints to guide in resource management. While these can be very useful, they are inherently limited to pre-conceived concepts of flexibility.

2.3 Organization

The rest of this paper presents a survey of recent research in extensible operating system design. Sections 3 through 6 cover projects that use grafts to extend the operating system: SPIN, VINO, Fox, and Kea. Sections 7 and 8 review the Exokernel and Protected Shared Library projects. These projects rely on user level libraries for most extensions. For each project we review the design of the extension system, discuss its safety characteristics, and summarize the design. Where possible, we also cover any significant applications and the performance of the operating system. Section 8 presents our conclusions.

3 SPIN

The SPIN operating system [BSP95, FB96, PB96] is an extensible kernel that relies on a type checking to provide safety. All kernel operations are modeled as event handlers. Kernel grafts can register new handlers for existing events and introduce new events at run time. SPIN has demonstrated good performance and easy extensibility.

In SPIN, grafts are called extensions. Grafts are written in Modula-3, and the compiler is used to enforce restrictions on graft access to other kernel structures.

3.1 Design & Implementation

SPIN uses *events* to bind together the different components within the kernel. One component can announce a change to the system by *raising* an event. Other components provide *handlers* that are executed when the event is raised. Handlers register with an event, and are dispatched when the event is raised.

When a graft is added to the system, it can both register handlers for existing events and define new events. For example, the TCP graft registers a handler for the IP.PacketRecv event and introduces a new TPC.PacketRecv event.

Each event handler can be associated with a set of guards that filter out unwanted handler invocations. Guards are often generated when a handler is registered with an event. For example, the Plexus network protocol architecture uses guards to ensure that a TCP packet handler is called only for packets destined for its TCP port. Guards are required to be free of side-effects and the order of evaluation can be defined.

The implementation of event registration and dispatch is central to the performance and flexibility of SPIN. Events are simply entry points to a dispatch routine and a kernel function triggers an event simply by calling the entry point. Registering a handler updates the list of handlers for the event and causes a customized dispatch routine to be generated at run-time. The dispatch routine for an event with a single handler is a direct subroutine call. The code generator can create dispatch routines for events with multiple handlers and guards by unrolling the loop that iterates over the handler list and inlining small guards and handlers.

3.2 Safety

Since all SPIN grafts are executed in the kernel, hardware protection is inadequate to guarantee the safety of the kernel. SPIN uses a type-safe language to restrict a grafts access to the kernel. All user installable grafts must be compiled in Modula-3 and may have to meet additional compiler enforced restrictions. Modula-3's strong type checking enforces adherence to public interfaces. The type checking is specifically exploited to safety implement capabilities as pointers.

Additional restrictions are enforced by a variety of link-time and run-time mechanisms. Capabilities are used to control which events a component can raise. Some events require that all handlers are side-effect free. Compiler generated information is used to enforce these restrictions.

In order for a graft to be accepted from a user level application, the executable must be generated and marked by the Modula-3 compiler. If this marking is a cryptographic signature, it will be hard to forge. A more general approval mechanism might have a set of signatures that indicate "safe" code.

3.3 Plexus Network Protocol Architecture

The Plexus architecture [FB96] is a case study in the construction of event based kernel extensions. At the basic level, all network activity is modeled as two standard events, PacketRecv and PacketSend, and access to these events is mediated by a protocol manager.

The protocol manager enforces protection against snooping (unauthorized receives) and spoofing (unauthorized sends). Snooping is prevented because grafts do not directly register handlers with network events. The protocol manager performs the registration, after verifying that the handler meets both functional and access requirements for the event. This registration may include the introduction of guards to limit handler invocations. Spoofing is prevented by limiting access to the PacketSend event. To send a packet via a given protocol, a function must first obtain the right to raise the PacketSend event.

Plexus uses a variety of access control mechanisms and can place a variety of restrictions on handlers. Plexus will generate guards that prevent a handler from snooping on packets. Plexus can also require that a handler does not allocate any system resources. This is enforced by a special compiler generated type-marking and is validated at when the handler is registered.

3.4 Applications

The SPIN group has constructed number of applications, include a web server. The web server uses a custom caching policy based on file size. Small files, which are accessed frequently, are cached on an LRU basis. Large files are not cached at all, since they are rarely accessed.

The SPIN event model makes it straightforward to implement a TCP packet forwarder. By adding a handler to the IP.PacketRecv event, TCP packets can be redirected to a secondary machine. Because this forwarding occurs in the transport layer, control packets and congestion control operate normally. A user level packet forwarder that splices together an input and output socket breaks both of these mechanisms.

3.5 Performance

The reported performance measures for SPIN focus on low-level measurements such as graft to kernel execution and network latency.

Overhead for event dispatch to a handler is approximately twice the intrinsic cost of a subroutine call. This is almost 50 times faster then typical system calls. The low cost of event dispatch makes it feasible to implement system services by repeated calls to low-level primitives.

Performance of applications appears to be good compared to the similar applications running on DEC OSF/1. Their web server is almost twice as fast a reference DEC OSF/1 web server. A video streaming application places about half the CPU load as the OSF/1 implementation. Unfortunately, no other application performance measures are available.

3.6 Summary

The SPIN event model is an interesting way to layer operating system abstractions. It provides a simple model of kernel behavior that allows applications to change and add to the built-in policies.

Overall, SPIN provides a convenient way to experiment with new interfaces and new abstractions for applications. The compiler enforced safety seems to be too weak for a production operating system but may be strong enough given more experience.

4 VINO

The VINO operating system [SES96] is an extensible kernel that relies on software fault isolation (SFI) [WLA93] and lightweight transactions to provide safety. SFI re-writes assembly language code to insert memory reference and other graft safety guarantees. This allows safe VINO grafts to be written in C++.

Lightweight transactions allow the VINO kernel to cope with resource hoarding. In contrast, the SPIN operating system has only limited defenses against a greedy graft.

4.1 Design & Implementation

The VINO operating system can be viewed as both a set of objects and as a set of events and handlers. The VINO grafting architecture supports two different extension techniques. The grafting technique used depends on the role of the graft.

Simple grafts can replace the implementation of a method on existing system objects. This is used to change default policy implementations such as cache replacement or read-ahead strategies. The modified object can limit which, if any, methods are replaceable.

Grafts that extend the services provided by the kernel can register handlers and introduce events to the kernel. This is similar to the event and handler structure of SPIN, but appears to lack the guards that are used in SPIN.

When a function is grafted into the kernel, a small wrapper function is generated to transactionalize the graft. The VINO transaction manager ensures atomicity, consistency, and isolation. All accesses or changes to kernel resources require locks and are mediated through accessor functions. It also provides support for nested transactions with an in-memory undo stack.

When a kernel thread invokes a graft, a transaction object is created. If a transaction is associated with a thread, any use of an accessor function requires a lock and causes an undo operation to be pushed onto the undo stack. If a graft is aborted, the undo stack is unwound. If the aborted graft replaced a method on an object, the original function is then called. If a nest transaction is committed, its undo stack and locks are merged with the parent transaction. Committing a non-nested transaction frees the locks, the transaction object, and the undo stack.

Transactions are aborted when a resource constraint is violated. Locks are *time-constrained* system resources. A transaction will be aborted if there is competition for a lock and a timeout value has been exceeded. Other resources such as memory are *quantity-constrained*. The kernel will abort transactions that exceed their quota on these resources.

4.2 Safety

Software fault isolation is used to prevent buggy or malicious code from corrupting the integrity of the kernel.

While the kernel can run code that has been written in any language, it only accepts grafts that have been processed by MiSFIT, a software fault isolation tool. MiSFIT rewrites the submitted machine code to add memory reference checks and validate subroutine calls. Overhead for loads and stores varies from two to five cycles.

To protect function calls, a graft can only call functions that are on the list of graft-callable functions. MiSFIT validates direct function calls. Indirect function calls are checked at runtime through a hash table. This adds 10 to 15 cycles per indirect function call.

VINO uses a cryptographic signature to ensure that all grafts have been processed by MiSFIT. This is similar to the Authenticode™ technology used by Microsoft for web browser extensions.

4.3 Performance

The primary costs of VINO grafts are the MiSFIT induced code checks and the transaction overhead. Reported types are based given in sec. For reference purposes, their hardware (120 Mhz Pentium) takes 0.3 sec. for a subroutine call and a checksum of a 4K memory block takes 137 sec.

Memory access checks introduced by MiSFIT vary on the load/store complexity of the graft. A naive (xor) encryption graft, with a high load/store versus computation ratio, has a modest MiSFIT overhead of 26 sec on 160 sec of computation (16%). For very simple grafts, the MiSFIT overhead is comparable to the basic operation. Even for these grafts, the overhead (2 sec) very small in real terms.

The transaction costs of VINO are more substantive. Basic overhead for the creating and committing a transaction runs to 68 sec. An additional 33 sec is typical for lock acquisition. This brings the total to over 100 sec. For simple grafts this can be a 50 fold overhead. In larger grafts, like encryption, this quickly drops to a factor of two.

4.4 Summary

The costs of Software Fault Isolation are modest for many grafts, and it may be a feasible alternative to type-safe languages. The flexibility to use any language and the introduction of explicit memory checks is a more comfortable approach to graft development and kernel safety.

VINO also addresses the problem of resource hoarding. A safe graft can still misbehave if it runs too long or allocates too many system resources. SPIN deals with this problem only partially. The SPIN use of side-effect free grafts can avoid specific problems, but fails to solve the general problem.

VINO's transaction mechanism provides a nice way to deal with resource hoarding. Unfortunately, the costs of the mechanism are substantial. The authors themselves point out the need to use cost-benefit analysis when considering the utility of writing or using a selected graft.

5 Fox and PCC

The Fox Project [HLP98] uses Proof Carrying Code (PCC) [NL96] to guarantee that kernel grafts are safe. A kernel defines a safety policy and the PCC mechanism ensures that grafts satisfy the published policy. This technique is still in development. It is currently being used only for limited extensions to existing kernels, primarily installable network packet filters.

5.1 Design & Implementation

In order to verify that the graft satisfies the safety policy, an abstract model of the machine is defined. This model defines state-transition functions that can be used by the proof system. For many machines these functions will describe a subset of the actual processor. Troublesome instructions and cache behavior can often be omitted without impact to the power of grafts.

The safety policy is then defined in terms of pre-conditions, post-conditions, and permissible resource access services. Safety policies may limit reads or writes to scratch areas, or enforce limits on resource usage. Preservation of data-abstractions may also be defined by safety policies.

A proof that the graft satisfies the safety policy is created by computing a safety predicate for the binary using the abstract machine model. This safety predicate is then proved to be true given the conditions and services defined by the safety policy.

To validate a binary, the kernel starts by re-computing the safety predicate for the binary. The safety predicate is then used to validate the proof that accompanies the binary. This validation is decidable, since it is logically equivalent to typechecking.

5.2 Performance

The PCC techniques are currently in use only for installable packet filters. The Fox project compared their PCC based binary code filters to interpreted packet filters (BSD Packet Filters) and two other binary code packet filers: type-safe packet filters (Modula-3), and SFI based packet filters.

All three binary code filters where significantly faster then the interpreted filter. The slowest binary code packet filter (Modula-3) averaged 5 times the speed of the BSD packet filter. The PCC packet filters were uniformly the fastest, averaging 10 times the BSD packet filter.

5.3 Summary

PCC has some important advantages over other systems for ensuring the safety of grafts. The basic mechanisms are independent of any implementation language and they induce no run-time overhead on the grafts. The proof system is powerful enough to deal with optimized code.

The complexity of PCC is likely to limit its application for some time. Generating the proofs, defining sound protection policies, and validating that the code corresponds to the proof are major open problems. In addition, PCC has only a modest (2 times) performance advantage over other binary code grafting techniques.

6 Kea

The Kea operating system allows grafts to be added to the kernel through a general purpose "service migration" facility. Kea explicitly avoids the safety issue. Code trust is treated as a policy issue rather then a technology issue.

6.1 Design & Implementation

Kea aims to reduce the cost of the operating system by reducing the cost of RPC between address spaces. Components within the same address space use a fast

RPC mechanism; components within the kernel address space have fast access to kernel functions.

Applications are assigned to run in address spaces and communicate with other address spaces via *portals*. Except for portal calls, address spaces are isolated from each other. This is the implementation of system security. Portals are implemented as operating system generated code stubs; portals to foreign address spaces include a kernel trap to change address space. The kernel is simply a well known address space with well known portals.

Each portal identifies the destination address space and the entry point for the requested function. Collections of portals are organized into *services*. Services can be moved from one address space to another with no changes to service callers or the service implementation. Since all inter-service communication is via portals, services have no direct knowledge of other services that may share their address space.

When a service is moved from one address space to another, the system rewrites the portal transfer. Portals for co-located services are rewritten to use a single indirect procedure call. This specifically eliminates the kernel trap overhead.

6.2 Safety

In Kea, safety is based strictly on address space protection. All threads and services running in an address space are assumed to trust each other. Untrusted applications are run in their own address space. They have complete, but slower, access to services in foreign address spaces.

Moving a service into the kernel address space means that the kernel trusts the grafted service. They point out that services that benefit most from reduced RPC overhead, such as database servers, are already often trusted by the system anyway.

6.3 Performance

Compared to FreeBSD, Kea demonstrates good file system performance on their hardware (100MHz Pentium). By bundling services into a variety of address spaces, they were able to independently measure the effects of different file system architecture. Table 1 shows how different co-location combinations affect performance.

Operation	Separate spaces	In Kernel space		FreeBSD
		IDE/FAT	All	
read	184	155	85	95
write	170	130	73	111

Table 1. File system performance for selected address space combinations.

Kea file access involves of three layered services: an IDE disk driver, a FAT file system, and a global File service. The File service supports integrated access through a variety of file systems, including a FFS service. With all three services in separate domains, steady state reads and writes take approximately twice the time as FreeBSD. With all components in the kernel, Kea is 12% to 52% faster then FreeBSD.

6.4 Summary

The Kea technique of moving and bundling services into a variety of address spaces is an interesting generalization of graft co-location. Their observation that trust and safety can be managed externally suggests that this is not a critical requirement for likely grafts.

7 Exokernel

The exokernel concept stresses application management of system resources [EKO95, KEG97]. User mode libraries (libOSes) implement all conventional operating system functions. Exokernels enable application management of resources by not abstracting, or hiding, physical resources.

An exokernel gives applications the means to safely and directly manipulate system resources. This allows the construction of application-specific libraries that implement system objects. With the applications responsible to manage physical resources the kernel is left to perform strictly multiplexing.

7.1 Design and Implementation

A key concept in the exokernel architecture is to multiplex resources, not abstract them. This means that each exokernel exports a platform specific API. Conventional abstractions like processes and virtual memory are implemented by a libOS. The libOS delivers the standard operating system API to an application.

An important mechanism for exokernels is exporting the physical names for resources to libOSes. This allows a libOS to request specific resources and removes a level of indirection. Physical names encode useful resource attributes that can be used to avoid cache conflicts and improve resource scheduling.

An exokernel uses three techniques to securely export resources to a libOS: secure bindings, visible revocation and abort protocols. Secure bindings allow an exokernel to protect resources without understanding them. Secure bindings are implemented as hardware mechanisms, software caching, or grafts. Visible revocation means that the libOS is notified with the exokernel must deallocate system resources that are held by the libOS. Well behaved libOSes will cooperate and return unneeded resources. The abort protocols handle ill-behaved libOS. When necessary, an exokernel can preempt a process or seize a resource. When this occurs, repossession vector is sent to the libOS so that it can update any mappings that use the resource.

Exokernels use application grafts primarily to test and validate protection. Grafts are used when table based tracking of system resources is not feasible. Certain large and complex resources, such as network packets and disk blocks, cannot be readily protected by memory based resource tables. Grafts also avoid the cost of a kernel mode crossing in situations where context switching to the application is infeasible.

An important application of exokernel grafts is the process of determining which application an incoming message should be delivered to. In the Aegis exokernel, dynamic code generation is used to create aggressively optimized packet filters from a restricted filtering language.

7.2 Safety

With the exokernel architecture, most extensions to the operating system can be made by changing the libOS. Since the libOS executes in user mode, changes to it are inherently more safe then changes to the kernel.

Exokernel grafts are protected by a variety of mechanisms. Packet filter grafts use a restricted language and runtime code generation to prevent poor behavior. Other grafting systems in exokernels use code inspection and sandboxing to ensure safety.

An exokernel does need to be carefully designed to prevent inadvertent exposure of application private data.

7.3 Two Exokernels

Aegis is an exokernel implemented for a MIPS based DECstation. ExOS is a UNIX emulation library operating system implemented that was originally developed for the MIPS. Aegis and ExOS has been used extensively to experiment with the exokernel concepts.

Aegis' protection of system resource relies on a processor environment structure that stores the information needed to deliver events to applications. Aegis delivers four kinds of events to application: exceptions, interrupts, protected control transfers, and address translations. Each event causes a direct transfer to the application. Normally a libOS will receive and handle the event.

ExOS is a libOS that supports most of the abstractions found in 4.4BSD UNIX. It manages the fundamental operating systems abstractions (e.g., virtual memory and process) at application level, completely with in the address space of the application that is using it. Many UNIX applications (cp, gcc, etc.) run on ExOS without modification and with better performance than a monolithic kernel on the same hardware.

The Xok exokernel is similar to Aegis that runs on Intel x86 based computers. This requires several changes to the kernel interface. For example, the x86 TLB cannot be controlled by the libOS. This slightly reduces the flexibility of Xok, but simplifies its implementation. Virtual memory remains the prerogative of the libOS. The standard libOS is ExOS.

7.4 The XN Stable Storage System

LibOSes that use the disk contain one or more library file systems (libFSes). Each libFS is responsible for managing its own files. The role of the exokernel is to give as much control as possible to the libFSes while still protecting the files from unauthorized access. This has proven difficult. Simple approaches, such as partitioning the disk for each file system, severely limit the flexibility of libOSes to manage the files.

XN, a 4th generation of design, provides stable storage at the level of disk blocks, a buffer cache registry, protection of file system metadata, and flexible access protection mechanisms.

XN controls access to disk blocks through a set of three grafts that are bound into a file system installable "template". These grafts are called untrusted deterministic functions (UDF) and are written in a restricted language that can be checked to ensure deterministic behavior. The *owns_udf* function must be deterministic and is used to track the owners of a disk block. The *acl_udf* and *size_udf* functions are not required to be deterministic – for example they may use time of day. The *acl_udf* function is used to determine file access permission, *size_udf* returns the size of the

7.5 Cheetah Web Server

The Cheetah web server is designed to exploit the exokernel concept to handle high HTTP traffic. The combination of Cheetah and the Xok exokernel has eight times the performance of NCSA 1.4.2 server on OpenBSD 2.0 and twice the performance of the fast Harvest cache.

The Cheetah avoids data touching by transmitting data directly from the file cache using pre-computed checksums. Because Cheetah tracks the status of the each request, it is able to merge the ACKs for client HTTP transmissions onto the document response. This is particularly valuable for small document sizes where it reduces the total number of packets by 20%. Cheetah also places files included in an HTML documents on adjacent disk blocks. This optimization increased throughput by a factor of two.

7.6 Performance

Experiments show that exokernels have good performance compared to monolithic UNIX operating systems. A variety of UNIX applications have been ported to Xok/ExOS, including cp, gcc, and diff. Experiments show that applications do not need to be re-written or even modified in order to take advantage of an exokernel.

One experiment measured performance on an I/O intensive workload that consisted of installing software, copying compressed files, uncompressing them, unpacking them, deleting binaries and deleting source tree. On a 200MHz Intel Pentium Pro, the Xok/ExOS run-time was 41 seconds, 19 seconds faster then either OpenBSD of FreeBSD.

Other experiments measured the performance of Xok/ExOS on the Modified Andrew Benchmark. On this benchmark, Xok/ExOS takes 11.5 seconds. This matches the FreeBSD time and beats OpenBSD. ExOS performance "suffers" on this benchmark due to a naive page table strategy that does not yet support shared page tables.

7.7 Summary

Traditional operating systems have significantly limited the performance, flexibility, and functionality of applications by abstracting physical hardware resources. The exokernel architecture avoids these problems by eliminating abstraction and exporting hidden, machine dependent, details. This allows applications and libOSes to effectively manage their own resources.

An exokernel is similar to a micro-kernel in the goal of placing only essential functions in the kernel. While standard micro-kernels export an API that abstracts the hardware resources [HHL97], exokernels export a machine specific interface. All abstractions are implemented in the libOS.

8 Protected Shared Libraries

Protected Shared Libraries (PSL) is a mechanism for building operating system libraries that safely run in user mode address space [BC94]. PSL exploits multiple memory segment registers, a common feature of modern computer architectures, to decouple protection domains and address spaces. This allows an application to safely share system resources and enjoy the benefits of a user mode operating system.

8.1 Design & Implementation

Modern computer architectures include both sophisticated memory management hardware and efficient mechanisms to transfer control to system mode. PSL combines these mechanisms into a "service call" mechanism that can change protection domains without causing heavyweight process or address space changes.

A simple kernel call can be implemented as a trap or software interrupt that switches to kernel mode and transfers control to the service code. User-supplied data needed by the call is left in user space and is accessed directly by the kernel-level service code. The high cost of many service calls is the overhead to support process and address space abstractions.

A service call is a simple variation on this technique that supports the safe sharing of operating system structures in user mode. A service call starts with a lightweight trap that changes the permissions and mappings used to limit access to memory. Rather then transfer control to a separate address space, the operating system's service code temporarily becomes part of the application's address space. By providing special control over separate stack and shared memory areas, safety can be ensured at a low cost.

A reasonable arrangement of memory protection defines four separate address domains: kernel code and data, library code, application code, and application data. PSL adds two additional address domains: common and service stack. The common domain provides efficient protected access to shared data. The service stack guarantees that service code has a safe control history. With this arrangement, a protected domain transfer requires only simple changes to the address space descriptor and a transfer back to the user mode at the service's entry point.

The PSL implementation on IBM's POWER architecture uses five of the 16 segment registers. While segment registers are a scarce resource, only four of the segment registers are used by normal applications. Each segment register refers to a subset of a 256Mbyte segment of the 52 bit address space.

8.2 Safety

PSL guarantees safety by carefully controlling the access rights to memory. It partitions memory into three different categories: private, shared, and common. A private region contains the client code and data and the stack used when the thread is running in the client code. The shared region contains the service code and service owned data. A common region is key to the flexible nature of protected shared libraries. Common memory is a unique to each task, but shared with the library. By placing the service stack in the common area, the library can be isolated

When an application is executing user code, library code and data are not accessible through the memory segment registers. When the library code is executing, private user data and code is protected.

8.3 Summary

Protected shared libraries make effective use of common hardware resources to quickly activate operating system functions. It does away with much of the overhead of protection domain transfers while ensuring the protection of both private application and shared operating system data. It seems well-suited to providing lightweight access to common operating system functions.

9 Conclusions

The pervasive presence of computers is leading to new and demanding applications. Like database managers and web servers before them, these applications will use computer resources in unexpected manners. Achieving adequate performance will require adapting resource management to the applications needs.

Extensible operating systems have a clear advantage to server applications. Database managers and web servers have very high resource demands and are often run on dedicated hardware. Their intense resource usage makes it reasonable to spend time and effort improving performance. Extensible operating systems allow these applications to be tuned to the hardware. Since they run on dedicated hardware, safety and compatibility with other applications is less important.

It is less clear that extensible operating systems are a good model for general purpose computers. Safe interaction among varied applications is very important in this environment. In addition, most applications are not a heavy resource drain on modern computers.

Due to their size and complexity, conventional monolithic kernels will be hard pressed to keep up with the new demands for custom resource management. The abstractions inherent in monolithic kernels penalize applications that don't use the mechanism. Since these costs are built in to the kernel, there is no way to avoid them.

Extensible operating systems directly support the application management of system resources. This provides an important way to speed up applications that have novel and distinctive patterns of resources use. An application can modify the operating system's interfaces and implementations to provide a better match between the needs of the application and the performance and functional characteristics of the system.

Extensible operating systems have demonstrated that information hiding by operating systems abstractions is a major performance bottleneck. It is striking that so many groups have demonstrated such significant performance improvements. Loosening the strong barriers to operating system internals seems to be an easy way to get performance improvements. As the range of applications expands, it becomes increasingly difficult to determine what information is unnecessary.

The exokernel architecture most directly addresses the causes of poor application performance. Rather then abstracting and hiding information about system resources, it strives to expose the hardware details. Nothing is hidden so applications are free to manage resources with full knowledge.

Building the operating system as a user mode library is an effective way to allow applications to select and build the abstractions that work best for their patterns of resource usage. The exokernel research indicates that tremendous performance increases are available to applications that aggressively optimize resource usage. This performance potential is available without penalty to unoptimized applications. Directly ported applications that use the UNIX library match their performance on monolithic kernels on identical hardware. Sophisticated resource sharing by libOSes is possible with the techniques of Protected Shared Libraries (PSL). This provides lightweight, coordinated behavior between applications.

Extensible operating systems encourage innovation in operating system interface design. User mode libraries and grafts allow ready hacking on the operating system. There are a lot of user level programmers and it is a much simpler development environment.

Research into extensible operating systems has already lead to important new operating system interfaces. The XN stable storage system is an excellent example of a new rich interface that safely offers rich and powerful features. The SPIN event and handler mechanisms are a clean and elegant way to construct operating system internals.

An important open problem is to ensure that grafts are safe. Applications that do not use a graft should not be affected by the extension, regardless of how ill-behaved the extension is. Current safety mechanisms for binary grafts, such as type-safe languages and SFI, remain awkward. Code inspection and runtime code generation are easier to use but have limited expressive power. Use of grafts will probably remain a privileged feature for some time.

Another problem extensible operating systems will have to address is the interaction of different extension. A common claim is that multiple extensions can be active on the system at the same time. It is not clear how different extensions with different resource models can cooperate. On many platforms, it is extremely difficult to build programs from a mix of languages because the run-time libraries are incompatible. Often it is a conflict in the abstract model of the IO system that creates the incompatibility. In operating systems that allow insertions to add new abstract models, the likelihood of an incompatibility seems high.

10 References

[BC94] A. Banerji, D. L. Cohn. Protected Shared Libraries. Technical Report TR-94-37, Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN, 1994.

[BSP95] B. N. Bershad, S. Savage, P. Paryak, E.G. Sirer, M. E. Fiucynski, D. Becker, C. Chambers. Extensibility, Safety and Performance in the SPIN Operating System, *Proceeding of the Fifteenth Symposium on*

Operating Systems Principles (SOSP-95), Dec 1995

[EKO95] D.R. Engler, M.F. Kaashoek, J. O'Toole Jr. Exokernel – An Operating System Architecture for Application-Level Resource Management, *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP-95)*, December 1995, pp. 251-266.

[FB96] M.E. Fiucynski, B.N. Bershad. An Extensible Protocol Architecture for Application-Specific Networking, *Proceedings of the 1996 Winter USENIX Conference*, San Diego, CA, January 1996., pp. 55-64.

[HHL97] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, J. Wolter. The Performance of -Kernel-Based Systems, *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP-97)*, Saint Malo, France, Oct 5-8, 1997, pp. 66-77.

[HLP98] R. Harper, P. Lee, F. Pfenning. The Fox Project: Advanced Language Technology for Extensible Systems, Technical Report CMU-CS-98-107, Carnegie Mellon University, January 1998

[KEG97] M. F. Kaashoek, D. R Engler, G. R. Ganger, H. M. Brice o, R. Hunt, D. Mazieres, T. Pinckney, R. Grimm, J. Jannotti, K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP-97)*, Saint Malo, France, Oct 5-8, 1997

[Lam71] B. W. Lampson. Protection. in *Proceeding of the Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pp. 437-443, reprinted in *Operating Systems Review*, 8(1), January 1974, pp. 18-24

[NL96] G. C. Necula, P. Lee. Safe Kernel Extensions Without Run-Time Checking. *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI-96)*, Seattle, WA, October 1996, pp. 229-243.

[PB96] P. Pardyak, B. N. Bershad. Dynamic Binding for an Extensible System, *Proceedings of the Second Symposiumon Operating Systems Design and Implementation (OSDI-96)*, Seattle, WA, October 1996, pp. 201-212.

[SES96] M.I. Seltzer, Y. Endo, C. Small, K.A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions, *Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI-96)*, Seattle, WA, October 1996.

[Sto91] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7), July 1991, pp. 412-418

[VH96] A.C. Veitch, N.C. Hutchinson. Kea – A Dynamically Extensible and Configuratble Operating System, *Proceeding of the Third Interational Conference on Configurable Distributed Systems*, Annapolis, Maryland, May 6-8, 1996

[VH98] A.C. Veitch, N.C. Hutchinson. Dynamic Service Reconfiguration and Migration in the Kea Kernel, *Proceeding of the Fourth Interational Conference on Configurable Distributed Systems*, Annapolis, Maryland, May 4-6, 1998

[WLA93] R. Wahbe, S Lucco, T. E. Anderson, S. L. Graham. Efficient Software-Based Fault Isolation. *Proceedings of the Fourteenth ACM Symposium on Operating System Priniciples (SOSP-93)*, 1993, pp. 203-216.