Building Fundamentally Extensible Application-Specific Operating Systems in SPACE [1]

Dave Probert and John Bruno

Computer Science Department
University of California
Santa Barbara, California 93106

contact: probert@cs.ucsb.edu

UCSB Computer Science — TRCS95–06 — March 1995

[1]This research was supported in part by a UC MICRO grant and the Xerox Corporation.

Building Fundamentally Extensible Application-Specific Operating Systems in SPACE [1]

Dave Probert and John Bruno

Abstract

We refine and extend previous work [1] towards a new approach to implementing operating systems. This approach, called SPACE*, uses processors, address spaces, and a generalized exception mechanism as the basic abstractions rather than processes, virtual memory, and interprocess communication. The result is that the facilities provided by the hardware architecture are exposed, and can be directly used by applications to achieve higher performance and more diverse functionality than is possible with conventional operating system abstractions. In* SPACE *the conventional abstractions can coexist with new application-specific abstractions, due to the fundamental extensibility of the* SPACE *primitives. Applications can implement their own versions of system services to satisfy critical functional and performance requirements while still benefiting from standard system services. We then give examples of implementing mechanisms in* SPACE*.*

1 Introduction

Historically operating systems were developed to provide two particular benefits to the application programmer: abstraction and protection. Abstraction provides a more convenient interface to the hardware architecture, improving uniformity and enhancing functionality. Protection limits access to physical resources in order to maintain the integrity of the system against errant or malicious applications. Protection is normally embedded within the abstractions provided by a system.

The most influential factor in the evolution of operating systems has been resource sharing by multiple applications. To support sharing, the primary tool has been to use abstractions that virtualize resources, thereby creating the illusion that there is no sharing. Processors, memory, and I/O became virtual processors (processes), virtual memory, and virtual I/O (files in filesystems).

1.1 Operating System Performance

One area where operating systems do not provide benefit is performance. Any single application can achieve at least as much performance by executing on the bare hardware, as by executing with the virtualized abstractions provided by the operating system. (Simply embed the O/S as part of the application and strip out all code and data it never uses).

Many applications will perform much better without the benefit of the operating system. Such applications access the operating system frequently, but use only a small subset of the available functionality. Performing frequent I/O or address space manipulations and use of multi-threading and/or multiple processors are characteristics of *service-intensive* applications which can experience high operating system overhead. These types of applications are increasing in importance as the changing economics of computing make dedication of multiple computing resources to single applications more attractive.

However it is not just the increased importance of service-intensive applications that are making operating system performance more critical. Operating systems themselves have been slowing down. This problem has been obscured by the juggernaut of rapidly improving raw processor speed, yet if operating system performance is normalized to keep application performance constant, a marked decrease in operating system performance becomes visible. One aspect of this decrease has been attributed to an increasing mismatch between hardware architecture and operating system abstractions [2].

[1] This research was supported in part by a UC MICRO grant and the Xerox Corporation.

A further source of the performance problem is due to increasing application requirements for operating system functionality. Some of these requirements are due to the need for interoperability between applications running on different platforms, as well as portability from one platform to another. Others arise from a need for new abstractions to simplify application implementation (e.g. distributed shared memory[3] and efficient user-level threading [4]). Performance also affects functionality requirements by necessitating changes in interfaces and implementation.

As application demands have been increasing, the complexity of the abstractions provided by operating systems are also increasing. More complexity means more code. Operating systems have become bloated. Even microkernels can exceed fifty thousand lines of source code[5]. The increasingly complex abstractions are multiplexed through a procedure-call system interface, and then internally demultiplexed according to the parameters and the current system state.

As an example of the impact of generic operating system abstraction on performance, consider the I/O subsystem in UNIX. The open/read/write/close system calls can reference communication paths of various types, many different types of raw I/O devices, as well as files and directories residing in a variety of filesystem types. Consider a read from a UNIX filesystem disk in System V Release 4. The read execution path passes through the following layers and interfaces[6]:

> *trap, syscall, vnode, ufs-read, inode, uio, trap, page-fault, address-space-fault, vnode-segfault, ufs-pagein, buffer-cache-read, raw-device-switch, and finally the device-driver*

Along the way much code is devoted to demultiplexing various operations and conditions, as well as checking for errors. The layering of the implementation allows the use of generic interfaces throughout the UNIX kernel, which provides modularization and thus extensibility. The goals of sharing and protection are achieved by this type of implementation, but the cost in performance is high. This cost can be quantified by considering systems which tailor system services for applications.

The Synthesis Kernel [7] synthesizes specialized code for read/write when an application opens a file. Performance (versus SunOS 3.5) for UNIX pipe operations was improved by 4-6 times for larger buffer sizes (4096-1024 bytes), and 56 times for single bytes.

The overhead of the I/O in the UNIX implementation is so high that it must be amortized over large transfers to even approach the performance of the specialized code. Batching of service requests has been applied not only to file I/O, but also to network protocols [8]. Unfortunately there are many applications where batching is difficult, such as client-server RPC using small messages, or fine grain concurrency in multiprocessors. To achieve performance here, the operating system overhead must be dramatically reduced.

1.2 Application-specific Operating Systems

The problem is that generic operating systems are required to do too much to do it efficiently. As service-intensive applications increase in importance, operating system overhead is becoming a more visible bottleneck. Applications are paying the price for a generality in sharing which is not always needed, as the economics of computing is making dedication of computing resources to single applications more common.

We believe that a promising approach to addressing the operating system performance issues we have described is to allow applications to customize the operating system to suit their own particular performance and functional requirements. However it is necessary that the historical operating system goals of sharing and protection still be achieved. Applications programmers do not want to reinvent entire operating systems every time they write a program. It is only in areas where performance or functionality is critical that an investment in extension of the operating system is worthwhile. Most operating system functionality is best provided by libraries that the application programmer can build upon.

We are developing a system, called SPACE, which provides a model of structuring operating system environments that are fundamentally extensible. SPACE is an acronym for *Systems Programming using Address spaces and Capabilities for Extensibility*. The conventional structuring technique for operating systems is to build them around a single

kernel which provides a set of basic abstractions. Since the kernel is unique its abstractions are difficult to replace or experiment with. When the kernel fails, the system fails.

In contrast SPACE supports building operating system services using multiple kernels implemented with the same tools used to build applications. A new system service can be implemented in a new kernel, while existing kernels continue to provide the previous functionality. Because there is no distinction between code implementing operating system services and code implementing applications, the abstraction of a privileged user is difficult to define. Instead SPACE uses capabilities to build protection mechanisms.

In section 2 we present SPACE. Our approach is then contrasted to other efforts at operating system extensibility in section 3. Section 4 illustrates SPACE by describing an implementation on the SPARC architecture. In section 5 we present some examples of implementing common mechanisms in SPACE. In the final sections we present our conclusions and discuss future directions.

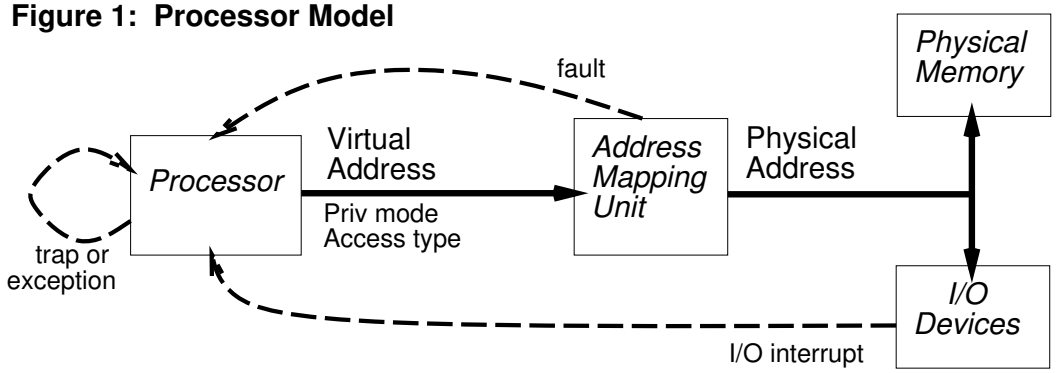
2 The SPACE Model

SPACE uses a generalized model of processor operation to provide abstractions that more closely match processor architectures than do conventional abstractions. The SPACE abstractions extend exception handling and privilege to allow software to be built using kernelized modules. In this section we present the SPACE model.

2.1 Processor Architecture

A simple block diagram of a processor is shown in figure 1. The processor execution normally consists of modifying the program counter, issuing loads and stores, and modifying internal registers. Loads and stores emit virtual addresses, which are translated by the Address Mapping Unit to physical addresses. A mapping of the set of all valid virtual addresses to physical addresses is a *context*. The AMU can support multiple contexts, each identified by a *context identifier*. The context currently in use is established by loading the corresponding context identifier into a hardware register.

Figure 1: Processor Model



The processor has more than one privilege mode (i.e. user and supervisor). The current privilege mode of the processor, as well as the access type (e.g. read/write/execute) is provided to the AMU along with the virtual address. The AMU declares an access fault if the requested access is not valid for the current privilege mode.

During normal execution one of several events may arise. An external I/O device presents an interrupt. The AMU may detect a fault in a mapping request. The processor may internally encounter an exception or trap condition (e.g. divide-by-zero or execution of a trap-instruction). For convenience we refer to any of these types of events as an *exception*.

When an exception occurs, processor execution is altered. The current program counter is saved somewhere (along with

other information related to the exception), and the processor begins execution at a new program counter associated with the exception. Often further exceptions are inhibited and the privilege mode changes. The current AMU context may grant the new privilege mode access to pages that were previously protected.

Upon return from an exception, the previous processor state (including the privilege mode and program counter) is simply restored.

2.2 SPACE generalizations

SPACE generalizes the processor architecture by abstracting *address spaces*, *privilege modes*, and *exceptions*. These mechanisms are used to construct software using *spaces*, *domains*, and *portals*. We describe the nature of the underlying mechanisms used by SPACE, and explain how these mechanisms provide fundamental extensibility.

2.3 Extended Privilege Modes

Address spaces correspond to AMU contexts, but instead of two privilege modes, SPACE supports an arbitrary number. Each privilege mode may have different access rights with respect to particular pages in the current address space. Since most AMU architectures support only two privilege modes, SPACE simulates privilege modes using overlapping AMU contexts. Distinct privilege modes within a single address space are represented by distinct AMU contexts which differ only in the access rights they grant.

Address spaces in SPACE are primitive compared to the *virtual address spaces* that programmers are most familiar with. The latter combine the address space supported by the AMU with a virtual memory implementation. SPACE decouples the virtual memory implementation from the address space. This allows multiple instances of virtual memory implementations to coexist within the system, or even within a single application. Virtual-memory-like mechanisms can be also used to implement other abstractions, such as garbage collectors, data-compression paging, and persistent stores [3].

The SPACE generalization of privilege provides for the implementation of protection domains. In single-kernel operating systems, each address space is part of two domains, corresponding to the two hardware privilege modes (user and supervisor). In SPACE an arbitrary number of domains can exist in an address space, each associated with a different privilege mode. Protection domains are useful for modularizing software [9], and insuring the integrity of shared data structures. Applications can be selective about which pages in their address space they expose to system services. System services can be implemented by applications.

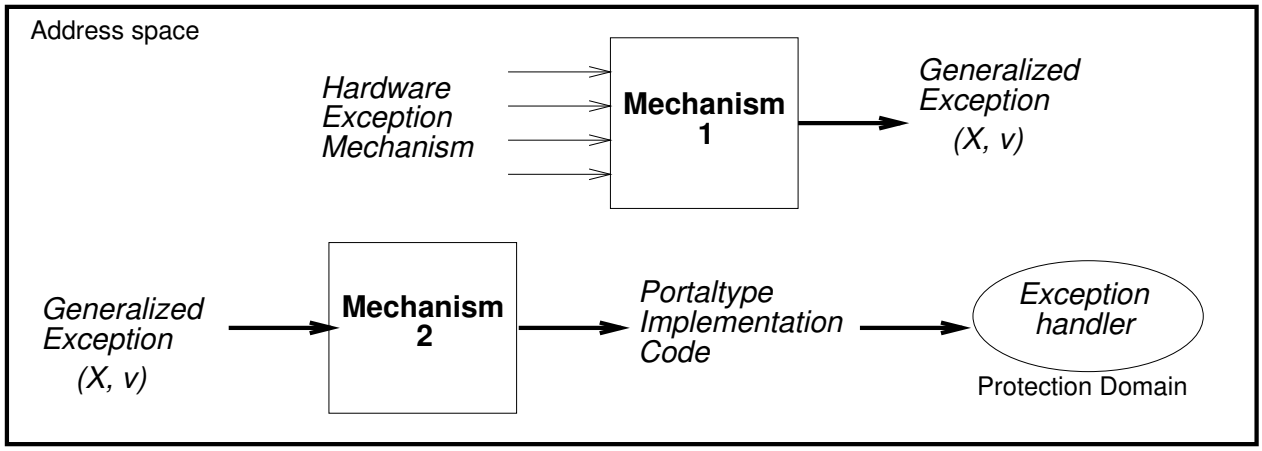


Figure 2: Generalized Exception and Portal Type Mechanisms

2.4 Generalized Exceptions

Exceptions are extended in SPACE through three related mechanisms, shown in figure 2.

The first mechanism translates a hardware exception and related information (e.g. processor exception registers) into a generalized exception and parameter pair *(X,v)*. The parameter *v* normally identifies the particular instance of the *generalized exception X* (e.g. trap number). However for exceptions generated by memory references, *v* is the virtual address that caused the hardware exception.

This generalized exception mechanism decouples the hardware view of exceptions from the requirements of the application. Hardware exceptions which directly satisfy application requirements are mapped into generalized exceptions without overhead. Examples of exceptions which may be directly mapped include trap-instructions and arithmetic exceptions.

In other cases a hardware exception may not be suitable for the application. Perhaps the application was originally designed for another architecture. Perhaps the hardware exception needs to be demultiplexed into several cases that are handled differently. A generalized exception can be synthesized from a hardware exception and auxiliary processor state (e.g. special registers and access bits in page table entries). Here are some examples of generalized exceptions that must be synthesized on most architectures:

Pseudo instructions: the hardware illegal instruction exception decoded into several different synthesized exceptions based on bits in the instruction.

Non-local procedure calls: text faults for valid but non-executable pages, for use in invoking procedures in other protection domains.

Copy-on-write: data faults on pages with page table entries marked read-only for user mode, but read-write for supervisor (supervisor pages are normally unreadable in user mode).

Descriptor faults: uses alignment restrictions to generate exceptions based on invalid low order address bits.

2.5 Portals

The second mechanism maps generalized exceptions into portals. A *portal* associates an *(X,v)* pair with a *generalized-exception handler*. It specifies a new value for the program counter, as well as a new address space and privilege mode in which to execute. For memory exceptions, where *v* is a virtual address, different handlers can be associated with different parts of an address space. This supports the coexistence of distinct virtual memory implementations within an application.

Invalid regions of an address space can be used as a name space for remote system services. This is often preferable to trying to manage the allocation of the relatively few trap-instructions that are available. These services can be implemented transparently, simulating local procedure calls and memory references. For example, a read operation to an invalid location appears to succeed when in fact no memory is mapped at that location. The reference was simulated by an exception handler.

Control transfer between domains requires a cross-domain call. The parameters to the call may be scalars, or pointers to shared memory areas. Cross-domain calls are invoked by causing an exception (e.g. a trap-instruction or invalid memory reference). Since the handler for the generalized exception is invoked in an arbitrary protection domain, it is the target of the cross-domain call.

2.6 Portal Types

Portals also specify the *portal type* implementation to use for traversing the portal to the exception handler and later returning. The portal type determines the conventions for saving state, synthesizing parameters, and resuming the interrupted execution upon return.

Examples of basic portal types include various forms of RPC, page-faults, I/O interrupts, and simulation of memory references. The portal type implementations differ in cost and generality.

Portal type implementations do not necessarily involve cross-domain calls. Some portal types may execute to completion without ever entering a new protection domain, *short-circuiting* the normal portal mechanism. Such portal types are useful for extension of the processor instruction set (e.g. compare&swap), as well as implementation of short operations critical to application performance. Short-circuit portal types are also used to implement specialized operations, such as *return-from-exception*.

Portal type implementations are not independent of each other. They may share common code and data structures. Also they must interact to accomplish some operations (e.g. *return-from-exception*). Address spaces connected by portals must have compatible implementations of generalized exceptions and portal types. A portal traversal from space A to a handler in space B will use a portal type defined in A to make the traversal. B must have a compatible *return-from-exception* mechanism if the exception handler in B is to successfully resume the execution in A.

Exception handlers must understand the conventions used by the portal types which will invoke them, in order to correctly save and restore registers, as well as interpret parameters. Each portal type may only be compatible with certain exceptions. For example, a portal type which implements I/O interrupts cannot make assumptions about the state of the processor, but must save/restore all state. In contrast, an intra-space version of an RPC portal type might assume that the current memory stack is accessible and use it.

Portal types may arbitrarily interpret the fields in the portal data structure. For example an I/O interrupt portal may extract an interrupt mask from the portal data. One interpretation of the portal fields which is particularly important is the implementation of a capability mechanism.

2.7 Address-based Capabilities

The use of a single kernel in conventional operating systems simplifies issues of protection. The kernel trusts itself. In a multi-kernel operating system, with users writing system services, a more general protection mechanism is needed, such as capabilities.

Implementation of low-level capabilities is built into portal types that need it. Part of the portal data contains a bitmap, with one bit corresponding to each privilege mode. The bitmap is used to determine whether the portal may be traversed from the current privilege mode. *The ability to traverse a portal is a capability*. This mechanism allows portals to be used to access protected services from one privilege mode without granting access to all privilege modes executing within the address space. In many cases this eliminates the need for an operating system service to independently validate the right to access the service.

2.8 Fundamental Extensibility

The third mechanism used to extend exceptions in SPACE allows different implementations of the generalized exception and portal type mechanisms to be associated with *each* address space. Applications thus have the freedom to modify even the fundamental abstractions of SPACE, creating new generalized exceptions and portal types as required.

The SPACE generalizations may seem only slight, but they are sufficient to build systems which are extensible at the most fundamental levels. The decoupling of virtual memory implementation from the abstraction of an address space gives applications direct access to the hardware primitives that underly virtual memory. Extending privilege modes allows multiple protection domains to exist, enabling multiple kernel implementations of system services. The extension of the exception mechanism is a natural method of supporting exception handling in an environment with multiple protection domains. Supporting SPACE implementations on a per-space basis allows applications to completely customize the abstraction of the hardware architecture.

SPACE offers a wide range of implementation possibilities. This is essential to providing applications as much flexibility as they may need to extend the system to suit their requirements. This is the inherent strength of the low-level access to the hardware architecture provided by SPACE. In choosing a particular implementation strategy,

application programmers must make trade-offs between performance and generality, between performance and safety, between investment and return. However, in making these trade-offs they will not be constrained by operating system abstractions that failed to anticipate their requirements.

3 Related Work

Extension of the user/supervisor privilege levels into an arbitrary number of protection domains is a very old idea (e.g. [9]). Multics [10] used eight hierarchical rings of protection. VAX/VMS used four hierarchical privilege modes [11]. Both Multics and VMS used a layered approach to O/S implementation, as popularized by Dijkstra [12]. Layered systems are successful at managing complexity, but have difficulty supporting alternative implementations of particular services. Spier used the four privilege modes on the PDP 11/45 to implement domain-based software, but the goal was to improve modularization and fault isolation in application software, rather than embedding operating system services.

The i432 architecture was designed to support O/S implementation using address spaces as independent protection domains, avoiding the user/supervisor dichotomy [13]. The system was segment and descriptor based, and used the descriptors to implement access control. Low-level operating system abstractions in the i432 were embedded in the microcode, but they were too expensive to achieve acceptable performance. The i432 is an example of operating system primitives which try to be too general to be generally useful.

Mach [14] made a substantial contribution to decentralizing operating system design by using a microkernel that retained process, virtual memory abstractions, and interprocess communication primitives while relegating other system services to servers. SPACE goes further by moving even process, VM, and IPC out of the kernel. The fundamental communication primitive in SPACE is a cross-domain call which can be implemented much more efficiently than the general message passing primitives in Mach, avoiding the performance problems which have been a major problem with the microkernel approach.

Several recent systems use protection domains. Opal [15] is an investigation into using a single large address space for running complex applications. Privilege domains are used to manage sharing and protection. Portals between domains are implemented using an explicit capability mechanism rather than an abstraction of the exception mechanism, as in SPACE.

Spring [16] is a microkernel-based operating system developed to build operating systems out of replaceable parts. Spring makes heavy use of cross-domain calls and capability-based portals (called *doors*), similar to the earlier version of SPACE [1] but without the use of the address space as a name space for services, and without the extension of privilege modes.

The Synthesis Kernel [7] demonstrated the value of eliminating operating system overhead through customization. However Synthesis generates code to efficiently implement a standard set of abstractions rather than building custom abstractions for embedding system services into applications.

Sandboxing [17] is a technique developed to allow embedding of untrusted application code inside trusted programs, such as an operating system kernel. The untrusted code is modified to control access to external code and data. This is a less general approach than SPACE, but is particularly valuable on architectures where cross-domain calls are very expensive.

A number of researchers have investigated providing mechanisms to allow applications to exploit hardware facilities, such as exception handling and memory management. Efficient handling of synchronous exceptions is explored by Thekkath and Levy [18]. The Tempest interface [19] provides low-level mechanisms allowing implementation of shared-memory semantics within applications. Earlier work with the Wisconsin Wind Tunnel interface [20] provided untrusted application code with protected access to memory management functions.

The SPIN operating system [21] has similar goals to SPACE. It supports application-specific operating system services through an extensible microkernel and application-specific *spindles*, which are *mounted* on the microkernel to encapsulate policy decisions. Spindles are a higher-level abstraction than the primitives in SPACE. However the use

of language mechanisms to ensure the safe integration of spindles with the microkernel is similar to the technique we are investigating for customizing generalized exceptions and portal types.

The Exokernel approach to operating systems [22] most closely mirrors the philosophy embodied in SPACE: make operating system abstractions optional. Exokernels replace standard abstractions, such as processes and virtual memory, with mechanisms in the kernel which export raw physical resources to applications directly. Policy decisions are made at the application level, though applications can also download code into the kernel to reduce the impact of crossing the kernel boundary. SPACE reduces the level of operating system further than exokernels, moving even resource allocation outside the kernel.

4 An Implementation of SPACE on the SPARC Architecture

To demonstrate SPACE we describe an implementation on the SPARC architecture. After describing the SuperSPARC processor we are using, we present the implementation of the SPACE mechanisms and give examples of the generalized exceptions and portal types employed.

4.1 The SuperSPARC Architecture

Most processors fit the general processor model assumed by SPACE. However there are architectural characteristics which are desirable if an application is to make full use of SPACE with reasonable performance:

- A low overhead hardware exception mechanism as found in most RISC processors.
- Low hardware context switch costs (e.g. tagged virtual or physically mapped caches, and tagged TLBs in the AMU).
- The AMU should provide enough contexts to efficiently implement multiple privileges within a single address space.

One architecture which has these characteristics is the SuperSPARC RISC processor [23], found in SPARCstation 10 workstations, the Meiko CS-2 scalable parallel computer, and elsewhere.

When a hardware exception occurs, the SuperSPARC processor begins execution of the instructions in the hardware trap vector within a few clocks of detecting the exception. The instruction and data caches are physical, and the TLBs are tagged so that neither the caches nor the TLB must be flushed during a context switch. These features allow cross-domain calls to be efficiently implemented.

The SuperSPARC uses a SPARC reference MMU [24], which tags the TLB entries with a 16-bit context identifier. Multi-level tablewalks are used to handle TLB misses in hardware, by walking the tree of page tables.

The 16-bits of context identifier is more than adequate for simulating privilege modes through overlapping AMU contexts. The overlapping contexts differ from one another only in the access rights for each page, not in the virtual-to-physical mapping.

For each page, there are only a small number of access modes supported by the hardware, and our implementation supports even fewer: *invalid, execute-only, read-only,* and *read-write*. No matter how many privilege modes are used within a particular space, these are the only possible access modes for each page. If addresses are allocated such that pages with different access modes are sufficiently separated, then many of the leaf page tables can be shared by the overlapping contexts.

4.2 SPACE on SPARC

An implementation of SPACE involves providing the SPACE mechanisms described earlier:

- Translation of hardware exceptions into generalized exception and parameter pairs *(X,v)*.
- Mapping of *(X,v)* to portals and implementation of the desired set of portal types.
- Implementation of mechanisms to allow different address spaces to execute different instances of the generalized exception and portal type mechanisms.

4.3 A SPARC Implementation

The SPARC architecture vectors hardware exceptions through a table pointed at by the *%tbr* register. The SPACE generalized exception mechanism is implemented on the SPARC by code that begins execution in the trap vector and ultimately branches to a code sequence specific to the generalized exception. For convenience we will refer to this final code sequence as the *GE-tail*. In the case of generalized exceptions which are not synthesized, the GE-tail begins at the hardware trap vector and the mechanism entails no overhead.

The portal mapping mechanism begins execution in the GE-tail. The generalized exception, *X*, is defined implicitly by the particular GE-tail that is executing. The *v* parameter (if needed) is provided explicitly in a register. For most exceptions the GE-tail can find the portal by indexing a table by *v*.

If *v* is a virtual address, a more involved mechanism than table lookup is required. We use a tree structure for mapping virtual addresses to portals. Each node in the tree corresponds to a range of addresses. A node can either represent a single portal, or point to a vector of portals, each corresponding to an address in the range. The latter mechanism is useful for naming remote system services. Addresses not described by tree nodes are associated with a default portal. There is a separate tree for each generalized exception *X* which uses *v* as a virtual address.

Once *(X,v)* is mapped into a portal, the portal type field is extracted and used to branch to the portal type implementation code (the *PT-code*). If the mapping from *(X,v)* to the portal type is unique, the PT-code begins execution in the GE-tail. If the generalized exception does not need to be synthesized, the GE-tail will begin in the hardware vector table, and thus the portal type code will begin to execute within a few cycles of the hardware detecting the exception.

PT-code can implement arbitrary functionality. On the SPARC we use portal types which implement various forms of cross-domain calls and simulated memory references, as well as conventional exception handlers. We use short-circuit portal types to implement *portal return*. Returning from a generalized exception handler requires that the previous state had been saved by the PT-code that made the original traversal. This state is associated with the new execution in such a way that it can be accessed by the PT-code that implements return.

The data in a portal is interpreted by the PT-code. Typically portals will specify a new protection domain in which to execute a generalized exception handler, but the fields can be given an arbitrary interpretations. For example, portal types implementing I/O handler use the portal to also specify the *processor interrupt level*.

Pointing the *%tbr* register at a different table can completely replace the SPACE generalized exception and portal type mechanisms. Implementation of address specific SPACE mechanisms is simply a matter of reloading *%tbr* when a new address space is entered.

Various internal data structures are used in this implementation. Processor state is saved in Processor Control Blocks (PCBs) which are allocated from a dynamic pool. Descriptors for address spaces point at the tables and trees used to implement the portal lookups. The privilege modes are implemented by indexing into a table of AMU context identifiers in the space descriptor.

Symbiont domains, which share these data structures in memory, are used to implement management functions such as creating portals and supplying AMU context identifiers. Synthesized exceptions are generated to communicate with the symbiont domains. Functionality related to resource management, including AMU access, is implemented as system services in protection domains.

5 Examples

In this section we describe the implementation of two mechanisms which illustrate the extensibility of SPACE.

5.1 Remote Procedure Call

Remote procedure calls extend the procedure call mechanism found in programming languages across a system boundary to access data that is not available in the current execution context. Though originally focussed on accessing remote services across a network [25], RPC is just as useful for implementing client-server mechanisms within a single machine.

Portal traversal is a natural mechanism for implementation of RPC between protection domains. Two types of RPC naturally occur in SPACE:

- invocation of a remote procedure with simple arguments passed in registers.
- invocation of a remote procedure with arguments passed in shared memory.

If an RPC cannot pass all arguments in registers, they are passed in shared memory. If the RPC is between two address spaces, the memory must be explicitly shared. For small amounts of data, the argument memory can be statically shared, but dynamically allocated. For large amounts of data, dynamic sharing of data is used, as in the Mach integration of IPC and virtual memory [26].

However, if the RPC domains are within a single address space, the sharing is done by allowing access to memory from both privilege modes. In many cases, where the remote procedure is trusted, the privilege mode of the remote procedure will have access to much of the calling address space, and arguments do not have to be marshaled. The remote invocation is then as efficient as passing simple parameters in registers. The situation is analogous to invoking kernel procedures in conventional operating systems.

The RPC implementation requires choosing/implementing appropriate generalized exceptions and portal types. The most efficient approach maps a specific hardware trap-instruction directly into a generalized exception, which is then mapped directly into a portal type specific to the remote procedure call. The portal type implementation allocates memory to save the current machine state, sets the AMU context to the domain where the remote procedure is to execute, and then uses the hardware return-from-exception mechanisms to being execution of the handler. The saved state is associated with the handler execution, so that when the return-from-exception occurs, the state can be restored.

This implementation is particularly efficient because the mappings are direct because the portal type code begins execution as soon as the exception is recognized by the hardware. However there are only a relatively small number of trap-instructions available, so it may not be possible to always use this approach.

An alternative implementation is useful for accessing remote procedures within the same space, but at a different privilege mode. A generalized exception, *intra-space RPC*, is synthesized when the hardware detects an instruction-fetch page-fault to a valid but inaccessible page. *v* is the address of the inaccessible procedure. *(X,v)* is mapped to a portal by looking up *v*. The portal specifies a portal type implementation which interprets the portal to obtain the privilege mode in which the procedure at *v* is executable. The procedure at *v* is then executed with the new privilege mode.

Use of the address space as a name space avoids conflict over allocation of the user trap-instructions. Additionally the invoking routine can be oblivious as to whether the invoked procedure is directly accessible or in a different domain. Implementation of a service as local or remote is resolved when the application is linked.

The simple form of RPC we have described here is particularly powerful because applications can make implementation trade-offs to suit their requirements. It can be as efficient as the hardware will allow. Microkernel/server implementations of operating systems, such as Mach [14], have suffered from performance limitations due to the overhead of calls to the servers. In SPACE this overhead can be reduced by using specialized portal types and exception mappings.

Even greater improvement is possible by the implementation of operating system servers as shared code and data in a separate privilege domain, but within the address space of each task, essentially making the server into a kernel. However, the server is also still written as an application. Perhaps one of many that occupy privilege modes within an address space.

5.2 Thread Support

The SPACE abstraction does not include an explicit notion of threads. However processor sequencing implicitly defines threads of execution. These implicit threads can be formalized into a threads package that supports thread execution across multiple domains.

To support application-specific threads implementations, a hierarchical allocation of processors is needed. Processors are allocated to thread packages for scheduling quanta. Within each quantum, the thread package can implement its own subquanta, and so on. For many applications, dedication of a set of processors for the entire execution is appropriate, and the processors may be scheduled exclusively by an application-specific thread package for the duration of the application's execution.

In the following example we illustrate a thread implementation for SPACE. This might be one of the abstractions available in a library or be supplied as a customized service by the application developer.

5.2.1 SPACE thread operations

The following thread operations are defined in this example:

- **thread creation operations:**
  - *Thread_id thr_create (Address)* – create a thread which begins execution by explicitly invoking the portal associated with *Address*.
  - *thr_exit (Exit_value)* – exit a thread
  - *thr_join (Thread_id)* – block until a specific thread exits
- **thread block/unblock operations:**
  - *thr_wait (Event_id)* – block until the specified event is signaled (may return prematurely)
  - *thr_signal(Event_id)* – signal that a particular event occurred
- **exception operations:**
  - *thr_portal (Exception_id, Address)* – establish a portal to use when the specified exception occurs. The portal must already be accessible and associated with *Address*
  - *thr_exception (Thread_id, Exception_id, parameter)* – interrupt the indicated thread with the indicated exception/parameter pair

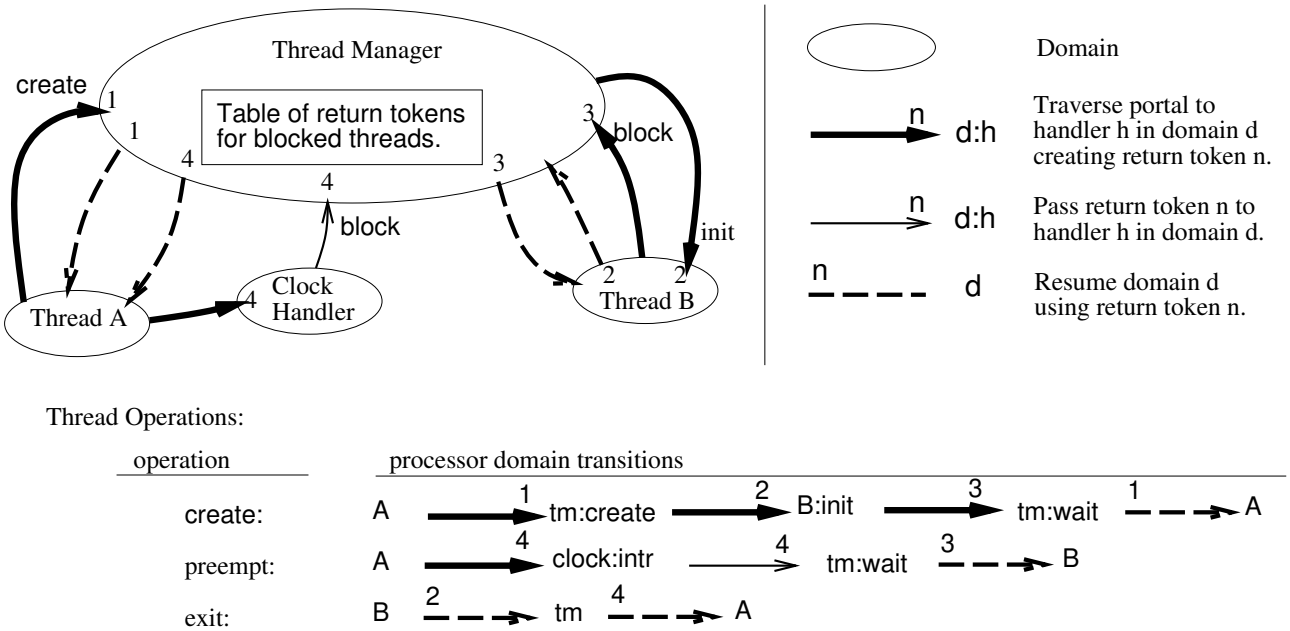
To implement this functionality several special portal types are useful:

- **explicit traversal:** a portal type that receives an *(X,v)* pair as a parameter, and traverses a portal as if the generalized exception *X* had occurred.
- **RPC with return token:** like the intra-space RPC portal type previously described, except that a *return token* is passed which allows an invoking execution to be explicitly resumed.
- **pass return token:** like explicit traversal, except that a subsequent return will return to the *predecessor* of the invoking domain rather than to the invoking domain itself.

- **return with token:** resume the execution specified by the argument *return token*.
- **return:** return to the previous execution.

Figure 4 shows an example illustrating *thread_create* and *thread_exit*, as well as preemption. The diagram shows four protection domains. Thread A uses an RPC to invoke the thr_create operation in the thread manager. *thr_create* is implemented using the *RPC with return token* exception. The new thread is begun by traversing the portal whose address is passed as a parameter (*init* in this example). *thr_create* must execute in the space of thread A in order to evaluate the portal address. (Alternatively a new version of the *explicit traversal* portal type which evaluates the portal address in the correct space could have been introduced).

Figure 3: Threads in SPACE



thr_create saves the return token in the thread table entry for the invoking thread, and links it to the dispatch queue. It then allocates a new thread table entry, and uses *explicit traversal* with an appropriate *X* value, and *v* set to the address of the *init* portal, to begin execution as thread B.

Eventually thread B suspends by invoking *thr_wait*, which traverses a portal of type *RPC with return token* to the thread manager. The thread manager uses the *return with token* portal type to resume the previous thread. The execution state to be resumed is referenced by the return token saved in thread A's thread table entry.

thr_wait is a general operation which, together with *thr_signal* implements functionality similar to the UNIX kernel operations sleep/wakeup.

Preemption is implemented via the clock exception, which occurs in figure 3 while thread A is executing. The handler for the clock exception is specified by a portal. If the scheduling quantum has expired, the clock handler uses the *pass return token* portal type to enter the *thr_wait* portal of the thread manager. This portal type arranges for the return token stored in thread A's thread entry to be used to return to thread A instead of the clock handler, since there is no reason to continue execution in the clock handler for this interrupt. The *thr_wait* handler will then dispatch the most worthy thread onto the current processor, which in this example is B.

thr_exit invokes a portal of type *return* which continues the original *thr_create* handler in the thread manager. The continuation code deallocates the thread entry and checks for outstanding *thr_join* operations. *thr_join* allows one thread to wait for another. Any *thr_join* operation pending on the exiting thread is completed by putting the waiting

thread back on the dispatch queue. Another thread is then dispatched onto the processor, which in the example is thread A.

Two other thread operations are *thr_portal* and *thr_exception*. They implement a mechanism similar to UNIX signals. These operations are very efficient in SPACE because they use the generalized exception mechanisms. *thr_portal* associates the indicated portal with an *(X,v)* pair. *thr_exception* will cause the exception to immediately appear to have occurred in the specified thread. If the thread is not running, it is given the current processor, and the caller of *thr_exception* is blocked. Only asynchronous exceptions can be invoked by *thr_exception*, since the portal types associated with synchronous exceptions may make faulty assumptions about saving state.

To support general software signal delivery, a new generalized exception *X* is introduced, with parameter *v* specifying a signal number. Since a particular *(X,v)* can map to an arbitrary portal type, it is straight forward to implement special signal types which guarantee separate delivery of each example or pass parameters to the handler.

6 Application-Specific Operating Systems in SPACE

SPACE was originally investigated as a set of abstractions for structuring operating systems, using multiple paradigms to support extensibility [1]. What we have described in this paper is a natural extension: embedding operating system services within applications. To support application-specific operating systems, SPACE has evolved by increasing the generality of its paradigms, as well as further lowering the level of its abstractions to provide direct access to the hardware.

There are lot of issues for application developers to consider in using this approach, many of which we are still learning about. However several general principles seem clear already:

- *Use custom operating services sparingly*. Operating system code can be very complex, adding to maintenance costs of the application. Use of customized services diminishes portability, as well as the interoperability of the application with the standard services. Custom services should only be used where they are critical for performance and/or functionality.
- *Flexibility isn't always helpful*. SPACE presents a much larger number of design options, since many constraints have been removed. The application designer must now wade through a much larger design space than is available in most systems. Additionally there can be multiple instances of even the standard system services to choose between. We are working on developing language mechanisms to help manage this complexity.
- *Application programmers aren't operating systems designers*. If embedding systems services requires that application designers become expert at writing operating systems code, the technology will be too hard to use. The embedding of system services needs to happen in the run-time environment of the program rather than through explicit programming in the application. Compiler and language designers need to use these techniques to improve performance by inventing new abstractions where the standard ones aren't sufficient. Also libraries of implementation paradigms need to be developed to enable application designers to utilize new paradigms without having to build them from scratch.

We believe what is needed is a general way to deal with the compatibility and composability of modules in a system. Object-oriented and other language mechanisms provide some help in organizing the potential complexity, but these approaches are not very successful at addressing issues such as concurrency, data sharing, capabilities and protection, distribution of computation, and resource sharing.

We are working toward a general design methodology for specifying the interactions of modules across their interfaces, which we call *Interfaces*.

7 Summary

This paper has presented an approach to implementing application-specific operating system services which allows performance and functionality goals to be achieved at the most fundamental levels of the system, while still preserving resource sharing and protection — though now, only in the measure needed by the application. No longer are application designers constrained by the ability of the operating system designer to anticipate application requirements.

In particular, we have removed virtualized resources as the fundamental abstraction presented to the application developer. Instead SPACE supports spaces, domains, and portals as structuring mechanisms, which can be used to implement system services in multiple protection domains.

Generalized exceptions and portal types in SPACE allow application-specific services to exploit the hardware at the lowest levels, even redefining the set of generalized exceptions and portal types associated with each address space, though we are currently researching how to do this safely.

The ability to redefine the fundamental SPACE mechanisms also represents an opportunity to do research on new hardware architectures for exception mechanisms. For the purpose of evaluating their usefulness, new exception mechanisms can first be implemented by synthesizing them in the generalized exception model of SPACE.

The key to SPACE is the exploitation of protection domains to structure the operating system in terms of multiple kernels. We have taken this concept further than other projects. By exposing the hardware interface we have made domains usable for implementing even the most basic operating system services. By extending privilege modes, SPACE comes close to obsoleting the monolithic single-kernel approach to structuring operating systems. Virtual machines can be constructed for running entire conventional operating systems inside a SPACE protection domains. Multiple versions of services can be readily supported. This wide ranging extensibility suggests SPACE is potentially a valuable platform for operating system research. Instead of constructing systems from scratch, or heavily modifying the supported paradigms in an existing system, a SPACE system can incrementally evolve for the purpose of doing experiments. New versions of standard paradigms are added at the level of applications.

8 Future Directions

There is a need to apply language mechanisms and compiler technology to make the customizing of operating system services by application programmers feasible. We are working on providing language support for building applications in terms of spaces, domains, and portals, as part of the *Interfaces* project.

We are also working to provide a safe environment for specifying and compiling specifications for generalized exceptions and portal types. This support is needed if applications are to be able to modify the system at these levels without compromising resource sharing and protection for other applications using the same system.

The *SPACE* abstraction of hardware architecture provides a model for building operating systems, but the approach would be hard to adopt if it required first building an operating system from scratch. Thus we are bootstrapping the functionality of *SPACE* by implementing a conventional operating system in a protection domain within SPACE. Particular issues we are addressing include the sharing of physical resources (e.g. physical memory, AMU contexts, and the cpu) between native SPACE applications and the innocent conventional operating system. Native applications need to gain access to conventional functionality without compromising their ability to exploit the hardware at low levels.

Other future work includes examining SPACE implementation on SPARC Version 9, as well other processor architectures, including those which use software to fetch missed TLBs. We are also applying SPACE to multiprocessors and scalable parallel computers, particularly for the purpose of using SPACE to implement very high-performance applications such as parallel machine communication primitives and high-speed networking.

References

[1] Dave Probert, John Bruno, and Murat Karaorman. Space: A new approach to operating system abstraction. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 133–137, October 1991.

[2] T.E. Anderson, H.M. Levy, B.N. Bershad, and E.D. Lazowska. The interaction of architecture and operating system design. In *ASPLOS IV*, pages 108–120, April 1991.

[3] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *ASPLOS IV*, pages 96–107, April 1991.

[4] Thomas. E. Anderson, Brian N. Bershad, Edward D. Lazowska, , and Henry M. Levy. Scheduler activations: Effecive kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[5] Brent Welch. The file system belongs in the kernel. In *USENIX Mach Symposium*, pages 233–250, Monterey, November 1991.

[6] Berny Goodheart and James Cox. *The Magic Garden Explained*. Prentice Hall, 1994.

[7] Henry Massalin. *Synthesis: an efficient implementation of fundamental operating system services*. PhD thesis, Columbia University, 1993.

[8] Adrian Nye, editor. *Volume 0, X Protocol Reference Manual*. The X Window System. O'Reilly & Associates, Sebastopol, CA, 1992.

[9] Michael J. Spier, Thomas N. Hastings, and David N. Cutler. A storage mapping technique for the implementation of protective domains. *Software-Practice and Experience*, 4:215–230, 1974.

[10] Michael D. Schroeder and Jerome H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, March 1972.

[11] Lawrence J. Kenah and Simon F. Bate. *VAX/VMS Internals and Data Structures*. Digital Press, Bedford, Massachusetts, 1984.

[12] E.W. Dijkstra. The structure of the "the"-multiprogramming system. *Communications of the ACM*, 11(5):341–360, May 1968.

[13] Elliott I. Organick. *A Programmer's View of the Intel 432 System*. McGraw Hill, 1983.

[14] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: a new kernel foundation for unix development. In *USENIX Conference Proceedings*, pages 93–112, Atlanta, Summer 1986.

[15] J.S. Chase, H.M. Levy, M.J. Feeley, and E.D. Lazowska. Sharing and protection in a single-address-space operating system. *Communications of the ACM*, 12(4):271–307, November 1994.

[16] Graham Hamilton and Panos Kougiouris. The spring nucleus: A microkernel for objects. In *USENIX Summer Conference Proceedings*, pages 147–160, June 1993.

[17] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, July 1993.

[18] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *ASPLOS VI*, October 1994.

[19] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994.

[20] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Second USENIX Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[21] Brian Bershad, Craig Chambers, Susan Eggers, and et. al. SPIN - an extensible microkernel for application-specific operating system services. Technical Report 94-03-03, Dept. of Comp. Sci. and Eng., University of Washington, Seattle, February 1994.

[22] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole Jr. Aegis: A secure programmable exokernel. In *Proceedings of the Sixth European SIGOPS workshop*, Wadern, Germany, September 1994.

[23] SPARC Technology Business. *SuperSPARC & MultiCache Controller User's Manual*. SUN Microsystems, April 1994.

[24] SPARC International, Inc. *The SPARC Architecture Manual – Version 8*. Prentice Hall, 1992. Appendix H.

[25] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[26] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, pages 63–76, November 1987.