Threads
A System for the Support of Concurrent Programming

Thomas W. Doeppner Jr.
Department of Computer Science
Brown University
Providence, RI 02912

June 16, 1987

Technical Report CS-87-11

Threads – A System for the Support of Concurrent Programming

Abstract

We describe a system, Threads, whose purpose is to make concurrency a practical programming concept. This system, which has been implemented on workstations and a shared-memory multiprocessor, supports the concept of a *thread*, which is an independent unit of execution, capable of concurrent execution with other threads. Threads have a very inexpensive implementation, allowing a programmer to write programs with a fair amount of concurrency. A major feature of the system is its support of concurrent programming abstractions – both the design of the particular abstractions we have implemented and how we allow the programmer to define new abstractions.

1. Introduction

The purpose of the Threads system is to make concurrency a practical programming concept. In order to achieve this, concurrency must be supported cheaply and there must be an attractive abstraction for using it. The Threads system supports the concept of a *thread of control* (or *thread*), which is an independent unit of execution, capable of concurrent execution with other threads. Our implementation is on top of workstations running Berkeley UNIX (it currently runs on Suns, MicroVAXes and Apollos) as well as on a shared-memory multiprocessor – the Encore Multimax. It has proved to be fast enough to satisfy the needs of several projects at Brown and is about to be distributed to researchers at other institutions. The programming interface provided by Threads insulates the user from such details as the number of processors being used: programs written for uniprocessors normally work correctly on multiprocessors. We provide a standard set of high-level programming abstractions and also provide facilities for the programmer to create his or her own abstractions.

The first version of Threads was completed in September 1985, and versions have been used by researchers other than the implementor since then. The first multiprocessor version of Threads was completed in March of 1987, two months after we acquired our Encore. This version has been used by others since April of 1987. A detailed tutorial on the use of the system is available [Doeppner 87].

The notion of cheap concurrency is often known as ‘‘lightweight processes.’’ To our knowledge, this concept was first introduced as part of Xerox’s Pilot system [Redell 80]. Other UNIX-based lightweight process implementations have been discussed in the past few years [Binding 85, Kepecs 85]. The system that comes closest to ours is Eric Cooper’s C-Threads [Cooper 87]. Recently an operating system-

supported notion of lightweight process, also known as a thread, has been implemented and used extensively at CMU as part of the Mach system [Tevanian 87].

What differentiates our system from the other UNIX-based systems is its complete support for systems concepts, including I/O, interrupts and exceptions. What differentiates our system from all of the other approaches is its support for concurrent programming abstractions – both the design of the particular abstractions we have implemented and how we allow the programmer to define new abstractions.

In this paper we first introduce the concept of concurrent threads, then discuss our approach to constructing the thread abstraction, and finally discuss the special issues that arose in the implementation of the concurrent version of Threads.

2. Threads and Concurrency

Concurrency is the notion that two or more things are happening at the same time. The designer of a processor is concerned about the concurrent use of various functional units such as adders, multipliers, etc. The architect of a computer system is concerned about the concurrent use of processors, memories, I/O devices, etc. The programmer is concerned about the concurrent execution of different control paths through a program. From the point of view of the computer architect, the *agents* which are doing things concurrently are processors. The point of view of the programmer is more abstract than that of the architect; the programmer is concerned not with processors, but with what we call *threads*. This notion of threads is independent of processors: we may have the concurrent execution of many threads on one processor or on many processors. In terms of the thread abstraction, the number of underlying processors does not matter (though it certainly does matter in the implementation of the abstraction).

In its purest form, a thread represents as little as possible: it is an abstraction of only the agent executing instructions; it is not an abstraction of either instructions or data. The execution of a thread causes local variables and a runtime stack to be allocated, but these are considered not a part of the thread itself but rather a part of the program being executed.

A number of issues must be dealt with in the design of a concurrent programming environment such as our Threads system: synchronization, scheduling, exceptions, interrupts and I/O.

Synchronization entails a thread suspending its own execution, usually waiting for some other thread to cause its execution to resume. For example, consider P and V operations on *semaphores*. A thread executing a P operation checks the value of the semaphore. If it is 0, then the thread causes itself to be suspended. At some later time, another thread executes a V operation and resumes the execution of the suspended thread.

Scheduling entails the assignment of processors to threads. Since the notion of ‘‘processor’’ does not exist in the threads abstraction, this is really an implementation concern, though an important one. How this assignment is done forms the binding of the world of threads to the world of processors.

Exceptions are a call for a thread to retreat – an unexpected event has occurred and the thread must cancel what it is doing and attempt to recover. For example, in a graphics system a thread might be computing the pixels within a certain portion of a picture when another thread is ‘‘informed’’ that the shape of the picture is to be changed. The second thread would raise an exception in the first, causing the first thread to stop what it is doing, go back to some earlier point in the computation, and start again with the new information.

An *interrupt* is part of the processor abstraction. It indicates that some sort of ‘‘event’’ has occurred, generated either externally or by the program, that requires immediate attention. If we want to give this ‘‘immediate attention’’ at the level of threads, then some suitable abstraction of interrupts has to be available at the threads level. For example, a mouse click might cause an interrupt. One way of dealing with this within the context of threads is to have a new thread be created in response to the mouse click; this new thread would execute the code which is to handle the mouse click.

I/O must be handled in some fashion so as to allow communication with the ‘‘outside world.’’ The technique we use to do this is to have threads make synchronous requests of I/O devices: a thread performing a read or write request is suspended until the request has completed. Note that the effect of asynchronous I/O can be achieved by introducing more threads: if it is desirable for a thread to continue execution while it is waiting for an I/O operation to take place, then a separate thread can be created for the purpose of performing the I/O operation.

2.1. Threads Are Lightweight

For a highly concurrent style of programming using threads to be useful, threads must be very inexpensive, i.e. the overhead required for creating, synchronizing and scheduling threads must be very low. A thread is not a traditional operating system process, which are typically very expensive to create and very expensive to synchronize. One reason for this expense is that processes are much more than just threads of control. They entail (usually) a separate address space and other protection boundaries which are time-consuming to set up. Another reason for the expense of processes is that they are managed by the operating system kernel; requests to perform operations such as synchronization must be passed to the kernel over a user-kernel boundary that is typically fairly expensive to cross (for example, in Berkeley UNIX running on a MicroVAX2, the overhead for a system call is 200 microseconds).

In the Mach system [Tevanian 87], threads are supported in the kernel, but appear to be cheap enough to qualify as lightweight processes (Mach threads are a lower-level abstraction than our threads). We are very interested in combining our approach with that of Mach, building our threads on top of Mach threads. Mach supplies a great deal more functionality than does Berkeley UNIX, yet is binary-compatible with it, so that porting our system to Mach should yield a number of insights and benefits.

Currently all of our Threads system runs as user-mode code. This has resulted in a minimal overhead due to system calls and has allowed our system to be ported fairly easily to other UNIX systems.

2.2. Threads Are Not Coroutines

Threads have access to their own representations. This allows synchronization and scheduling to be handled by the very threads being synchronized and scheduled. In this sense, threads are much like *coroutines*.

Coroutines, like threads, are independent units of execution. A coroutine is created by a *call* statement; the caller causes a new coroutine to come into existence and simultaneously transfers control to it. A coroutine may transfer control to a specified existing coroutine by executing a *resume* statement. Thus synchronization and scheduling are combined into a single construct: the programmer not only specifies the synchronization conditions (i.e. decides when a coroutine should suspend itself via a resume), but also explicitly specifies the scheduling function (i.e. chooses the next thread to execute).

The problem with this approach is that synchronization and scheduling are in fact separate concepts that should be treated separately. One can build synchronization constructs (such as operations on semaphores, monitors, etc.), which determine when it is *permissible* for a thread to execute, independently of schedulers, which determine which threads to execute from a set of threads for which execution *is* permissible (this was discussed formally in [Doeppner 76]). As will be discussed, we provide in Threads a set of basic operations from which both synchronization and scheduling can be constructed, allowing one to build one's own synchronization and scheduling abstractions.

A further difference between threads and coroutines is that threads are capable of dealing with *asynchronous* activities. If a coroutine performs an I/O request, all coroutines are blocked until the request has completed. A thread waiting on an I/O request has no such direct effect on other threads. Threads may be preemptively scheduled: if an interrupt occurs, the execution of one thread can be preempted, allowing another thread to execute. Finally, coroutines do not permit the simultaneous execution of multiple coroutines; their semantics require that only one coroutine be executing at a time. Threads have no such restriction: as many threads as are ready to execute may be executing.

3. Building the Thread Abstraction

The Thread abstraction is built in layers using a style that is essentially a procedure-calling dual of the (message-based) object-oriented approach. Put very simply, in the object-oriented approach, a system is organized around *objects* which communicate with each other by exchanging *messages*. Each object is an *instance* of a particular *class*. A class contains a number of *methods* representing operations which can be performed on individual objects. The internal state of an object consists of a number of *instance variables*, which are private for the particular object, as well as a number of *class variables*, which are global (shared) among all objects belonging to a particular class. To perform an operation involving a particular object, one sends that object a message which contains an indication of the method to be used and data to be supplied to the method. Thus one thinks of ''asking an object to perform an operation on itself.''

In [Lauer 78] it is shown that a system based on a message-passing paradigm has an equivalent, dual system which is based on a procedure-calling paradigm. In the spirit of [Lauer 78], the dual of an object-oriented system is a system in which *procedure-calling* replaces message-passing, *procedures* replace

methods, and ‘‘*protected*’’ *data structures* replace the instance variables and class variables which comprise an object. Thus the analogue of a class is a collection of procedures and the analogue of an object is a protected data structure. To perform an operation, one calls a procedure, passing to it, along with other parameters, an indication of which protected data structure is to be operated upon. Thus one thinks of ‘‘telling a procedure to operate on a particular data structure.’’

Object-oriented systems have the concept of *inheritance*, meaning that one can define a new class of objects by creating a subclass of an existing class, adding new methods to those provided by the existing class. In our dual approach this corresponds to adding new procedures to an existing collection and adding additional fields to the protected data structure. This results in a layered approach to the design of a system.

The Threads system is built in two layers; the user is given the ability to add further layers. The bottom layer, which is built on top of the UNIX process, implements the basic notion of a thread as an independent entity. A thread at this level presents a fairly low-level procedural interface which allows it to be manipulated directly. In the next layer, the thread abstraction is extended to supply the functionality needed to give the programmer the types of programming constructs expected in a high-level language; this layer can be thought of as the runtime library for such a language. A user of the Threads system may add additional layers, building on top of the lower layers by supplying additional procedures and adding additional fields to the thread data structures.

Each of our layers is divided into a number of components. Each component contains a collection of procedures, defines a number of global variables (corresponding to class variables) and defines fields of the protected thread data structure (corresponding to instance variables).

3.1. The Bottom Layer

The functionality defined in this layer includes scheduling and context switching, low-level synchronization, interrupt processing, exception handling and stack handling. In addition, this layer provides the routines employed for protection from interrupts and for the locking of data structures when used on a multiprocessor.

3.1.1. Context and Queueing

The most basic components of the system are those implementing the execution context of a thread and queueing of threads. The queueing component contains procedures which link threads in queues and contributes to the thread data structure the fields necessary to link a thread into a queue. Its primary procedures are:

movetoqueue(THREAD, QUEUE)
pullfromqueue(THREAD, QUEUE)

The execution context component contains the procedure which performs context switching of threads and contributes to the thread data structure the fields needed to contain the context of a thread, e.g. the processor registers. The only procedure of this component is the context-switching routine:

run()

These procedures would be typically used for synchronization. For a thread to suspend its execution, it would execute code similar to:

```
movetoqueue(CURRENT, WAITQUEUE);
run();
```

Here, *CURRENT* represents the data structure of the currently executing thread and *WAITQUEUE* represents a queue of threads. The caller puts itself on the queue of threads and then calls the context-switching *run* routine to have its processor state saved in its data structure and to relinquish its processor. If other threads are waiting for a processor, then *run* will select one of them for execution.

3.1.2. Exceptions

Exceptions are dealt with partly in the bottom layer and partly in the second layer. In the bottom layer, the exception-handling component is responsible for storing the address of the exception handler and for invoking the handler in response to an exception. (What happens when the handler is invoked is the responsibility of the second layer, since this invocation may affect other components in that layer.) The bottom-layer exception-handling component supports two procedures:

setexception(NEWHANDLER, OLDHANDLER)
raiseexception(THREAD, PARAMETER)

The first routine, *setexception*, establishes its first argument as the current exception handler and returns in its second (result) parameter the address of the previous handler, if any. This allows one to maintain a chain of exception handlers. The second routine, *raiseexception*, forces an exception to occur in the indicated thread. If an exception occurs, the procedure *exceptioncall* defined in the second layer is called.

3.1.3. Interrupt-Handling

The purpose of the interrupt-handling component is to respond to interrupts, mapping them into suitable actions within the context of Threads. Since the Threads system is built on top of a UNIX process, interrupts are presented to us in the form of *signals*. We respond to these signals as requested by the user. The most important procedure at this level is:

registersignal(SIGNO, RESPONSE, OLDRESPONSE)

This establishes the response to a particular signal (interrupt), returning in the final (result) parameter the previous response. The types of responses that may be chosen will be discussed in the section on the second layer (Section 3.2).

3.1.4. Stacks

The next component to be discussed is the one that handles the runtime stack. Strictly speaking, the stack is not part of a thread itself but is a result of the thread's execution. The stack consists of a sequence of stack frames, in which each stack frame contains the thread's context within a procedure (e.g. linkage to the caller, local variables, parameter pointers, etc.). Whenever a thread enters a procedure, it is necessary to allocate a new stack frame. In many operating systems there is support for exactly one stack (per process). Compilers can assume an unbounded stack and thus need not deal with stack allocation: the kernel automatically extends a process's stack as it references beyond its current extent, until a resource limit is reached. At that point the process is typically terminated. Since this is implemented with the aid of memory management hardware, there is no extra cost associated with this technique for checking for stack overflow.

With threads, on the other hand, a number of stacks are needed, one for each thread. To handle the stack allocation problem one could modify the compilers so that they generate code for allocating stack frames, one could modify the operating system so that it supports each additional stack in the same manner as it supports a single stack, or one could essentially ignore the problem, preallocate sufficient stack space and occasionally check to see if any stack has grown beyond its limit.

The first technique, modifying compilers, is probably the best, but we have chosen the last because of its simplicity. A stack is preallocated for each thread and its bounds are made part of the thread control block. Whenever the execution context routines are called by a thread (i.e. when a context switch is being performed), a check is made to determine if the thread has exceeded the bounds of its stack. If so, then the system is usually terminated.

3.1.5. Mutual Exclusion

In both the bottom layer and the second layer it is necessary to provide fairly low-level means of guaranteeing mutually exclusive access to shared data. On multiprocessor systems we have to guard against the in which threads running on different processors attempt to access and modify the same data structure simultaneously. On both multiprocessor systems and uniprocessor systems, we have to guard against interrupts occurring at inopportune moments: a thread might be using a data structure when the processor on which it is executing is interrupted, invoking an interrupt handler which causes actions that involve the same data structure.

The user of Threads need only be concerned about the synchronization of *threads* – the concepts of processors and interrupts do not appear at this level of abstraction. It is strictly in the implementation that we need to worry about processors and interrupts. (Of course, if a user of Threads wishes to design his or her own synchronization constructs, then processors and interrupts become issues.)

Code that must to be protected from interrupts is bracketed with calls to:

```
protect();
unprotect();
```

Interrupt protection has no effect, however, on other processors in a multiprocessor environment. For such environments, we make use of *locks*. All shared data structures used by the code implementing

threads (e.g. Thread control blocks, headers of queues, etc.) have a lock field associated with them. A thread attempting use the data structured ‘‘spins’’ on the lock (i.e. continually tests it) until the lock is granted; at this point it sets the lock, as part of an atomic instruction which makes the last (successful) test and then does the set.

The use of locks results in code that is prone to deadlock, unless care is taken in those cases in which a thread may hold multiple locks. We establish a partial order on locks, requiring related locks to be taken in a fixed order. If this is not possible, then a thread is required to release its locks before waiting for one to become available.

3.2. The Second Layer

In this second level of our implementation we build up a set of programming constructs from the low-level interface of the bottom layer. This layer can be viewed as the runtime library of a concurrent version of C: we support the concept of a monitor, for example, by supplying a set of procedures which, if used ‘‘as intended,’’ will correctly provide the standard semantics of a monitor; but we do not do the syntax-checking necessary to insure that these procedures are used correctly.

3.2.1. Synchronization

The design of synchronization constructs is very straightforward. We have provided three such constructs (semaphores, monitors and synchronized families of threads); it is fairly easy to construct other types. All of these constructs are implemented in the second level, building on top of the facilities of the low level.

The implementation of a semaphore consists of a data structure that must be allocated to hold the value of the semaphore and the header of the semaphore’s wait queue. A thread performing a P operation on the semaphore simply tests the value of the semaphore; if it is 0, the thread places itself on the wait queue by calling a queue linkage routine and then calls the execution context module so that another thread will be scheduled. When a thread performs a V operation on the semaphore, it calls queue linkage routines to determine if any thread is on the wait queue and, if so, to put that thread on the run queue.

A monitor is equally straightforward: a monitor data structure is allocated to contain the state of the monitor, including headers for an entry queue and any number of condition queues. To enter a monitor, a

thread calls the *monitorentry* routine, which will return only when it is safe for the thread to enter the monitor; in the meantime, the thread is kept on the entry queue. A thread exiting a monitor calls the *monitorexit* routine; if there are threads waiting to enter the monitor, this routine causes one of them to return from the *monitorentry* routine by taking it off of the the entry queue and putting it on the run queue. *Wait* and *signal* constructs are implemented using the condition queues in a similar fashion.

The third type of synchronization construct involves synchronization between parent and child threads on termination. When a thread is created, it may be specified to be *nondetached*. This allows the child thread to ‘‘return’’ a value to the parent when the child terminates. The parent thread may execute a call to wait until any of its children terminates, and then retrieve the returned value of that thread. If the parent attempts to terminate before its children, its termination will be delayed until its children terminate.

The implementation of nondetached threads requires a new field in the thread data structure: there must be headers for queues of all extant child threads and terminated but not-yet-synchronized-with-the-parent child threads, as well as space to hold a thread’s return value. The procedures associated with this component are called upon when a thread is created and destroyed and also when a parent thread waits for a child to terminate so that it can pick up the child’s return value.

3.2.2. Exceptions

When an exception is raised in a thread, we want the thread to stop its normal execution, call an exception handler to ‘‘clean up’’ after itself, and then resume execution at some ‘‘safe point’’ outside of the module in which the thread was executing when the exception was raised. To make these actions possible, a thread must register an exception handler, giving the address of an exception-handling routine. The point at which the call to register the exception occurs is the safe point to return to after an exception occurs: the *setexception* routine returns a 0 after it is called, but after an exception has occurred and the handler has been called, the thread again returns from the *setexception* routine, this time returning a 1 (technique similar to the *setjmp/longjmp* facility of UNIX).

When an exception is raised, a parameter is passed to the exception handler which serves to identify the type of the exception. Exceptions may be raised explicitly by a *raiseexception* routine which can be used by one thread to raise an exception in another (or itself), or may be raised in response to program

errors, such as arithmetic problems (in this case, the exception parameter serves to identify the error type).

If an exception occurs while a thread is inside a monitor, care must be taken since other threads that use the monitor might be adversely affected. The state of the monitor must be changed to reflect the fact that the thread has left the monitor, and if there are threads waiting to enter the monitor, one of them must be allowed to enter. Monitors are typically used to protect data that is shared by a number of threads; it is assumed that if a thread enters a monitor, this data will be left in a consistent state on exit. But if an exception is raised in the thread, it might abandon the monitor, leaving it in an inconsistent state. Because of this problem, each monitor may define a cleanup function which is called to put the data of the monitor back into a consistent state if an exception is raised in a thread in the monitor.

The implementation of this additional aspect of exception handling requires information to be kept for each thread for each monitor it has entered (there may be nested calls to monitors). Thus another component of the thread control block is added which deals with *managers*. Whenever a thread enters a monitor (or any similar type of construct that one might wish to define), a manager is allocated and pushed onto a stack of managers referred to by the thread control block. (Managers are typically allocated on the runtime stack, so that their allocation is very efficient.) Whenever an exception is raised in a thread, those managers within the scope of the exception handler (i.e. those that correspond to monitors entered since the exception handler was registered) are popped off the manager stack and the cleanup routines referred to by them are called.

3.2.3. Input and Output

Since the Threads system runs on top of an underlying operating system, most aspects of I/O are taken care of for us. The function of our implementation is then to convert the I/O abstraction presented by the operating system into one suitable for threads. We take the synchronous I/O system calls of UNIX and make them synchronous with respect to individual threads. That is, a thread's execution is blocked while an I/O operation it requested is in progress, but no other thread is affected. The difficulty here is to make certain that the underlying UNIX process does not block, but is available for the execution of other threads. This is accomplished by using the asynchronous I/O facilities of UNIX. A thread attempts an I/O operation; if it cannot be completed without blocking, then it puts itself on a wait queue (associated with the I/O

device). At some later time, UNIX (through a *SIGIO* signal) signals that the operation can now take place without blocking, so the thread is taken off the wait queue (by a thread responding to the signal) and put onto the run queue.

A thread that is blocked for an I/O operation will affect other threads if it is inside a monitor. Normally, when a thread inside a monitor finds it must wait for something, it does so by taking itself out of the monitor by calling the monitor wait routine with a particular condition. Then, when some other thread determines that what the first thread was waiting for has happened, it brings that thread back into the monitor by calling the monitor signal routine with that condition. If a thread is waiting for an I/O operation to take place, we would like it to be able to wait on some condition which will be signaled when the I/O operation can take place. We provide an addition to monitors that does exactly that. A thread may call *monitorwaitevent* from inside a monitor, specifying a condition, a number of I/O devices of interest, and a timeout period. The thread will be taken out of the monitor, waiting on the condition. When any of the indicated I/O devices is ready for a transfer or the timeout period has expired, then the condition will be signaled and the thread will be brought back into the monitor when the thread is first in line for that condition.

3.2.4. Interrupts

Although at the processor level of abstraction an interrupt interrupts the current computation, at the threads level of abstraction this is not the only possibility. We provide three separate methods for bringing interrupts into the threads abstraction. In many cases an interrupt is an indication that some action needs to be performed, such as popping up a window on a display or recording an event in a log. In such instances, the interrupt causes a thread to be created which executes concurrently with other threads – it does not ‘‘interrupt’’ them. In other cases an interrupt indicates some type of error in the execution of a thread. In these instances the interrupt causes an exception to be raised in the thread which caused the error. The final means for dealing with interrupts is actually to interrupt a particular thread: suspend its execution while a new thread executes. When that thread terminates, the original thread is resumed.

In our implementation, interrupts are represented by UNIX signals. For each type of signal, the Threads programmer may specify which of the above methods is to be used for dealing with it. For

exception-causing signals, the signal number is supplied to the exception handler as an argument. For interrupt-causing signals, a programmer-supplied routine is called to determine which thread should be interrupted.

There are more complications in suspending an interrupted thread than are immediately apparent. If a thread has been placed on a wait queue and is suspended, what is to happen when another thread attempts to move the suspended thread from the wait queue to another queue? If a thread on a monitor's condition queue is suspended and the condition is then signaled, what should the status of the monitor become? Our solution to these problems is that being suspended is orthogonal to being on a queue: a suspended thread may be moved to and from any queue, including the run queue (the queue of threads that are ready to execute, but are not currently executing). That a thread is suspended means only that it is not a candidate for execution.

3.2.5. Example

To illustrate some of the Threads concepts discussed above, we look at a very simple example involving an oversimplified workstation application. We assume that we have a window-managed display in which each window is represented by a data structure which contains the attributes of the window, including a monitor which is to synchronize access to the window. A simple procedure for interacting with a window is *promptandread*, which outputs a prompt to the window and then reads a response:

```
promptandread(window, prompt, response, length)
WINDOW window;
char *prompt, *response;
int length;
{
  MANAGER manager;

  monitorentry(window.monitor, &manager);

  write(window.out, prompt, strlen(prompt));
  read(window.in, response, length);

  monitorexit(window.monitor);
}
```

window represents a data structure describing the window. It includes a *monitor* field, which we use for synchronization, as well as *in* and *out* fields which are used to identify to the UNIX *read* and *write* system calls the files which represent the display. The *manager* is allocated on the thread's stack for use in the

event of an exception while in the monitor. *monitorentry* is used to enter the monitor; thus the caller is suspended until it is granted mutually exclusive access to the window. The calls to *write* and *read* cause I/O to be performed; the calling thread may be suspended as a result of these calls, but other threads are not affected. Finally, the call to *monitorexit* releases the window so that it can be used by other threads.

Now, suppose that in response to a particular type of interrupt, we would like a thread to be created which will ‘‘pop up’’ a new window, prompt for a command, then process the command. To do this, we first need to tell the Threads system how to respond to the interrupt:

```
SIGHANDLER response = {CREATETHREAD, 0, popup};
registersignal(SIGINT, &response, NULL);
```

The first line declares *response* to be of type SIGHANDLER and gives it a value. The call to *registersignal* requests that whenever an interrupt of type SIGINT occurs, the response should be to create a new thread which executes the procedure *popup*. Each time this interrupt occurs, a new thread is created which executes:

```
popup() {
  WINDOW newwindow = create_window(...);
  char request[80];

  promptandread(newwindow, "yes? ", request, 80);

  handle_request(request);
}
```

This creates a new window by calling *create_window* which, among other things, must allocate a new monitor to synchronize the window. It then calls *promptandread* to prompt for a command name. It finally calls *handle_request* to carry out the command. Even though *popup* was called in response to an interrupt, the thread created to execute *popup* is no different from any other thread. If other threads need to access the window created in *popup*, then these threads, as well as the interrupt thread, will all be synchronized by using the window’s monitor. Furthermore, this code will work no matter how many processors are employed.

4. Experiences with a Multiprocessor

Threads was designed with the intent that the concept of ‘‘processor’’ should be invisible to the user. Thus a goal of our multiprocessor implementation was to bring this about. A programmer using threads on the Encore Multimax specifies the number of processors to be used when the system is initialized. From that point onwards, all further interaction is strictly in terms of threads; the number of processors used is not relevant for correctness (though it is certainly important for performance).

Porting Threads to the shared-memory multiprocessor environment of the Encore Multimax was fairly straightforward, mainly because we had envisioned a multiprocessor implementation from the beginning. The trickiest part of getting a logically correct port was to deal with the effect of events that might occur asynchronously on one processor (e.g. interrupts) on a thread running on another processor.

The addition of locks for interprocessor synchronization was a fairly mechanical process, once a deadlock-preventing ordering of lock acquisition had been decided upon. The locks themselves are instrumented to detect deadlock (nobody’s perfect ...) and to measure their effect on performance (of course, the code added to the locks to instrument them become a major factor affecting performance).

A mistake we made with locks at first was to use them for too much. Our original strategy for dealing with idle processors was to have them spin on the lock guarding the run queue. With this strategy in place, a test program which did nothing but create a large number of threads, each of which promptly terminated, *slowed down* linearly with respect to the number of processors used! Adding another lock, so that there was one for use by idle processors and a separate one for guarding the run queue, turned the performance curve around.

4.1. Performance

At this writing, using code that is not optimized, it takes just under a millisecond to create a thread on the Encore (this is with National 32032 processors). Approximately 20% of this is overhead resulting from the multiprocessor port, since it takes approximately 800 microseconds to create a thread on the Encore using the single-processor version of threads (as opposed to the multiprocessor version parameterized for one processor). We are continuing to work on improving these figures and feel that much improvement is possible. For comparison, in an optimized single processor version of Threads running on a MicroVAX-2

(a processor which is about 33% faster than the 32032), it takes approximately 250 microseconds to create a thread.

5. References

[Binding 85] Binding, C., ‘‘Cheap Concurrency in C,’’ *SIGPLAN Notices*, Vol. 20, No. 9, September 1985.

[Cooper 87] Cooper, E.C., Draves, R.P., ‘‘C Threads,’’ Draft of Carnegie Mellon University/Computer Science Report, March 1987.

[Doeppner 76] Doeppner, T.W., ‘‘On Abstractions of Parallel Programs,’’ *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, May 1976.

[Doeppner 87] Doeppner, T.W. Jr., ‘‘A Threads Tutorial,’’ Computer Science Technical Report CS-87-06, Brown University, 1987.

[Kepecs 85] Kepecs, J., ‘‘Lightweight Process for UNIX/Implementation and Applications,’’ *USENIX Association Summer Conference Proceedings*, June 1985.

[Lauer 78] Lauer, H.C., Needham, R.M., ‘‘On the Duality of Operating System Structures,’’ *Proceedings of Second International Symposium on Operating Systems*, IRIA, October 1978, reprinted in *Operating Systems Review*, Vol. 13, No. 2, April 1979.

[Redell 80] Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G., Purcell, S.C., ‘‘Pilot: An Operating System for a Personal Computer,’’ *Communications of the ACM*, Vol. 23, No. 2, February 1980.

[Tevanian 87] Tevanian, A., Rashid, R.F., Golub, D.B., Black, D.L., Cooper, E., Young, M.W., ‘‘Mach Threads and the Unix Kernel: The Battle for Control,’’ *USENIX Association Summer Conference Proceedings*, June 1987.