PPoPP '90

MULTI-MODEL PARALLEL PROGRAMMING IN PSYCHE

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

Computer Science Department
University of Rochester
Rochester, New York 14627
{scott,leblanc,marsh}@cs.rochester.edu

Abstract

Many different parallel programming models, including lightweight processes that communicate with shared memory and heavyweight processes that communicate with messages, have been used to implement parallel applications. Unfortunately, operating systems and languages designed for parallel programming typically support only one model. Multi-model parallel programming is the simultaneous use of several different models, both across programs and within a single program. This paper describes multi-model parallel programming in the Psyche multiprocessor operating system. We explain why multi-model programming is desirable and present an operating system interface designed to support it. Through a series of three examples, we illustrate how the Psyche operating system supports different models of parallelism and how the different models are able to interact.

1. Introduction

The widespread use of distributed systems since the late 1970's and the growing importance of multiprocessor systems in the 1980's has spurred the development of programming environments for parallel and distributed computing. Emerging programming models have provided many different styles of communication and process primitives. Individually-routed synchronous and asynchronous messages, unidirectional and bidirectional message channels, remote procedure calls, global buffers [13], and shared address spaces with semaphores, monitors, or spin locks have all been used for communication and synchronization. Coroutines, lightweight run-to-completion threads, lightweight blocking threads, heavyweight single-threaded processes, and heavyweight multi-threaded processes have been used to express concurrency. These communication and process primitives, among others, appear in many combinations in the parallel and distributed programming environments in use today.

Each parallel programming environment defines a model of processes and communication. Each model makes assumptions about communication granularity and frequency, synchronization, the degree of concurrency desired, and the need for protection. Successful models make assumptions that are well-matched to a large class of applications, but no existing model has satisfied all applications, and there is little hope that one ever will. Problems therefore arise when attempting to use a multiprocessor as a general-purpose parallel machine, because the traditional approach to operating system design adopts a single model of parallelism and embeds it in the kernel.

This work is supported in part by NSF research grant number CDA-8822724, Darpa/ETL contract number DACA76-85-C-0001, and a DARPA/NASA Graduate Research Assistantship in Parallel Processing.

When the model of parallelism enforced by an operating system does not meet the needs of an application, the programmer must either (1) distort the algorithm to fit the available model, (2) modify or augment the operating system, or (3) move to a system with a different model. Changing the algorithm to fit the model may or may not be practical, and often results in a loss of both clarity and efficiency. Modifying the operating system is seldom worthwhile for a single application. Augmenting the system from outside (in a library or language run time package) may be difficult as well; depending on the specific primitives provided and the functionality needed, even minor changes can be surprisingly hard to achieve [22]. Moving to another system is often not an option, and even when possible will generally preclude running applications to which the old system was well suited.

To meet the needs of different applications, many different models of parallel programming must be implemented on the same underlying hardware. To meet the needs of different applications simultaneously, these models must be implemented on a single operating system. To meet the needs of applications that use more than one model internally, the operating system must allow program fragments written under different models to interact in efficient and well-defined ways.

Applications that need not run concurrently can be supported by separate operating systems. Building these systems from scratch would involve enormous expense and duplication of effort, but the development of customizable operating systems (such as Choices [10] or the *x*-Kernel [15]) may make their construction practical. It may even prove practical to customize operating systems at run time, so that models can be changed without rebooting. Even so, it is unlikely that applications will be able to communicate across models supported by disjoint portions of the kernel interface.

An alternative approach is to assume a single kernel interface and then construct a library or language run time package for each model. We find this approach attractive, but believe it to be infeasible in existing systems, because existing interfaces are too narrow and high-level, and because environments that attempt to adapt these interfaces to their needs do so in different and incompatible ways.

Operating system designers have tended to target their interfaces to application programmers, not library and language implementors, and have therefore developed high-level mechanisms for communication and process management. These mechanisms are seldom amenable to change, and may not be well-matched to a new model under development, leading to awkward or inefficient implementations. The traditional Unix interface, for example, has been used beneath many new parallel

programming models [9, 19, 26]. In most cases the implementation has needed to compromise on the semantics of the model (*e.g.*, by blocking all threads in a shared address space when any thread makes a system call) or accept enormous inefficiency (*e.g.*, by using a separate Unix process for every lightweight thread).

Programming environments that need mechanisms not provided among the high-level operating system primitives may restrict themselves to some subset of the kernel interface (e.g. signals and non-blocking I/O in Unix), and then use that subset in stylized ways. This approach may allow the implementation of more than one model, but it can be awkward, and does not generally support interaction across models. Working outside the conventions of the kernel interface standard, the designers of independently-developed programming environments are almost certain to make incompatible decisions. They may use synchronization mechanisms, for example, that depend upon the details of a user-level thread package. Even if they express operations in terms of the kernel interface, they may use primitives (Unix domain sockets, for example) that are understood in one environment but not in another. Easy cooperation across environment boundaries would require that synchronization mechanisms and kernel primitives used in any environment be available in all, enlarging and potentially corrupting the associated programming models. Unless the set of kernel primitives is very small and general, inconsistency in the use of primitives by different programming environments will make cooperation extremely ad-hoc and difficult.

Since 1984 we have explored these issues while developing parallel programming environments for the BBN Butterfly multiprocessor. Using the Chrysalis operating system [4] as a low-level interface, we have created several new programming libraries and languages and ported several others [18]. We were able to construct efficient implementations of many different models of parallelism because Chrysalis allows the user to manage memory and address spaces explicitly, and provides efficient low-level mechanisms for communication and synchronization. Chrysalis processes are heavyweight, however, so lightweight processes must be encapsulated inside a heavyweight process. The resulting special-purpose scheduling packages and the need to avoid blocking kernel calls make it difficult to interact with the processes of other programming models. Moreover, Chrysalis provides no acceptable way to share code across processors, and no abstraction mechanism to provide conventions for the sharing of data.

Each of our programming models was developed in isolation, without support for interaction with other models. These experiences led to us design a new multiprocessor operating system, called Psyche, that would provide both an appropriate interface for implementing multiple models and conventions for interaction across models. The flexibility of Psyche stems from a kernel interface based on fundamental concepts supported by shared-memory hardware: data, processors, control flow via subroutine calls, and protection between address spaces. Rather than providing a fixed set of parallel programming abstractions to which programming environments would have to be adapted, this interface provides an abstraction *mechanism* from which many different user-level abstractions can be built *in user space*.

We believe that Psyche forms an ideal foundation for *multi-model parallel programming* — the simultaneous use of disparate models of parallelism, both across programs and within individual programs. Multi-model programming allows each application component to use the model most appropriate to its own individual needs. It also introduces the ability to interact across models when applications must communicate with server processes that use a different model and when a single application is composed of components using different models.

2. Related Work

Multi-model programming is related to, but distinct from, the work of several other researchers. Remote procedure call systems have often been designed to work between programs written in multiple languages [5, 14, 16, 20]. RPC-based systems support a single style of process interaction, and are usually intended for a distributed environment; there is no obvious way to extend them to fine-grained process interactions. Synchronization is supported only via client-server rendezvous, and even the most efficient implementations [7] cannot compete with the low latency of direct access to shared memory.

The Agora project [8] defines additional mechanisms for process interaction in a distributed environment. It allows processes to interact through events and certain kinds of stylized memory sharing. Its emphasis is on optimizing static relationships between processes and on providing the illusion of memory sharing through extensive software support. Because it is implemented on top of an existing operating system (Mach [1]), Agora must limit process interactions to those that can be represented effectively in terms of encapsulated Mach primitives.

Mach is representative of a class of operating systems designed for parallel computing. Other systems in this class include Amoeba [21], Chorus [2], Topaz [27], and V [11]. To facilitate parallelism within applications, these systems allow more than one kernel-supported process to run in one address space. To implement minimal-cost threads of control, however, or to exercise control over the representation and scheduling of threads, coroutine packages must still be used within a single kernel process. Psyche provides mechanisms unavailable in existing systems to ensure that threads created in user space can use the full range of kernel services (including those that block), without compromising the operations of their peers. In contrast to existing systems, Psyche also emphasizes data sharing between applications as the default, not the exception, distributes access rights without kernel assistance, and checks those rights lazily. Protection is provided only when the user indicates a willingness to pay for it; Psyche presents an explicit tradeoff between protection and performance.

Washington's Presto system [6] is perhaps the closest relative to Psyche, at least from the point of view of an individual application. Presto runs on a shared-memory machine (the Sequent Symmetry), and allows its users to implement many different varieties of processes and styles of process interaction in the context of a single C++ application. As with Agora, however, Presto is implemented on top of an existing operating system, and is limited by the constraints that that system imposes. Where Agora relies on operations supported across protection boundaries by the operating system, Presto works within a single language and protection domain, where a wide variety of parallel programming models can be used. Psyche is designed to provide the flexibility of Presto without its limitations, allowing programs written under different models (e.g. in different languages) to interact while maintaining protection.

This paper uses a series of examples to illustrate how different parallel programming models can be built on top of Psyche. In the following section we overview the Psyche kernel interface, explaining how it differs from more conventional systems. (Additional details and design rationale can be found in other papers [24, 25].) Using two different models as examples (lightweight threads in a single shared address space and heavyweight processes that communicate with messages), we show how dissimilar models can be implemented on top of Psyche. We then describe the implementation of a parallel data structure that can be shared by programs that use different models. Finally we summarize the specific features of Psyche that facilitate multi-model programming.

3. Psyche Overview

Psyche is intended to provide a common substrata for parallel programming models implemented by libraries and language run-time packages. As a result, Psyche has a low-level kernel interface. We do not expect application programmers to use the kernel interface directly. Rather, we expect that they will use languages and library packages that implement their favorite programming models. The purpose of the low-level interface is to allow new packages to be written on demand (by somewhat more sophisticated programmers), and to provide well-defined underlying mechanisms that can be used to communicate between models when desired.

3.1. Kernel Interface Description

The Psyche kernel interface is based on four abstractions: the *realm*, the *protection domain*, the *virtual processor*, and the *process*. Realms form the unit of code and data sharing. Protection domains are a mechanism for limiting access to realms. Processes are user-level threads of control. Virtual processors are kernel-level abstractions of physical processors, on which processes are scheduled. Processes are implemented in user space; the other three abstractions are implemented in the kernel.

Each *realm* contains code and data. In general, it is expected that the code will provide a collection of operations that constitute a protocol for accessing the data. If a realm has no code then any needed protocol must be established by convention among the users of the data. Since all code and data is encapsulated in realms, all computation consists of the invocation of realm operations. Interprocess communication is effected by invoking operations of realms accessible to more than one process.

Depending on the degree of protection desired, an invocation of a realm operation can be as fast as an ordinary procedure call, termed *optimized* invocation, or as safe as a remote procedure call between heavyweight processes, termed *protected* invocation. Unless the caller explicitly asks for protection, the two forms of invocation are initiated in exactly the same way, with the native architecture's jump-to-subroutine instruction. The kernel implements protected invocations by catching and interpreting page faults.

A *process* in Psyche represents a thread of control meaningful to the user. A *virtual processor* is a kernel-provided abstraction on top of which processes are implemented. There is no fixed correspondence between virtual processors and processes. One virtual processor will generally schedule many processes. Likewise, a given process may run on different virtual processors at different points in time. As it invokes protected operations, a process moves through a series of *protection domains*, each of which embodies a set of access rights appropriate to the invoked operation. Each domain has a separate page table, which includes precisely those realms for which the right to perform optimized invocations has been verified by the kernel in the course of some past invocation.

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.

In order to verify access rights, the kernel checks to see whether an appropriate key can be found in a special data structure shared (writably) between the user and the kernel. Once authorization has been granted, access checks are not performed for subsequent realm invocations, even those that are protected (and hence effected by the kernel). To facilitate sharing of arbitrary realms at run time, Psyche arranges for every realm to have a unique system-wide virtual address. This *uniform addressing* allows processes to share pointers without worrying about whether they might refer to different data structures in different address spaces.

Every protection domain has a distinguished initial realm, called its *root*. When a process performs a protected invocation of some operation of a realm, it moves to the protection domain rooted by that realm. The domain therefore contains processes that have moved to it as a result of protected invocations, together with processes that were created in it and have not moved. Within each domain, the representations of processes are created, destroyed, and scheduled by user-level code. As a process moves among domains, it may be represented in many different ways — as lightweight threads of various kinds, or requests on the queue of a heavyweight server. The kernel keeps track of the call chains of processes that have moved between protection domains (in order to implement returns correctly), but it knows nothing about how processes are represented or scheduled inside domains, and is not even aware of the existence of processes that have not moved.

In order to execute processes inside a given protection domain, the user must ask the kernel to create a collection of *virtual processors* on which those processes can be scheduled. The number of virtual processors in a domain determines the maximum level of physical parallelism available to the domain's processes. On each physical node of the machine, the kernel time-slices among the virtual processors currently located on that node. (The allocation of virtual processors to physical processors is usually established by the kernel, but may be specified by certain user-level programs.) Because of time-slicing, the progress of a given virtual processor may be erratic, but each receives a comparable share of available CPU resources. A data structure shared between the kernel and the user contains an indication of which process is being served by the current virtual processor. This indication can be changed in user code, so it is entirely possible (in fact likely) that when execution enters the kernel the currently running process will be different from the one that was running when execution last returned to user space. The kernel's help is not required to create or destroy processes within a single protection domain, or to perform context switches between those processes.

Communication from the kernel to the virtual processors takes the form of signals that resemble software interrupts. A software interrupt occurs when a process moves to a new protection domain, when it returns, and whenever a kernel-detected error occurs. In addition, user-level code can establish interrupt handlers for wall clock and interval timers.

The interrupt handlers of a protection domain are the entry points of a scheduler for the processes of the domain. Protection domains can thus be used to provide the boundaries between distinct models of parallelism. Each scheduler is responsible for the processes in its domain at the current point in time, managing their representations and mapping them onto the virtual processors of the domain. Realms are the building blocks of domains, and define the granularity at which domains can intersect.

3.2. Psyche Programming Environment

Like conventional systems such as Unix, Psyche assumes that programs are executed in the context of a standard programming environment. The uniform virtual address space requires that programs be position independent, or else be dynamically relocated when loaded into memory. It also presents the opportunity to dynamically link new applications to already-loaded code. Name service is needed for running programs that need to find each other, and command interpreters are required for interactive use.

The standard execution environment for Psyche programs is illustrated in figure 1. It includes a shell, a linker, and a name server. The linker is created at system boot time as the primordial realm, with a corresponding protection domain and initial virtual processor. As part of its initialization, the linker creates the rest of the execution environment, including the default shell and the name server. The linker maintains a table of distinguished symbols that can be used to resolve external references in applications. Among these symbols are the entry points

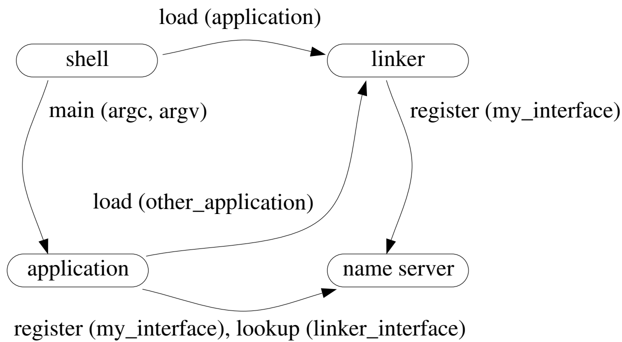

Figure 1: Psyche Execution Environment

of the name server and of the linker itself. Additional symbols might include a default file server or reentrant C libraries. Once initialization is complete, the linker enters server mode, in which it accepts external requests from the shell or other user programs.

The shell accepts commands from a user. It communicates with the linker to load new applications, and with the loaded applications to begin their execution. The name server provides a string registry that can be accessed via protected invocations. Its principal purpose is to maintain a listing of services that are not universal enough to be included in the linker's symbol table, or that may need to be accessed by already-loaded programs. The *name_server.register* operation creates an association between a key and a value string, typically a symbolic service name and a realm operation address. The *name_server.lookup* operation can be used to retrieve associations.

Applications are compiled into relocatable object files, suitable for dynamic linking. In response to a typed command, the shell asks the linker to load a user program. The linker reads the header of the object file to discover the size of the program. It uses a system call to create an empty realm (which the kernel places at an address of its choosing), and then loads the program into the realm, resolving external symbols. At this point the linker returns to the shell. The realm itself is still passive. The shell uses another system call to create an initial virtual processor in the protection domain of which the new realm is the root. The kernel immediately provides this virtual processor with a software interrupt at a handler address defined in a table at the beginning of the realm. This handler initializes the program to accept protected invocations before re-enabling software interrupts. The shell performs a protected invocation to the program's main entry point, whose address was returned by the linker. In most programs it is expected that the main entry point will be in a library routine that initializes standard I/O and repackages command line arguments before branching to the user's main routine. If desired, the main routine can then register an external interface with the name server.

3.3. Discussion

As Unix-like systems are developed for multiprocessors, a consensus is emerging on multiple kernel-supported processes within an address space. Amoeba, Chorus, Mach, Topaz, and V all take this approach. Most support some sort of memory sharing between processes in different address spaces, but message passing or RPC is usually the standard mechanism for synchronization and communication across address-space boundaries.

On the surface there is a similarity between the Psyche approach and these modern conceptions of Unix. A protection domain corresponds to an address space. A virtual processor corresponds to a kernel-provided process. Protected procedure calls correspond to RPC. The correspondence breaks down, however, in three important ways.

Ease of Memory Sharing. Uniform addressing means that pointers do not have to be interpreted in the context of a particular address map. Without uniform addressing, it is impossible to guarantee that an arbitrary set of processes will be able to place a shared data structure at a mutually-agreeable location at run time. The key and access list mechanism, with its lazy checking of access rights, means that processes do not have to pay for checks on things they don't actually use, nor do they have to realize when they are using something for the first time, in order to ask explicitly for access. Pointers in distributed data structures can be followed without worrying about whether access checking has yet been performed for the portion of the data they reference.

Uniformity of Invocation. Optimized and protected invocations share the same syntax and, with the exception of protection and performance issues, the same semantics. No stub generators or special compiler support are required to implement protected procedure calls. In effect, an appropriate stub is generated by the kernel when an operation is first invoked, and is used for similar calls thereafter. As with the lazy checking of access rights, this late binding of linkage mechanisms facilitates programming techniques that are not possible with remote procedure call systems. Function pointers can be placed in data structures and can then be used by processes whose need for an appropriate stub was not known when the program was written.

First Class User-Level Threads. Because there are no blocking kernel calls, a virtual processor is never wasted while the user-level process it was running waits for some operation to complete. Protected invocations are the only way in which a process can leave its protection domain for an unbounded amount of time, and its virtual processor receives a software interrupt as soon as this occurs. These software interrupts provide user-level code with complete control over the implementation of lightweight processes, while allowing those processes to make standard use of the full set of kernel operations. In section 6 we describe a convention that provides the same degree of control for operations that are initiated by optimized invocations, but must block for condition synchronization.

In a system with more than one kernel-level process per address space, the programmer who wants more than one user-level process must choose how to map user processes onto those provided by the kernel. One option is to map them one-to-one. Unfortunately, this approach implies significant costs for process creation, destruction, synchronization, and context switching. Though these costs are lower than the corresponding costs for an old-style Unix process (because address space changes are no longer involved), they are still large enough to be a serious concern. A second option is to use a coroutine-style package to map many user-level processes onto one or more kernel-level processes [12]. This option either forces an entire collection of user-level processes to wait when one of them performs a blocking operation, or else forces the programmer to cast all operations in terms of some non-standard non-blocking subset of the kernel interface. In the latter case it becomes difficult or impossible to interact with programming environments based on other subsets of the kernel interface, to share data structures that require condition synchronization with other environments, or to use standard library packages.

A third possible mapping uses a hybrid of the other two options, multiplexing user-level processes on top of one or more kernel-level processes up to the point at which a blocking operation must be performed, and then interacting with a separate pool of kernel-level processes that serve as blockable surrogates for their peer(s). This option requires that the user decide up front how many blocking processes to create, or else adjust the size of the pool dynamically. It also requires that user code know ahead of time which operations will block. It requires communication and synchronization between kernel-level processes in order to pass a blocking operation off to a surrogate.

The following three sections illustrate the advantages of the Psyche approach through a series of examples.

4. Lightweight Processes in a Shared Address Space

Programming models in which a single address space is shared by many lightweight processes, such as Presto [6] and the Uniform System [28], can be implemented easily and efficiently on Psyche. A lightweight process scheduler can be implemented as a library package that is linked into an application, creating a single realm and protection domain whose virtual processors share both the scheduling code and a central ready list. Virtual processors can run whatever process is available, regardless of where it ran last. Access to the ready list must be synchronized, but in most other respects the scheduler resembles a traditional coroutine package. Each process is represented by a context block that includes room to save volatile registers and any other data (such as display pointers) that the source language requires.

The scheduler interface contains routines to *create* a new process, *destroy* the current process, *block* the current process, and *unblock* a specified process. The *destroy* and *block* routines cause a context switch to another process, busy-waiting if none is available. All four routines require little more than a subroutine call, with no assistance from the kernel.

For the sake of simplicity, we assume that our code is designed to be used in applications started by the shell, which cooperates with the linker to create the application's realm, protection domain, and initial virtual processor. The virtual processor receives an initialization interrupt from the kernel. It initializes the ready list and any other scheduler data structures, and then creates a virtual processor on every physical processor of the machine, ensuring the application maximum available parallelism. Each new virtual processor receives its own initialization interrupt and waits for processes to appear on the ready list.

After creating the initial virtual processor, the shell creates the first lightweight process in the address space by invoking the application's main entry point. The kernel delivers an invocation interrupt to a virtual processor in the application's protection domain. The code to handle this interrupt uses the *create* operation to allocate space for a process that has moved into the protection domain of the lightweight process package from outside (in this case from the shell). To avoid copying arguments, the invocation handler could allocate a new interrupt stack for future use and give the old stack (with arguments in place) to the newly-arrived process. Other processes can be created from within the application (by simply calling the *create* operation) or can arrive from outside the application via an invocation interrupt (as in the case of the shell providing the initial process).

To synchronize access to shared data structures (including those of the scheduler), we can build semaphores in user space using spin locks and the scheduler's *block* and *unblock* routines. We assume the availability of an atomic hardware instruction like test-and-set. A mechanism for ensuring that no process is preempted while holding a spin lock is described in section 6.

4.1. Enhancements

The scheduling package above is deliberately simplistic; it illustrates how simple scheduling policies and mechanisms can be implemented with very little code and no interaction with the kernel. Virtual processors spin when the ready list is empty. They share a single list. They switch processes only when the current process blocks or terminates explicitly. Each of these simplifying assumptions can be relaxed with a small amount of additional code. In addition, the basic kernel interface allows a process to invoke operations of other protection domains without blocking its virtual processor. It also allows processes in other protection domains to invoke operations provided by the lightweight process application.

Yielding When the Ready List is Empty

A virtual processor can spin when the ready list is empty, just as a physical processor executes an idle task when it has nothing else to do. Since physical processors are multiprogrammed, a spinning virtual processor only chews up cycles to which it would be entitled in any event. An obvious enhancement is for a virtual processor to relinquish the physical processor for the rest of its scheduling quantum when there is nothing on the ready list. To avoid the overhead of checking the list once per quantum, the virtual processor could use a kernel call to block pending interrupts instead. In this case, the lightweight process package would need to keep a count of the number of virtual processors that are blocked, and awaken one when inserting a process onto a previously empty ready list.

Multiple Ready Lists

A single ready list is simple, and allocates work fairly among virtual processors, achieving the effect of load balancing without any effort on the kernel's part. A single ready list may be inappropriate, however, on a machine in which there is a non-trivial cost associated with running a process on a different processor than the one on which it last ran. Such cost may be due to state built up in a cache or TLB, or in pages that have been moved to local memory on an architecture with non-uniform memory access speeds. It is straightforward to replace the single ready list with a collection of lists, one per virtual processor. New processes created by the application (and newly-arrived processes from the invocation interrupt handler) are added to the shortest ready list. Each virtual processor adds and deletes processes on its own ready list, examining other lists only when their lengths become heavily unbalanced.

Preemption

Each process can be given a quantum at the end of which the virtual processor will be given to another process. The kernel interface associates interval timers with virtual processors for this purpose. The timer value lies in data structures shared between the virtual processor and the kernel. The kernel decrements it on every hardware clock interrupt. The user may also set it to any desired value. When the interval has expired, the virtual processor receives a software interrupt. The timer expiration handler will yield the virtual processor, putting the current process at the end of the ready list, resetting the interval timer, and selecting a new process to run. The *block* routine will also reset the timer before transferring to another process. The kernel must decrement the current timer value on each clock tick and deliver an interrupt at expiration. All other aspects of lightweight process preemption occur in user space.

4.2. Crossing Protection Domains

In Psyche, I/O and all other potentially blocking operations are implemented in user space through protected invocations (*e.g.*, of read and write operations in a file server realm). When a lightweight process invokes a protected operation the kernel provides its virtual processor with an interrupt indicating that the process has moved to another protection domain. The process state local to the invoking domain can be saved until the invocation returns. In the interim, the virtual processor can be used to execute other processes. When the invoking process finally returns, the kernel delivers a software interrupt to the virtual processor that last ran the process in its old protection domain. The handler for that interrupt looks up the context of the process and unblocks it, placing it back on the ready list for later execution.

We have already mentioned the interrupt handler that allows our lightweight process package to handle invocations from other protection domains (it allowed the shell to provide our initial process). The invocation handler routine allocates space for the newly-arrived process as if it had been created locally, and adds it to the ready list. While executing the requested operation, a virtual processor indicates in data structures shared with the kernel that it is running the newly-arrived process.

To switch processes in user mode, a virtual processor simply changes the name of the current process in the data structure it shares with the kernel. The kernel is therefore always able to tell

which process was responsible when a trap occurs. If the trap represents a protected invocation, the kernel remembers that the process has moved to another protection domain. Any future trap in the current domain that claims to be caused by the departed process is treated as an error. Because a process is merely a name from the kernel's point of view, new processes can be created simply by inventing new names. So long as lightweight processes execute inside their own protection domain, they can be created, destroyed, and scheduled without intervention by the kernel. When one of these processes invokes an operation outside the protection domain, the kernel delivers an interrupt to allow the scheduler to take action.

5. Heavyweight Processes with Message Passing

Psyche can be used to implement heavyweight processes communicating with messages. Consider a system with the following characteristics. Each process operates in its own address space. Processes are connected by two-way communication channels called *links*. There are three main message-passing primitives: *send*, *receive*, and *wait*. *Send* and *receive* each specify a buffer address, a length, and a link. *Send* indicates that the contents of the specified buffer should be sent to the process at the other end of the link. *Receive* indicates that the specified buffer should be filled by a message received from the process at the other end of the link. *Wait* blocks the process until some outstanding send or receive request has completed.

These semantics are a subset of the message-passing primitives of the Charlotte distributed operating system [3]. They can be implemented as follows. Each Charlotte process is implemented as a separate protection domain with a single virtual processor, running a single Psyche process. A diagram of one such protection domain (and its interactions with others) appears in figure 2. Each link is implemented as a realm containing only data. The process that creates the link realm also invents a key that will permit optimized access to the link. It gives both the address of the realm and the key to the processes at the ends of the link, allowing those processes to open the realm for optimized access from their protection domains. Each link realm contains two buffers, one for a message in each direction, and a small amount of status information that indicates which buffers are full and which processes have an outstanding receive request on the link.

To receive a message, a process looks in the link realm to see if the appropriate buffer is full. If so, it copies the data into its own local buffer and sets a flag indicating that the link buffer is now empty. It then performs a (protected) invocation of an operation in the sender's protection domain to inform it that the send request has completed. (It finds the address of this operation, together with a key permitting protected invocation, in data structures of the link realm.) If on the other hand the link buffer is empty when the receiver first inspects it, the receiver leaves status information indicating that it has a receive request outstanding, specifying the location and size of its local buffer. In either event, the *receive* operation does not block.

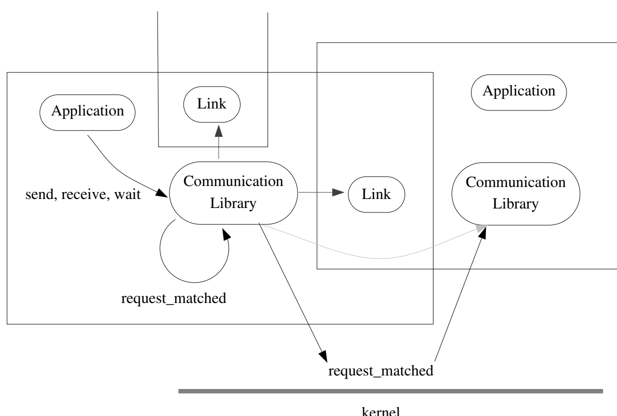

Figure 2: Implementation of a Message-Passing Model

To send a message, a process gathers its data into the appropriate buffer in the link realm and sets a flag indicating that the message is available. If then checks to see if the process at the other end of the link has a receive request outstanding. If so, it performs a (protected) invocation of an operation in the receiver's protection domain to inform it that the receive request has completed. Whether the receiver has a request outstanding or not, the *send* operation returns without blocking.

When a *send* or *receive* operation discovers that a matching request is already outstanding in the process at the other end of the link, it makes a note in data structures local to the current process, indicating that a transaction has occurred, and can be reported by a subsequent *wait* operation. As noted above, a protected call is also made into the other process, allowing it to make a similar note in its own data structures. When either process performs a *wait* operation, it can tell if any transactions have already occurred. If none have, it calls a Psyche kernel operation to block the virtual processor pending the arrival of interrupts. It will receive an interrupt (for a protected invocation) as soon as some communication partner performs an operation to match one of its outstanding send or receive requests.

When a process performs a protected invocation to inform one of its communication partners that a send or receive request has been matched, that process moves to the partner's protection domain. In the previous section we allocated a context block and stack to represent each new arrival. In this new example, we can treat each arriving Psyche process as simply a request for service. We need not give it state. If requests do not interfere with current activity, they can be serviced immediately by the interrupt handler. Otherwise, the handler can place them in a queue for synchronous examination later.

If we prefer a more sophisticated approach to process management and message passing, the communication library and link realms can easily be modified to provide the communication semantics of the Lynx distributed programming language [23]. Lynx subdivides each heavyweight process into lightweight threads of control, not to improve performance through parallelism, but as a program structuring tool. The threads use coroutine semantics: one thread runs until it blocks, then another runs. Lynx also uses an RPC style of message passing: messages come in matched request-reply pairs. A thread that initiates an RPC waits until a reply message is received, but a control transfer allows another thread to run in the meantime, so the heavyweight process is not blocked. Seen from outside, a heavyweight Lynx process resembles a Charlotte process closely, though not exactly.

Lynx was first implemented on top of Charlotte at the University of Wisconsin. The implementation was complex and difficult, because the operations provided by the operating system were not quite what the language wanted, and their semantics could not be changed [22]. For example, there are times when a Lynx process is only interested in receiving replies to RPCs it has initiated, and is not willing to service RPCs requested by anybody else. Charlotte semantics provide no way to differentiate between requests and replies, so unwanted requests must be returned to the sender through a complicated protocol.

In Psyche, the link realm described above can be modified to include separate buffers for requests and replies in each direction, with appropriate operations to fill and empty them. The communication library realms can be modified to provide *rpc*, *receive_request*, and *send_reply* operations instead of *send* and *receive*. The changes are straightforward, and produce an implementation almost identical to that used to support Lynx on top of

the Chrysalis operating system on the BBN Butterfly.

The ease with which the Charlotte style of message passing can be converted to the Lynx style of message passing illustrates the flexibility of Psyche. Implementing Lynx on Charlotte was difficult because the communication semantics were fixed by the kernel and not subject to even slight modification. The implementation of Lynx on Chrysalis was comparatively easy, but it suffers from problems of its own. It depends on an unsafe Chrysalis queue mechanism to achieve the effect of protected calls to match send and receive requests. It is incompatible with other models of programming on the Butterfly because Chrysalis provides no conventions for synchronizing access to shared data without understanding how other models are implemented. Finally, it implements Lynx threads as coroutines outside the knowledge of the operating system. If a thread performs a blocking operation (like I/O), it blocks the entire Lynx process. In Psyche, each Lynx thread would use a separate Psyche process name, and any blocking operation it performed would result in a software interrupt that would allow the virtual processor that represents the Lynx process to switch to a different Psyche process, representing a different Lynx thread.

6. Shared Parallel Priority Queue

Psyche can be used to implement parallel data structures that can be accessed from several different programming models simultaneously. In this example we describe a parallel priority queue implementation. This example illustrates how data structures are shared among different programming models and also how the Psyche interface incorporates cooperation between the operating system and the application in the implementation of scheduling.

The algorithm for concurrent priority queues can be adapted from [17]. The external interface of the queue includes two operations: *insert* and *deletemin*. The implementation uses a heap ordered binary tree. *Deletemin* removes and returns the smallest element in the queue by deleting the root of the tree and melding the two remaining subtrees. *Insert* adds a new element to the queue by melding the existing tree with a new one item tree. Operations on the priority queue are allowed to proceed concurrently; each operation need only hold exclusive access to some subset of the queue's data structure. Exclusive access is implemented using binary semaphores associated with nodes in the heap.

To implement a parallel priority queue in Psyche, we must first consider how the queue will be used. If the queue is intended for use within a single program or protection domain, where compiler protection is sufficient, efficiency may be more important than protection. If the queue will be used simultaneously by processes in different protection domains, under different programming models, then the implementation should include firewalls to protect the abstraction.

These implementation issues are analogous to those that arise in modular programming languages. In Modula-2, for example, the implementor of an abstraction must decide whether or not the abstraction should be represented by an opaque type. The choice is usually based on how the abstraction is to be used and on the likelihood of changes in the implementation of the abstraction. With an opaque type, implementation details are hidden by the abstraction; the interface must provide all operations that allocate and manipulate instances of the type. As a result, changes to the implementation can be made without affecting the client, but a procedure call, and its associated overhead, is required for every operation on the type. If a transparent type is used, clients can easily augment or modify the implementation to tailor it to specific cases. Although transparent types introduce the potential for corruption of the abstraction, they allow the client to bypass the procedural interface and optimize the implementation for a particular need.

In the original version of the parallel priority queue algorithm, a transparent type was used, so that clients could allocate data items among their own variables and pass pointers to them to the priority queue, rather than passing the contents of the items. The implementation could then simply chain together items from the clients. Such an implementation is appropriate in Psyche if all clients reside in a single protection domain, where mutual trust exists. In that case, clients could use optimized invocations to access the data structure; only procedure call overhead would be incurred.

If the data structure is to be used across several programming models and protection domains, a priority queue server realm with its own local allocation of data items would be more appropriate. (Pointers to client-created data items could not be used, because clients in one protection domain cannot follow pointers into an inaccessible portion of another.) The priority queue realm could be accessed via either protected or optimized invocations. In the former case, the queue would be isolated in its own protection domain. In the latter case, it would lie in the intersection of the domains of all of its clients.

Placing the queue in its own protection domain would ensure that modifications to its data structures happened *only* as a result of legitimate invocations of queue operations. Placing the queue in the intersection of its clients' domains would only establish conventions. In either case, malicious clients would be able to send each other invalid information through the queue, but would be unable to modify each other's data structures directly. In the protected invocation case, clients would also be unable to destroy the structural integrity of a properly-implemented queue. With optimized invocations they would be able to modify the data structures of the queue directly, bypassing the normal access routines. The advantage of optimized invocations, of course, is that they are much faster.

It is worth noting that trust relationships between the data structure and its clients are not reflected in the calls to the operations on the data structure. Optimized and protected invocations both take the form of ordinary procedure calls. However, as in modular programming, trust relationships dictate how and where storage is allocated, and how data is represented within the data structure.

Let us assume that the priority queue realm will be included in the protection domains of its clients, who will access it with optimized invocations. No protection boundary will normally be crossed. The virtual processors that run processes executing the code of the queue will belong to client protection domains. The level of concurrency within the realm will depend on the level of concurrency among the clients, and on synchronization constraints in the code.

In an earlier example we assumed an implementation of semaphores for use by threads within a shared address space. In this example, different types of processes may access the data structure and be forced to wait on a semaphore. Since the semantics of blocking may differ among the various types of processes that access the semaphore, our semaphore implementation must provide some mechanism to dynamically tailor the implementation to the type of process.

By convention, each Psyche protection domain defines procedures to block and unblock its processes. These procedures are part of the scheduling code of the domain. Pointers to them are stored in each process context block. When a process attempts to perform a P operation on an unavailable semaphore, the code for that operation writes the address of the *unblock* routine of the current process into the data of the semaphore and then calls the *block* routine. Because it affects the current process, *block* is always called within the current protection domain, and requires no kernel intervention. When the semaphore implementation must unblock the process (because some other process has performed a V operation), the appropriate (saved) *unblock* routine is called. This call may be either an optimized or protected invocation, depending on whether the process performing the V operation is in the same protection domain as the process that called the P operation. Because it manipulates the data structures used to represent processes, the *unblock* routine is unlikely to be made available for optimized access from outside its domain.

A diagram of our implementation appears in figure 3. Solid arrows represent optimized invocations (*i.e.* procedure calls) and light grey arrows (with accompanying arrows into and out of the kernel) represent protected invocations. In this figure, however, the arrow representing a call to *unblock* may also be optimized, if the process to be unblocked is in the current protection domain.

The discussion accompanying the original parallel priority queue algorithm pointed out that the most appropriate implementation for mutual exclusion is spin locks, not binary semaphores. Unfortunately, spin locks in the presence of preemption[1] can cause a significant performance penalty [29]. A process holding the lock could be preempted, causing other processes to spin on a lock that cannot be released. In addition, pent-up demand for the lock increases contention when it is finally released. As a result, spin locks are rarely used unless the operating system dedicates processors to a single application. Fortunately, Psyche supports the ''close alliance between the system's processor allocator and the application's thread scheduler'' called for in [29] to solve the problem of spin locks in the presence of preemption. Since the kernel and user share a data structure describing the state of the executing virtual processor, it is easy for the clock handler to set a flag when the virtual processor is close to the end of its quantum. The application can examine this flag to determine if preemption within a critical section is likely. The code to acquire a naive test-and-set lock would take the following form:

```
repeat
    if preemption_flag
        yield
until test_and_set (lock) = 0
```

7. Conclusions

We have attempted with the examples in this paper to illustrate how special features of Psyche facilitate the construction of multi-model programs. We can summarize the effect of these features as facilitating sharing within and between protection domains, simplifying communication with the kernel and between protection domains, and moving process scheduling out of the kernel and into user-level code.

Facilitating Sharing

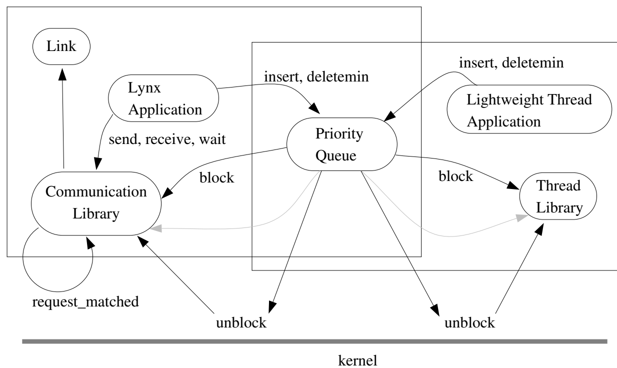

Figure 3: Implementation of a Shared Priority Queue

[1]Although our data structure might be used by a programming model that provides non-preemptable processes, those processes execute on virtual processors that can be preempted by the kernel. The use of spin locks in any programming environment on Psyche must take this fact into account.

Processes interact via shared memory and procedure calls. Popular communication mechanisms have straightforward implementations, and any programming model that supports procedure calls can use this common base to invoke operations implemented under any other model.

Protection domains share a uniform virtual address space. Every realm has a unique address, and can be incorporated without conflict into any protection domain with the right to access it.

Access rights are checked only when necessary. Lazy examination of key and access lists means that a process pays for access checking only on the data and operations it actually uses, not for the ones it might potentially use. We can afford to disseminate keys liberally, because we don't need the kernel's help to do so. Because access checking occurs automatically, the user need not keep track of which realms have already been accessed.

Simplifying Communication

Protected and optimized invocations are syntactically the same. In the Lynx example, communication routines can use the same syntax to invoke the local and remote *request_matched* operations. In the priority queue example, an *insert* operation on an empty queue can call an *unblock* operation without worrying about whether that operation is in the current protection domain or not.

The kernel and the user share data. Descriptive data structures allow the kernel to obtain key lists, access lists, process names, software interrupt vectors, and realm interface descriptions without requiring the user to provide them as explicit arguments to system calls. A virtual processor preemption warning flag allows us to use spin locks efficiently in user space. Shared flags and counters allow us to disable interrupts or ask for timer interrupts without a kernel trap. A *current_process_name* variable allows us to switch between processes without a kernel trap. Kernel-maintained data allows us to read the time, compute load averages, or examine other run-time statistics without a kernel trap.

Scheduling in User Space

There is a standard interface for block and unblock routines. Shared data structures like the priority queue can incorporate synchronization code that works with any kind of process.

A virtual processor receives an interrupt when its process blocks for a protected invocation. It can then switch to another, runnable process. This means that lightweight processes get first-class treatment from the kernel.

Other scheduling events cause interrupts as well. Timer expiration interrupts allow preemption. Protected invocation interrupts allow asynchronous communication from other protection domains. The kernel interface itself is entirely non-blocking. User-level code controls every aspect of scheduling other than the implementation of virtual processors.

Our first major user application was implemented in the department's robotics laboratory in the fall of 1989. It combines binocular camera input, a pipelined image processor, a 6-degree-of-freedom robot arm, and a competing agent model of motor control to bounce a balloon suspended from the ceiling on a string. We expect to tune the features of our kernel interface as we accumulate experience with additional applications written by programmers outside the Psyche group. Our experience with previous systems and applications indicates that multi-model programming is important. Our example programs and our work to date in the robot lab suggest that Psyche supports it well.

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, pp. 93-112.

[2] R. Armand, M. Gien, R. Herrmann and M. Rozier, ''Revolution 89, or 'Distributing UNIX Brings it Back to its Original Virtues','' *Proceedings of the First USENIX Workshop on Experiences Building Distributed and Multiprocessor Systems*, 5-6 October, 1989, pp. 153-174.

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

[4] BBN Advanced Computers Incorporated, ''Chrysalis® Programmers Manual, Version 4.0,'' Cambridge, MA, 10 February 1988.

[5] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo and M. Schwartz, ''A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems,'' *IEEE Transactions on Software Engineering SE-13*:8 (August 1987), pp. 880-894.

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

[7] B. N. Bershad, T. E. Anderson, E. D. Lazowska and H. M. Levy, ''Lightweight Remote Procedure Call,'' *ACM Transactions on Computer Systems 8*:1 (February 1990), pp. 37-55. Also in *ACM SIGOPS Operating Systems Review 23*:5; originally presented at the *Twelfth ACM Symposium on Operating Systems Principles*, 3-6 December 1989.

[8] R. Bisiani and A. Forim, ''Multilanguage Parallel Programming,'' *Proceedings of the 1987 International Conference on Parallel Processing*, 17-21 August 1987, pp. 381-384.

[9] A. P. Black, ''Supporting Distributed Applications: Experience with Eden,'' *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, 1-4 December 1985, pp. 181-193. In *ACM SIGOPS Operating Systems Review 19*:5.

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

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

[12] T. W. Doeppner, Jr., ''Threads: A System for the Support of Concurrent Programming,'' Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[13] D. Gelernter, ''Generative Communication in Linda,'' *ACM Transactions on Programming Languages and Systems 7*:1 (January 1985), pp. 80-112.

[14] R. Hayes and R. D. Schlichting, ''Facilitating Mixed Language Programming in Distributed Systems,'' *IEEE Transactions on Software Engineering SE-13*:12 (December 1987), pp. 1254-1264.

[15] N. C. Hutchinson and L. L. Peterson, ''Design of the *x*-Kernel,'' *Proceedings of the SIGCOMM '88 Symposium*, August 1988, pp. 65-75.

[16] M. B. Jones, R. F. Rashid and M. R. Thompson, ''Matchmaker: An Interface Specification Language for Distributed Processing,'' *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, January 1985, pp. 225-235.

[17] D. W. Jones, ''Concurrent Operations on Priority Queues,'' *Communications of the ACM 32*:1 (January 1989), pp. 132-137.

[18] 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, Languages and Systems*, 19-21 July 1988, pp. 161-172.

[19] B. Liskov, D. Curtis, P. Johnson and R. Scheifler, ''Implementation of Argus,'' *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, 8-11 November 1987, pp. 111-122. In *ACM SIGOPS Operating Systems Review 21*:5.

[20] B. Liskov, R. Bloom, D. Gifford, R. Scheifler and W. Weihl, ''Communication in the Mercury System,'' *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, January 1988, pp. 178-187.

[21] 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.

[22] 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.

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

[24] M. L. Scott, T. J. LeBlanc and B. D. Marsh, ''Design Rationale for Psyche, a General-Purpose Multiprocessor Operating System,'' *Proceedings of the 1988 International Conference on Parallel Processing*, V. II – Software, 15-19 August 1988, pp. 255-262.

[25] M. L. Scott, T. J. LeBlanc and B. D. Marsh, ''Evolution of an Operating System for Large-Scale Shared-Memory Multiprocessors,'' TR 309, Computer Science Department, University of Rochester, March 1989.

[26] M. Shapiro, ''Prototyping a Distributed Object-Oriented OS on UNIX,'' *Proceedings of the First USENIX Workshop on Experiences Building Distributed and Multiprocessor Systems*, 5-6 October, 1989, pp. 311-331.

[27] C. P. Thacker and L. C. Stewart, ''Firefly: A Multiprocessor Workstation,'' *IEEE Transactions on Computers 37*:8 (August 1988), pp. 909-920. Originally presented at the *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 5-8 October 1987.

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

[29] J. Zahorjan, E. D. Lazowska and D. L. Eager, ''The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Processors,'' TR 89-07-03, Department of Computer Science, University of Washington, July 1989.