20. Is optimistic concurrency control more or less restrictive than using timestamps? Why?

21. Does using timestamping for concurrency control ensure serializability? Discuss.

22. We have repeatedly said that when a transaction is aborted, the world is restored to its previous state, as though the transaction had never happened. We lied. Give an example where resetting the world is impossible.

23. The centralized deadlock detection algorithm described in the text initially gave a false deadlock, but was later patched up using global time. Suppose that it has been decided not to maintain global time (too expensive). Devise an alternative way to fix the bug in the algorithm.

24. A process with transaction timestamp 50 needs a resource held by a process with transaction timestamp 100. What happens in:

(a) Wait-die?
(b) Wound-wait?

4

Processes and Processors in Distributed Systems

In the preceding two chapters, we have looked at two related topics, communication and synchronization in distributed systems. In this chapter we will switch to a different subject: processes. Although processes are also an important concept in uniprocessor systems, in this chapter we will emphasize aspects of process management that are usually not studied in the context of classical operating systems. In particular, we will look at how the existence of multiple processors is dealt with.

In many distributed systems, it is possible to have multiple threads of control within a process. This ability provides some important advantages, but also introduces various problems. We will study these issues first. Then we come to the subject of how the processors and processes are organized and see that several different models are possible. Then we will look at processor allocation and scheduling in distributed systems. Finally, we consider two special kinds of distributed systems, fault-tolerant systems and real-time systems.

4.1. THREADS

In most traditional operating systems, each process has an address space and a single thread of control. In fact, that is almost the definition of a process. Nevertheless, there are frequently situations in which it is desirable to have

multiple threads of control sharing one address space but running in quasi-parallel, as though they were separate processes (except for the shared address space). In this section we will discuss these situations and their implications.

4.1.1. Introduction to Threads

Consider, for example, a file server that occasionally has to block waiting for the disk. If the server had multiple threads of control, a second thread could run while the first one was sleeping. The net result would be a higher throughput and better performance. It is not possible to achieve this goal by creating two independent server processes because they must share a common buffer cache, which requires them to be in the same address space. Thus a new mechanism is needed, one that historically was not found in single-processor operating systems.

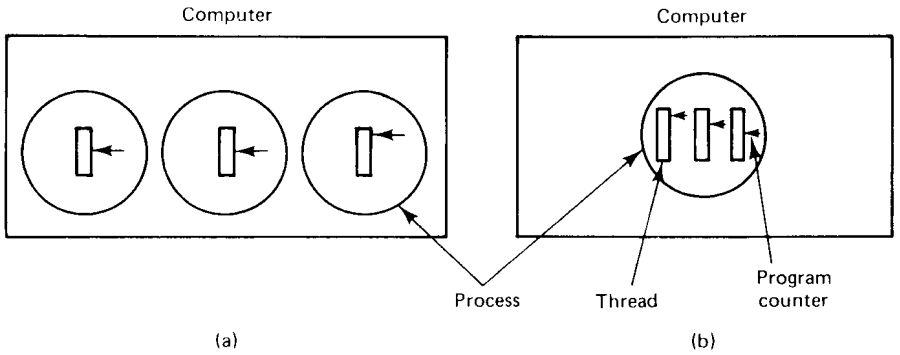

Fig. 4-1. (a) Three processes with one thread each. (b) One process with three threads.

In Fig. 4-1(a) we see a machine with three processes. Each process has its own program counter, its own stack, its own register set, and its own address space. The processes have nothing to do with each other, except that they may be able to communicate through the system's interprocess communication primitives, such as semaphores, monitors, or messages. In Fig. 4-1(b) we see another machine, with one process. Only this process contains multiple threads of control, usually just called **threads**, or sometimes **lightweight processes**. In many respects, threads are like little mini-processes. Each thread runs strictly sequentially and has its own program counter and stack to keep track of where it is. Threads share the CPU just as processes do: first one thread runs, then another does (timesharing). Only on a multiprocessor do they actually run in parallel. Threads can create child threads and can block waiting for system calls to complete, just like regular processes. While one thread is blocked, another thread in

the same process can run, in exactly the same way that when one process blocks, another process in the same machine can run. The analogy: thread is to process as process is to machine, holds in many ways.

Different threads in a process are not quite as independent as different processes, however. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every virtual address, one thread can read, write, or even completely wipe out another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary. Unlike different processes, which may be from different users and which may be hostile to one another, a process is always owned by a single user, who has presumably created multiple threads so that they can cooperate, not fight. In addition to sharing an address space, all the threads share the same set of open files, child processes, timers, and signals, etc. as shown in Fig. 4-2. Thus the organization of Fig. 4-1(a) would be used when the three processes are essentially unrelated, whereas Fig. 4-1(b) would be appropriate when the three threads are actually part of the same job and are actively and closely cooperating with each other.

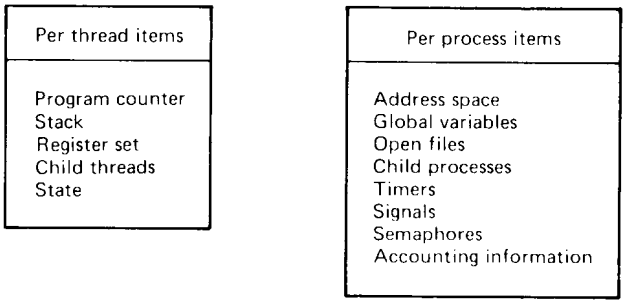

Fig. 4-2. Per thread and per process concepts.

Like traditional processes (i.e., processes with only one thread), threads can be in any one of several states: running, blocked, ready, or terminated. A running thread currently has the CPU and is active. A blocked thread is waiting for another thread to unblock it (e.g., on a semaphore). A ready thread is scheduled to run, and will as soon as its turn comes up. Finally, a terminated thread is one that has exited, but which has not yet been collected by its parent (in UNIX terms, the parent thread has not yet done a WAIT).

4.1.2. Thread Usage

Threads were invented to allow parallelism to be combined with sequential execution and blocking system calls. Consider our file server example again. One possible organization is shown in Fig. 4-3(a). Here one thread, the

dispatcher, reads incoming requests for work from the system mailbox. After examining the request, it chooses an idle (i.e., blocked) **worker thread** and hands it the request, possibly by writing a pointer to the message into a special word associated with each thread. The dispatcher then wakes up the sleeping worker (e.g., by doing an UP on the semaphore on which it is sleeping).

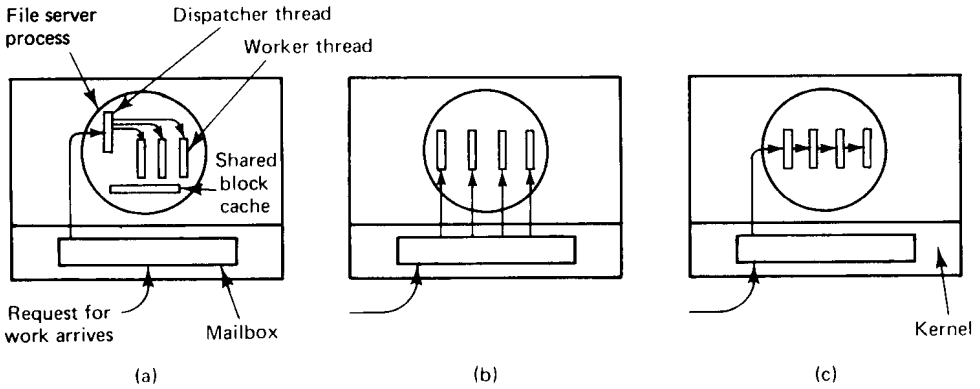

Fig. 4-3. Three organizations of threads in a process. (a) Dispatcher/worker model. (b) Team model. (c) Pipeline model.

When the worker wakes up, it checks to see if the request can be satisfied from the shared block cache, to which all threads have access. If not, it sends a message to the disk to get the needed block (assuming it is a READ) and goes to sleep awaiting completion of the disk operation. The scheduler will now be invoked and another thread will be started, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

Consider how the file server could be written in the absence of threads. One possibility is to have it operate as a single thread. The main loop of the file server gets a request, examines it, and carries it out to completion before getting the next one. While waiting for the disk, the server is idle and does not process any other requests. If the file server is running on a dedicated machine, as is commonly the case, the CPU is simply idle while the file server is waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus threads gain considerable performance, but each thread is programmed sequentially, in the usual way.

So far we have seen two possible designs: a multithreaded file server and a single-threaded file server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. A third possibility is to run the server as a big finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the cache, fine, but if not, a message must be sent to the disk.

However, instead of blocking, it records the state of the current request in a table and then goes and gets the next message. The next message may either be a request for new work or a reply from the disk about a previous operation. If it is new work, that work is started. If it is a reply from the disk, the relevant information is fetched from the table and the reply processed. Since it is not permitted to send a message and block waiting for a reply here, RPC cannot be used. The primitives must be nonblocking calls to *send* and *receive*.

In this design, the "sequential process" model that we had in the first two cases is lost. The state of the computation must be explicitly saved and restored in the table for every message sent and received. In effect, we are simulating the threads and their stacks the hard way. The process is being operated as a finite-state machine that gets an event and then reacts to it, depending on what is in it.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls (e.g., RPC to talk to the disk) and still achieve parallelism. Blocking system calls make programming easier and parallelism improves performance. The single-threaded server retains the ease of blocking system calls, but gives up performance. The finite-state machine approach achieves high performance through parallelism, but uses nonblocking calls and thus is hard to program. These models are summarized in Fig. 4-4.

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-thread process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Fig. 4-4. Three ways to construct a server.

The dispatcher structure of Fig. 4-3(a) is not the only way to organize a multithreaded process. The **team** model of Fig. 4-3(b) is also a possibility. Here all the threads are equals, and each gets and processes its own requests. There is no dispatcher. Sometimes work comes in that a thread cannot handle, especially if each thread is specialized to handle a particular kind of work. In this case, a job queue can be maintained, with pending work kept in the job queue. With this organization, a thread should check the job queue before looking in the system mailbox.

Threads can also be organized in the **pipeline** model of Fig. 4-3(c). In this model, the first thread generates some data and passes them on to the next thread for processing. The data continue from thread to thread, with processing going on at each step. Although this is not appropriate for file servers, for other

problems, such as the producer-consumer, it may be a good choice. Pipelining is widely used in many areas of computer systems, from the internal structure of RISC CPUs to UNIX command lines.

Threads are frequently also useful for clients. For example, if a client wants a file to be replicated on multiple servers, it can have one thread talk to each server. Another use for client threads is to handle signals, such as interrupts from the keyboard (DEL or BREAK). Instead of letting the signal interrupt the process, one thread is dedicated full time to waiting for signals. Normally, it is blocked, but when a signal comes in, it wakes up and processes the signal. Thus using threads can eliminate the need for user-level interrupts.

Another argument for threads has nothing to do with RPC or communication. Some applications are easier to program using parallel processes, the producer-consumer problem for example. Whether the producer and consumer actually run in parallel is secondary. They are programmed that way because it makes the software design simpler. Since they must share a common buffer, having them in separate processes will not do. Threads fit the bill exactly here.

Finally, although we are not explicitly discussing the subject here, in a multiprocessor system, it is actually possible for the threads in a single address space to run in parallel, on different CPUs. This is, in fact, one of the major ways in which sharing is done on such systems. On the other hand, a properly designed program that uses threads should work equally well on a single CPU that timeshares the threads or on a true multiprocessor, so the software issues are pretty much the same either way.

4.1.3. Design Issues for Threads Packages

A set of primitives (e.g., library calls) available to the user relating to threads is called a **threads package**. In this section we will consider some of the issues concerned with the architecture and functionality of threads packages. In the next section we will consider how threads packages can be implemented.

The first issue we will look at is thread management. Two alternatives are possible here, static threads and dynamic threads. With a static design, the choice of how many threads there will be is made when the program is written or when it is compiled. Each thread is allocated a fixed stack. This approach is simple, but inflexible.

A more general approach is to allow threads to be created and destroyed on-the-fly during execution. The thread creation call usually specifies the thread's main program (as a pointer to a procedure) and a stack size, and may specify other parameters as well, for example, a scheduling priority. The call usually returns a thread identifier to be used in subsequent calls involving the thread. In this model, a process starts out with one (implicit) thread, but can create one or more threads as needed, and these can exit when finished.

Threads can be terminated in one of two ways. A thread can exit voluntarily when it finishes its job, or it can be killed from outside. In this respect, threads are like processes. In many situations, such as the file servers of Fig. 4-3, the threads are created immediately after the process starts up and are never killed.

Since threads share a common memory, they can, and usually do, use it for holding data that are shared among multiple threads, such as the buffers in a producer-consumer system. Access to shared data is usually programmed using critical regions, to prevent multiple threads from trying to access the same data at the same time. Critical regions are most easily implemented using semaphores, monitors, and similar constructions. One technique that is commonly used in threads packages is the **mutex**, which is a kind of watered-down semaphore. A mutex is always in one of two states, unlocked or locked. Two operations are defined on mutexes. The first one, LOCK, attempts to lock the mutex. If the mutex is unlocked, the LOCK succeeds and the mutex becomes locked in a single atomic action. If two threads try to lock the same mutex at exactly the same instant, an event that is possible only on a multiprocessor, on which different threads run on different CPUs, one of them wins and the other loses. A thread that attempts to lock a mutex that is already locked is blocked.

The UNLOCK operation unlocks a mutex. If one or more threads are waiting on the mutex, exactly one of them is released. The rest continue to wait.

Another operation that is sometimes provided is TRYLOCK, which attempts to lock a mutex. If the mutex is unlocked, TRYLOCK returns a status code indicating success. If, however, the mutex is locked, TRYLOCK does not block the thread. Instead, it returns a status code indicating failure.

Mutexes are like binary semaphores (i.e., semaphores that may only have the values 0 or 1). They are not like counting semaphores. Limiting them in this way makes them easier to implement.

Another synchronization feature that is sometimes available in threads packages is the **condition variable**, which is similar to the condition variable used for synchronization in monitors. Each condition variable is normally associated with a mutex at the time it is created. The difference between mutexes and condition variables is that mutexes are used for short-term locking, mostly for guarding the entry to critical regions. Condition variables are used for long-term waiting until a resource becomes available.

The following situation occurs all the time. A thread locks a mutex to gain entry to a critical region. Once inside the critical region, it examines system tables and discovers that some resource it needs is busy. If it simply locks a second mutex (associated with the resource), the outer mutex will remain locked and the thread holding the resource will not be able to enter the critical region to free it. Deadlock results. Unlocking the outer mutex lets other threads into the critical region, causing chaos, so this solution is not acceptable.

One solution is to use condition variables to acquire the resource, as shown

in Fig. 4-5(a). Here, waiting on the condition variable is defined to perform the wait and unlock the mutex atomically. Later, when the thread holding the resource frees it, as shown in Fig. 4-5(b), it calls *wakeup*, which is defined to wakeup either exactly one thread or all the threads waiting on the specified condition variable. The use of WHILE instead of IF in Fig. 4-5(a) guards against the case that the thread is awakened but that someone else seizes the resource before the thread runs.

```
lock mutex;
    check data structures;
    while (resource busy)
        wait (condition variable);
    mark resource as busy;
unlock mutex;
```

(a)

```
lock mutex;
    mark resource as free;
unlock mutex;
wakeup (condition variable);
```

(b)

Fig. 4-5. Use of mutexes and condition variables.

The need for the ability to wake up all the threads, rather than just one, is demonstrated in the reader-writer problem. When a writer finishes, it may choose to wake up pending writers or pending readers. If it chooses readers, it should wake them all up, not just one. Providing primitives for waking up exactly one thread and for waking up all the threads provides the needed flexibility.

The code of a thread normally consists of multiple procedures, just like a process. These may have local variables, global variables, and procedure parameters. Local variables and parameters do not cause any trouble, but variables that are global to a thread but not global to the entire program do.

As an example, consider the *errno* variable maintained by UNIX. When a process (or a thread) makes a system call that fails, the error code is put into *errno*. In Fig. 4-6, thread 1 executes the system call ACCESS to find out if it has permission to access a certain file. The operating system returns the answer in the global variable *errno*. After control has returned to thread 1, but before it has a chance to read *errno*, the scheduler decides that thread 1 has had enough CPU time for the moment and decides to switch to thread 2. Thread 2 executes an OPEN call that fails, which causes *errno* to be overwritten and thread 1's access code to be lost forever. When thread 1 starts up later, it will read the wrong value and behave incorrectly.

Various solutions to this problem are possible. One is to prohibit global variables altogether. However worthy this ideal may be, it conflicts with much existing software, such as UNIX. Another is to assign each thread its own private global variables, as shown in Fig. 4-7. In this way, each thread has its own private copy of *errno* and other global variables, so conflicts are avoided. In effect, this decision creates a new scoping level, variables visible to all the

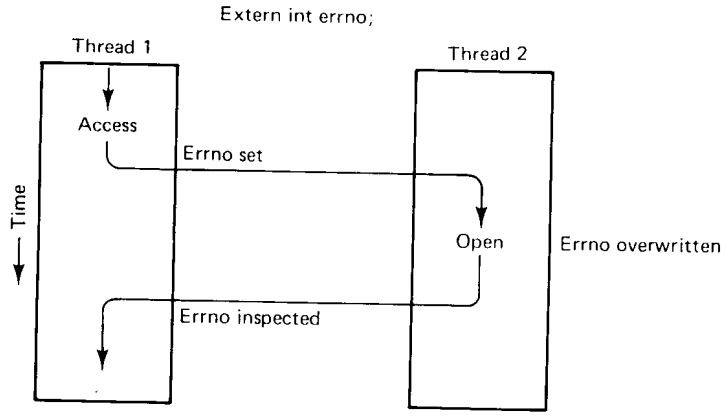

Fig. 4-6. Conflicts between threads over the use of a global variable.

procedures of a thread, in addition to the existing scoping levels of variables visible only to one procedure and variables visible everywhere in the program.

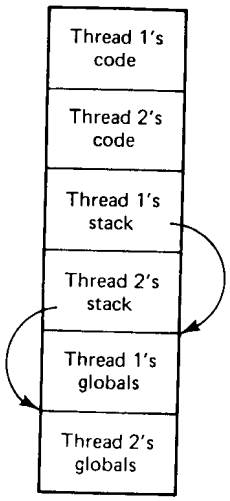

Fig. 4-7. Threads can have private global variables.

Accessing the private global variables is a bit tricky, however, since most programming languages have a way of expressing local variables and global variables, but not intermediate forms. It is possible to allocate a chunk of memory for the globals and pass it to each procedure in the thread, as an extra parameter. While hardly an elegant solution, it works.

Alternatively, new library procedures can be introduced to create, set, and read these thread-wide global variables. The first call might look like this:

```
create_global("bufptr");
```

It allocates storage for a pointer called *bufptr* on the heap or in a special storage area reserved for the calling thread. No matter where the storage is allocated, only the calling thread has access to the global variable. If another thread creates a global variable with the same name, it gets a different storage location that does not conflict with the existing one.

Two calls are needed to access global variables: one for writing them and the other for reading them. For writing, something like

```
set_global("bufptr", &buf);
```

will do. It stores the value of a pointer in the storage location previously created by the call to *create_global*. To read a global variable, the call might look like

```
bufptr = read_global("bufptr");
```

This call returns the address stored in the global variable, so the data value can be accessed.

Our last design issue relating to threads is scheduling. Threads can be scheduled using various scheduling algorithms, including priority, round robin, and others. Threads packages often provide calls to give the user the ability to specify the scheduling algorithm and set the priorities, if any.

4.1.4. Implementing a Threads Package

There are two main ways to implement a threads package: in user space and in the kernel. The choice is moderately controversial, and a hybrid implementation is also possible. In this section we will describe these methods, along with their advantages and disadvantages.

Implementing Threads in User Space

The first method is to put the threads package entirely in user space. The kernel knows nothing about them. As far as the kernel is concerned, it is managing ordinary, single-threaded processes. The first, and most obvious, advantage is that a user-level threads package can be implemented on an operating system that does not support threads. For example, UNIX originally did not support threads, but various user-space threads packages were written for it.

All of these implementations have the same general structure, which is illustrated in Fig. 4-8(a). The threads run on top of a runtime system, which is a

collection of procedures that manage threads. When a thread executes a system call, goes to sleep, performs an operation on a semaphore or mutex, or otherwise does something that may cause it to be suspended, it calls a runtime system procedure. This procedure checks to see if the thread must be suspended. If so, it stores the thread's registers (i.e., its own) in a table, looks for an unblocked thread to run, and reloads the machine registers with the new thread's saved values. As soon as the stack pointer and program counter have been switched, the new thread comes to life again automatically. If the machine has an instruction to store all the registers and another one to load them all, the entire thread switch can be done in a handful of instructions. Doing thread switching like this is at least an order of magnitude faster than trapping to the kernel, and is a strong argument in favor of user-level threads packages.

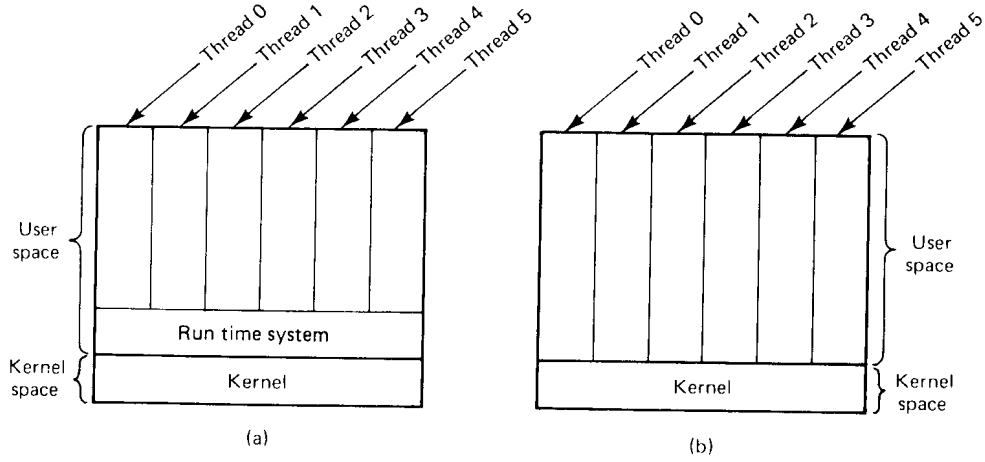

Fig. 4-8. (a) A user-level threads package. (b) A threads packaged managed by the kernel.

User-level threads also have other advantages. They allow each process to have its own customized scheduling algorithm. For some applications, for example, those with a garbage collector thread, not having to worry about a thread being stopped at an inconvenient moment is a plus. They also scale better, since kernel threads invariably require some table space and stack space in the kernel, which can be a problem if there are a very large number of threads.

Despite their better performance, user-level threads packages have some major problems. First among these is the problem of how blocking system calls are implemented. Suppose that a thread reads from an empty pipe or does something else that will block. Letting the thread actually make the system call is unacceptable, since this will stop all the threads. One of the main goals of

having threads in the first place was to allow each one to use blocking calls, but to prevent one blocked thread from affecting the others. With blocking system calls, this goal cannot be achieved.

The system calls could all be changed to be nonblocking (e.g., a read on a empty pipe could just fail), but requiring changes to the operating system is unattractive. Besides, one of the arguments for user-level threads was precisely that they could run with *existing* operating systems. In addition, changing the semantics of READ will require changes to many user programs.

Another alternative is possible in the event that it is possible to tell in advance if a call will block. In some versions of UNIX, a call SELECT exists, which allows the caller to tell whether a pipe is empty, and so on. When this call is present, the library procedure *read* can be replaced with a new one that first does a SELECT call and then only does the READ call if it is safe (i.e., will not block). If the READ call will block, the call is not made. Instead, another thread is run. The next time the runtime system gets control, it can check again to see if the READ is now safe. This approach requires rewriting parts of the system call library, is inefficient and inelegant, but there is little choice. The code placed around the system call to do the checking is called a **jacket**.

Somewhat analogous to the problem of blocking system calls is the problem of page faults. If a thread causes a page fault, the kernel, not even knowing about the existence of threads, naturally blocks the entire process until the needed page has been fetched, even though other threads might be runnable.

Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU. Within a single process, there are no clock interrupts, making round-robin scheduling impossible. Unless a thread enters the runtime system of its own free will, the scheduler will never get a chance.

An area in which the absence of clock interrupts is crucial is synchronization. It is common in distributed applications for one thread to initiate an activity to which another thread must respond and then just sit in a tight loop testing whether the response has happened. This situation is called a **spin lock** or **busy waiting**. This approach is especially attractive when the response is expected quickly and the cost of using semaphores is high. If threads are rescheduled automatically every few milliseconds based on clock interrupts, this approach works fine. However, if threads run until they block, this approach is a recipe for deadlock.

One possible solution to the problem of threads running forever is to have the runtime system request a clock signal (interrupt) once a second to give it control, but this too is crude and messy to program. Periodic clock interrupts at a higher frequency are not always possible, and even if they are, the total overhead may be substantial. Furthermore, a thread might also need a clock interrupt, interfering with the runtime system's use of the clock.

Another, and probably most devastating argument against user-level threads is that programmers generally want threads in applications where the threads block often, as, for example, in a multithreaded file server. These threads are constantly making system calls. Once a trap has occurred to the kernel to carry out the system call, it is hardly any more work for the kernel to switch threads if the old one has blocked, and having the kernel do this eliminates the need for constantly checking to see if system calls are safe. For applications that are essentially entirely CPU bound and rarely block, what is the point of having threads at all? No one would seriously propose to compute the first n prime numbers or play chess using threads because there is nothing to be gained by doing it that way.

Implementing Threads in the Kernel

Now let us consider having the kernel know about and manage the threads. No runtime system is needed, as shown in Fig. 4-8(b). Instead, when a thread wants to create a new thread or destroy an existing thread, it makes a kernel call, which then does the creation or destruction.

To manage all the threads, the kernel has one table per process with one entry per thread. Each entry holds the thread's registers, state, priority, and other information. The information is the same as with user-level threads, but it is now in the kernel instead of in user space (inside the runtime system). This information is also the same information that traditional kernels maintain about each of their single-threaded processes, that is, the process state.

All calls that might block a thread, such as interthread synchronization using semaphores, are implemented as system calls, at considerably greater cost than a call to a runtime system procedure. When a thread blocks, the kernel, at its option, can run either another thread from the same process (if one is ready), or a thread from a different process. With user-level threads, the runtime system keeps running threads from its own process until the kernel takes the CPU away from it (or there are no ready threads left to run).

Due to the relatively greater cost of creating and destroying threads in the kernel, some systems take an environmentally correct approach and recycle their threads. When a thread is destroyed, it is marked as not runnable, but its kernel data structures are not otherwise affected. Later, when a new thread must be created, an old thread is reactivated, saving some overhead. Thread recycling is also possible for user-level threads, but since the thread management overhead is much smaller, there is less incentive to do this.

Kernel threads do not require any new, nonblocking system calls, nor do they lead to deadlocks when spin locks are used. In addition, if one thread in a process causes a page fault, the kernel can easily run another thread while waiting for the required page to be brought in from the disk (or network). Their

main disadvantage is that the cost of a system call is substantial, so if thread operations (creation, deletion, synchronization, etc.) are common, much more overhead will be incurred.

In addition to the various problems specific to user threads and those specific to kernel threads, there are some other problems that occur with both of them. For example, many library procedures are not reentrant. For example, sending a message over the network may well be programmed to assemble the message in a fixed buffer first, then to trap to the kernel to send it. What happens if one thread has assembled its message in the buffer, then a clock interrupt forces a switch to a second thread that immediately overwrites the buffer with its own message? Similarly, after a system call completes, a thread switch may occur before the previous thread has had a chance to read out the error status (*errno*, as discussed above). Also, memory allocation procedures, such as the UNIX *malloc*, fiddle with crucial tables without bothering to set up and use protected critical regions, because they were written for single-threaded environments where that was not necessary. Fixing all these problems properly effectively means rewriting the entire library.

A different solution is to provide each procedure with a jacket that locks a global semaphore or mutex when the procedure is started. In this way, only one thread may be active in the library at once. Effectively, the entire library becomes a big monitor.

Signals also present difficulties. Suppose that one thread wants to catch a particular signal (say, the user hitting the DEL key), and another thread wants this signal to terminate the process. This situation can arise if one or more threads run standard library procedures and others are user-written. Clearly, these wishes are incompatible. In general, signals are difficult enough to manage in a single-threaded environment. Going to a multithreaded environment does not make them any easier to handle. Signals are typically a per-process concept, not a per-thread concept, especially if the kernel is not even aware of the existence of the threads.

Scheduler Activations

Various researchers have attempted to combine the advantage of user threads (good performance) with the advantage of kernel threads (not having to use a lot of tricks to make things work). Below we will describe one such approach devised by Anderson et al. (1991), called **scheduler activations**. Related work is discussed by Edler et al. (1988) and Scott et al. (1990).

The goals of the scheduler activation work are to mimic the functionality of kernel threads, but with the better performance and greater flexibility usually associated with threads packages implemented in user space. In particular, user

threads should not have to be make special nonblocking system calls or check in advance if it is safe to make certain system calls. Nevertheless, when a thread blocks on a system call or on a page fault, it should be possible to run other threads within the same process, if any are ready.

Efficiency is achieved by avoiding unnecessary transitions between user and kernel space. If a thread blocks on a local semaphore, for example, there is no reason to involve the kernel. The user-space runtime system can block the synchronizing thread and schedule a new one by itself.

When scheduler activations are used, the kernel assigns a certain number of virtual processors to each process and lets the (user-space) runtime system allocate threads to processors. This mechanism can also be used on a multiprocessor where the virtual processors may be real CPUs. The number of virtual processors allocated to a process is initially one, but the process can ask for more and can also return processors it no longer needs. The kernel can take back virtual processors already allocated to assign them to other, more needy, processes.

The basic idea that makes this scheme work is that when the kernel knows that a thread has blocked (e.g., by its having executed a blocking system call or caused a page fault), the kernel notifies the process' runtime system, passing as parameters on the stack the number of the thread in question and a description of the event that occurred. The notification happens by having the kernel activate the runtime system at a known starting address, roughly analogous to a signal in UNIX. This mechanism is called an **upcall**.

Once activated like this, the runtime system can reschedule its threads, typically by marking the current thread as blocked and taking another thread from the ready list, setting up its registers, and restarting it. Later, when the kernel learns that the original thread can run again (e.g., the pipe it was trying to read from now contains data, or the page it faulted over has been brought in from disk), it makes another upcall to the runtime system to inform it of this event. The runtime system, at its own discretion, can either restart the blocked thread immediately, or put it on the ready list to be run later.

When a hardware interrupt occurs while a user thread is running, the interrupted CPU switches into kernel mode. If the interrupt is caused by an event not of interest to the interrupted process, such as completion of another process' I/O, when the interrupt handler has finished, it puts the interrupted thread back in the state it was in before the interrupt. If, however, the process is interested in the interrupt, such as the arrival of a page needed by one of the process' threads, the interrupted thread is not restarted. Instead, the interrupted thread is suspended and the runtime system started on that virtual CPU, with the state of the interrupted thread on the stack. It is then up to the runtime system to decide which thread to schedule on that CPU: the interrupted one, the newly ready one, or some third choice.

Although scheduler activations solve the problem of how to pass control to

an unblocked thread in a process one of whose threads has just blocked, it creates a new problem. The new problem is that an interrupted thread might have been executing a semaphore operation at the time it was suspended, in which case it would probably be holding a lock on the ready list. If the runtime system started by the upcall then tries to acquire this lock itself, in order to put a newly ready thread on the list, it will fail to acquire the lock and a deadlock will ensue. The problem can be solved by keeping track of when threads are or are not in critical regions, but the solution is complicated and hardly elegant.

Another objection to scheduler activations is the fundamental reliance on upcalls, a concept that violates the structure inherent in any layered system. Normally, layer n offers certain services that layer $n + 1$ can call on, but layer n may not call procedures in layer $n + 1$.

4.1.5. Threads and RPC

It is common for distributed systems to use both RPC and threads. Since threads were invented as a cheap alternative to standard (heavyweight) processes, it is natural that researchers would take a closer look at RPC in this context, to see if it could be made more lightweight as well. In this section we will discuss some interesting work in this area.

Bershad et al. (1990) have observed that even in a distributed system, a substantial number of RPCs are to processes on the same machine as the caller (e.g., to the window manager). Obviously, this result depends on the system, but it is common enough to be worth considering. They have proposed a new scheme that makes it possible for a thread in one process to call a thread in another process on the same machine much more efficiently than the usual way.

The idea works like this. When a server thread, S, starts up, it exports its interface by telling the kernel about it. The interface defines which procedures are callable, what their parameters are, and so on. When a client thread C starts up, it imports the interface from the kernel and is given a special identifier to use for the call. The kernel now knows that C is going to call S later, and creates special data structures to prepare for the call.

One of these data structures is an argument stack that is shared by both C and S and is mapped read/write into both of their address spaces. To call the server, C pushes the arguments onto the shared stack, using the normal procedure passing conventions, and then traps to the kernel, putting the special identifier in a register. The kernel sees this and knows that the call is local. (If it had been remote, the kernel would have treated the call in the normal manner for remote calls.) It then changes the client's memory map to put the client in the server's address space and starts the client thread executing the server's procedure. The call is made in such a way that the arguments are already in place,

so no copying or marshaling is needed. The net result is that local RPCs can be done much faster this way.

Another technique to speed up RPCs is based on the observation that when a server thread blocks waiting for a new request, it really does not have any important context information. For example, it rarely has any local variables, and there is typically nothing important in its registers. Therefore, when a thread has finished carrying out a request, it simply vanishes and its stack and context information are discarded.

When a new message comes in to the server's machine, the kernel creates a new thread on-the-fly to service the request. Furthermore, it maps the message into the server's address space, and sets up the new thread's stack to access the message. This scheme is sometimes called **implicit receive** and it is in contrast to a conventional thread making a system call to receive a message. The thread that is created spontaneously to handle an incoming RPC is occasionally referred to as a **pop-up thread**. The idea is illustrated in Fig. 4-9.

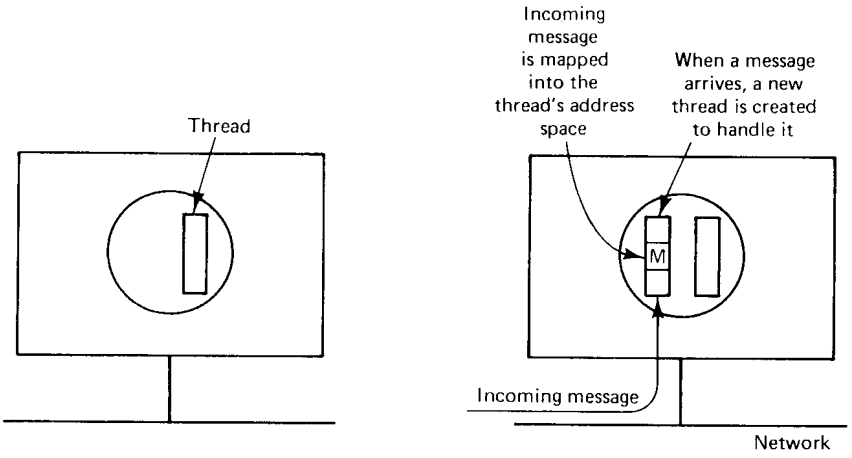

Fig. 4-9. Creating a thread when a message arrives.

The method has several major advantages over conventional RPC. First, threads do not have to block waiting for new work. Thus no context has to be saved. Second, creating a new thread is cheaper than restoring an existing one, since no context has to be restored. Finally, time is saved by not having to copy incoming messages to a buffer within a server thread. Various other techniques can also be used to reduce the overhead. All in all, a substantial gain in speed is possible.

Threads are an ongoing research topic. Some other results are presented in (Marsh et al., 1991; and Draves et al., 1991).