Psyche: A General-Purpose Operating System for Shared-Memory Multiprocessors

Michael L. Scott
(716) 275-7745
scott@cs.rochester.edu

Thomas J. LeBlanc
(716) 275-5426
leblanc@cs.rochester.edu

University of Rochester
Computer Science Department
Rochester, NY 14627

July 1987

ABSTRACT

The Psyche project at the University of Rochester aims to develop a high-performance operating system to support a wide variety of models for parallel programming on a shared-memory multiprocessor. It is predicated on the assumption that no one model of process state or style of communication will prove appropriate for all applications, but that a shared-memory machine can and should support all models. Conventional approaches, such as shared memory or message passing, can be regarded as points on a continuum that reflects the degree of sharing between processes. Psyche enables fully dynamic sharing by providing a user interface based on passive data abstractions in a uniform virtual address space. It ensures that users pay for protection only when it is required by permitting lazy evaluation of protection using keys and access lists. The data abstractions define conventions for sharing the uniform address space; the tradeoff between protection and performance determines the degree to which those conventions are enforced. In the absence of protection boundaries, access to a shared abstraction can be as efficient as a procedure call or a pointer dereference.

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.

1. Introduction

Though shared-memory multiprocessors have existed for over 20 years, the design of operating systems for such machines has seldom been the subject of research. For one thing, individual processors have tended to be very few in number, or less than general-purpose. With the notable exception of projects at CMU [27,38], it is only in recent years that multiprocessors have been constructed with relatively large numbers of equally powerful nodes. It is understandable, then, that the parallel operating systems community has for the past decade focused its attention on loosely-coupled systems, in which more-or-less conventional processors exchange messages over a local area network.

With the advent of large-scale commercial multiprocessors, several vendors have adapted the UNIX operating system for use on parallel machines. The IBM RP3 project [33] has adopted a similar approach, with a parallel version of UNIX to be produced by the Ultracomputer group at NYU's Courant Institute [15]. Most message-based operating systems can be implemented on shared-memory machines as well. The Mach project [1] at CMU represents, to a large extent, the merger of Berkeley UNIX with the Accent network operating system. Mach now runs on several multiprocessors, including DEC, Encore, and Sequent machines.

As we see it, the principal danger in adapting an existing operating system for use on a multiprocessor is that it may fail to realize the full potential of the hardware. Though messages and shared memory can each be used to implement the other, the simulation is efficient in one direction only. Implementations of message-passing using hardware-supported shared memory can be every bit as fast as the interprocess messages of a hypercube (for example), but the simulation of shared memory on a fundamentally message-based machine is invariably too slow to use for fine-grained access. The principal advantage of a shared-memory machine is its ability to support fast implementations of a wide variety of models for process and communication structure. Valuable opportunities will be lost if the operating system provides a single model and precludes the use of others.

The Psyche project at the University of Rochester is founded on the conviction that there are many useful paradigms for process interaction, ranging from loosely-coupled message passing to tightly-coupled shared memory. No one of these paradigms will be best for all applications. Our goal is to allow each application, or part of an application, to be written under the model of parallelism most appropriate for its own particular needs.

Psyche is, conceptually, a layered operating system. Its lowest, *kernel* layer provides those operations that must execute in a privileged state in order to provide protection and to access hardware resources. The surrounding *supervisor* layer provides abstractions that facilitate the implementation of communication mechanisms and establish conventions for interaction between pieces of an application that rely on different mechanisms. A pilot implementation is under construction.

This paper discusses the principles on which Psyche is based. Section 2 explains the motivation for our work. It is followed by a description of Psyche primitives (section 3) and implementation issues (section 4). Section 5 contains additional discussion and comparison to other projects. The conclusion summarizes our current status and plans.

2. Motivation

2.1. Shared Memory and Messages

Conventional wisdom holds that parallel processes must communicate either by sharing memory or by exchanging messages. These alternatives are generally viewed as incompatible opposites. It is our contention, however, that conventional approaches are better regarded as points on a *continuum* that reflects the *degree of sharing* between processes. The full spectrum includes many different styles of message passing, as well as monitors, path expressions, remote procedure calls, atomic and parallel data structures, and unconstrained shared memory. In a pure shared-memory approach, processes share everything; in a pure message-passing approach, they share nothing. The other options lie somewhere in-between.

The continuum has not been widely recognized. Parallel programming environments have tended to present a single user view, often one directly supported by the underlying hardware. But a kernel interface is more than just a mechanism for accessing physical resources. It is also a programming abstraction that profoundly influences the algorithms that can be implemented on top of it.

Two years ago, our department acquired a 128-node BBN Butterfly Parallel Processor, still the largest shared-memory machine available, and one that also provides firmware support for message passing. Since then, a major thrust of our work has been the comparison of solutions to common problems under various programming models (see for example [9] and [24]). We are convinced that no one model of parallelism will prove appropriate for all applications. Some algorithms will be easier to implement with fully shared memory. Others are most clearly conceived with message passing. Still others need an intermediate option, such as monitors. Some applications may even benefit from the ability to use different models in different software modules. A computer vision system, for example, may be easiest to construct with shared memory at the lowest levels, where processes are operating in parallel on common pixel maps, and message passing at higher levels, where the emphasis is on feature integration in order to recognize objects.

The need for flexibility in the communication structures of parallel programs is illustrated by an analogy to the information structures of sequential programs. In sequential programming, information can be made available in one of two forms: a data structure that contains the information or a function that computes it. Since either approach can be used to implement the other, the choice depends on the attributes of the application. Information that is hard to compute, but easy to store and access, is encoded in a data structure. Information that is easy to compute, or would require too much space to store, is encoded in a function. A data structure might also be used in situations where the raw data is easy to compute, but the relationship between data items, as encoded in the data structure, may be difficult to recreate. Complex information structures, such as the symbol table in a compiler, often use combinations of both mechanisms.

Message passing is analogous to information exchange via functions, in that both impose a *value-oriented* semantics. Processes may only communicate values, some of which might require the exchange of an environment in which to interpret the value. The implicit communication required to establish an environment will often dominate the cost of interpreting a value within the environment. In the case of

functions, a value-oriented semantics guarantees the absence of side-effects, but requires the environment to be passed as a parameter.[1] As with message passing, the cost of passing the environment as a parameter can dominate the cost of function execution.

Another property shared by message passing and functions is that both offer a form of abstraction. A function computes a value without requiring the caller to know any details of how the value is computed. Similarly, message passing offers a recipient the contents of a message without requiring it to know the details of how the message values were computed, when the message was sent, or what buffering operations were involved.

On the other hand, communication using shared memory is analogous to information exchange via data structures. Each computation (process) has access to the results of previous computations that have been stored (cached) in the shared memory, just as each procedure may have access to previous results stored in global data structures. Computation units (processes or procedures) have reduced fixed overhead, since they can inherit a context implicitly (an address space or a global data structure). There is little abstraction involved since both shared memory and data structure access require the user to have detailed knowledge of the location and format of information.

The analogy between communication structures and information structures is useful because it points out the inadvisability of any attempt to impose a single model of communication on all applications. Sequential programming systems do not attempt to dictate the choice of information structure; they provide functions, data structures, and hybrid combinations. Existing parallel programming systems tend to allow only a single communication structure. Psyche is designed to be more flexible, providing shared memory, message passing, and options in-between.

2.2. Lightweight Process Models

The processes scheduled by an operating system tend to be bulky objects with a large amount of state. Context switching between them is relatively expensive. Though many parallel algorithms are most easily realized with a very large number of processes, the cost of heavyweight context switches (as well as the space required for process state) makes straightforward implementation impossible. Lightweight processes, with a limited amount of explicit state, have been provided by several operating systems, including Mach [1] and Amoeba [31], and by an even larger number of parallel programming languages and library packages. The precise semantics of lightweight processes, however, differ nearly as much from system to system as do the semantics of interprocess communication.

As with IPC semantics, we believe that the choice of a lightweight process model must be left to the writers of individual applications. Certainly an operating system that intends to allow the implementation of LISP futures [20], Ada tasks [40], LYNX threads [36], Emerald objects [8], Modula-2 coroutines [41], and Argus [26] or SR [4] processes cannot insist on the use of a single, fixed model for lightweight process management. Psyche provides a notion of thread that is independent of process weight, and that eliminates

[1] We are assuming *pure* functions that do not have access to an implicit environment. Functions that reference global data are considered a hybrid form of information structure.

the need for kernel intervention when switching between mutually-trusting threads.

2.3. Project Goals

The overall aim of Psyche is to support "general-purpose" parallel computing. By this we mean that the operating system will run almost any application for which the hardware is appropriate, and will usually run it well. We also mean that Psyche will not be a back-end system. In addition to individual, highly-parallel applications, it will support large numbers of users with smaller applications, in the style of conventional time-sharing.

Our work to date has focused on the abstractions provided by Psyche. Their design is an attempt to balance three competing goals:

Flexibility

> It should be possible to implement a wide variety of models for interprocess communication and lightweight process structure. It should be easy for processes using different models to interact, that is to arrange dynamically to share access to arbitrary abstractions.

Protection

> It should be possible to associate a protocol with a shared abstraction in such a way that access to the abstraction is possible *only* by executing the protocol.

Performance

> The cost of a simple operation on a shared abstraction should be much closer to that of a procedure call than to that of a sending a message in current network operating systems.

Protection and performance are rather conventional goals (though the techniques we have adopted to pursue them are not). By contrast, our emphasis on flexibility is somewhat unusual. We are driven by the observation that an operating system kernel must provide a lowest common denominator for the things that will be built upon it. For general-purpose parallel computing, the kernel interface must be fairly low-level. It may not be especially convenient to use in its raw form, but the assumption is that most programmers will never attempt to do so. Instead, they will rely on a collection of standard libraries and language support packages for process management and communication. The operating system may facilitate the construction of higher-level software by promoting the use of conventions (as in the Psyche supervisor), but the lowest-level primitives must remain available. The purpose of the kernel is to provide protection and to hide the most unpleasant idiosyncrasies of the hardware, while leaving the bulk of its power available to the language and library builder.

This conception of the role of the operating system does not appear to have guided most recent research projects. Message-based operating systems, such as Eden [3], Mach [1], and V [11], have tended to provide a kernel interface that is too low-level to be used directly (witness the proliferation of remote procedure call stub generators), yet too high-level to permit alternative approaches to naming, buffering, error recovery, or flow control (we argue this point in [35]). Similarly, most implementations of parallel programming languages have either employed a special-purpose kernel (as in Argus [26], SR [4], StarMod [23], Linda [10], or NIL [37]), or have been built on top of an existing uniprocessor operating system,

most often UNIX. We are unaware of any work specifically addressing the design of a kernel or supervisor to support multiple programming models.

3. Psyche Overview

The kernel layer of Psyche is intended to be minimal. It provides operations to create, destroy, and manipulate three basic abstractions: the segment, the address space, and the thread of control. The rest of this section is concerned with the interface provided by the Psyche supervisor layer. Though this interface is considerably higher level than that of the underlying kernel, we do not expect ordinary programmers to make use of it on a regular basis. The preferred technique is to rely on a library package or compiler that implements the conventions of a favorite parallel programming model. Direct use of the supervisor layer allows programmers to circumvent these conventions in order to communicate between models.

3.1. Basic Concepts

The **realm** is the central abstraction provided by the supervisor layer. Each realm includes data and code. The code constitutes a protocol for manipulating the data. The intent is that the data should not be accessed except by obeying the protocol. In effect, a realm is an abstract data object. Its protocol consists of operations on the data that define the nature of the abstraction. Invocation of these operations is the principal mechanism for communication between parallel threads of control.

The **thread** is the principal abstraction for control flow and scheduling. Each thread is represented by a **context realm** that contains a stack and space to store registers and other volatile information when the thread is blocked. The relationship between realms and threads is somewhat unusual: the conventional notion of an anthropomorphic process has no analog in Psyche. Realms are passive objects, but their code controls all execution. Threads merely animate the code; they have no "volition" of their own.

Depending on the degree of protection desired, an invocation of a realm operation can be as fast as an ordinary procedure call or as slow as a heavyweight process switch. We call the inexpensive version an *optimized* invocation; the safer version is a *protected* invocation. In the case of a trivial protocol or truly minimal protection, Psyche also permits direct external access to the data of a realm. One can think of direct access as a mechanism for in-line expansion of realm operations. By mixing the use of protected, optimized, and in-line invocations, the programmer can obtain (and pay for) as much or as little protection as desired.

Keys and **access lists** are the mechanisms used to implement protection. 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. New keys, with a pseudo-random distribution affording probabilistic protection, can be obtained from the supervisor. The distribution of keys and the management of access lists is under user control, enabling the implementation of many different protection policies.

3.2. Memory Management

If optimized (particularly in-line) invocations are to proceed quickly, they must avoid modification of memory maps. Every realm visible to a given thread must therefore occupy a different location from the point of view of that thread. In addition, if pointers are to be stored in realms, then every realm visible to multiple threads must occupy the same location from the point of view of each of those threads. Since we want threads to be able, at run time, to obtain access to arbitrary realms, we must generally arrange for all coexistent realms to occupy disjoint virtual addresses. Psyche therefore presents its users (conceptually at least) with a single, global, virtual address space. Each thread may run with a different view of this address space, in the sense that different subsets may be marked accessible, but the mapping from virtual to physical addresses will be uniform. Virtual addresses suffice for naming, and pointers can (with appropriate permissions) be used without regard to the realm into which they point.

The view of an executing thread is embodied in the hardware memory map. It always includes both the context realm of the thread and the realm in which the thread is executing (the "current realm"). Execution proceeds unimpeded until an attempt is made to access something not included in the view. The kernel passes the resulting protection fault upward into the supervisor, whose job it is to either (1) announce an error, (2) update the current view and restart the faulting instruction, or (3) create a new thread to perform the attempted operation in a separate protection domain (i.e., with a separate view).

In effect, Psyche uses conventional memory-management hardware as a cache for software-managed protection. Case (2) above corresponds to optimized invocation. Future invocations of the same realm from the same view will proceed without kernel or supervisor intervention. Case (3) corresponds to protected invocations. The choice between cases is controlled by the keys and access lists.

A protected invocation of a realm operation creates a new thread of control. The stack of the thread contains a copy of the top frame on the calling stack, making value parameters accessible. The view of the new thread includes, at minimum, the context realm of the thread, the realms in which reference parameters reside, and the realm of the invoked operation. The view may grow over time as the thread invokes operations in realms to which optimized access is permitted.

In general, the right to invoke an operation is verified at the latest possible moment. This "lazy evaluation" approach to protection allows us to use the same syntax for both optimized and protected invocations. It also allows those invocations to use the same linkage conventions as ordinary procedure calls. In the optimized case, all but the first invocation can execute the same sequence of instructions as an ordinary procedure call. Callers need not know whether a given invocation will be optimized or protected, although they can insist on protection if desired.

3.3. Threads and Scheduling

A protected invocation creates a new thread and suspends the old thread. Information about the old thread is placed at the bottom of the stack of the new thread, along with a copy of the invocation's parameters. Under most circumstances the new thread will want to resume the old one when its work is completed. There are no general restrictions, however, on when threads execute. The supervisor keeps a ready

list of threads that can execute in parallel, possibly on different physical processors. It does not insist that all runnable threads appear on this list. In particular, execution can move without the assistance of the kernel or supervisor to any thread visible in the current view, simply by changing a pointer to the current thread and using that thread's stack. Mutually-accessible threads can therefore be scheduled with lightweight, user-level code.

The context realm of each thread contains the addresses of realm operations to block and unblock the thread. Arbitrary realms can invoke these operations to enforce synchronization constraints. The supervisor provides default routines that switch between threads on its own ready list, but more lightweight mechanisms can be implemented by providing block and unblock routines in user-level "scheduler realms." A scheduler can keep its own ready list of mutually-accessible threads and can switch between them very quickly. Simplistic threads can always use the standard routines that manipulate the supervisor's ready list. Any user-level routine, moreover, will want to call the supervisor when it can find nothing else to do.

3.4. Access Lists, Keys, and Protection

From the caller's point of view, protected and optimized calls will usually look the same. The exception is that a caller can insist that an invocation be protected when it does not trust the realm it is calling. In effect, Psyche has separated the dimensions of protection and performance from the semantics of realm invocation. Unless explicitly requested by the caller, the choice between the two is based on the access list of the realm being called.

When a thread attempts to invoke an operation of a realm that is not in the current view, the supervisor 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. The same protection mechanism is used when creating and destroying realms. Rights contained in access lists include:

invoke operation X
invoke operation X optimized
fork new thread executing operation X
read/write data of realm directly
copy realm (to create a new realm)
destroy realm

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.

The precise mechanism used by the supervisor for checking keys has not yet been determined. We are considering a scheme in which each realm contains a list of keys at a location known to the supervisor. When a fault occurs the supervisor can compare the key lists of the current realm and the current thread's context against the access list of the realm in question. The principal drawback of this strategy is that with long key lists the cost of the comparison could be quite high. An alternative would be for the supervisor to

perform an upcall [13] to a key manager that could use thread-specific knowledge to find an appropriate key. The address of an appropriate key manager could be kept with the block and unblock routines in each thread.

4. Implementation Considerations

In order to support an implementation of Psyche, a target multiprocessor architecture must have a number of characteristics. These characteristics are assumed by the overview above.

(1) All or most of the memory of the machine must be sharable. All memory may be equally distant from all processors, or there may be a hierarchy of locality, but in any event it must be possible to access the code and data of any realm from any processor.

(2) The virtual address space of the machine must be at least as large as, and preferably much larger than, the physical address space. The memory management hardware must support sparse address spaces efficiently.

(3) There must be a very large number of segments or pages, with hardware access rights that can be altered individually.

(4) It must be possible to change quickly into and out of a privileged state in which hardware access rights can be changed.

Commercial multiprocessors that are likely candidates for Psyche implementations include the Sequent Balance, Encore Multimax, multiprocessor VAX, and BBN Butterfly machines. Of these, the Butterfly has by far the largest number of processing nodes and the most interesting memory architecture, in terms of varying locality. Unfortunately, the Butterfly, like many other commercial machines, currently supports only 24-bit virtual addresses, too narrow to address all of physical memory at once.[2] Although announced successors to the current generation of multiprocessors will reduce the severity of this problem by supporting larger virtual address spaces, software techniques for coping with the scarcity of virtual addresses will still be necessary; they are discussed in section 5.1.

Our implementation effort has deliberately focused on issues that are independent of the choice of a particular target machine. These include:

- Facilities for passing protection information between realms and for specifying which realms remain accessible during a realm operation must be designed in such a way that performance does not suffer. For example, interpretation of a large number of parameters during each realm operation is unlikely to prove acceptable.
- Protection mechanisms must allow the kernel to limit access to both memory and the CPU. Even with full protection of currently-accessible memory, an invoked operation must be trusted to return the processor. Clock interrupts provide a mechanism for implementing preemption, which can be used to

[2] The Balance, Multimax, and Butterfly currently have 24-bit virtual addresses, enough to access 16 megabytes. A fully-configured, 256-node Butterfly would contain one gigabyte of memory. The Balance can have up to 28 megabytes of memory, the Multimax up to 128 megabytes.

enforce supervisor control of the processor without requiring the kernel to implement scheduling.

- Linkage information for realm operations must be managed in such a way that users cannot compromise protection. The supervisor cannot trust information stored in the user's stack. Information stored in protected space may interact with lightweight process management.
- Mechanisms for resource recovery will be needed to ensure that both physical memory and virtual address space is reclaimed when no longer needed. The flexibility of sharing relationships in Psyche will make it difficult to determine precisely when recovery should happen.
- Facilities such as the determination of an appropriate address for a newly-created realm (and relocation of the load image to run at that address) will require non-trivial "engineering" solutions, if not original research.

As with most work in operating systems, the proof of concept for Psyche will lie in the implementation. It remains to be seen whether the mechanisms required for protection will indeed permit the most conservative realm invocations to perform significantly better than the higher-level facilities of, say, a message-based operating system. It will be important to ensure that common, optimized operations do not pay for the full generality of the protected case.

5. Discussion

The design of Psyche is based on the observation that access to shared memory is the fundamental mechanism for interaction between processes on a multiprocessor. Any other abstraction that can be provided on the machine must be built from this basic mechanism. An operating system whose kernel interface is based on direct use of shared memory will thus in some sense be universal. The task of the kernel is to make sharing convenient and to provide facilities to guarantee protection.

5.1. The Uniform Virtual Address Space

A uniform virtual address space is a consequence of our assumptions about sharing. Active entities (by whatever name) must be able to arrange, at run time, to share access to arbitrary realms. For at least certain applications, specification of what is shared cannot occur before run time. For a very large class of applications, sharing must be as efficient as the hardware makes possible. In particular, pointers into shared realms must be permitted. As a result:

(1) Any two realms that are to be accessible to the same thread at the same time must occupy different virtual addresses.

(2) Each realm must occupy the same virtual addresses in every thread that uses it. Otherwise compilers will have to generate position-independent code, and addresses will not be usable as pointers.[3]

Any attempt to squeeze active realms into an address space smaller than their total physical size will result in overlaps. The assignment of addresses to realms will then become an exercise in graph coloring, an

[3] In fact, we must generate position-independent code anyway if we wish to share code segments among separate instances of the same type of realm.

exercise that will often have no solution. To make matters worse, it is unlikely that the operating system will be able to predict at realm creation time just which other realms must be avoided. The graph to be colored may not even be known.

The simplest solution to the graph coloring problem is to give each realm a different color (*i.e.*, virtual address). That is, realms reside in a uniform virtual address space shared by all users. Not all portions of that space will be accessible at all times, but the mapping between virtual addresses and realms will be the same from everyone's point of view. This simple solution has one major disadvantage: virtual addresses will be a limiting factor on most current architectures. Neither the 24-bit virtual addresses currently available nor the 32-bit virtual addresses expected soon will be sufficient to address every realm of every program. Therefore, although the conceptual model provided by Psyche is that of a single, uniform address space, any practical implementation must treat virtual addresses as a scarce resource. Two important techniques for managing a scarce resource are (1) multiplex the resource among different programs and (2) reclaim the resource when it is not in use.

Traditional operating systems multiplex both physical memory and virtual addresses among processes. Paging allows two different processes to occupy the same physical memory over time. Virtual memory allows two different processes to use the same virtual address for two different purposes. In contrast, Psyche requires that two different threads use the same virtual address for the same purpose. Nevertheless, the implementation can choose to violate this assumption, as long as it remains transparent to the user. That is, a portion of the virtual address space must be dedicated to any realm that might be shared, but two realms that are never simultaneously accessible can be placed at the same virtual address. For example, stacks and some code may never be shared. The stack and non-shared code of one program may overlap the stack and code of another in the virtual address space. We expect in practice to be able statically to identify significant portions of most programs as non-sharable.

No system can run indefinitely while allowing resources to be consumed and never released. The critical resources to be reclaimed in Psyche are virtual addresses. The problem is that we wish to support long-lived sharing relationships; we cannot delete a realm simply because no thread is currently accessing it. Garbage collection presupposes that all references to realms can be found, an assumption we are not willing to make. Other solutions adopted in traditional operating systems don't work because either sharing must be established explicitly beforehand, long-lived sharing relationships are not allowed, or resources that can be shared long-term are never reclaimed by the system. For example, in most operating systems memory is reclaimed when a process terminates. The file system must be used for long-term sharing (often defined to be any sharing that spans process boundaries) and file space is reclaimed only by human intervention. In Psyche, we plan to use a combination of explicit deallocation by the user and implicit deallocation via an ownership hierarchy to reclaim virtual address space. Explicit deallocation allows the user to micro-manage the virtual address space; implicit deallocation based on ownership guarantees that the system has ultimate control over resource reclamation.

One outgrowth of the uniqueness of realm locations in Psyche is that it becomes trivial for threads to share copies of common libraries. Each such library constitutes a realm without data (except, of course,

for read-only structures and constants). The sharing of library realms avoids the all-to-common situation in which traditional processes are linked to individual copies of the file system, language support routines, mathematical libraries, and other subroutine packages. Of course, on machines like the Butterfly the slower speed of remote memory references will encourage the creation of local copies, but each copy can be shared by all threads on one node. Moreover, the existence of copies need not be visible to the user. We expect our Butterfly implementation to make extensive use of automatic replication for code and read-only data. From the point of view of a user, every thread in existence will be able to share a single copy of, say, the trigonometry package.

5.2. Keys and Access Lists

Psyche realm invocations resemble both Eden object invocations [3] and Hydra procedure calls [14]. Eden and Hydra both use capabilities to implement protection. For Psyche, however, the use of capabilities would have several serious problems:

(1) The tight association between names and rights within a capability would require most pointers into realms to be accompanied in every data structure by an appropriate capability.

(2) Given appropriate rights, our goal for optimized access is to map a realm into the current address space in such a way that further proof of rights is never needed. Under these circumstances we expect accesses to occur frequently enough to make even the presentation of a capability unacceptable.

(3) The use of an explicit *open* primitive, while eliminating the need to present capabilities for routine access, would force the Psyche user to keep track of which realms are currently accessible. Our experience with the Butterfly's Chrysalis operating system [6] has convinced us that this burden will be unacceptable for ordinary programmers and undesirable for the implementors of communication models. Opening realms at the earliest possible moment (rather than waiting until just before the first access) is also unattractive, because the set of realms that might potentially be accessed is likely to be very much larger than the set that will actually be accessed.

(4) While the rights over a memory block are essentially limited to read, write, and execute permissions, the rights to an arbitrary realm are much more difficult to describe. They are significantly easier to represent in data structures associated with the realm itself than in a standard format capability.

Traditional access lists solve these problems, but have other limitations of their own. They require that we be able to name the entities to whom access should be granted. They can require a great deal of space to list all valid names. They make it difficult or impossible to pass rights on to a third party without kernel intervention. By introducing keys as an additional level of indirection, we obtain the advantages of access lists while avoiding their disadvantages. Keys can be moved from place to place without kernel intervention. A single key can convey an arbitrarily complex set of rights over an arbitrary set of realms to an arbitrary set of clients. The rights associated with a key can even be changed without the knowledge of the clients. The primary disadvantage of keys is that they make it impossible to control distribution, but we see no way to avoid this problem in any scheme that transfers rights between protection domains without

the help of the kernel.

5.3. Locality

The fact that Psyche will run on a multiprocessor raises locality issues not encountered in uniprocessor systems. Some of the architectures currently available (the Sequent and Encore machines, for example) make all memory equally accessible from all places. This equality is extremely convenient — it facilitates automatic load balancing, for example, in parallel versions of UNIX — but it is not likely to be feasible in machines with hundreds or thousands of processors. We are committed in the Psyche group to developing software that will scale comfortably to very large machines with non-uniform memory access times. Architectures such as the Butterfly, the IBM RP3, and the Illinois Cedar machine [22] recognize that while it may be possible to make all memory equally distant from a large number of processors, it is not possible to make it all equally close. The former option is no less expensive than an intermediate solution in which each processor is close to some of the memory, and distant from the rest.

It is tempting to suggest that an operation should always execute on the processor closest to its data, but such a suggestion ignores the fact that in some cases the cost of transferring control to a thread on the appropriate processor may exceed the cost of accessing the data remotely, or even of executing the code itself out of non-local memory. A system that provides remote procedure calls as the basic communication mechanism on a multiprocessor precludes the more primitive option. Psyche, on the other hand, retains both options by permitting direct execution when appropriate.

We fully expect that Psyche will make use of memory management techniques to maximize locality whenever possible. The code for a realm operation will probably be replicated on most of the nodes that invoke it. Where operating systems like Accent make heavy use of copy-on-write techniques [18], Psyche is likely to provide for copy on remote access. Where machines like the Butterfly already provide a block transfer operation for high-speed copying between local memories, Psyche may add a copy-on-write block transfer.

For use in a network environment, we see competing approaches to accommodating Psyche. The first is to support realm invocations and even direct access across machines by simulating memory sharing. This approach has the advantage of functional transparency, but serves to hide costs that we believe should be explicit. Our inclination at present is to pursue a second option, in which cross-machine operations are supported by a local realm that serves as a network interface. This approach is similar in style to remote procedure call stub generators and to the network server processes of Accent and Mach. It reflects, rather than hides, the costs involved in inter-machine communication.

5.4. Synchronization

Given the provision for direct execution, we expect that more than one thread, and indeed more than one processor, may be running in a realm at once. As with locality, Psyche addresses the issue of synchronization by permitting each individual realm to adopt the mechanism most appropriate to its own semantics. It would be possible, for example, to insist that each realm be a monitor, and that only one

thread at a time be allowed to execute inside. Such an approach would preclude useful parallelism in many applications. Since the requirements for synchronization vary greatly from program to program [12], it seems better to provide the hooks with which to implement various policies than to force a particular model on every application.

Each multiprocessor architecture will provide some sort of atomic memory-level operations. From these basic operations, users will be able to construct a variety of synchronization routines. Once libraries (realms) exist for the basic synchronization mechanisms, such as semaphores, monitors, path expressions, and read/write locks, it should be straightforward to incorporate the appropriate mechanism in each more complicated realm. Exotic parallel data structures may require the construction of special-purpose synchronization routines, but this requirement should not be regarded as a liability. An operating system that insisted on a single synchronization model would not support the code at all.

5.5. Examples of the Use of Realms

For both locality and synchronization, the philosophy of Psyche is to provide a fundamental, low-level mechanism from which a wide variety of higher-level facilities can be built. Realms, with directly-executed operations, can be used to implement the following:

(1) Pure shared memory in the style of the BBN Uniform System [7]. A single, large, static realm would be shared by all threads. The access protocol, in an abstract sense, would permit unrestricted reads and writes of individual memory cells.

(2) Packet-switched message passing. Each message would be a separate realm. To send a message one would make the realm accessible to the receiver and inaccessible to the sender.

(3) Circuit-switched message passing, in the style of Demos [29], Accent [34], Charlotte [5], or LYNX [36]. Each communication channel would be realized as a realm accessible to a limited number of threads, and would contain buffers manipulated by protocol operations.

(4) Synchronization mechanisms such as monitors, locks, and path expressions. As described above, each can be written once as a library routine that is instantiated as a realm by each abstraction that needs it.

(5) Parallel data structures. Special-purpose locking could be implemented in a collection of realms scattered across the nodes of the machine, in order to reduce contention [16, 17]. For certain kinds of data structures, (the Linda tuple space [2], for example), the entry routines of the data structure as a whole might be fully parallel, able to be executed without synchronization until access was required to particular pieces of the data.

5.6. Relationship to Previous Work

Psyche resembles Hydra [42] in its use of protected procedure calls for the execution of operations in separate protection domains. Our approach differs in its emphasis on multiple programming models, its integration of code and data in realms, and its provision for optimized access. Objects in Hydra can be either procedures or data. Realms in Psyche are both. Our approach is more in keeping with current use of

the term "object-oriented," in that data is never separated from the protocol for its access.[4] Sharable data in Hydra can be accessed only through the use of capabilities, so very fine-grain operations, even without the need for protection, cannot be made efficient.

The structural difference between Hydra objects and Psyche realms is best viewed as a difference in approaches to building abstractions. The association between data and procedures in Hydra is established by convention. Protocols are enforced by giving a procedure the ability to *amplify* the rights of capabilities for certain types of data objects. User programs hold capabilities that do not permit them to access the internals of the data objects; only the amplifying procedures can do so. Psyche abstractions, by contrast, are provided directly by the Psyche supervisor. No amplification mechanism is needed in order to enforce the use of protocols. Where a Hydra user would ask the "pop" procedure to return an item from stack object X, a Psyche user would ask the "stack X" object to pop itself and return the result. By analogy to programming languages, the Hydra approach to abstraction resembles an Ada package that exports an opaque type, while Psyche abstractions resemble Smalltalk objects [19].

Psyche also bears a resemblance to the StarOS [21] and Medusa [32] operating systems for Cm*. It is closer to Hydra than to StarOS, and closer to StarOS than to Medusa. StarOS emphasizes the asynchronous execution of operations by remote processes. As in Hydra, code and data comprise separate objects, but a number of special object types (dequeues, mailboxes, events) are built into the kernel and supported with microcode. A mechanism is provided for mapping an object into one of a limited number of *windows*, but the result is much less general than the inclusion of Psyche realms in views. In any event the use of a uniform virtual address space would not have been an option on the Cm* hardware, which only supported 16-bit addresses. Medusa adopts an essentially message-based approach to process interaction, with only a limited form of data sharing permitted within multi-process task forces.

Perhaps the best-known current work in multiprocessor operating systems is the Mach project [1], again at CMU. In comparison to Mach, Psyche has both a different motivating philosophy and a different set of resulting abstractions. Psyche is not meant to be UNIX compatible. It is also not meant to run on networks, though it could be extended to do so. Its real focus is on shared-memory multiprocessors, for which we believe it can make significantly better use of the hardware than is possible with a primarily message-based system.

Psyche adopts a passive view of objects, as opposed to the active view of Mach. Where Mach provides messages as the basic communication mechanism, Psyche provides data sharing and protected procedure calls. Where the notion of threads within a task is built into Mach at the kernel level, the threads of Psyche can be scheduled in user code and can move between mutually-accessible realms. Where data can be shared in Mach only between related tasks in the task creation tree, Psyche permits realms to be shared among arbitrary groups of threads. All of these differences make Psyche a lower-level, less structured operating system, but at the same time one that will admit a wider variety of user applications with a finer grain of interaction.

---

[4] The fundamentally passive nature of a realm, the unusual protection mechanism, and the lack of inheritance lead us to avoid the adjective "object-oriented" ourselves.

We feel that the closest parallels to Psyche can be found in the so-called *open* operating systems developed for uniprocessors by groups at Xerox and MIT. In Cedar [39] (no relation to the Illinois Cedar project) and Swift [13], all the software of the machine runs in a single address space, with no protection provided by the kernel. Processes are prevented from interfering with each other by relying on the compiler for a "safe" programming language. The Cedar system was built with the Cedar language, a successor to Mesa [30]; Swift was built with CLU [25]. Psyche can be regarded as an attempt to provide the advantages of an open operating system without relying on a single programming language. It is also an attempt to extend support to multiple processing nodes, though the Cedar group is moving in the same direction [28].

The comparison to Swift is particularly apt. The notion of an upcall corresponds directly to the invocation of a realm operation. The multi-process modules of Swift are very much like realms. Both Psyche and Swift are designed to separate the crossing of functional boundaries (i.e. between realms) from the expense of context switching. The solution may be more successful in Swift, since the CLU compiler can provide cost-free protection when calling an untrusted module. Psyche invocations that go "down" into a trusted realm like the file system will be easier to optimize than invocations that go "up" into untrusted user code.

6. Status and Plans

As of summer 1987, we have finished designing the Psyche kernel and have begun a pilot implementation, while continuing design work on the supervisor. Our principal goal for the short term is to obtain an environment as quickly as possible in which we can experiment with multi-model programs.

We expect our work to evolve into a number of interrelated projects. Interesting research could be performed in memory management, lightweight process structure, implementation and evaluation of communication models, and parallel language design. The latter subject is of particular interest. We have specifically avoided language dependencies in the design of the Psyche kernel and supervisor. It is our intent that many languages, with widely differing process and communication models, be able to coexist and cooperate on a Psyche machine. We are interested, however, in the extent to which the Psyche philosophy itself can be embodied in a programming language.

The communications facilities of a language enjoy considerable advantages over a simple subroutine library. They can be integrated with the naming and type structure of the language. They can employ alternative syntax. They can make use of implicit context. They can produce language-level exceptions. For us the question is: to what extent can these advantages be provided without insisting on a single communication model at language-design time? Though these questions are beyond the scope of our current work, we expect them to form the basis of a future, follow-on project.

Acknowledgments

The work reported herein owes much to the efforts of the entire Psyche group: Rob Fowler, Alan Cox, Lawrence Crowl, Peter Dibble, Neal Gafter, John Kerber, Brian Marsh, and John Mellor-Crummey. Jerry Feldman provided useful comments on an earlier draft of this paper.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for UNIX Development," *Proceedings of the Summer 1986 USENIX Technical Conference and Exhibition*, June 1986.

[2] S. Ahuja, N. Carriero, and D. Gelernter, "Linda and Friends," *Computer 19*:8 (August 1986), pp. 26-34.

[3] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, "The Eden System: A Technical Review," *IEEE Transactions on Software Engineering SE-11*:1 (January 1985), pp. 43-59.

[4] G. R. Andrews, R. A. Olsson, M. Coffin, I. J. P. Elshoff, K. Nilsen, and T. Purdin, "An Overview of the SR Language and Implementation," *ACM TOPLAS*, to appear. Available as TR 86-6a, Department of Computer Science, University of Arizona, 23 June 1986.

[5] Y. Artsy, H.-Y. Chang, and R. Finkel, "Interprocess Communication in Charlotte," *IEEE Software 4*:1 (January 1987), pp. 22-28.

[6] BBN Advanced Computers Incorporated, "Chrysalis® Programmers Manual, Version 3.0," 28 April 1987.

[7] BBN Laboratories, "The Uniform System Approach to Programming the Butterfly® Parallel Processor," BBN Report #6149, Version 2, Cambridge, MA, 16 June 1986.

[8] A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System," *OOPSLA'86 Conference Proceedings*, 29 September - 2 October 1986, pp. 78-86. In *ACM SIGPLAN Notices 21*:11 (November 1986).

[9] C. M. Brown, 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.

[10] N. Carriero and D. Gelernter, "The S/Net's Linda Kernel," *ACM TOCS 4*:2 (May 1986), pp. 110-129. Originally presented at the *Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985.

[11] D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, 10-13 October 1983, pp. 129-140. In *ACM Operating Systems Review 17*:5.

[12] D. R. Cheriton, "Problem-oriented Shared Memory: A Decentralized Approach to Distributed System Design," *Proceedings of the Sixth International Conference on Distributed Computing Systems*, 19-23 May, 1986, pp. 190-197.

[13] D. Clark, "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.

[14] E. Cohen and D. Jefferson, "Protection in the Hydra Operating System," *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, November 1975, pp. 141-160.

[15] J. Edler, A. Gottlieb, and J. Lipkis, "Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3," Ultracomputer Note #91, Courant Institute, N. Y. U., December 1985.

[16] C. Ellis, "Concurrent Search and Insertion in 2-3 Trees," *Acta Informatica 14* (1980), pp. 63-86.

[17] C. Ellis, "Concurrent Search and Insertion in AVL Trees," *IEEE Transactions on Computers C-29*:9 (September 1980), pp. 811-817.

[18] R. Fitzgerald and R. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM TOCS 4*:2 (May 1986), pp. 147-177. Originally presented at the *Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985.

[19] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[20] R. H. Halstead, Jr., "Parallel Symbolic Computing," *Computer 19*:8 (August 1986), pp. 35-43.

[21] A. K. Jones, R. J. Chansler, Jr., I. Durham, K. Schwans, and S. R. Vegdahl, "StarOS, a Multiprocessor Operating System for the Support of Task Forces," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, December 1979, pp. 117-127.

[22] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh, "Parallel Supercomputing Today and the Cedar Approach," *Science 231* (28 February 1986), pp. 967-974.

[23] T. J. LeBlanc, R. H. Gerber, and R. P. Cook, "The StarMod Distributed Programming Kernel," *Software — Practice and Experience 14*:12 (December 1984), pp. 1123-1139.

[24] T. J. LeBlanc, "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. Expanded version available as BPR 3, Computer Science Department, University of Rochester, January 1986.

[25] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*, Lecture Notes in Computer Science #16, Springer-Verlag, Berlin, 1981.

[26] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS 5*:3 (July 1983), pp. 381-404.

[27] H. H. Mashburn, "The C.mmp/Hydra Project: An Architectural Overview," pp. 350-370 (chapter 22) in *Computer Structures: Principles and Examples*, ed. D. P. Siewiorek, C. G. Bell, and A. Newell, McGraw-Hill, New York, 1982.

[28] E. McCreight, "The DRAGON CPU," *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 5-8 October 1987. To appear.

[29] B. P. Miller, D. L. Presotto, and M. L. Powell, "DEMOS/MP: The Development of a Distributed Operating System," *Software — Practice and Experience 17* (April 1987), pp. 277-290.

[30] J. G. Mitchell, W. Maybury, R. Sweet, and J. R. Herz, Jr., "Mesa Language Manual, version 11.0," Xerox Office Systems Division, June 1984.

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

[32] J. D. Ousterhout, D. A. Scelza, and S. S. Pradeep, "Medusa: An Experiment in Distributed Operating System Structure," *CACM 23*:2 (February 1980), pp. 92-104.

[33] G. R. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proceedings of the 1985 International Conference on Parallel Processing*, 20-23 August 1985, pp. 764-771.

[34] R. F. Rashid 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.

[35] M. L. Scott, "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.

[36] M. L. Scott, "Language Support for Loosely-Coupled Distributed Programs," *IEEE Transactions on Software Engineering SE-13*:1 (January 1987), pp. 88-103.

[37] R. E. Strom and S. Yemini, "The NIL Distributed Systems Programming Language: A Status Report," *ACM SIGPLAN Notices 20*:5 (May 1985), pp. 36-44.

[38] R. J. Swan, S. H. Fuller, and D. P. Siewiorek, "Cm* — A Modular Multi-Microprocessor," *Proceedings of the AFIPS 1977 NCC 46*, AFIPS Press (1977), pp. 637-644.

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

[40] United States Department of Defense, "Reference Manual for the Ada® Programming Language," (ANSI/MIL-STD-1815A-1983), 17 February 1983. Available as Lecture Notes in Computer Science #106, Springer-Verlag, New York, 1981.

[41] N. Wirth, *Programming in Modula-2*, Third, Corrected Edition. Texts and Monographs in Computer Science, ed. D. Gries, Springer-Verlag, Berlin, 1985.

[42] W. A. Wulf, R. Levin, and S.P. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, New York, 1981.