Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System

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

University of Rochester
Department of Computer Science
Rochester, NY 14627

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. It is predicated on the conviction that no one model of process state or style of communication will prove appropriate for all applications, but that shared-memory multiprocessors (particularly the scalable ''NUMA'' variety) 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 facilitates 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 policies implemented with 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.

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 [1, 8], 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. Most message-based operating systems can be implemented on shared-memory machines as well. The Mach project [24] at CMU represents, to a large extent, the merger of Berkeley UNIX with the Accent network operating system [7]. Mach now runs on several multiprocessors, including DEC, Encore, and Sequent machines.

Our aim in the Psyche project is to develop a programming environment (starting with an operating system) that supports truly 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. As with the parallel UNIX designs, 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.

We see at least two dangers in adapting an existing operating system for use on a multiprocessor. First, it may fail to provide abstractions that are appropriate for certain applications. Second, it may fail to make effective use of the hardware. Through the course of considerable experience with application and system software, we have become convinced that no one model of process state or style of communication will be best for all parallel applications. Just as a general-purpose operating system for a uniprocessor must support a wide variety of models (e.g. programming languages) for sequential computing, so too must a general-purpose operating system for a multiprocessor support a wide variety of models for parallel computing. Since parallel computing involves concepts (such as scheduling and interprocess communication) that have traditionally been the province of operating systems, parallel versions of traditional operating systems are unlikely to provide the flexibility required by users.

Our first goal for Psyche is therefore *flexibility*: users should be able to implement a wide variety of models for interprocess communication and lightweight process structure. Pieces of an application written under different models should be able to interact easily, that is, to arrange dynamically to share access to arbitrary abstractions. Since Psyche is to be a multiuser system, our second goal is *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. Finally, since multiprocessors are attractive primarily for speed, our third goal is *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 sending a message in current network operating systems.

Though protection and performance are conventional goals, our emphasis on flexibility is distinctive and unusual. In order to permit user-level control over processes and communication, we have adopted a kernel/user interface consisting of unusually low-level primitives. We do not expect this interface to be easy to use, but the assumption is that most programmers will never attempt to use it. Instead, they will rely on preexisting libraries and language support packages for process management and communication. We have adopted the position that an operating system kernel should provide only the lowest common denominator for things that will be built upon it. 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 [15], Mach [24], and V [10], 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 [28]). Similarly, most implementations of parallel programming languages have either employed a special-purpose kernel (as in SR [37], StarMod [11], or Linda [16]), or have been built on top of an

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. We thank the Xerox Corporation University Grants Program for providing equipment used in the preparation of this paper.

existing uniprocessor operating system, most often UNIX. We are unaware of any work specifically addressing the design of a kernel to support multiple programming models.

Motivation

Shared Memory Versus 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.

Three years ago, our department acquired a 128-node BBN Butterfly™ Parallel Processor [17], 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 [18, 19, 35, 36]. 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. A data structure might also be used in situations where the relationship between data items, as encoded in the data structure, may be difficult to recreate. Information that is easy to compute, or would require too much space to store, is encoded in a function. 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.

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 [24] and Amoeba [20], 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 [21], Ada tasks [9], LYNX threads [29], Emerald objects [22], Modula-2 coroutines [12], and SR [37] 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 the need for kernel intervention when switching between mutually-trusting threads.

[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.

Psyche Overview

The design of Psyche is based on the observation that access to shared memory is the fundamental mechanism for interaction between threads of control 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.

Basic Concepts

The **realm** is the central abstraction provided by the Psyche kernel. Each realm includes data and code. The code constitutes a protocol for manipulating the data and for scheduling threads of control. 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 abstraction for control flow and scheduling. All threads that begin execution in the same realm reside in a single **protection domain**. That domain enjoys access to the original realm and any other realms for which access rights have been demonstrated to the kernel. The layout of a thread context block is defined by the kernel, but threads themselves are created and scheduled by the user. The kernel time-slices on each processor between protection domains in which threads are active, providing upcalls [13] at quantum boundaries and whenever else a scheduling decision is required.

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 affording probabilistic protection. The creation and distribution of keys and the management of access lists are all under user control, enabling the implementation of many different protection policies.

Memory Model

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. Satisfying these two conditions simultaneously constitutes an exercise in bipartite graph coloring. In order to accommodate arbitrary changes to the graph at run time, 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 protection domain may have 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 a protection domain is embodied in the hardware memory map. Execution proceeds unimpeded until an attempt is made to access something not included in the view. The resulting protection fault is fielded by the kernel, whose job it is to either (1) announce an error, (2) update the current view and restart the faulting instruction, or (3) perform an upcall into the protection domain associated with the target realm, in order to create a new thread to perform the attempted operation. In effect, Psyche uses conventional memory-management hardware as a cache for software-managed protection. Case (2) corresponds to optimized invocation. Future invocations of the same realm from the same protection domain will proceed without kernel intervention. Case (3) corresponds to protected invocations. The choice between cases is controlled by the keys and access lists.

The major disadvantage of the uniform virtual address space is that address bits will be a scarce resource on most current architectures. Neither the 24-bit virtual addresses of many current machines nor the 32-bit virtual addresses now becoming available 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 take special measures to economize on virtual addresses. As with all scarce resources, it becomes important to (1) multiplex the resource among different programs and (2) reclaim the resource when it is not in use.

A Psyche implementation need only maintain the *appearance* of a uniform virtual address space. It can multiplex addresses if it knows that certain realms will never be simultaneously visible. Realms that will not be shared at all can clearly overlap. Substantial amounts of code and data are likely to fall into this category in practice. Asking the user to identify unshared realms to the kernel runs counter to the Psyche philosophy, but is likely to produce benefits that outweigh its conceptual cost. In addition, realms that are accessed only through protected invocations can be located somewhere other than where the user thinks they are, and in fact can overlap. Since the kernel is involved in every invocation, it can map a dense range of virtual addresses onto the operations of the overlapped realms.

In order to reuse virtual addresses, a kernel implementation must be able to tell when a realm is no longer needed. Since we want to support long-term sharing relationships, we cannot delete a realm simply because no thread is currently accessing it. Genuine garbage collection is also impractical, since it presupposes that all references to realms can be found. Other solutions adopted in traditional operating systems don’t work either because sharing must be established explicitly beforehand, because long-lived sharing relationships are not allowed, or because 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 micromanage the virtual address space; implicit deallocation based on ownership guarantees that the system has ultimate control over resource reclamation.

Threads and Scheduling

Each realm in Psyche is the root of exactly one protection domain. All threads that begin execution in the same realm belong to the protection domain rooted in that realm. They share a common view of memory, that is, a single memory map. Initially, the view includes only the data of the original realm of the protection domain. When an attempt is first made to access another realm from that domain, the kernel checks access rights and implicitly *opens* the new realm for access by threads in the domain.[2] If optimized access is permitted, the new realm is added to the view.

The kernel time-slices on each processor between protection domains in which threads are active, providing each with an equal percentage of the CPU. On a given processor, each protection domain will be represented by at most one of its threads at any point in time. The identity of this thread can be changed in user code, so that the thread suspended at the end of a quantum may well be different from the one that was resumed at the beginning of the quantum. In effect, the kernel and user schedule exactly the same abstraction.

Each realm is required to provide routines for thread management tasks that involve the kernel. The kernel performs upcalls to these routines whenever user-level scheduling may be required. For example, upcalls occur when (1) an invocation of one of the realm's operations has occurred in a protection domain in which the realm is open for protected access (so that it may be appropriate to create a new thread to perform the requested operation), (2) a protected invocation by the current thread in the realm's own protection domain has caused that thread to block (so that it may be appropriate to run a different thread), (3) a protected invocation has completed in some other protection domain (so that a local thread may be unblocked), (4) a user-specified time limit has expired (so that preemption of the current thread may be required), and (5) a hardware fault has occurred (so that it may be appropriate to raise an exception in the current thread). None of these upcalls is expected to return. The state of the machine at the time of the upcall is saved by the kernel in the context block of the current thread. After performing its scheduling operation, the upcall routine is expected to jump immediately into the execution of an appropriate thread.

Upcalls execute in user mode, running code provided by the user. Their work space is allocated out of a static area established by the kernel when the realm is created. Each realm exercises complete control over the threads in its own protection domain. The kernel makes no assumption about the nature (or even the existence) of stacks for the threads themselves.

Since a realm can be opened for optimized access from more than one protection domain, it is possible for threads of many different kinds to be executing in the realm at once. In order to facilitate synchronization of these threads, each root realm of a protection domain is expected to provide a pair of routines to be called in user mode to block the current thread and to unblock a specified thread. These routines are in addition to the kernel-required upcall interface. When execution of a realm operation cannot proceed because of a synchronization constraint, the approved course of action is to call the thread blocking routine of the current protection domain, after saving the address of the unblock routine in an appropriate data structure. Low-level, architecture-specific primitives (such as test-and-set or compare-and-swap instructions) can be used to maintain atomicity of the scheduling operations.

[2] This ''lazy evaluation'' approach to protection frees the programmer from keeping track of which realms have been opened, and allows us to limit the cost of access rights verification to cases in which the realm in question will actually be used.

In comparison to the library-based coroutine packages of traditional operating systems, the parameterized thread management of Psyche allows a protection domain to schedule other threads when the current thread has blocked, and permits time-slicing between user threads in a completely natural way. In comparison to the kernel-supported threads of Mach [24], or Amoeba [20], the Psyche mechanism provides the speed of a coroutine package for voluntary context switches within a protection domain and, given sufficient overlap of domains, for unblock operations that span thread types. In addition, the Psyche mechanism allows us to use the syntax and linkage conventions of ordinary procedure calls for both protected and optimized invocations. Once a realm is opened, it allows the optimized invocations to exhibit the same performance as ordinary procedure calls. Finally, the Psyche mechanism provides a much higher degree of flexibility than is possible with either other approach. Reference parameters can be used for protected invocations if the caller trusts the callee. Synchronization of operations in shared realms can be provided to dissimilar threads. User specification of the code to be executed by upcalls means that a realm can implement an explicitly message-based style of serving external requests, dispatching invocations to waiting server threads rather than creating new threads implicitly.

Keys and Access Lists

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 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.

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 context block of 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. The principal drawback of this strategy is the potential cost of matching when both the key list and the access list are long. Since matching occurs only when realms are opened, there is reason to believe that any cost incurred will be amortized over enough operations to make it essentially negligible. Moreover, we believe that most programmers will use keys in either a capability or access-list style, so that either the key list or the access list will generally be short. In cases where multi-way matching is expected to be unacceptably slow, programmers will have the option of calling an explicit open operation, with explicit presentation of a key.

In the early stages of our design work, before adopting our system of keys, we had planned to use capabilities for protection in Psyche. This seemed to be a reasonable choice; realm invocations bore a superficial resemblance to mechanisms employing capabilities in several other systems, including the object invocations of Eden [15] and the procedure calls of Hydra [6]. Upon further examination, however, it became clear that the use of capabilities in Psyche would pose 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, resulting in unacceptable space overhead.

(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 the cost of presenting a capability on every access unacceptable, even if no actual verification is performed.

(3) Mandatory use of an explicit *open* primitive would eliminate the need to present capabilities for routine access, but would also force the Psyche user to keep track of which realms are currently accessible. Our experience with the Chrysalis operating system [30] 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.

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. 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 help of the kernel.

Locality

The fact that Psyche is intended to run on a large-scale multiprocessor raises locality issues not encountered in uniprocessors or in bus-based multiprocessors. Machines that will scale to hundreds or thousands of nodes must clearly have NUMA (non-uniform memory access) architectures. Current designs include the BBN Butterfly [17, 31], IBM RP3 [14], Illinois Cedar [23], and Encore Ultramax [32]. Given hardware or firmware support for microsecond access to remote memory, hypercube designs would qualify as well. Optimal performance on these NUMA machines depends critically on maximizing locality, so that data accessed frequently is also accessed quickly. Unfortunately, the research community has yet to develop any general purpose memory management strategy that achieves the desired result. Attacking this so-called ''NUMA problem'' will be a crucial task for Psyche.

Psyche realms provide a strong notion of locality in our current implementation. All the data of a given realm resides at a single location, equally close or equally far from each individual thread. Applications that need to manage locality explicitly can create multiple realms. Allowing the data of a realm to be scattered across the machine would require either (1) a successful solution to the NUMA problem (in the form of kernel-managed, automatic, optimal data distribution), or (2) the introduction of a new abstraction to represent the pieces of a realm. We are reluctant to accept the latter; we do not yet have the former. We see our current approach as a reasonable first cut that will permit further experimentation.

Whether realms span NUMA boundaries or not, protection domains clearly must do so, since they consist of multiple realms. As a result, interactions between realms may span NUMA boundaries. In most cases performance will be maximized by executing realm operations on a processor close to the data. These operations must be performed by a thread co-located with the data. In some cases, however, the cost of transferring control to a thread on an appropriate processor may exceed the cost of accessing the data remotely. If the appropriate code is replicated, these operations can be performed by any thread, which then accesses the data remotely.

In our current implementation, we permit the user to specify which operations are data-intensive enough to justify the cost of co-locating code and data. The code for these operations is kept out of the page table to force the use of protected invocations, thereby transferring control to a thread in a domain that is close to the data. As with scattering of data, we consider this approach to be a first cut that will support later experimentation with more sophisticated realm or thread migration strategies.

The opportunity to perform migration occurs in several places. When a realm is first opened for optimized access from a given protection domain, the kernel can consider moving the realm to be closer to other realms in the domain. When a protected invocation provides reference parameters, the kernel can consider moving either the target realm or the realm of the parameters in order to optimize access. When a very large data structure is incorporated in the views of more than one protection domain, the kernel may use virtual memory techniques to copy on reference or even move on reference. The optimal mix of techniques for automatic data (re)location is far from obvious; the NUMA problem is very much an open research issue.

Although Psyche is not designed explicitly for loosely coupled networks, it could be extended to accommodate them in at least two different ways. The first is to incorporate networks into the NUMA model by simulating remote operations in software (i.e. in the kernel). This approach has the considerable advantage of functional transparency. Protected invocations would be implemented in much the same way as on a shared-memory machine. Optimized invocations would need to make use of automatic migration in order to obtain acceptable performance. Work by Li and Hudak suggests that this first approach is tractable for certain usage patterns [38]. In other cases, however, it might serve to hide costs that would be better kept explicit. An alternative would be to write network interface realms to support cross-machine operations. This second approach would require no kernel modifications. It is similar in style to remote procedure call stub generators and to the network server processes of Accent and Mach.

Examples of the Use of Realms

For both locality and communication, 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 [25]. A single large collection of realms 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 Accent [7], Charlotte [33], or Lynx [29]. 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. Each of these 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 [3, 4]. For certain kinds of data structures, (the Linda tuple space [26], for example), the entry routines of the data structure as a whole might be fully parallel, able to be executed without synchronization until access is required to particular pieces of the data.

Machine Requirements

In order to support an implementation of Psyche, a target multiprocessor must have certain characteristics. All or most of its memory must be sharable — the architecture may be UMA or NUMA, but it must be possible to access the code and data of any realm from any processor. The virtual address space must be at least as large as, and preferably much larger than, the physical address space. There must be a very large number of individually protected segments or pages. Support must be provided for very sparse address spaces.

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. The new Butterfly 1000 series [31] also provides 32 bit virtual addresses, more than either Sequent or Encore.[3] Even with 32 bits, however, software techniques for coping with the scarcity of virtual addresses will still be necessary. Our implementation effort has deliberately focused on issues that are independent of the choice of a particular target machine.

Relationship to Previous Work

Psyche resembles Hydra [6] 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 kernel. 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 [9] that exports an opaque type, while Psyche abstractions resemble Smalltalk objects.

Psyche also bears a resemblance to the StarOS [2] and Medusa [5] 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 [24], again at CMU. In comparison to Mach, Psyche has both a different motivating philosophy and a different set of resulting abstractions. Psyche is not constrained to be UNIX compatible. It is also not designed specifically for networks, though it could be extended to run in a loosely-coupled world. Its real focus is on scalable 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 Mach supports data sharing primarily between related tasks in the task creation tree, Psyche facilitates dynamic sharing relationships between arbitrary threads. Where Mach relies on the kernel to control the use of capabilities, Psyche provides probabilistic protection with keys in user space. 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.

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 [27] (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’’

---

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

[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.’’

programming language. 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 [34].

The comparison to Swift is particularly apt. The multi-process modules of Swift are very much like realms. Upcalls between modules in Swift resemble optimized realm invocations. 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.

Status and Plans

Design of the low-level kernel routines for Psyche was completed in the summer of 1987. Implementation of these routines has proceeded in parallel with the design of higher layers. We have recently acquired a 24-node Butterfly 1000 Parallel Processor (a.k.a. Butterfly Plus) on which we are continuing development. With its Motorola 68851-based memory management system, this new machine permits the large sparse address spaces we require. Our principal goal for the coming year 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 (particularly for the automatic management of memory with non-uniform access times), 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. 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

Many members of the Rochester systems group have contributed to the work reported herein. The authors extend their thanks to Rob Fowler, Bill Bolosky, Alan Cox, Lawrence Crowl, Peter Dibble, Neal Gafter, John Kerber, and John Mellor-Crummey.

References

[1] 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.

[2] 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.

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

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

[5] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, ''Medusa: An Experiment in Distributed Operating System Structure,'' *Communications of the ACM 23*:2 (February 1980), pp. 92-105.

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

[7] 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 SIGOPS Operating Systems Review 15*:5.

[8] 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.

[9] 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.

[10] 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 SIGOPS Operating Systems Review 17*:5.

[11] 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.

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

[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 SIGOPS Operating Systems Review 19*:5.

[14] 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.

[15] 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.

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

[17] BBN Laboratories, ‘‘Butterfly® Parallel Processor Overview,’’ BBN Report #6148, Version 1, Cambridge, MA, 6 March 1986.

[18] 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, October 1986.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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, pp. 93-112.

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

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

[27] D. Swinehart, P. Zellweger, R. Beach, and R. Hagmann, ‘‘A Structural View of the Cedar Programming Environment,’’ *ACM Transactions on Programming Languages and Systems 8*:4 (October 1986), pp. 419-490.

[28] 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.

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

[30] BBN Advanced Computers Incorporated, ‘‘Chrysalis® Programmers Manual, Version 3.0,’’ Cambridge, MA, 28 April 1987.

[31] BBN Advanced Computers Incorporated, ‘‘Inside the Butterfly Plus,’’ Cambridge, MA, 16 October 1987.

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

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

[34] R. R. Atkinson and E. M. McCreight, ‘‘The Dragon Processor,’’ *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operat ing Systems*, 5-8 October 1987, pp. 65-71.

[35] T. J. LeBlanc, ‘‘Problem Decomposition and Communication Tradeoffs in a Shared-Memory Multiprocessor,’’ in *Numerical Algorithms for Modern Parallel Computer Architectures*, IMA Volumes in Mathematics and its Applications #16, Springer-Verlag, 1988.

[36] T. J. LeBlanc, M. L. Scott, and C. M. Brown, ‘‘Large-Scale Parallel Programming: Experience with the BBN Butterfly Parallel Processor,’’ *Proceedings of the First ACM Conference on Parallel Programming: Experience with Applications, Languag es and Systems*, 19-21 July 1988, pp. 161-172.

[37] G. R. Andrews, R. A. Olsson, M. Coffin, I. J. P. Elshoff, K. Nilsen, T. Purdin, and G. Townsend, ‘‘An Overview of the SR Language and Implementation,’’ *ACM Transactions on Programming Languages and Systems 10*:1 (January 1988), pp. 51-86.

[38] K. Li and P. Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Transactions on Computer Systems 7*:4 (November 1989), pp. 321-359. Earlier version presented at the Fifth Annual ACM Symposium on Principles of Distributed Computing, 11-13 August 1986.