U of R C.S. TR 309

Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors

Michael L. Scott
Thomas J. LeBlanc
Brian D. Marsh

University of Rochester
Department of Computer Science
Rochester, NY 14627

scott@cs.rochester.edu
leblanc@cs.rochester.edu
marsh@cs.rochester.edu

March 1989

ABSTRACT

Scalable shared-memory multiprocessors (those with non-uniform memory access times) are among the most flexible architectures for high-performance parallel computing, admitting efficient implementations of a wide range of process models, communication mechanisms, and granularities of parallelism. Such machines present opportunities for general-purpose parallel computing that cannot be exploited by existing operating systems, because the traditional approach to operating system design presents a virtual machine in which the definition of processes, communication, and grain size are outside the control of the user. Psyche is an operating system designed to enable the most effective use possible of large-scale shared memory multiprocessors. The Psyche project is characterized by (1) a design that permits the implementation of multiple models of parallelism, both within and among applications, (2) the ability to trade protection for performance, with information sharing as the default, rather than the exception, (3) explicit, user-level control of process structure and scheduling, and (4) a kernel implementation that uses shared memory itself, and that provides users with the illusion of uniform memory access times.

This work was supported in part by NSF CER grant number DCR-8320136, Darpa/ETL contract number DACA76-85-C-0001, and an IBM Faculty Development Award.

Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors

March 1989

ABSTRACT

Scalable shared-memory multiprocessors (those with non-uniform memory access times) are among the most flexible architectures for high-performance parallel computing, admitting efficient implementations of a wide range of process models, communication mechanisms, and granularities of parallelism. Such machines present opportunities for general-purpose parallel computing that cannot be exploited by existing operating systems, because the traditional approach to operating system design presents a virtual machine in which the definition of processes, communication, and grain size are outside the control of the user. Psyche is an operating system designed to enable the most effective use possible of large-scale shared memory multiprocessors. The Psyche project is characterized by (1) a design that permits the implementation of multiple models of parallelism, both within and among applications, (2) the ability to trade protection for performance, with information sharing as the default, rather than the exception, (3) explicit, user-level control of process structure and scheduling, and (4) a kernel implementation that uses shared memory itself, and that provides users with the illusion of uniform memory access times.

1. Introduction

The future of high-speed computing depends on parallel computing, which in turn is limited by the scalability and flexibility of parallel architectures and software. For the past five years, we have been engaged in the implementation and evaluation of systems software and applications for large-scale shared-memory multiprocessors, accumulating substantial experience with scalable, NUMA (non-uniform memory access time) machines. Based on this experience, we are convinced that NUMA multiprocessors have tremendous potential to support general-purpose, high-performance parallel computing. We are also convinced that existing approaches to operating system design, with a model of parallelism imposed by the operating system, are incapable of harnessing this potential.

With the advent of multiprocessors, parallelism has become fundamental both to the programmer's conceptual model and to the effective use of the underlying hardware. The definition of processes, the mechanisms for communication and scheduling, and the protection boundaries that prevent unwanted sharing can no longer be left to the sole discretion of the operating system. Since these concepts lie at the core of traditional operating systems, shared-memory multiprocessors require a radically new approach, one that provides the user with an unprecedented degree of control over parallelism, sharing, and protection, while limiting the operating system to operations that must occur in a privileged hardware state.

We have designed and implemented an operating system called Psyche that embodies this new approach. Our design incorporates innovative mechanisms for user-level scheduling, authorization, and NUMA memory management. Its implementation provides the foundation for an ambitious project in real-time computer vision and robotics, undertaken jointly

with the department's vision and planning researchers. In the vision lab and elsewhere, Psyche offers a level of support for multi-model parallel computing unmatched by previous systems.

The Psyche design is the direct outgrowth of five years of hands-on experience with multiprocessor systems and applications. This paper traces the evolution of that design, giving the rationale for our major design decisions and describing how the results achieve our goals of user flexibility and efficient use of NUMA hardware.

2. Motivating Experience

The Computer Science Department at the University of Rochester acquired its first shared-memory multiprocessor, a 3-node BBN Butterfly® machine, in 1984. Since that time, departmental resources have grown to include four distinct varieties of Butterfly (one with 128 nodes) and an IBM ACE multiprocessor workstation. From 1984 to 1987, our work could best be characterized as a period of experimentation, designed to evaluate the potential of NUMA hardware and to assess the need for software support. In the course of this experimentation we ported three compilers to the Butterfly, developed five major and several minor library packages, built two different operating systems, and implemented dozens of applications. A summary of this work can be found in [16].

2.1. Architecture

As we see it, the most significant strength of a shared-memory architecture is its ability to support efficient implementations of many different parallel programming models, encompassing a wide range of grain sizes of process interaction. Local-area networks and more tightly-coupled multicomputers (the various commercial hypercubes, for example) can provide outstanding performance for message-based models with large to moderate grain size, but they do not admit a reasonable implementation of interprocess sharing at the level of individual memory locations. Shared-memory multiprocessors can support this fine-grained sharing, and match the speed of multicomputers for message passing, too.

We have used the BBN Butterfly to experiment with many different programming models. BBN has developed a model based on fine-grain memory sharing [26]. In addition, we have implemented remote procedure calls [15], an object-oriented encapsulation of processes, memory blocks, and messages [7], a message-based library package [13], a shared-memory model with numerous lightweight processes [24], and a message-based programming language [23].

Using our systems packages, we have achieved significant speedups (often nearly linear) on over 100 processors with a range of applications that includes various aspects of computer vision, connectionist network simulation, numerical algorithms, computational geometry, graph theory, combinatorial search, and parallel data structure management. In every case it has been necessary to address the issues of locality and contention, but neither

of these has proven to be an insurmountable obstacle.[1] Simply put, a shared-memory multiprocessor is an extremely flexible platform for parallel applications. The challenge for hardware designers is to make everything scale to larger and larger machines. The challenge for systems software is to keep the flexibility of the hardware visible at the level of the kernel interface.

Each of our programming models on the Butterfly was implemented on top of BBN's Chrysalis operating system. We have been successful in constructing these implementations primarily because Chrysalis provides for user-level access to memory-mapping operations, interlocked queues, block transfers, and similar low-level facilities. Unfortunately, Chrysalis imposes a heavyweight process model that cannot be circumvented. Lightweight processes can be simulated with coroutines inside a heavyweight process, but without the ability to invoke kernel operations or to interact with other kinds of simulated processes independent of their peers. In addition, the usefulness of the Chrysalis memory-mapping operations is compromised by their relatively high cost and by the lack of uniform addressing. Finally, Chrysalis provides little in the way of protection, and could not easily be modified to do more without significantly reducing the efficiency of its low-level operations.

2.2. Programming Models

A major focus of our experimentation with the Butterfly has been the evaluation and comparison of multiple models of parallel computing [3,12,16]. Our principal conclusion is that while every programming model has applications for which it seems appropriate, no single model is appropriate for every application. In an intensive benchmark study conducted in 1986 [3], we implemented seven different computer vision applications on the Butterfly over the course of a three-week period. Based on the characteristics of the problems, programmers chose to use four different programming models, provided by four of our systems packages. For one of the applications, none of the existing packages provided a reasonable fit, and the awkwardness of the resulting code was a major impetus for the development of yet another package [24]. It strikes us as highly unlikely that any predefined *set* of parallel programming models will be adequate for the needs of all user programs.

In any environment based on memory sharing, we believe it will be desirable to employ a uniform model of addressing. Even if the programmer must deal explicitly with local caching of data, uniform addressing remains conceptually appealing. It is, for example, the principal attraction of the Linda programming languages [10]. The Linda "tuple space" is not a conventional shared memory. Its operations are not transparent, nor are they efficient enough to be used for fine-grained sharing. The tuple space does, however, allow processes to name data without worrying about their location. In our own experience, BBN's Uniform

[1] Contention in the switching network has not proven to be a problem [21]. Locality of memory references and contention for memory banks are facets of the general memory-management problem for NUMA machines (what we call the "NUMA problem"). We are investigating strategies to address this problem [2,9,14], but they are beyond the scope of this paper.

System [26] is the most popular programming package on the Butterfly for much the same reason. Its single address space allows data items, *including pointers* to be copied from the local memory of one process to that of another without any intermediate translation. The fact that ordinary assignments and variable references can be used to effect the copies makes the global name space even more attractive.

We have found it particularly useful to establish sharing relationships at run time. Low-cost establishment of sharing is also important, as is the ability to share things of arbitrary size. Among the existing environments on the Butterfly, the Uniform System provides the highest degree of support for dynamic, fine-grain sharing — a shared heap in a single address space with a parallel allocator. There is a conflict, however, between protection and performance: data visible to more than one process can be written by *any* process. It may be possible under certain programming models to provide compiler-enforced protection at a very fine granularity with little or no run-time cost. Others have adopted this approach in the context of "open" operating systems [6, 25]. For us to count on compiler protection, however, would be inconsistent with the desire to support as many programming models as possible, particularly with multiple users on a single parallel machine. We are therefore careful to distinguish between the notion of a single address space (which we reject) and that of *uniform* addressing, in which there are multiple address spaces, each containing a potentially different subset of the data in the world, but each individual datum has a unique address that is the same in every address space that includes it.

Clearly any programmer who wants to ignore protection or to provide it with a compiler for a "safe" language should be free to do so, and should pay no penalty for the protection needs of others. When desired, however, the operating system should also provide protection, even if it can only do so with page-size granularity and with the overhead of kernel intervention when protection boundaries are crossed. In order to pay this overhead as infrequently as possible, we believe it will be desirable to evaluate access rights in a *lazy* fashion, so that processes pay for the things they actually share, rather than the things they might potentially share. We therefore distinguish between the distribution of access rights (which must be cheap) and the exercise of those rights.

Other researchers have recognized the need for multiple models of parallel computing. Washington's Presto project [1], for example, supports user-definable processes and communication in a modular, customizable, C++ library package. Presto is not an operating system; it is linked into the code for a single application (written in a single language), and provides no protection beyond that which is available from the compiler. At the operating system level, the Choices project at Illinois [4] allows the kernel itself to be customized through the replacement of C++ abstractions. The University of Arizona's *x*-Kernel [11] adopts a similar approach in the context of communication protocols for message-based machines. Both Choices and the *x*-Kernel are best described as *reconfigurable* operating systems; they provide a single programming model defined at system generation time, rather than supporting multiple models at run time. We are unaware of any project that provides kernel-level protection and run-time flexibility comparable to that of Psyche.

3. Evolution of the Psyche Design

Motivated by our experience with the Butterfly, the Psyche project has from the beginning stressed two fundamental goals: to provide users of the kernel interface with an unprecedented level of programming flexibility, and to permit efficient use of large-scale NUMA machines. Implicit throughout has been the assumption that the kernel must also provide a high degree of protection to those applications that require it.

Original goals:
User flexibility
Efficient use of NUMA hardware
(Protection without compiler support)

These first two goals are complementary. Flexibility is the most significant feature of NUMA

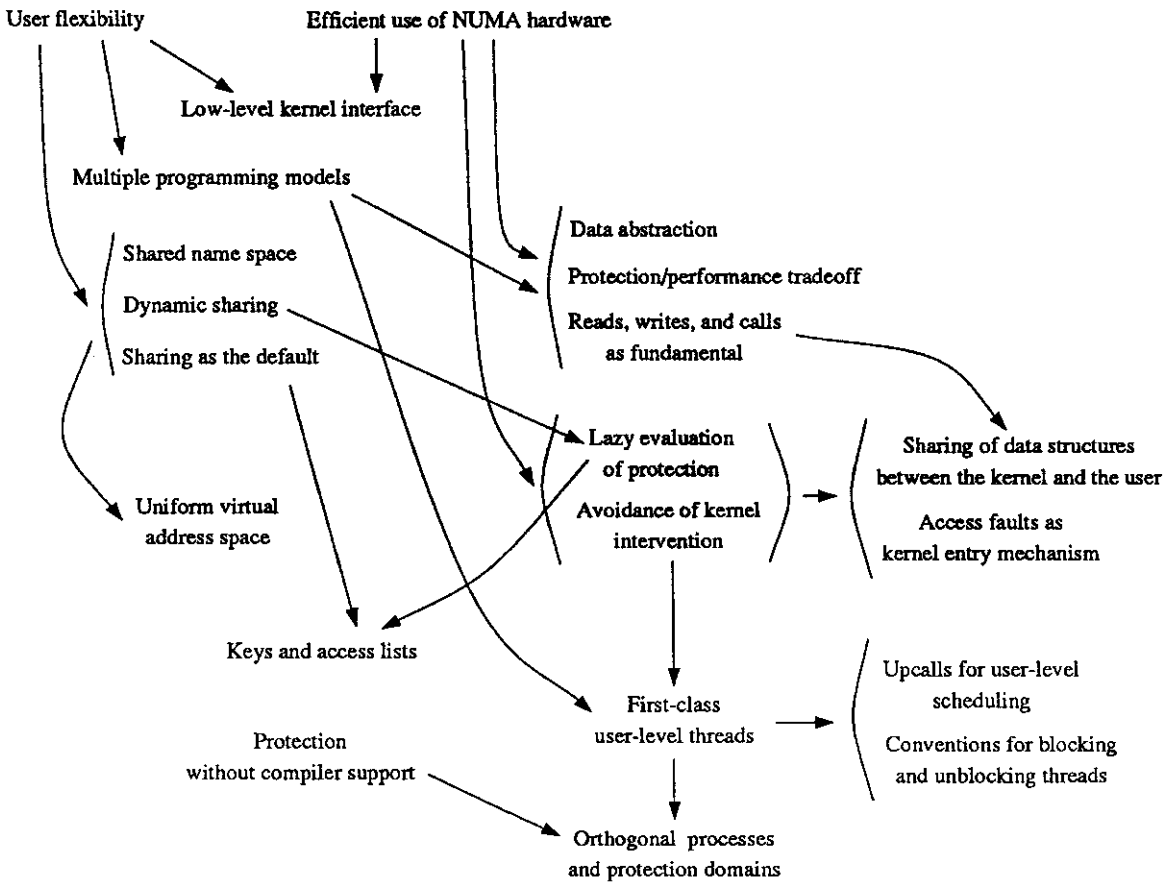
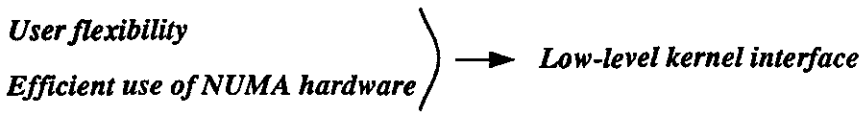


Figure 1: Evolution of Psyche Design Decisions

machines, and the one most difficult to exploit. Our emphasis on flexibility has determined the style of our kernel interface and the abstractions it provides. Our emphasis on hardware exploitation has underlined the importance of efficiency, and has played a dominant role in implementation strategy.

The development of our ideas is summarized in figure 1. Each of the major design decisions visible to the user can be seen as the outgrowth of more basic concepts, tracing back to project goals. The text of this section serves as commentary on the figure, portions of which are reproduced along with the corresponding discussion.



There are two principal strategies by which an operating system can attempt to provide both efficiency and user flexibility. One is to implement a variety of programming models directly in the kernel. This strategy has the advantage of allowing unsophisticated programmers to use the kernel interface directly. It suffers, unfortunately, from a tendency toward the "kitchen sink" syndrome; no small set of models will satisfy all users, and even the smallest deviations from predefined abstractions can be very difficult to obtain (see [7] and [22] for examples). A more attractive approach is to implement a set of primitive building blocks on which practically anything can be built.

We decided very early that Psyche would have a low-level kernel interface. In some sense this places us in the tradition of the "minimal" kernels for distributed operating systems such as Accent [19], Charlotte [8], and V [5]. Our emphasis, however, is different. Message-passing kernels provide a clean and narrow interface at a relatively high level of abstraction. By confining the kernel to lower-level operations, we can cover a much wider spectrum of programming models with an equally clean and narrow interface. (See figure 2).

We do not expect everyday programmers to use the Psyche interface directly. Rather, we expect that they will use languages and library packages that implement their favorite programming models. The purpose of the low-level interface is to allow new packages to be written on demand (by somewhat more sophisticated programmers), and to provide well-defined underlying mechanisms that can be used to communicate between models when desired.

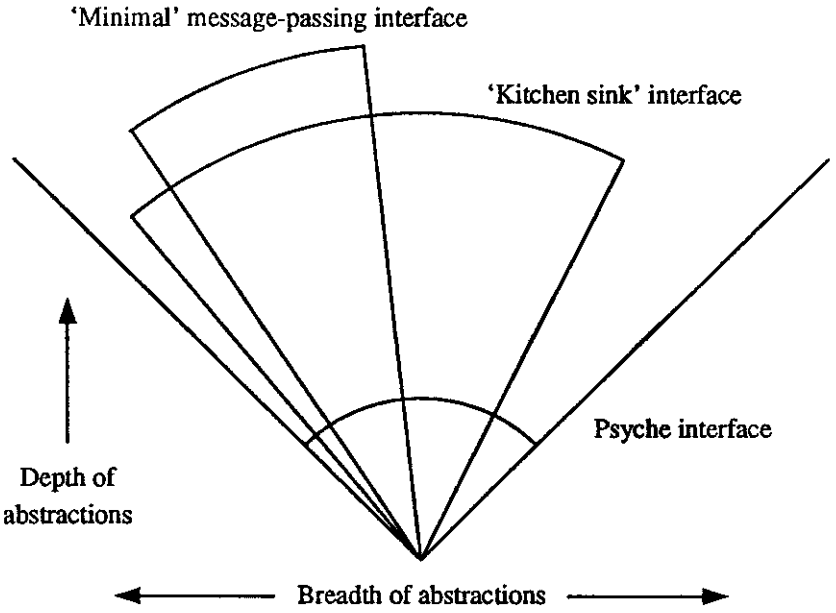
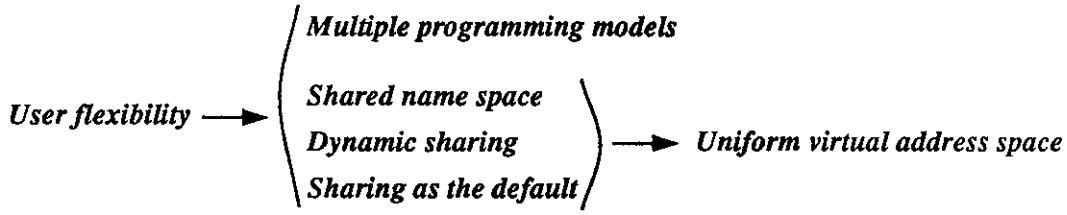
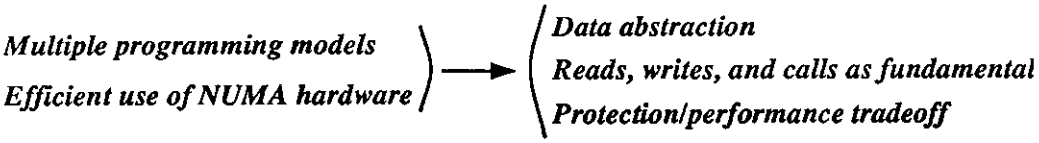


Figure 2: Levels of Kernel Interface



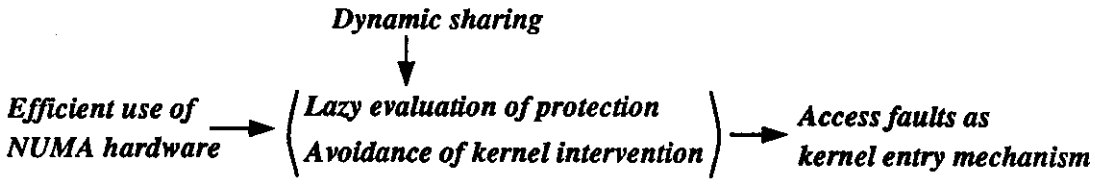
Our commitment to user flexibility is manifested primarily in the need for multiple programming models and in an emphasis on sharing. This emphasis on sharing has in turn led us to establish the convention of uniform addressing. If processes are to share pointers, then any data object visible to two different processes must appear at the same virtual address from each point of view. Moreover, any two data objects simultaneously visible to the same process must have different virtual addresses from that process's point of view. Satisfying these constraints is an exercise in graph coloring. Since the graph may change at run time (as sharing relationships change), the only general solution is one in which every data object has its own color — its own unique address. On existing 32-bit machines, of course, the available virtual address space is likely to be too small to hold all user programs at once, but simple heuristics based on *a priori* knowledge about some of the graph nodes can be used to overlap large amounts of space while maintaining for the user the illusion of uniform addressing. We discuss these heuristics in section 5.3.



If programmers are to implement multiple programming models outside the kernel, they must be provided with tools for building abstractions. This is particularly true if the implementations of different models are to be similar enough in conception to allow their processes to interact. To allow user-level code to execute efficiently, it is also important that the abstraction-building tools reflect the characteristics of the underlying hardware. We have therefore adopted the concept of data abstraction as the fundamental building block in Psyche. Programmers are accustomed to building sequential applications with abstract data types. In our experience with parallel systems, we have found data abstractions (with appropriate synchronization) to be well-suited to parallel applications as well. Not only do they provide a natural means of expressing communication, they also permit an implementation based on ordinary data references and subroutine calls, the most primitive and efficient operations on a shared-memory multiprocessor.

In Psyche, a data abstraction is known as a *realm*. Together with its data, each realm includes a protocol. The intent is that the data should not be accessed except by obeying the protocol. Invocation of the protocol operations of shared realms is the principal mechanism for communication between processes in Psyche. A concurrent data structure (a 2-3 tree for example) can be built from a node abstraction with appropriate read and write locks. A monitor or a module protected by path expressions is by nature a data abstraction; its entry procedures are its protocol. A message channel or mailbox can also be built as a realm. Its protocol operations control the reading and writing of buffers. For connectionless message passing, each individual message may be realized as a realm, "sent" and "received" by changing access rights.

To protect the integrity of realms, the kernel must be prepared to ensure that protocols are enforced. It must not, however, insist on such enforcement. Some applications may not care about protection. Others may provide it with a compiler. Psyche therefore provides an explicit tradeoff between protection and performance. Realms are grouped together into (often overlapping) *protection domains*. Invocations of realm operations within a protection domain are *optimized* — as fast as an ordinary procedure call. Invocations between protection domains are *protected* — as safe as a remote procedure call between traditional heavyweight processes. In the case of a trivial protocol or truly minimal protection, Psyche also permits direct external access to the data of a realm, through *in-line* invocations. In all cases, the Psyche invocation mechanism is an ordinary data reference or a jump-to-subroutine instruction.



A kernel call by its very nature requires a context switch into and out of the operating system. Optimizing this context switch is a common goal of computer architects and kernel designers, but even the most trivial kernel call inevitably costs significantly more than a subroutine call and return. Extremely fine-grain interactions between processes must therefore occur without kernel intervention. The elimination of both explicit kernel calls and implicit kernel operations is important to Psyche's goal of maximal hardware utilization. Furthermore, functionality provided outside the kernel can be changed more easily than functionality inside the kernel, an advantage in keeping with our goal of user flexibility. As in several "minimal" message-based kernels, we are implementing device management, file systems, virtual memory backing store, and network communications in user-level software. We also permit interprocess communication to be implemented outside outside the kernel, along with the bulk of scheduling.

As explained above, we believe that for the sake of efficient dynamic sharing it is desirable to delay the evaluation of access rights as late as possible, and to cache those rights once they have been established. We therefore wait until a process actually attempts to touch a realm before verifying its right to do so. Since a valid access cannot be distinguished from an invalid one until this verification has occurred, initial references to realms appear to the kernel as access faults — invalid reference traps from the address-translation hardware.

Psyche's access-fault handler is the most important kernel entry point. It must distinguish between a large number of possible situations. Like most operating systems, Psyche uses access faults to implement demand paging. It also uses them to implement page migration to maximize locality on NUMA multiprocessors.[2] Both of these functions are semantically invisible. From the point of view of the user-level programmer, there are only three possibilities:

(1) The reference may truly be invalid. Its virtual address may not refer to any realm at all, or it may refer to a realm to which access is not permitted. These two cases are indistinguishable to the user.

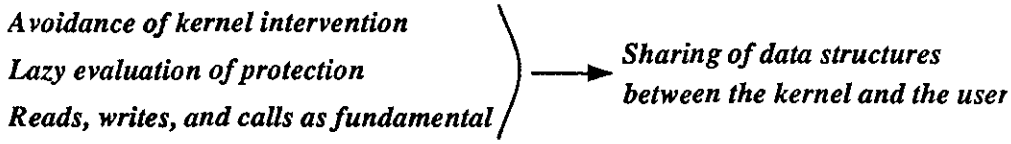
(2) The reference may be valid, but may refer to a realm that has not been accessed before.

(3) The virtual address may refer to a realm that may only be accessed through protected invocations.

[2] The Psyche memory management system is described in a companion paper [14].

Case (2) provides the hook for lazy evaluation of access rights. If optimized or in-line invocations are permitted, the referenced realm is added to the current memory map. We say that the realm has been *opened* for optimized access. The faulting instruction is restarted. Future invocations will proceed without kernel intervention. They will, in fact, be ordinary subroutine calls. If protected invocations (only) are permitted, the kernel arranges for the calling process to *move to another protection domain*, in which it may execute the requested operation. It also makes a note to itself indicating that the realm has been opened for protected access. Future invocations from the same protection domain will fall under case (3) above, and will not cause a re-evaluation of access rights.

In all cases, protected and optimized, invocations are requested by executing ordinary subroutine call instructions (or data references, in the case of in-line invocations). Since protected and optimized invocations look the same, the choice between them can be made simply by modifying access rights (at run time) without rewriting code or recompiling. In effect, Psyche has separated the issues of protection and performance from the semantics of realm invocation. In a program that never attempts to circumvent realm protocols, protection levels can be changed without changing the appearance or behavior of the program.



Psyche places a heavy emphasis on sharing of data structures between the kernel and the user. This sharing is consistent with our desire to perform work only when we know it is necessary, to perform it outside the kernel whenever possible, and to express it in terms of ordinary data access. Shared data structures allow the user to obtain useful information without asking the kernel explicitly. For example, the kernel maintains information (read-only in user space) that identifies the current processor, current protection domain, current process, and time of day. Future extensions might include run-time performance statistics or referencing information of use to user-level migration strategies. In a similar vein, user-level code can provide information to the kernel without performing kernel calls. Shared data structures are used to specify access rights, control scheduling mechanisms and policy, set wall time and countdown timers, and describe the operation interface supported by each realm.

Explicit transfers of control between the user and the kernel occur only when synchronous interaction is essential. Kernel calls exist to create and destroy realms, return from protected invocations, and forcibly revoke rights to a realm that has been opened by another protection domain. In the other direction, the kernel provides *upcalls* to user-level code in response to various error conditions, and whenever a user-level scheduling operation is required. The latter case covers calls and returns from protected invocations, expiration of timers, and imminent end of a scheduling quantum.

Multiple programming models
Avoidance of kernel intervention → *First-class user-level threads*

Many parallel algorithms are most easily realized with a very large number of lightweight processes, or *threads*. For reasons of both semantics and efficiency, it is important that these threads be implemented outside the operating system. No single kernel-provided process abstraction is likely to meet the needs of such diverse languages as Ada, Emerald, Lynx, MultiLisp, and SR. Different programming models include different ideas about where to place stacks (if any), what state to save on context switches, how to schedule runnable threads, and how to keep track of unrunnable threads. Moreover, no kernel is likely to provide the performance of user-level code to create, destroy, block, and unblock threads. Operating systems such as Mach [18] and Amoeba [17] have attempted to reduce the cost of process operations by separating the scheduling abstraction from the address-space abstraction, but the result is still significantly less efficient than the typical implementation of user-level threads.

With a traditional operating system it is always possible to implement lightweight threads inside a single heavyweight kernel process, but the operating system is then unable to treat these threads as first-class entities. They cannot run in parallel and they cannot make use of kernel services (e.g. blocking operations) independent of their peers. Psyche addresses this problem with a novel approach to first-class, lightweight, user-level threads.

First-class user-level threads
Protection without compiler support → *Orthogonal processes and protection domains*

A process in Psyche is an anthropomorphic entity that moves between protection domains as a result of protected invocations. A thread is the realization of a process within a given protection domain. As a process moves among domains, it may appear as many different kinds of threads. In one domain it may be an Ada task. In another domain it may be an Actor or a MultiLisp future. In a simple server domain there may be no recognizable notion of thread at all; processes may be represented over time by the actions of a state machine. The kernel keeps track of which processes are currently in which protection domains, but it knows nothing about how the threads that represent those processes are scheduled inside those domains. It does not keep track of process state. Threads are created, destroyed, and scheduled in user-level code. The kernel assists by providing upcalls whenever scheduling decisions may be required.

To bootstrap Psyche, the kernel creates a single primordial realm in a single protection domain, containing a single user-level process. This process executes code to create additional realms. Creating a realm also implicitly creates a protection domain, of which the created realm is said to be the *root*. Protected invocations of realm operations cause the

current process to move to the protection domain of which the realm is the root. Among the arguments to the **make-realm** operation is a specification of the number of processes in the domain that should be allowed to execute simultaneously. The kernel creates this many *activations* of the domain. Users for the most part need not worry about activations. They serve simply as placeholders for processes that are running simultaneously. In effect, they are virtual processors. From the user's point of view, Psyche behaves as if there were one physical processor for each activation.

On each node of the physical machine, the kernel time-slices between activations currently located on its node. A data structure shared between the kernel and the user contains an indication of which process is being served by the current activation. This indication can be changed in user code, so it is entirely possible (in fact likely) that when execution enters the kernel the currently running process will be different from the one that was running when execution last returned to user space.

To implement a protected invocation of a realm operation (see figure 3), the kernel (1) makes a note that the invoking process has moved to the protection domain rooted by the target realm of the invocation, (2) provides an upcall to some activation of that domain, telling it that the process has moved and asking it to perform the realm operation, and (3) provides an

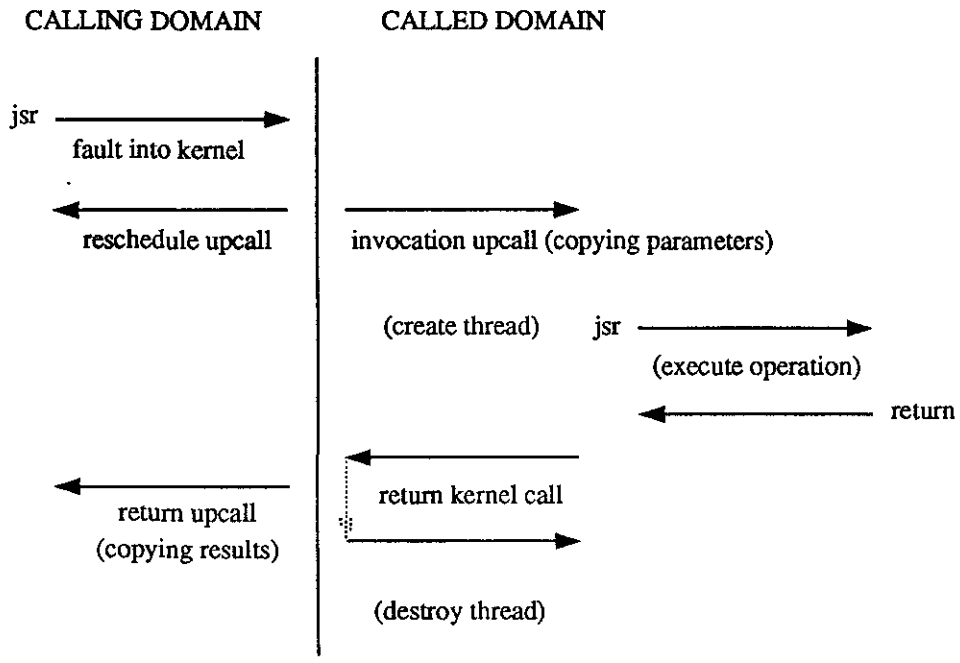


Figure 3: Protected Invocation

upcall to the activation that had been running the process, telling it that the invocation is indeed protected, that the process has moved to another protection domain, and that something else could run in the meantime. The target activation (if multi-threaded) creates a new thread of an appropriate kind to execute the requested operation on behalf of the newly-acquired process. The thread is initialized in such a way that when it returns from the operation it will execute a **return-from-invocation** kernel call. At that point, the kernel will look in its internal data structures to see which domain the process came from, and provide some activation of that domain with an upcall indicating that the process has returned. A simplistic, single-threaded protection domain (similar to a traditional Unix process) can choose to ignore the reschedule and return upcalls. In this case the activation simply blocks until the invocation has completed.

For both the invocation and the return, the kernel copies parameters from one protection domain to the other. It assumes that the parameters can be found at a predefined offset from a predefined register (commonly used as the stack pointer). The sizes of the parameters are defined in the user/kernel data structure that describes the interface of the invocation's target realm. **In** parameters are copied into the upcall stack of the target activation. That activation may choose to copy the parameters into space belonging to the newly-created thread, or it may simply reassign the upcall stack to the thread, so that the kernel's copy operation is the only one required. When the invocation returns, the kernel copies **out** parameters back into the frame of the invoking thread. Reference parameters are treated the same as pointer value parameters; the callee must possess appropriate rights in order to dereference them.

Process names are ordered pairs consisting of the name of the protection domain in which the process was created and a serial number managed by user-level code. When execution enters the kernel, it is therefore possible to distinguish between a newly-created process (created in user space!), a process already known to be in the current domain, or a bogus claim to be a process that cannot be in the domain. The latter case has occurred when (1) the process name indicates that it was created elsewhere, but the kernel has no indication that it moved to the current domain, or (2) the process name indicates that it was created in the current domain, but the kernel has a note indicating that it subsequently moved to some other domain. Otherwise, the kernel takes the word of the user regarding which process is currently running. In fact, the process name variable written by the user constitutes the *definition* of which process is currently running.

First-class user-level threads → { *Upcalls for user-level scheduling*; *Conventions for blocking and unblocking threads* }

To switch between processes in user mode, the user need only change the current process name. In most cases, the user will want to save and restore registers and other state as well (as part of a context switch between threads), but the kernel imposes no such requirements. To permit time slicing, the kernel implements both wall clock and interval timers for each domain activation. Each time the clock ticks, the kernel's interrupt handler decrements the interval timer value in the data structures of the current activation. If the value reaches zero, it provides a timer upcall to the activation. It also provides an upcall if the current time exceeds the activation's wall clock timer value. The interval timer counts down and the wall clock timer is checked only when the activation is actually running. The activation is therefore guaranteed to get an interval timer upcall as soon as the specified amount of actual execution has elapsed, and a wall clock timer upcall no later than the beginning of the next activation quantum after the specified time is reached.

Since realms may be shared (for optimized access) by protection domains whose processes are realized as different kinds of threads, Psyche users are encouraged to follow a convention that permits threads of different kinds to block and unblock each other. Pointers to block and unblock routines can be found in the user/kernel shared data structures for each protection domain. To illustrate the use of these routines, consider a realm that implements a bounded buffer between threads of different kinds. The code for the insert operation will check to see whether the buffer is currently full. If it is, it will look through the data structures of the current protection domain to find the block and unblock routines appropriate for the current kind of thread. It will then write the pointer to the unblock routine into the synchronization data structures of the buffer and call the block routine. When some other thread (possibly of a different kind) removes an element of the buffer, it will examine the synchronization data structures, discover that the first thread is waiting to continue, and call its unblock routine. If the buffer is shared between protection domains that trust each another, the block and unblock routines may be available for optimized invocation. Everything still works, however, if they require protected invocation. Insert and delete operations will still complete very quickly when the buffer is neither full nor empty.

Summarizing the Psyche model, memory consists of a uniform address space divided into realms. Each realm is the root of a single protection domain, which may encompass other realms as well. Processes are anthropomorphic entities that move around between protection domains executing code on behalf of user applications. Processes run on virtual processors called activations, which are created in numbers adequate to provide user-specified levels of true parallelism within protection domains. Execution and context switching of processes by a single activation proceeds without kernel intervention until an attempt is made to access something not in the address space (or *view*) of the current protection domain. The kernel then either (1) announces an error, (2) opens the accessed realm for optimized or

protected invocation, as appropriate, or (3) effects a protected invocation by moving the current process to the protection domain rooted by the accessed realm.

Each realm includes an access list consisting of <key, right> pairs. The right to invoke an operation of a realm is conferred by possession of a key for which appropriate permissions appear in the realm's access list. A key is a large uninterpreted value affording probabilistic protection. The creation and distribution of keys and the management of access lists are all under user control.

When a thread attempts to invoke an operation of a realm for the first time, the kernel performs an implicit *open* operation on behalf of the protection domain in which the thread is executing. In order to verify access rights, the kernel checks to see whether the thread possesses a key that appears in the realm's access list with a right that would permit the attempted operation. Once a realm has been opened from a given protection domain, access checks are *not* performed for individual realm invocations, even those that are protected (and hence effected by the kernel).

Rights contained in access lists include: initialize realm (change protocol), destroy realm, invoke protected, invoke optimized (or in-line), and invoke optimized read-only.

The user/kernel data structure for each thread contains a pointer to the key list to be used when checking access rights. When a fault occurs, the kernel matches the key list of the current thread against the access list of the target realm. Since matching occurs only when realms are opened, any cost incurred will usually be amortized over enough operations to make it essentially negligible. Moreover, we believe that in most cases either the key list or the access list will be short. Our current implementation of the matching operation is based on hashing, and consumes expected time linear in the size of the shorter list. In cases where multi-way matching is expected to be unacceptably slow, programmers have the option of calling an explicit open operation, with explicit presentation of a key.

Psyche keys constitute a compromise between traditional capabilities and traditional access lists, providing most the advantages of both while avoiding their disadvantages. If one imagines an access matrix with rows for protection domains and columns for realms, a key can be associated with an equivalence class consisting of arbitrary entries in the matrix. In a capability system, adding a protection domain to a class requires time linear in the number of realms in the class. In an access list system, adding a realm to a class requires time linear in the number of protection domains in the class. Under both approaches the amount of space required to represent the class is quadratic. With Psyche-style keys, the addition or deletion of protection domains or realms in an equivalence class requires constant time, and the total amount of space required to represent the class is linear. Since the value of a key depends on neither the holder nor on the realm(s) to which it confers rights, it is possible to

(1) possess a key that grants rights to a large number of realms, (2) change the rights conferred by a key without notifying the holder(s), and (3) change the holders of a key without notifying the realm(s) to which the key grants access. Moreover, the use of probabilistic protection allows the operations affecting access rights to occur without the assistance of the kernel. While it is not in general possible to *prevent* a thread from passing its keys on to a third party, we see no way to avoid this problem in any scheme that transfers rights between protection domains without the kernel's help.

One characteristic of the Psyche protection scheme is that keys and access lists control the right to *open* a realm for access from a given protection domain, not the right to access it *per se*.[3] The distinction is moot when distributing new rights, but is very important when revoking rights. Removing a key from a key list or a key/right pair from an access list prevents that key or pair from being used to open a realm in the future. It does *not* prevent continued access to realms that have already been opened. For users who require hard revocation of rights, Psyche provides an explicit **revoke** kernel call. **Revoke** is a potentially expensive operation. To implement it, the kernel keeps track of which keys and access list entries were used to perform each *open* operation.

4. The Psyche Kernel Interface

Figure 4 contains a diagram of Psyche data structures shared between the kernel and the user. The "magic locations" at the left of the figure behave as read-only pseudo-registers. The current-domain and current-activation pointers are changed by the kernel on context switches between activations. By following pointers, both the kernel and user can find any of the data structures associated with a protection domain or activation.

Each realm is represented by a data structure containing its access list and a description of the interface to its protocol operations. The **make-realm** kernel call allocates space for this data structure and for the corresponding protection domain and activation data structures. It is the user's responsibility to allocate the remaining data structures out of the data space of the newly-created realm. The kernel keeps a mapping, invisible to the user, that allows it to find the realm data structure given a pointer to anything inside the realm. This permits it to consult the protocol description when performing a protected invocation.

The separation between activation and thread data structures, with the former pointing to the latter, is purely a matter of convenience. Each activation runs only one thread at a time. We have put into the activation data structure the information we think the user is most likely to want to leave the same when changing between threads (and the processes they represent) and have put into the thread data structure the information we think the user is most likely to want to change when changing threads. To permit the coexistence of

[3] It would be feasible to check access rights on every protected invocation. We have chosen not to do so because (1) it would add a noticeable amount of overhead to every such invocation, and more importantly (2) it would introduce a semantic difference between optimized and protected invocations, something we want very much to avoid.

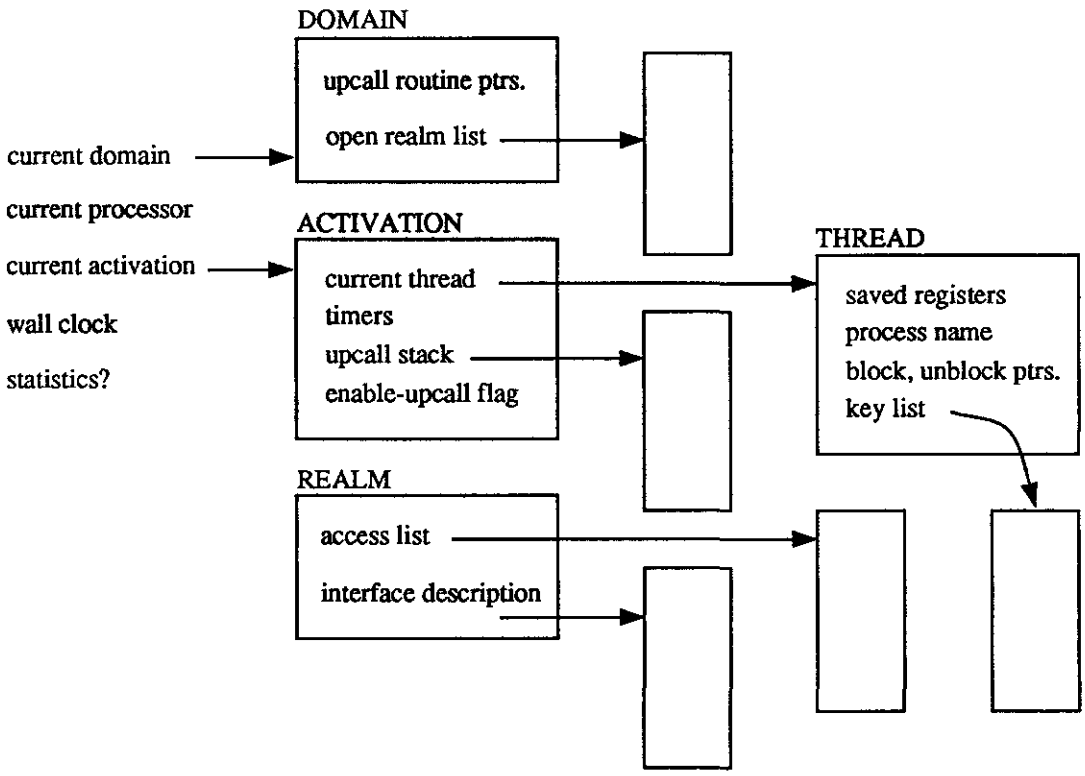


Figure 4: User/Kernel Shared Data Structures

different kinds of threads in a single protection domain, we have chosen to identify block and unblock routines with the thread being executed by the current activation, rather than with the current protection domain.

Complete utilization of the upcall and data structure interface to the kernel requires a sophisticated body of user-level code. Since much of the generality of the interface will not be needed by simple applications, Psyche defines default behavior to cover cases in which some or all of the upcalls are not wanted, or in which pieces of the shared data structures are not provided. A realm that serves as a simple shared memory block, for example, may be accessed solely through in-line invocations. It has no protocol operations of its own. Its protection domain is never used. It has no activations. The only portions of figure 4 that will actually appear are the boxes labeled domain, realm, and access list, and the latter will be trivial. The total space required for kernel/user data structures will be on the order of 50 bytes. Since these data structures are writable in user space, they can share a page with realm data.

Psyche provides the following principal kernel calls:

make-realm — Takes as parameters the desired amount of code and data space, the desired number of activations, and a so-called "master key." Returns pointers to the code space, the data space, and the realm, domain, and activation data structures. The master key can be used by the creator of the realm to initialize both code (which is normally read-only) and data. Program loaders are outside the kernel, and may be integrated with external pagers.

destroy-realm — Takes as parameters a pointer to a realm and a key that should authorize destruction.

return-from-invocation — Takes as parameters the values that would be in the frame pointer and function return registers if we were returning from a subroutine. The kernel can tell where to return to (and what return parameters to copy) by examing the current-process pointer in shared data structures.

block-pending-upcalls — Takes no arguments. Blocks the current activation until an upcall is received.

open-realm — Requires an explicit key. Provided for users who want to avoid implicit searching of key lists.

close-realms — Takes as parameters a list of open realms. The open realm list data structure in figure 4, if allocated by the user, is used by the kernel to list all open realms and the circumstances under which they were opened. Users can use this information to decide when to perform close operations.

invoke-protected — Provided for users who want to make sure they use a protected invocation, even if they have a key that would permit optimized invocation.

revoke — Takes as parameter an access list entry and a pointer to a realm. Forcibly closes the realm in any protection domain that used the access list entry to open it. Requires a key conferring revocation rights.

A small number of additional calls are provided for I/O, external pagers, "ownership," and "attachment." The external pager mechanism is similar in spirit to that provided in Mach [28], but with an interface based on shared memory instead of message passing. The ownership mechanism allows automatic reclamation of realms that are no longer needed. The ownership graph is a DAG; we destroy any realm whose owners have all been destroyed. The attachment mechanism reduces the cost of obtaining access to a multi-realm data structure. If realm B is attached to realm A, then B is automatically opened for access from any protection domain that opens A.

Psyche's kernel-provided upcalls include:

invocation — Provides an indication of the calling realm, domain, and process, the desired operation, and its arguments.

reschedule — Indicates that the current process has moved temporarily to another protection domain.

return-from-invocation — Indicates that a process has returned and may proceed.

interval-timeout, wall-time-alarm — Indicates that a countdown or wall-time timer has elapsed.

end-of-quantum — Indicates that preemption of the current activation will occur after a brief (implementation-dependent) delay. The activation is guaranteed this amount of execution time in which to perform any desired clean-up operations. The end-of-quantum warning permits the use of such mechanisms as spin locks in user-level code, without worrying about untimely preemption.

fault — Indicates that some detectable program error has occurred. Examples include invalid references and arithmetic faults.

Additional upcalls are provided for external pagers and I/O notifications.

Upcalls never return. They are analogous to signals in Unix, except that they use a special stack to avoid interference with the storage management mechanisms used by user-level threads. Before making an upcall, the kernel saves the machine state (registers) of the currently-running thread in the appropriate data structure in user space. Upcalls can be disabled temporarily by setting a flag in the data structure of the current activation. The kernel queues upcalls (other than faults) until they are enabled. A fault that occurs when upcalls are disabled causes the destruction of the current activation.

5. Implementation

5.1. Organization

The Psyche implementation is highly machine independent, and should port easily to different types of shared-memory multiprocessors. To accommodate large-scale parallel computing we have adopted a model of memory that includes non-uniform access times. A Psyche host machine is assumed to consist of *clusters*, each of which comprises processors and memories with identical locality characteristics. A Sequent or Encore machine consists of a single cluster. On a Butterfly, each node is a cluster unto itself. The proposed Encore Ultramax [27] would consist of several non-trivial clusters. Scheduling and memory-management data structures are allocated in the kernel on a per-cluster basis.

For the sake of scalability, each cluster contains a separate copy of the bulk of the kernel code. Kernel functions are performed locally whenever possible. The only exceptions are device managers (which must be located where interrupts occur) and certain of the virtual memory daemons (which consume fewer resources when run on a "regional" basis). To communicate between processors, both within and between clusters, the implementation makes extensive use of shared memory in the kernel. Ready lists, for example, are manipulated remotely in order to implement protected invocations. The alternative, a message-passing scheme in which instances of the kernel would be asked to perform the manipulations themselves, was rejected as overly expensive. Most modifications to remote data structures can be performed asynchronously — the remote kernel will notice them the next time it reads the data. Synchronous inter-kernel interrupts are used for I/O, TLB shootdown, and insertion of

high-priority processes in ready queues.

Since each instance of the kernel must be able to interact with each other instance, scalability dictates that a great deal of address space be devoted to kernel data structures. Since the kernel also shares data structures with the user, the entire Psyche uniform address space must be visible to the kernel as well. No available machine provides enough address space for both of these needs. Psyche therefore employs a two-address-space kernel organization (see figure 5). The code and data of the local kernel instance are mapped into the same locations in both address spaces, permitting easy address space changes. The user/kernel address space also contains all of user space, and the kernel/kernel address space contains the data of every kernel instance. Local data appears at two different locations in the kernel/kernel space. A typical kernel call works in the user/kernel space until it has finished examining user information, then transfers to the kernel/kernel space to obtain access to kernel data on other clusters.

In practice, few data structures are private to a single processor. Most are at least cluster global. In fact, the cases in which a processor enjoys exclusive access to a data structure are so few that we have opted to allow activations to be preempted while executing in the kernel. The same synchronization mechanisms that prevent conflicting accesses from other processors also prevent conflicting accesses by other activations on the same processor. Spin locks (our most widely-used synchronization mechanism) disable preemption in order to ensure that the operation they protect completes quickly.

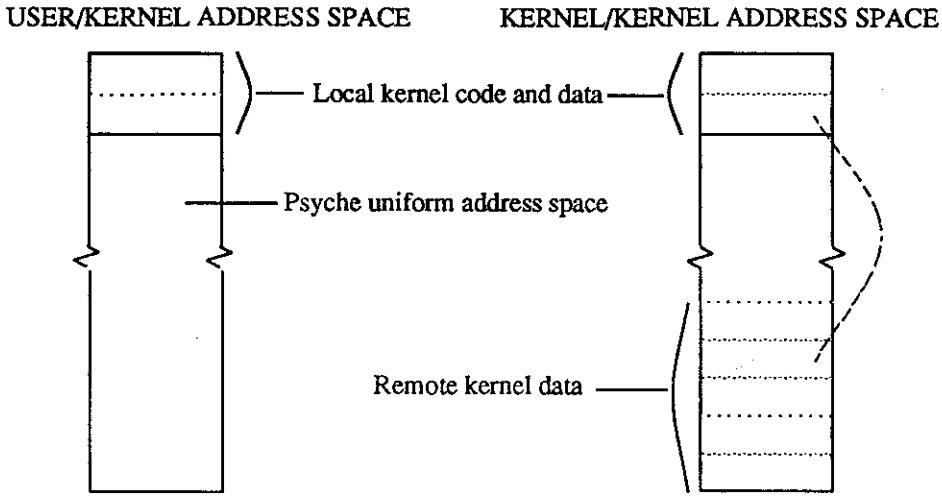


Figure 5: Kernel address spaces

5.2. Memory management

The largest and most sophisticated portion of the kernel is devoted to memory management [14], comprising four distinct abstraction layers. The lowest (NUMA) layer provides an encapsulation of physical page frames and tables. The second (UMA) layer provides the illusion of uniform memory access times through page replication and migration. The third (VUMA) layer provides a default pager for backing store and a mechanism for user-level pagers. The final (PUMA) layer implements Psyche protection domains and upcalls. Page faults may indicate events of interest to any of the layers; they percolate upward until handled.

The PUMA layer maintains a mapping (currently a splay tree) that allows it to identify the realm that contains a given virtual address. This mapping is consulted when a page fault propagates to the PUMA layer, and allows the kernel to determine whether an attempt to touch an inaccessible realm constitutes an error, an open, or a protected invocation. The UMA layer is strictly divided between policy and mechanism. It is not yet clear how best to decide when to replicate and migrate pages, and this division facilitates experiments. There is no notion of location attached to a realm; the placement of its pages is under the complete control of UMA-layer policies. High-quality policies are likely to depend on the judicious use of hints from user-level software.

5.3. Address Space Limitations

Like most modern microprocessor-based machines, the Butterfly Plus employs 32-bit virtual addresses (using the Motorola 68851 memory management unit). The resulting 4 Gigabyte address space is too small to contain all applications. We expect the 32-bit limit to be lifted by emerging architectures. In the meantime, we have devised techniques to economize on virtual addresses.

The **make-realm** kernel call currently accepts a parameter that specifies whether the new realm will be *normal*, *paranoid*, or *private*. A paranoid realm can only be accessed through protected invocations. A private realm can only be accessed through optimized invocations, and only from a single protection domain.

The Psyche uniform address space is divided into four separate areas, two large and two small. One small area holds the code and data of the local kernel. One large area holds the normal realms. All paranoid realms lie on top of each other in a single small area. The private portions of all protection domains lie on top of each other in the remaining large area. Each paranoid realm is represented by a small (conventionally addressable) jump table in normal space that contains information used by the kernel to remap the paranoid portion of the address space when performing a protected invocation. The jump table is inaccessible to every protection domain, so every attempt to access it will cause an access fault. On every context switch between protection domains the kernel also remaps the private portion of the address space.

Above the level of the kernel interface, private and paranoid realms have a limited conceptual impact. They constitute an up-front assertion that optimized sharing will not occur,

which runs counter to the Psyche philosophy. They also require special mechanisms to make them available to user-level pagers.

6. Project Status

Our implementation of Psyche is written in C++ and runs on the BBN Butterfly Plus multiprocessor. We have completed the major portions of the kernel and are experimenting with user-level software.

To facilitate kernel development we have implemented a remote, source-level debugger in the style of the Topaz TeleDebug facility [20]. The front end for the debugger runs on a Sun workstation and communicates via UDP and serial lines with a low-level debugging stub that underlies the Psyche kernel. The low-level debugger was the first piece of the kernel to be written, and has proven extremely valuable.

In concert with members of the computer vision and planning groups within the department, we have undertaken a major integrated effort in the area of real-time active vision and robotics. Our laboratory includes a custom binocular robot "head" on the end of a PUMA robot "neck." Images from the robot's "eyes" feed into a special-purpose pipelined image processor. Higher-level vision, planning, and robot control have been implemented on a uniprocessor Sun. Real-time response, however, will require extensive parallelization of these functions. The Butterfly implementation of Psyche provides the platform for this work. Effective implementation of the full range of robot functions will require several different models of parallelism, for which Psyche is ideally suited. In addition, practical experience in the vision lab will provide feedback on the Psyche design.

Current activity in the Psyche group is focussed on

(1) Cooperation with members of the vision group to ensure that Psyche meets the needs of real-time applications.

(2) Implementation and analysis of alternative strategies for NUMA page migration and replication.

(3) Performance analysis and tuning.

References

[1] Bershad, B. N., E. D. Lazowska, H. M. Levy, and D. B. Wagner, "An Open Environment for Building Parallel Programming Systems," *Proceedings of the ACM/SIGPLAN PPEALS 1988 — Parallel Programming: Experience with Applications, Languages and Systems*, 19-21 July 1988, pp. 1-9. In *ACM SIGPLAN Notices 23*:9.

[2] Bolosky, W. J., M. L. Scott, and R. P. Fitzgerald, "Simple but Effective Techniques for NUMA Memory Management," submitted for publication, March 1989.

[3] Brown, C. M., R. J. Fowler, T. J. LeBlanc, M. L. Scott, M. Srinivas, and others, "DARPA Parallel Architecture Benchmark Study," BPR 13, Computer Science Department, University of Rochester, October 1986.

[4] Campbell, R., G. Johnston, and V. Russo, "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)," *ACM Operating Systems Review 21*:3 (July 1987), pp. 9-17.

[5] Cheriton, D., "The V Kernel — A Software Base for Distributed Systems," *IEEE Software 1*:2 (April 1984), pp. 19-42.

[6] Clark, D., "The Structuring of Systems Using Upcalls," *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 171-180. In *ACM Operating Systems Review 19*:5.

[7] Crowl, L. A., "Shared Memory Multiprocessors and Sequential Programming Languages: A Case Study," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988.

[8] Finkel, R. A., M. L. Scott, Y. Artsy, and H.-Y. Chang, "Experience with Charlotte: Simplicity and Function in a Distributed Operating System," *IEEE Transactions on Software Engineering*, June 1989 (to appear). Extended abstract presented at the *IEEE Workshop on Design Principles for Experimental Distributed Systems*, Purdue University, 15-17 October 1986.

[9] Fowler, R. J. and A. L. Cox, "An Overview of PLATINUM, A Platform for Investigating Non-Uniform Memory (Preliminary Version)," TR 262, Computer Science Department, University of Rochester, November 1988.

[10] Gelernter, D., "Generative Communication in Linda," *ACM TOPLAS 7*:1 (January 1985), pp. 80-112.

[11] Hutchinson, N. C. and L. L. Peterson, "Design of the *x*-Kernel," TR 88-16, Department of Computer Science, The University of Arizona, 9 March 1988.

[12] LeBlanc, T. J., "Shared Memory Versus Message-Passing in a Tightly-Coupled Multiprocessor: A Case Study," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986, pp. 463-466.

[13] LeBlanc, T. J., "Structured Message Passing on a Shared-Memory Multiprocessor," *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988, pp. 188-194.

[14] LeBlanc, T. J., B. D. Marsh, and M. L. Scott, "Memory Management for Large-Scale NUMA Multiprocessors," submitted for publication, March 1989.

[15] LeBlanc, T. J., J. M. Mellor-Crummey, N. M. Gafter, L. A. Crowl, and P. C. Dibble, "Experiences in the Development of a Multiprocessor Operating System," *Software — Practice and Experience*, to appear, 1989.

[16] LeBlanc, T. J., M. L. Scott, and C. M. Brown, "Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor," *Proceedings of the ACM SIGPLAN PPEALS 1988 — Parallel Programming: Experience with Applications, Languages, and Systems*, 19-21 July 1988, pp. 161-172.

[17] Mullender, S. J. and A. S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System," *The Computer Journal 29*:4 (1986), pp. 289-299.

[18] Rashid, R. F., "Threads of a New System," *UNIX Review 4*:8 (August 1986), pp. 37-49.

[19] Rashid, R. F. and G. G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, 14-16 December 1981, pp. 64-75. In *ACM Operating Systems Review 15*:5.

[20] Redell, D., "Experience with Topaz TeleDebugging," *Proceedings, ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 5-6 May 1988, pp. 35-44. In *ACM SIGPLAN Notices 24*:1 (January 1989).

[21] Rettberg, R. and R. Thomas, "Contention is No Obstacle to Shared-Memory Multiprocessing," *CACM 29*:12 (December 1986), pp. 1202-1212.

[22] Scott, M. L., "The Interface Between Distributed Operating System and High-Level Programming Language," *Proceedings of the 1986 International Conference on Parallel Processing*, 19-22 August 1986, pp. 242-249.

[23] **Scott, M. L. and A. L. Cox, "An Empirical Study of Message-Passing Overhead,"** *Proceedings of the Seventh International Conference on Distributed Computing Systems*, 21-25 September 1987, pp. 536-543.

[24] Scott, M. L. and K. R. Jones, "Ant Farm: A Lightweight Process Programming Environment," in *Applications on the Butterfly® Parallel Processor*, ed. Pamela J. Waterman, Pitman Publishing/MIT Press, London, England, April 1989. Also available as BPR 21, Computer Science Department, University of Rochester, August 1988.

[25] Swinehart, D., P. Zellweger, R. Beach, and R. Hagmann, "A Structural View of the Cedar Programming Environment," *ACM TOPLAS 8*:4 (October 1986), pp. 419-490.

[26] Thomas, R. H. and W. Crowther, "The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors," *Proceedings of the 1988 International Conference on Parallel Processing II – Software* (15-19 August 1988), pp. 245-254.

[27] Wilson, A. W. Jr., "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Fourteenth Annual International Symposium on Computer Architecture*, 2-5 June 1987, pp. 244-252.

[28] Young, M., A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron, "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 8-11 November 1987, pp. 63-76. In *ACM Operating Systems Review 21*:5.