OPERATING SYSTEMS

THREADS

3 Threads

2

Each process has its address space and starts with a single thread. Nevertheless, multiple threads run in the same address space in the same process, as if they were separate processes.

Why would anyone want to have a kind of process within a process? There are several reasons for having these miniprocesses, called threads.

3.1 Thread Usage

3

The main reason for having threads is that multiple activities are going on at once. To become simpler the programming model, an application is decomposed into multiple sequential threads that run in quasi-parallel.

With threads, we add only a new element: the ability for the parallel entities to share an address space and all of its data among themselves.

3.1 Thread Usage

4

The second advantage is that threads are easier to create and destroy than processes. In many systems, creating a thread goes 10-100 times faster than creating a process.

A third reason is performance. Threads can yield performance gain when there is substantial computing and also substantial I/O, having threads allows these activities to overlap, thus speeding up the application. Finally, threads may supply real parallelism.

3.1 Thread Usage

5

Suppose that the user suddenly deletes one sentence from page 1 of an 800-page document. Then to make another change, he types a command to go to page 600. The word processor is now forced to reformat the entire book up to page 600 on the spot because it does not know what the first line of page 600 will be until it has processed all the previous pages. There may be a sizeable delay before page 600 can be displayed, leading to an unhappy user.

3.1 Thread Usage

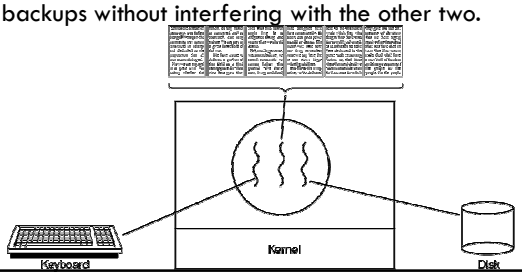
6

For the word processor is written as a two-threaded program, one thread interacts with the user and the other handles reformatting in the background. As soon as the sentence is deleted from page 1, the interactive thread tells the reformatting thread to run the whole book. Meanwhile, the interactive thread continues to listen to the input devices and responds to simple commands like scrolling page 1 while the other thread is computing madly in the background. If the reformatting is completed before the request to see page 600, it can be displayed without delay.

3.1 Thread Usage

7

Many word processors have a feature of automatic saving the entire file to disk every few minutes to protect the data. The third thread can handle the disk backups without interfering with the other two.



3.1 Thread Usage

8

If the program were single-threaded, then whenever a disk backup started, commands from the keyboard and mouse would be ignored until the backup was finished. The user would suppose the system as slow performance. Alternatively, keyboard and mouse events could interrupt the disk backup, allowing good performance but leading to a complex interrupt-driven programming model. With three threads, the programming model is much simpler.

3.1 Thread Usage

9

It should be clear that having three separate processes would not work here because all three threads need to operate on the document. By having three threads instead of three processes, they share a common memory and thus all have access to the same document being edited.

3.1 Thread Usage

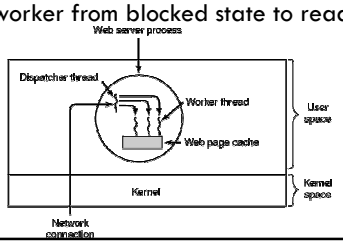
10

Now consider a web server. Requests for pages come in and the requested page is sent back. Some pages are more commonly accessed than others. e.g., Samsung's home page is accessed far more than a page containing the technical specifications of any particular camcorder. Web servers use this fact to improve performance by maintaining a collection of heavily used pages in main memory. Such a collection is called a cache.

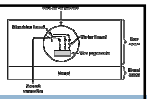
3.1 Thread Usage

11

In figure, one thread (the dispatcher) reads incoming requests for work from the network. After examining the request, it chooses an idle worker thread and hands it the request. The dispatcher then moves sleeping worker from blocked state to ready state.



3.1 Thread Usage



12

When the worker wakes up, it checks if the request can be satisfied from the cache. If not, it starts a read operation to get the page from the disk and blocks until the disk operation completes. When the thread blocks on the disk operation, another thread is chosen to run, possibly the dispatcher, in order to acquire more work, or possibly another worker that is now ready to run.

3.1 Thread Usage

13

A rough outline of the code is given in figure. Here, TRUE is assumed to be the constant 1. Also, *buf* and *page* are structures appropriate for holding a work request and a web page, respectively.

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, Spage);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(Spage);
}
```

3.1 Thread Usage

14

Consider how the Web server could be written in the absence of threads or with only a single thread. The main loop of the Web 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 incoming requests. If the Web server is running on a dedicated machine, the CPU is simply idle while the Web server is Waiting for the disk. The net result is that many fewer requests/sec can be processed. Thus threads gain considerable performance.

3.1 Thread Usage

15

So far we have seen two possible designs: a multithreaded and a single-threaded Web server. Suppose that threads are not available but the system designers find the performance loss due to single threading unacceptable. If a nonblocking version of the read system call is available, a third approach is possible. 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 nonblocking disk operation is started.

3.1 Thread Usage

16

The server records the state of the current request in a table and then goes and gets the next event. The next event 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. With nonblocking disk I/O, a reply probably will have to take the form of a signal or interrupt.

3.1 Thread Usage

17

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 every time the server switches from working on one request to another. In effect, we are simulating the threads and their stacks the hard way. A design like this, in which each computation has a saved state, and there exists some set of events that can occur to change the state is called a finite-state machine. This concept is widely used throughout computer science.

3.1 Thread Usage

18

A third example is applications used very large amounts of data. The normal approach is to read in a block of data, process it, and then write it out again. The problem here is that if only blocking system calls are available, the process blocks while data are coming in and data are going out. Having the CPU go idle when there is lots of computing to do is clearly wasteful and should be avoided if possible.

3.1 Thread Usage

19

Threads offer a solution. The process could be structured with an input thread, a processing thread, and an output thread. The input thread reads data into an input buffer. The processing thread takes data out of the input buffer, processes them, and puts the results in an output buffer. The output buffer writes these results back to disk. In this way, input, output, and processing can all be going on at the same time. Of course, this model only works if a system call blocks only the calling thread, not the entire process.

3.2 The Classical Thread Model

20

One important concepts in the process model is a thread of execution. The thread has a program counter that follows which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its process are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

3.2 The Classical Thread Model

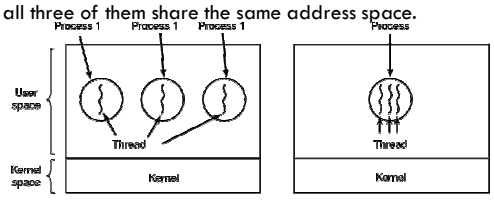
21

What threads add to the process model is to allow multiple executions to take place in the same process environment. Having multiple threads running in parallel in one process is similar to having multiple processes running in parallel in one computer. While the threads share an address space and other resources, processes share physical memory, disks, printers, and other resources. Because threads have some of the properties of processes, they are sometimes called lightweight processes.

3.2 The Classical Thread Model

22

In figure (a) we see three traditional processes. Each process has its own address space and a single thread of control. In contrast, in figure (b) we see a single process with three threads of control. Although in both cases we have three threads, in figure (a) each of them operates in a different address space, whereas in figure (b) all three of them share the same address space.



(a) Three processes each with one thread. (b) One process with three threads.

3.2 The Classical Thread Model

23

When a multithreaded process is run on a single-CPU system, the threads take turns running. By switching back and forth among multiple processes, the system gives the illusion of separate sequential processes running in parallel. Multithreading works the same way. With three compute-bound threads in a process, the threads would appear to be running in parallel, each one on a CPU with one-third the speed of the real CPU.

3.2 The Classical Thread Model

24

Different threads are not as independent as different processes. All threads have the same address space, set of open files, child processes, alarms, and signals. Since every thread can access every memory address within the process' address space, one thread can read, write, or even destroy another thread's stack. There is no protection between threads because (1) it is impossible, and (2) it should not be necessary. A process is always owned by a single user, who has created multiple threads so that they can cooperate, not fight.

3.2 The Classical Thread Model

25

The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

3.2 The Classical Thread Model

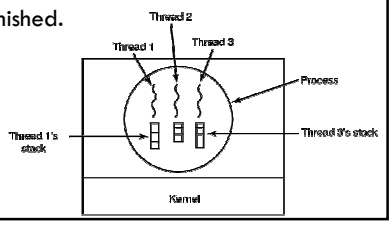
26

Like a process with only one thread, a thread 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 some event to unblock it. For example, when a thread performs a system call to read from the keyboard, it is blocked until input is typed. A thread can block waiting for some external event to happen or for some other thread to unblock it. A ready thread is scheduled to run and will as soon as its turn comes up.

3.2 The Classical Thread Model

27

It is important to realize that each thread has its own stack, as illustrated in figure below. Each thread's stack contains one frame for each procedure called but not yet returned from. This frame contains the procedure's local variables and the return address to use when the procedure call has finished.



3.2 The Classical Thread Model

28

When multithreading is present, processes normally start with a single thread. This thread has the ability to create new threads by calling a library procedure, for example, *thread_create*. A parameter to *thread_create* specifies the name of a procedure to run. It is not possible to specify anything about the new thread's address space, since it automatically runs in the address space of the creating thread. Sometimes threads are hierarchical, with a parent-child relationship, but often no such relationship exists, with all threads being equal.

3.2 The Classical Thread Model

29

When a thread has finished its work, it can exit by calling a library procedure, say, *thread_exit*. It then disappears and is no longer schedulable. In some thread systems, one thread can wait for a specific thread to exit by calling a procedure, for example, *thread_join*. This procedure blocks the calling thread until a specific thread has exited. In this regard, thread creation and termination is very much like process creation and termination, with the same options as well.

3.2 The Classical Thread Model

30

Another common thread call, *thread_yield*, allows a thread to voluntarily give up the CPU to let another thread run. Such a call is important because there is no interrupt to enforce multiprogramming as there is with processes. Thus it is important for threads to be polite and voluntarily abandon the CPU from time to time to give other threads a chance to run. Other calls allow one thread to wait for another thread to finish some work, for a thread to announce that it has finished some work.

3.2 The Classical Thread Model

31

Besides threads introduce a number of problems into the programming model. To start with, consider the effects of the fork system call.

- If the parent process has multiple threads, should the child also have them? If not, the process may not work properly, since all of them may be essential. However, if the child process gets as many threads as the parent, what happens if a thread in the parent was blocked on a read call, say, from the keyboard? Are two threads now blocked on the keyboard, one in the parent and one in the child? When a line is typed, do both threads get a copy of it? Only the parent? Only the child?

3.2 The Classical Thread Model

32

Another class of problems is related to the fact that threads share many data structures.

- What happens if one thread closes a file while another one is still reading from it? Suppose that one thread notices that there is too little memory and starts allocating more memory. Just then, a thread switch occurs, and the new thread also notices that there is too little memory and also starts allocating more memory. Memory will probably be allocated twice. These problems can be solved with some effort, but careful thought and design are needed to make multithreaded programs work correctly.

3.3 POSIX Threads

33

IEEE has defined a standard (IEEE1003.1c) for threads to write portable threaded programs. The threads package is called *Pthreads*. Most UNIX systems support it. The standard defines over 60 function calls. Instead we will just describe a few of the major ones to give an idea of how it works.

3.3 POSIX Threads

34

The calls we will describe are listed as follow.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

3.3 POSIX Threads

35

```
#include <stdio.h>
#include <sys/syscall.h>
#include <pthread.h>
void *myID()
{
  printf("I am a thread.");
}
main()
{
  pthread_t   thred;
  int th_no;
  th_no = pthread_create(&thred, NULL, myID, NULL)
  pthread_exit(&thred);
}
```

3.4 Implementing in User Space

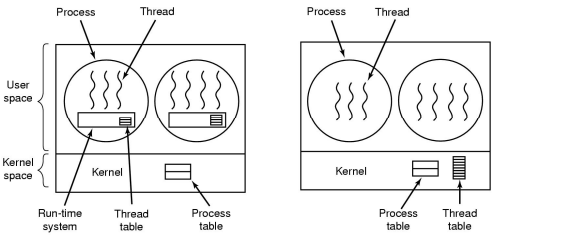
36

There are two main ways to implement a threads package: in user space and in the kernel. The first method is to put the threads package entirely in user space. The kernel knows nothing about them. The most obvious advantage is that a user-level threads package can be implemented on an operating system that does not support threads. With this approach, threads are implemented by a library.

3.4 Implementing in User Space

37

All of these implementations have the same general structure, which is illustrated in figure.

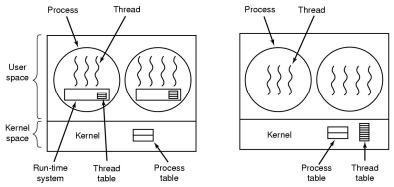


(a) A user-level threads package. (b) A threads package managed by the kernel.

3.4 Implementing in User Space

38

The threads run on top of a run-time system, which is a collection of procedures that manage threads. We have seen four of these already: *pthread_create*, *pthread_exit*, *pthread_join*, and *pthread_yield*.



(a) A user-level threads package. (b) A threads package managed by the kernel.

3.4 Implementing in User Space

39

When threads are managed in user space, each process needs its own private thread table to keep track of them. This table is similar to the kernel's process table, except that it keeps track only of the per-thread properties, such as each thread's program counter, stack pointer, registers, state, and so forth. The thread table is managed by the run-time system. When a thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table, exactly the same way as ones in the process table.

3.4 Implementing in User Space

40

When a thread does something that may cause it to become blocked locally, it calls a run-time system procedure. This procedure checks that if the thread is in blocked state. If so, it stores the its own registers in the thread table, looks in the table for a ready 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.

3.4 Implementing in User Space

41

However, there is one key difference with processes. When a thread calls *thread_yield*, the code of *thread_yield* can save the thread's information in the thread table itself. Then it can call the thread scheduler to pick another thread to run. The procedure that saves the thread's state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call. Among other issues, no trap is needed, no context switch is needed, the memory cache need not be flushed, and so on. This makes thread scheduling very fast.

3.4 Implementing in User Space

42

Besides, user-level threads 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 always require some table space and stack (space in the kernel which can be a problem if there are a very large number of threads).

3.4 Implementing in User Space

43

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 the keyboard. Letting the thread make the system call is unacceptable, since this will stop all the threads. One of the main goals was to allow each one to use blocking calls, but to prevent one blocked thread from affecting the others. With blocking system calls, it is hard to see how this goal can be achieved.

3.4 Implementing in User Space

44

The system calls could all be changed to be nonblocking (e.g., a read on the keyboard would just return 0 bytes if no characters were already buffered), 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.

3.4 Implementing in User Space

45

Somewhat analogous to the problem of blocking system calls is the problem of page faults. Briefly, computers may not load all of a program in main memory at once. If the program jumps to an instruction that is not in memory, a page fault occurs and the operating system will get the missing instruction from disk. This is called a page fault. The process is blocked while the necessary instruction is being located. If a thread causes a page fault, the kernel naturally blocks the entire process until the disk I/O is complete, even though other threads maybe runnable.

3.4 Implementing in User Space

46

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 it impossible to schedule processes round-robin fashion (taking turns). Unless a thread enters the run-time system of its own free will, the scheduler will never get a chance.

3.4 Implementing in User Space

47

One possible solution to the problem of threads running forever is to have the run-time 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 run-time system's use of the clock.

3.4 Implementing in User Space

48

Another, and really the most destroying, argument against user-level threads is that programmers generally want threads precisely in applications where the threads block often. 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 making select system calls that check to see if read system calls are safe.

3.5 Implementing in the Kernel

49

Now let us consider having the kernel know about and manage the threads. No run-time system is needed in each. Also, there is no thread table in each process. Instead, the kernel has a thread table that keeps track of all the threads in the system. 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 by updating the kernel thread table.

3.5 Implementing in the Kernel

50

The kernel's thread table holds each thread's registers, state, and other information. The information is the same as with user-level threads, but now kept in the kernel instead of in user space (inside the run-time system). This information is a subset of the information that traditional kernels maintain about their single threaded processes, that is, the process state. In addition, the kernel also maintains the traditional process table to keep track of processes.

3.5 Implementing in the Kernel

51

All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure. When a thread blocks, the kernel can run either another thread in ready from the same process or a thread from a different process. With user-level threads, the run-time 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).

3.5 Implementing in the Kernel

52

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.

3.5 Implementing in the Kernel

53

Kernel threads do not require any new, nonblocking system calls. In addition, if one thread in a process causes a page fault, the kernel can easily check to see if the process has any other runnable threads, and if so, run one of them while waiting for the required page to be brought in from the disk. Their main disadvantage is that the cost of a system call is substantial, so if thread operations (creation, termination, etc.) are common, much more overhead will be incurred.

3.5 Implementing in the Kernel

54

While kernel threads solve some problems, they do not solve all problems. For example, what happens when a multithreaded process forks? Does the new process have as many threads as the old one did, or does it have just one? In many cases, the best choice depends on what the process is planning to do next. If it is going to call exec to start a new program, probably one thread is the correct choice, but if it continues to execute, reproducing all the threads is probably the right thing to do.

3.5 Implementing in the Kernel

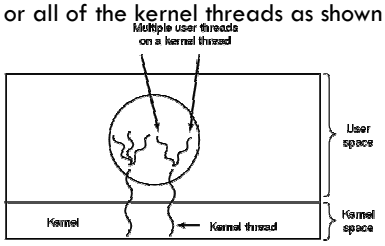
55

Another issue is signals. Remember that signals are sent to processes, not to threads, at least in the classical model. When a signal comes in, which thread should handle it? Possibly threads could register their interest in certain signals, so when a signal came in it would be given to the thread that said it wants it. But what happens if two or more threads register for the same signal. These are only two of the problems threads introduce, but there are more.

3.6 Hybrid Implementations

56

To combine the advantages of user-level threads with kernel-level threads, one way is use kernel-level threads and then multiplex user-level threads onto some or all of the kernel threads as shown in figure.

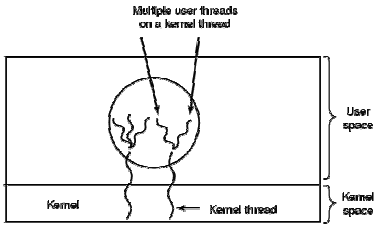


Multiplexing user-level threads onto kernel-level threads.

3.6 Hybrid Implementations

57

When this approach is used, the programmer can determine how many kernel threads to use and how many user-level threads to multiplex on each one.



Multiplexing user-level threads onto kernel-level threads.

3.6 Hybrid Implementations

58

With this approach, the kernel is aware of only the kernel-level threads and schedules those. Some of those threads may have multiple user-level threads multiplexed on top of them. These user-level threads are created, destroyed, and scheduled just like user-level threads in a process that runs on an operating system without multithreading capability. In this model, each kernel-level thread has some set of user-level threads that take turns using it.

3.7 Scheduler Activations

59

Although kernel threads are better than user-level threads in some key ways, they are slower. The goals of the scheduler activation work are to mimic the functionality of kernel threads, but with the better performance and greater flexibility in user space. Efficiency is achieved by avoiding unnecessary transitions between user and kernel space.

3.7 Scheduler Activations

60

When a hardware interrupt occurs while a user thread is running, the CPU switches into kernel mode. When the interrupt handler has finished, if the process is not interested in the interrupt, it puts the interrupted thread back in its before state; but if the process is interested in the interrupt, the interrupted thread is not restarted. Instead, the run-time system is started on that virtual CPU, with the state of the interrupted thread. It is then up to the run-time system to decide which thread to schedule on that CPU: the interrupted one, the newly ready one, or some third choice.

3.8 Pop-Up Threads

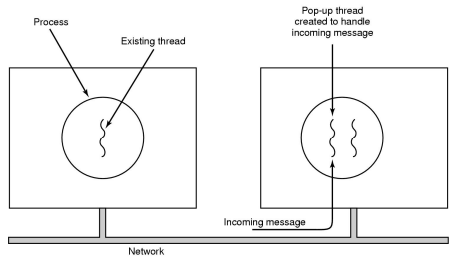
61

Threads are frequently useful in distributed systems. An important example is how incoming messages, for example requests for service, are handled. The traditional approach is to have a process or thread that is blocked on a receive system call waiting for an incoming message. When a message arrives, it accepts the message and processes it. However, a completely different approach is also possible, in which the arrival of a message causes the system to create a new thread to handle the message.

3.8 Pop-Up Threads

62

Such a thread is called a pop-up thread and is illustrated in figure.

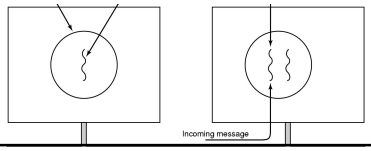


Creation of a new thread when a message arrives. (a) Before the message arrives. (b) After the message arrives.

3.8 Pop-Up Threads

63

A key advantage of pop-up threads is that since they are brand new, they don't have any history (registers, stack, etc.). This makes it possible to create such a thread quickly. The new thread is given the incoming message to process. The result of using pop-up threads is that the latency between message arrival and the start of processing can be made very short.



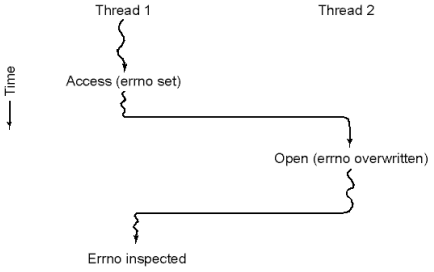
3.9 Single-Thread to Multithread

64

Converting programs written as a single-threaded to multithreading requires some tricks. As a start, the code of a thread normally consists of multiple procedures, just like a process. These may have local variables, global variables, and 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 are a problem. These are variables that are global in the sense that many procedures within the thread use them, but other threads should logically leave them alone.

3.9 Single-Thread to Multithread

65



Conflicts between threads over the use of a global variable

3.9 Single-Thread to Multithread

66

In figure, Thread1 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 Thread1, but before it has a chance to read *errno*, the scheduler decides that Thread1 has had enough CPU time for the moment and decides to switch to Thread2. Thread2 executes an open call that fails, which causes *errno* to be overwritten and Thread1's access code to be lost forever. When Thread1 starts up later, it will read the wrong value and behave incorrectly.

3.9 Single-Thread to Multithread

67

The next problem is that many library procedures are not reentrant. That is, they were not designed

to have a second call made to any given procedure while a previous call has not yet finished.

Similarly, memory allocation procedures maintain crucial tables about memory usage. If a thread switch occurs while the tables are inconsistent and a new call comes in from a different thread, an invalid pointer may be used, leading to a program crash.

3.9 Single-Thread to Multithread

68

Next, consider signals. Some signals are logically thread specific, whereas others are not. For example, if a thread calls alarm, it makes sense for the resulting signal to go to the thread that made the call. However, when threads are implemented entirely in user space, the kernel does not even know about threads and can hardly direct the signal to the right one.

3.9 Single-Thread to Multithread

69

One last problem introduced by threads is stack management. In many systems, when a process' stack overflows, the kernel just provides that process with more stack automatically. When a process has multiple threads, it must also have multiple stacks. If the kernel is not aware of all these stacks, it cannot grow them automatically upon stack fault.