

The Classical Thread Model

Bindu Singh

Department of Computer Engineering
Vadodara Institute of Engineering

Abstract-One way of looking at a process is that it is a way to group related resources together. A process has an address space containing program text and data, as well as other resources. This resource may include open files, child processes, pending alarms, signal handlers, accounting information, and more. By putting them together in the form of a process, they can be managed more easily. The other concept a process has is a thread of execution, usually shortened to just thread.

Keywords-Thread, User space, Kernel Space, Process, Program counter

I. INTRODUCTION

A thread is the smallest unit of processing that can be performed in an OS. In most modern operating systems, a thread exists within a process - that is, a single process may contain multiple threads. The thread has a program counter that keeps track of 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. What threads add to the process model is to allow multiple executions to take place in the same process environment, to a large degree independent of one another. Having multiple threads running in parallel in one process is analogous to having multiple processes running in parallel in one computer. In the former case, the threads share an address space and other resources. In the latter case, processes share physical memory, disks, printers, and other resources. Because threads have some of the properties of processes, they are sometimes called lightweight processes. The term multithreading is also used to describe the situation of allowing multiple threads in the same process. Some CPUs have direct hardware support for multithreading and allow thread switches to happen on a nanosecond time scale. When a multithreaded process runs 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. The CPU switches rapidly back and forth among the threads,

providing the illusion that the threads are running in parallel, albeit on a slower CPU than the real one. 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.

Different threads in a process are not as independent as different processes. All threads have exactly the same address space, which means that they also share the same global variables. Since every thread can access every memory address within the process' address space, one thread can read, write, or even 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 can share the same set of open files, child processes, alarms, and signals, and so on. Per process items

- | | |
|------------------|-----------------------------|
| 1) | Address space |
| 2) | Global variables |
| 3) | Open files |
| 4) | Child processes |
| 5) | Pending alarms |
| 6) | Signals and signal handlers |
| 7) | Accounting information |
| Per Thread items | |
| 8) | Program counter |
| 9) | Registers |
| 10) | Stack |
| 11) | State |

The items in the first column are process properties, not thread properties. For example, if one thread opens a file, that file is visible to the other threads in the process and they can read and write it. This is logical, since the process is the unit of resource management, not the thread. If each thread had its own address space, open files, pending alarms, and so on, it would be a separate process. What we are trying to achieve with the thread concept is the ability for multiple threads of execution to share a set of resources so that they can work together closely to perform some task. Like a traditional process (i.e., 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. The transitions between thread states are the same as the transitions between process states. It is important to realize that each thread has its own stack. 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. For example, if procedure X calls procedure Y and Y calls procedure Z, then while Z is executing, the frames for X, Y, and Z will all be on the stack. Each thread will generally call different procedures and thus have a different execution history. This is why each thread needs its own stack.

When multithreading is present, processes normally start with a single thread present. This thread has the ability to create new threads by calling a library procedure, for example, `thread ^create`. A parameter to `thread ^create` typically specifies the name of a procedure for the new thread to run. It is not necessary (or even 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. With or without a hierarchical relationship, the creating thread is usually returned a thread identifier that names the new thread. When a thread has finished its work, it can exit by calling a library procedure, say, `thread-exit`. It then vanishes 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 approximately the same options as well. Another common thread call is `thread-yield`, which allows a thread to voluntarily give up the CPU to let another thread run. Such a call is important because there is no clock interrupt to actually enforce multiprogramming as there is with processes. Thus it is important for threads to be polite and voluntarily surrender 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, and so on. While threads are often useful, they also introduce a number of complications into the programming model. To start with, consider the effects of the UNDC fork system call. If the parent process has multiple threads, should the child also have

them? If not, the process may not function 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? The same problem exists with open network connections. 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. Partway through, 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.

II. IMPLEMENTING THEAD IN USER SPACE

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. We will now describe these methods, along with their advantages and disadvantages. 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. All operating systems used to fall into this category, and even now some still do. With this approach, threads are implemented by a library. All of these implementations have the same general structure. 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. When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process. This table is analogous 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 the kernel stores information about processes in the process table. When a thread does something that may cause it to become blocked locally, for example, waiting for another thread in its process to complete some

work, it calls a run-time system procedure. This procedure checks to see if the thread must be put into blocked state. If so, it stores the thread's registers (i.e., its own) 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. 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 just a handful of instructions. Doing thread switching like this is at least an order of magnitude—maybe one—faster than trapping to the kernel and is a strong argument in favor of user-level threads packages. However, there is one key difference with processes. When a thread is finished running for the moment, for example, when it calls `thread_yield`, the code of `thread_yield` can save the thread's information in the thread table itself. Furthermore, it can then 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.

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 the keyboard before any keys have been hit. 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, it is hard to see how this goal can be achieved readily.

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. 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 system call, `select`, exists, which allows the caller to tell whether a prospective read will block. 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 run-time 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 or wrapper.

Somewhat analogous to the problem of blocking system calls is the problem of page faults. It is sufficient to say that computers can be set up in such a way that not all of the program is in main memory at once. If the program calls or jumps to an instruction that is not in memory, a page fault occurs and the operating system will go and get the missing instruction (and its neighbors) from disk. This is called a page fault. The process is blocked while the necessary instruction is being located and read in. If a thread causes a page fault, the kernel, not even knowing about the existence of threads, naturally blocks the entire process until the disk I/O is complete, 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 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. 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. Another, and really the most devastating, argument against user-level threads is that programmers generally want threads precisely in applications where the threads block often, as, for example, in a multithreaded Web 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 making `select` system calls that check to see if `read` 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 computing the first n prime numbers or playing chess using

threads because there is nothing to be gained by doing it that way.

III. IMPLEMENTING THEAD IN KERNEL SPACE

Now let us consider having the kernel know about and manage the threads. No run-time system is needed in each, as shown in Fig. 2-16(b). 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. 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. Ail 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, 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 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) 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. 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. 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. 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.

IV. CONCLUSION

This paper gives brief introduction about various terms used in thread. These terms are useful to get the detail of thread. Threads are minute processes that are working in coordination with each other involved in any process.

REFERENCES

- [1] G. O. Young, —Synthetic structure of industrial plastics (Book style with paper title and editor), in *Plastics*, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964, pp. 15–64.
- [2] W.-K. Chen, *Linear Networks and Systems* (Book style). Belmont, CA: Wadsworth, 1993, pp. 123–135.
- [3] H. Poor, *An Introduction to Signal Detection and Estimation*. New York: Springer-Verlag, 1985, ch. 4.
- [4] B. Smith, —An approach to graphs of linear forms (Unpublished work style), unpublished.
- [5] E. H. Miller, —A note on reflector arrays (Periodical style—Accepted for publication), *IEEE Trans. Antennas Propagat.*, to be published.
- [6] J. Wang, —Fundamentals of erbium-doped fiber amplifiers arrays (Periodical style—Submitted for publication), *IEEE J. Quantum Electron.*, submitted for publication.