SPACE: A New Approach to Operating System Abstraction

D. Probert
probert@cs.ucsb.edu

J. L. Bruno
bruno@cs.ucsb.edu

M. Karaorman
murat@cs.ucsb.edu

Department of Computer Science — University of California at Santa Barbara

Abstract

Object-oriented operating systems, as well as conventional O/S designs, present an overly restrictive level of abstraction to the programmer. Models of objects, processes, concurrency, etc., are embedded within the system in such a way that they are difficult to extend or replace.

SPACE is an extensible operating system being developed for research into object-oriented and distributed systems design. SPACE uses capability mechanisms based on the manipulation of address spaces to provide low-level kernel primitives from which higher-level abstractions can be constructed.

Standard micro-kernel abstractions such as processes, virtual memory, interprocess communication, and object models are built outside the kernel in SPACE, using the SPACE-kernel primitives: spaces, domains, *and* portals. *Multiple versions of the standard O/S abstractions can coexist and interact.*

1 Introduction

This paper provides an overview of the approach being taken in the design of the SPACE Operating System. The description is broken into three sections. First the nature of operating system abstractions is discussed. Next, the particular abstractions used in SPACE are described, and we explain how standard O/S abstractions are built on top of them. In the final section we sketch the building of objects.

2 Operating System Abstractions

Historically, operating systems were developed to allow the sharing of physical resources (processors, memory, devices), insulate programmers from the details of the underlying hardware architecture, and provide for system integrity and protection. The primary strategy used in O/S design is to replace the low-level mechanisms available in the hardware by a uniform set of higher-level abstractions (e.g. processes, virtual memory, filesystems). The integrity of the system is maintained by using memory management to isolate the system data structures and reserving the use of certain hardware functions to the operating system.

As the functional demands on computer systems have increased, operating systems have provided increasing functionality. Loaders became compilers, libraries and dynamic linkers. Filesystems acquired database mechanisms. Interactive computing evolved, changing job control languages into command interpreters. Terminal handlers became window systems.

To control the complexity, operating systems, beginning with the THE system[1], were developed as layers of abstractions. As operating systems continued to grow, and become more unwieldy, O/S functionality was divided between a single, central portion (the kernel) and separate (user-mode) processes and programs (e.g. UNIX[2]).

The choice of which abstractions to place in the kernel is a fundamental one. Normally only a single abstraction of each type is supported, and the abstractions are generally difficult to extend or replace. Attempting to extend a kernel abstraction to user-mode (e.g. Mach external pagers [3]) encounters significant constraints and inefficiencies.

3 Abstractions in SPACE

The SPACE operating system is taking a new approach to O/S abstractions by moving the standard O/S primitives outside the kernel, and implementing them in terms of the SPACE primitives: **spaces**, **domains**, and **portals**.

Spaces

Spaces provide the address mapping for *processor* addresses. Each *space* specifies the translation of pro-

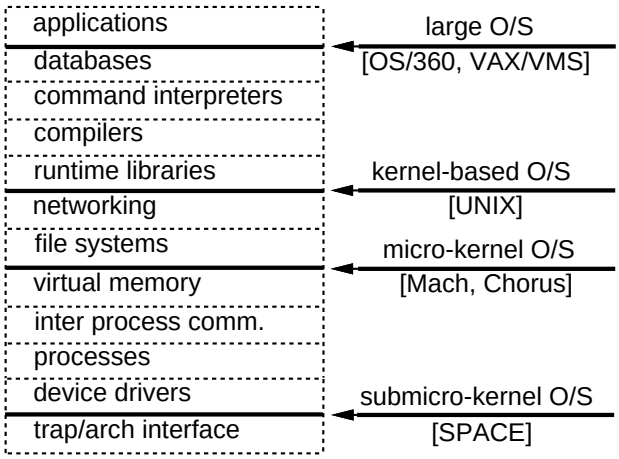

Figure 1: The diagram shows different types of O/S abstractions. The arrows indicate locations where the boundary between kernel and user-mode implementation is made for various systems.

cessor addresses into bus addresses and *portals*. Bus addresses give the processor access to physical memory or i/o. Portals are generalizations of traditional pagefault handling mechanisms, and are described below.

A space may map a given processor address into both a bus address and a portal. Whether the bus address mapping or the portal mapping is used is not determined by the space alone, but depends on the protections associated with the bus address mapping, and the type of access being made.

SPACE uses a protection scheme which qualifies each space into a set of *domains*.

Domains

SPACE extends the conventional protection mechanism used when mapping addresses. Instead of providing a single pair of valid and writable bits for each mapping, SPACE provides bit-vectors, which are indexed by the *protection-id* associated with the processor.

The qualification of a *space* by a *protection-id* is called a *domain*. For a given type of access, the current domain determines a mapping from each processor address into either a bus address or a portal.

Many systems have supported limited mechanisms similar to domains. The page table entries used in VAX/VMS support four sets of page protection bits, which have effect depending on whether the processor mode (analogous to the protection-id in SPACE) is user, supervisor, executive, or kernel. Most other systems (including VAX/UNIX) support at least protection bits for user and kernel. However, the different sets of protection bits are usually hierarchical. For example, user write-access implies kernel write-access.

Domains are more general in both the number of sets of protection bits, and the lack of a hierarchical relationship. Consider a set of domains within the same space (i.e. domains for which only the protection-ids vary). Each domain can have data which is only accessible when the processor is executing in that domain, or data which is accessible from all domains, or in general, data for which only some set of the other domains have read and/or write access. The inter-accessibility of the data between domains is determined by the bit vector of protections associated with each bus address mapping in the common space. The level of access control provided by this facility is particularly useful for building protected objects, as will be discussed in the final section.

Portals

Portals are the mechanism which allow processor execution to switch domains. Whenever processor execution encounters a trap (or fault or interrupt or exception), the SPACE kernel executes briefly to lookup the portal corresponding to the trap (portals are associated with addresses in the case of pagefaults), and then the processor continues execution in the domain specified by the portal.

In addition to specifying the new domain (space and protection-id), portals specify an address at which to begin execution. (Portals also have a set of flags which govern interaction with architectural details such as interrupts).

Because interrupts are mapped into portals, device drivers in SPACE are built outside the kernel within domains. Due to the nature of interrupts, their portals must always be available. Thus interrupt portals belong to a special class of *global portals* which are the same in all domains. Exceptions which are related to the execution of the processor (e.g. floating-point-underflow), are mapped into portals that are specific to each domain.

Most hardware architectures support explicit trap instructions, which can be used to enter the operating system from a user program. In SPACE most trap instructions are mapped into portals. This facilitates the implementation of system calls that are executed within domains, rather than within the kernel. However a few of the trap instructions are reserved for implementing calls into the SPACE kernel. These *ker-*

nel calls are used to provide explicit access to the two basic kernel primitives.

3.1 SPACE Kernel Primitives

There are two primary operations supported by the SPACE kernel: *portal_entry* which is used to move between domains, and *resume_pcb* which resumes execution at a point where a portal_entry had occurred. Portal_entry can be implicit (e.g. a pagefault or i/o interrupt), or explicit through a kernel call. Resume_pcb is always an explicit operation.

When a processor switches domains by passing through a portal, the code sequence that was previously running usually needs to be resumed at some time. This requires that the processor state be saved. The SPACE kernel saves the processor state, as well as the identity of the current domain, in a data structure called the *processor control block* (pcb). A new pcb may be allocated for processor execution at every portal_entry, but it is only used if the code sequence in the new domain also enters a portal.

Each pcb has a unique *token* associated with it. The new domain is recorded in the pcb as the owner of the token, so that only code sequences running in the new domain can use the token to do a *resume_pcb*.

The code sequence executed due to a portal_entry is provided a set of parameters. These parameters vary, depending on the type of portal_entry, but always include the pcbtoken and previous domain. The parameters can also include the arguments to an explicit portal_entry, which is useful for implementing system calls.

Resume_pcb is the operation that reverses a portal_entry, resuming an interrupted code sequence. The pcbtoken passed as a parameter identifies the pcb from which the kernel restores the processor state and domain. Other parameters influence details such as whether the current instruction is retried, and how to supply read data.

The pcb associated with the code sequence calling resume_pcb is deallocated, and the pcb belonging to the resumed sequence is marked so that it's token is no longer valid.

It will sometimes happen that a processor will pass through one portal, initiating a code sequence that decides the trap really needs to be dealt with by yet another portal (e.g. a floating-point trap deciding to invoke a generic signal handling mechanism, or a clock interrupt deciding to invoke a processor scheduler). Rather than require that each code sequence in the chain be resumed, SPACE provides a variant of portal_entry called *pass_pcb*. *Pass_pcb* allows ownership of a pcb to be passed through a portal. The pcb for the code sequence that passed the pcbtoken is deallocated, and the new domain is recorded as the owner of the original pcbtoken.

3.2 Building Standard Abstractions

Standard operating system abstractions, such as processes, virtual memory, interprocess communication, etc., are implemented outside the kernel in SPACE. They are built on top of the kernel primitives spaces, domains and portals. Different implementations of these abstractions can coexist and interact. As illustrations, we describe some of the ideas behind possible implementations of single-threaded unix-like processes and virtual memory.

Processes

The model of a process we are using in this illustration requires an addressing environment, an execution thread, access to system calls, implementation of signals, and process scheduling.

Each process is uniquely associated with a space. The space contains one domain in which all of the process' text, data, and stack are accessible, and in which normal (user-mode) execution of the process takes place. Any reasonable register is chosen as the stack pointer. Other domains within the process' space are used to implement system calls. The process' text, data, and stack are accessible from the system call domains.

All addresses within the process' text, data, and stack map into a portal that executes the pagefault code within the virtual memory implementation. (The pagefault portal is only invoked when the hardware reports a fault on the MMU.)

All other addresses (not reserved for the UNIX system) map into a portal to the signal implementation. The signal implementation has its code and data in memory that is accessible from one of the system domains within the the process' space. Since the system domains can access the process' stack, the signal implementation can prepare the stack to run the process' signal handler. The signal handlers themselves are invoked through portal_entry, and returning from a signal is accomplished with resume_pcb.

Preemption of processes is arbitrated by the clock interrupt portal. If the code sequence on the other side of the clock portal determines that the process' cpu quantum has expired, it passes the pcbtoken for the process through a scheduler portal. The sched-

uler saves the pcbtoken, and then resumes a previously saved pcb on the current processor.

Virtual Memory

Implementing virtual memory requires managing physical resources, such as memory pages, swap disk space, and MMU contexts. The VM code runs both on demand, in response to pagefaults, and at regular time intervals, i.e. to age pages.

The VM implementation is given its own space, consisting of a primary domain, plus what other domains are needed to provide access to other system services. The pagefault portals in the spaces of UNIX processes switch to this domain. The clock also periodically enters the page-aging portal, which gives the VM an opportunity to run code that examines reference bits on pages.

The SPACE kernel views the MMU as a set of registers that can be made to point at the mappings for different domains simply by storing the right set of values. The kernel keeps a cache of these values, but invokes a global *MMU context fault* portal if it tries to switch to a domain for which it doesn't have the context values.

The context fault portal points to a code sequence in the VM domain (whose MMU context must obviously never be deleted). The VM code uses a kernel call to load the missing MMU context values into the kernel.

A single process can have multiple VM implementations associated with different regions of its addresses, simply by using different pagefault portals for each region. This allows new VM implementations to be experimented with in an incremental fashion. The SPACE mechanism for extending VM differs from the external-pager approach in Mach, in that the entire VM system can be supplanted by a new one.

Sharing of data between processes is a common feature of VM systems, and SPACE VM implementations may also share data (i.e. between spaces). However there are applications of shared memory where the conventional VM approach is inadequate because dynamically mapping/unmapping of shared data incurs too much overhead. We look at an example of this in the final section, where we consider the passing of data to protected objects.

4 SPACE Objects

As with the standard O/S abstractions, like processes and virtual memory, there is no object abstraction within the SPACE kernel. Instead a variety of object models can be supported within the system at the same time.

We believe that the underlying abstractions of SPACE, together with its inherent extensibility, provide some special facilities that make it possible to resolve several important issues regarding the development of object models.

4.1 Operating System Conformance

Some earlier work on implementing concurrency in the Eiffel programming language[4][5] has demonstrated the difficulty of interfacing object models with the conventional abstractions of process, virtual memory, and interprocess communication available in operating systems without explicit object support. On the otherhand, explicit object support embeds a particular object abstraction into the system, making the system perhaps no more valuable for experimenting with new object paradigms than a non-object operating system.

One of the reasons for building SPACE is to be able to modify operating system abstractions to better conform to the requirements of OOPLs. The implementation of concurrent objects requires more control over processor scheduling and interprocess communication and synchronization than is available in conventional systems. Within SPACE the interface provided by the O/S to the OOPL can be tailored to fit by modifying, extending, and replacing inadequate abstractions.

4.2 Methodical Attributes

The portal abstraction provides a potentially useful tool for the implementation of OOPLs. In SPACE, methods that take no arguments, and return a single value can be replaced by accesses to addresses that are mapped into portals. The portals result in execution of a method associated with the object, yet the invoking program can treat the method invocation as a pointer reference.

One use of this mechanism is in synchronizing with active objects. Consider a program which invokes a method that will run concurrently in an active object. The method returns immediately, providing a pointer to the result. This pointer initially points to an invalid address, but when the active object completes execution of the method, it will arrange for the address in the pointer to become valid, pointing at the result.

Meanwhile, the original program continues execution. If it accesses the pointer before the active object finishes, the address will be invalid and the program

will enter a portal designed to block until the active object is done.

4.3 Protected Objects

Protection is an important issue in the implementation of object oriented systems. Many OOPL implementations (e.g. Eiffel[6] and C++[7]) provide protection only through the compilers. However in building larger systems it is necessary to use the architectural features of the hardware to control access to objects. We refer to such objects as *protected objects*. Protected objects contain private data whose integrity is to be guaranteed by the hardware (MMU) rather than by the compiler.

A problem that arises with protected objects, is how to pass them unprotected objects in an efficient way. Copying objects is not very efficient unless they are of very small size. Statically mapping all unprotected objects to be accessible to each protected object is likely to overflow the available address size. Dynamically mapping/unmapping the unprotected objects incurs too much overhead.

The solution in SPACE is to statically map protected objects into different domains within each space that contain unprotected objects which reference them. The unprotected objects have their protections set so that they are accessible by all domains within the space, whereas the private data of each protected object is only accessible within the domain assigned to that object. The methods of the protected objects are accessed via portal_entry, which switches to the protected domain. The unprotected objects are accessible from the unprotected domain, and thus can be passed to protected methods by pointer reference.

Acknowledgements

The original notions for SPACE were inspired by the ideas about associating general semantics with segments in Clouds/Ra[8]. (Clouds also helped inspire the name *SPACE*). Domains were developed as a solution to object copying in Clouds. The use of virtual memory to provide capabilities is similar to the implementation of the MCP (O/S) on the segmented architecture of the Burroughs B5500/6700, and from ideas developed later in the Intel i432.

We first heard of micro-kernels (though not by that name) a decade ago in a 4.2BSD futures talk. Mach, Chorus, and Clouds/Ra are contemporary systems that successfully exploit the idea. A talk on 4.2BSD also suggested the name *portals* which in that context were intended to be files that invoked processes when referenced.

References

[1] E. W. Dijkstra, "The Structure of the THE Multiprogramming System," *CACM,* Volume 11, Number 5 (May 1968), pp 341-346.

[2] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *CACM* Volume 21, Number 2 (February 1978), pp 120-126.

[3] A. Tevanian, et. al., "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach," Technical Report, CMU (July 1987)

[4] M. Karaorman, J. L. Bruno, "Introducing Concurrency to a Sequential Object-Oriented Language", Technical Report, Department of Computer Science, UC Santa Barbara, 1990.

[5] M. Karaorman, J. L. Bruno, "Concurrent Programming with Eiffel", Proc. of Ninth International Eiffel User Conference, Santa Barbara, CA, August 1991.

[6] B. Meyer, Object-Oriented Software Construction, Prentice Hall, New York, 1988.

[7] B. Stroustrup, The C++ Programming Language, Addison-Wesley, 1986.

[8] P. Dasgupta, et. al., "The Design and Implementation of the *Clouds* Distributed Operating System," *Technical Report,* School of Information and Computer Science, Georgia Institute of Technology.