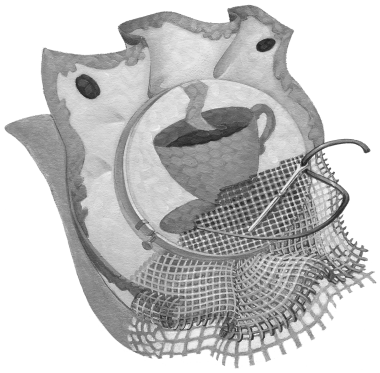
Chapter 2



Concepts

In which the reader is introduced to the basic concepts of multitasking operating systems and of multithreading as it compares to other programming paradigms. The reader is shown reasons why multithreading is a valuable addition to programming paradigms, and a number of examples of successful deployment are presented.

Background: Traditional Operating Systems

Before we get into the details of threads, it will be useful for us to have some clear understanding of how operating systems without threads work. In the simplest operating system world of single-user, single-tasking operating systems such as DOS, everything is quite easy to understand and to use, although the functionality offered is minimal.

DOS divides the memory of a computer into two sections: the portion where the operating system itself resides (*kernel space*[1]) and the portion where the programs reside (*user space*). The division into these two spaces is done strictly by the implicit agreement of the programmers involved—meaning that nothing stops a user program from accessing data in kernel space. This lack of hardware enforcement is good, because it is simple and works well when people write perfect programs. When a user program needs some function performed for it by kernel code (such as reading a file from a disk), the program can call the DOS function directly to read that file.

Each program has some code that it runs (which is just a series of instructions, where the *program counter* points to the current instruction), some data (global and local) that it uses, and a stack where local data and return addresses are stored (the *stack pointer* designates the current active location on the stack).

Figure 2–1 illustrates the traditional DOS operating system memory layout. Thus, as shown in Figure 2–1, the division between user space and kernel space is a division by agreement of the programmers; there is no hardware enforcement of the policy at all. The drawbacks to this technique are significant, however. Not all programs are written flawlessly, and a programming mistake (or virus!) here can bring down the entire machine or, worse, destroy valued data. Neither can a machine run more than one program at a time, nor can more than one user log in to the machine at a time. Dealing with networks from DOS machines is somewhat awkward and limited.

[1] *Kernel space* is UNIX lingo for this concept, but the concept is valid for all operating systems.

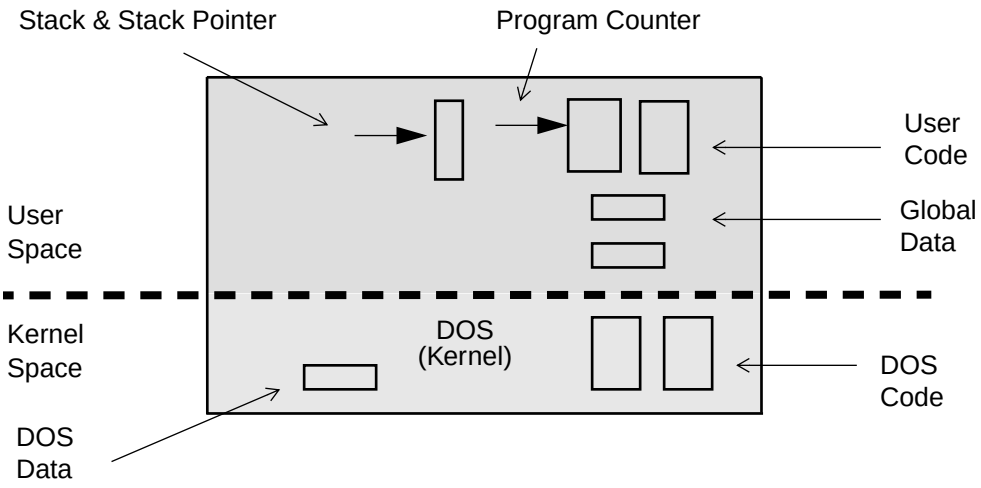


Figure 2–1 *Memory Layout for DOS-Style Operating Systems*

In a typical multitasking operating system such as VMS, UNIX, Windows NT, etc., this dividing line between the user space and the kernel space is solid (Figure 2–2); it's enforced by the hardware. There are actually two different modes of operation for the CPUs: *user mode*, which allows normal user programs to run, and *kernel mode*, which also allows some special instructions to run that only the kernel can execute. These kernel-mode instructions include I/O instructions, processor interrupt instructions, instructions that control the state of the virtual memory subsystem, and, of course, the *change mode* instruction.

So a user program can execute only user-mode instructions, and it can execute them only in user space. The data it can access and change directly is also limited to data in user space. When it needs something from the kernel (say, it wants to read a file or find out the current time), the user program must make a *system call*. This is a library function that sets up some arguments, then executes a special *trap* instruction. This instruction causes the hardware to trap into the kernel, which then takes control of the machine. The kernel figures out what the user wants (based upon the data that the system call set up) and whether the user has permission to do so. Finally, the kernel performs the desired task, returning any information to the user process.

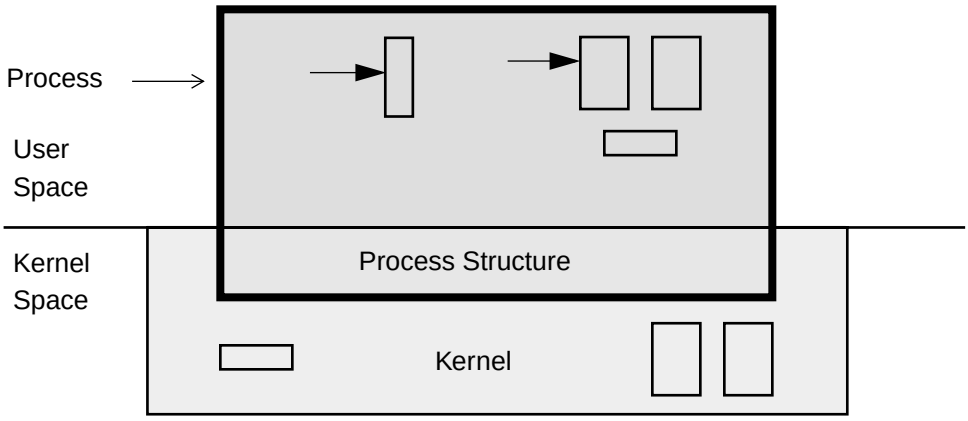


Figure 2–2 *Memory Layout for Multitasking Systems*

Because the operating system has complete control over I/O, memory, processors, etc., it needs to maintain data for each process it's running. The data tells the operating system what the state of that process is—what files are open, which user is running it, etc. So, the concept of *process* in the multitasking world extends into the kernel (see Figure 2–2), where this information is maintained in a *process structure*. In addition, as this is a multitasking world, more than one process can be active at the same time, and for most of these operating systems (notably, neither Windows NT nor OS/2), more than one user can log in to the machine independently and run programs simultaneously.

Thus, in Figure 2–3, process P1 can be run by user Kim while P2 and P3 are being run by user Dan, and P4 by user Bil. There is also no particular restriction on the amount of memory that a process can have. P2 might use twice as much memory as P1, for example. It is also true that no two processes can see or change each other's memory unless they have set up a special *shared memory* segment.

For all the user programs in all the operating systems mentioned so far, each has one stack, one program counter, and one set of CPU registers per process. So each of these programs can do only one thing at a time. They are *single threaded*.

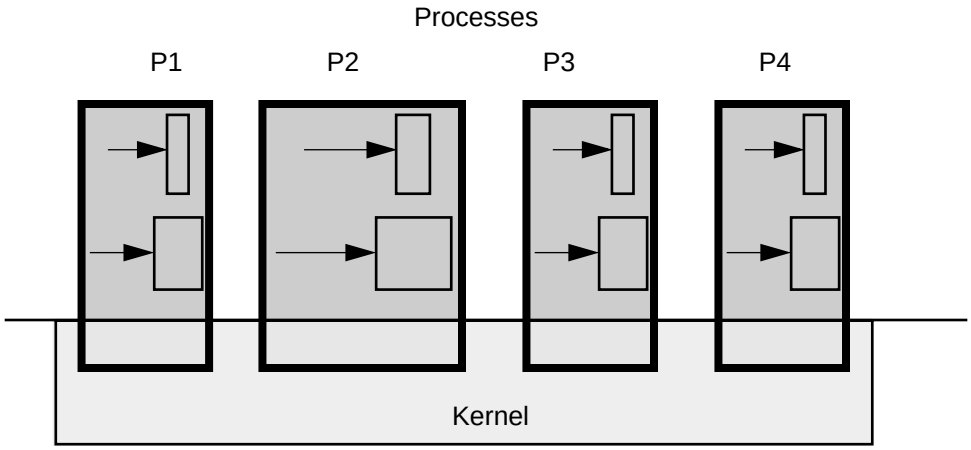


Figure 2–3 *Processes on a Multitasking System*

What Is a Thread?

Just as multitasking operating systems can do more than one thing concurrently by running more than a single process, a process can do the same by running more than a single *thread*. Each thread is a different stream of control that can execute its instructions independently, allowing a multithreaded process to perform numerous tasks concurrently. One thread can run the GUI while a second thread does some I/O and a third performs calculations.

A thread is an abstract concept that comprises everything a computer does in executing a traditional program. It is the program state that gets scheduled on a CPU; it is the "thing" that does the work. If a process comprises data, code, kernel state, and a set of CPU registers, then a thread is embodied in the contents of those registers—the program counter, the general registers, the stack pointer, etc., and the stack. A thread, viewed at an instant of time, is the state of the computation.

"Gee," you say, "That sounds like a process!" It should. They are conceptually related. But a process is a heavyweight, kernel-level entity and includes such things as a virtual memory map, file descriptors, user ID, etc., and each process has its own

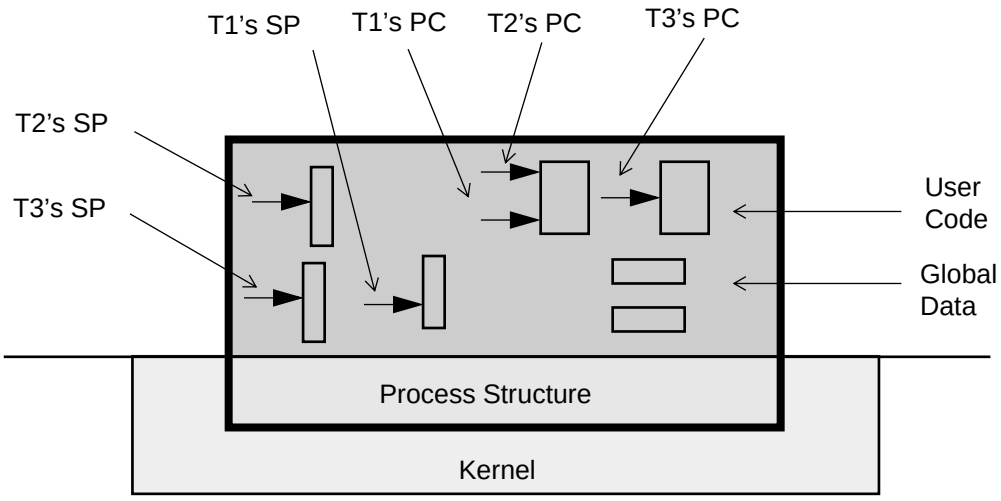


Figure 2–4 *Relationship between a Process and Threads*

collection of these. The only way for your program to access data in the process structure, to query or change its state, is via a system call.

All parts of the process structure are in kernel space (Figure 2–4). A user program cannot touch any of that data directly. By contrast, all of the user code (functions, procedures, etc.), along with the data, is in user space and can be accessed directly.

A thread is a lightweight entity, comprising the registers, stack, and some other data. The rest of the process structure is shared by all threads: the address space, file descriptors, etc. Much (and sometimes all) of the thread structure is in user space, allowing for very fast access.

The actual code (functions, routines, signal handlers, etc.) is global, and it can be executed on any thread. In Figure 2–4 we show three threads (T1, T2, and T3), along with their stacks, stack pointers (SP), and program counters (PC). T1 and T2 are executing the same function. This is a normal situation, just as two different people can read the same road sign at the same time.

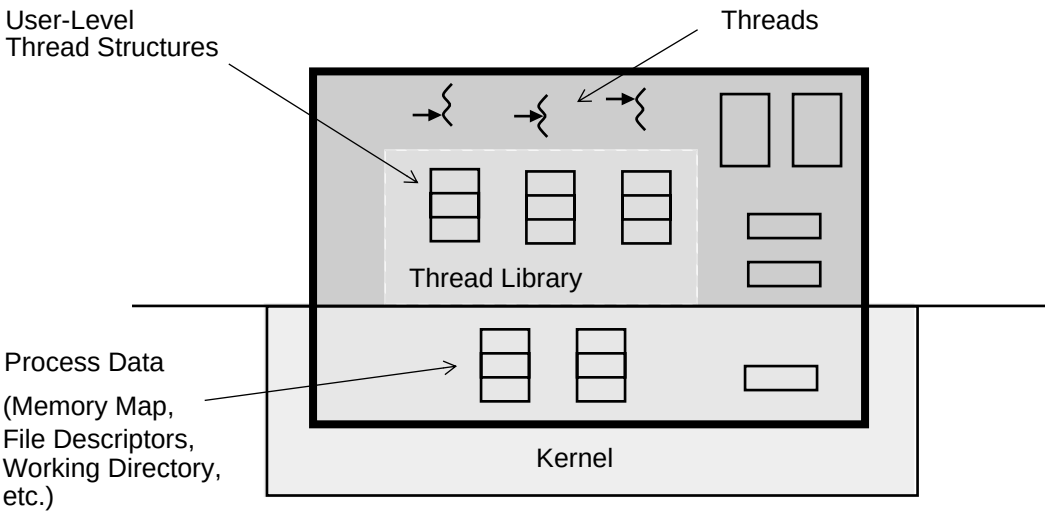


Figure 2–5 *Process Structure and Thread Structures*

All threads in a process share the state of that process (Figure 2–5[2]). They reside in exactly the same memory space, see the same functions, and see the same data. When one thread alters a process variable (say, the working directory), all the others will see the change when they next access it. If one thread opens a file to read it, all the other threads can also read from it.

Let's consider a human analogy: a bank. A bank with one person working in it (traditional process) has lots of "bank stuff," such as desks and chairs, a vault, and teller stations (process tables and variables). There are lots of services that a bank provides: checking accounts, loans, savings accounts, etc. (the functions). With one person to do all the work, that person would have to know how to do everything, and could do so, but it might take a bit of extra time to switch among the various tasks. With two or more people (threads), they would share all the same "bank stuff," but they could specialize in their different functions. And if they all came in and worked on the same day, lots of customers could get serviced quickly.

[2]From here on, we will use the squiggle shown in the figure to represent the entire thread—stack, stack pointer, program counter, thread structure, etc.

To change the number of banks in town would be a big effort (creating new processes), but to hire one new employee (creating a new thread) would be very simple. Everything that happened inside the bank, including interactions among the employees there, would be fairly simple (user space operations among threads), whereas anything that involved the bank down the road would be much more involved (kernel space operations between processes).

When you write a multithreaded program, 99% of your programming is identical to what it was before—you spend your efforts in getting the program to do its real work. The other 1% is spent in creating threads, arranging for different threads to coordinate their activities, dealing with thread-specific data, etc. Perhaps 0.1% of your code consists of calls to thread functions.

Kernel Interaction

We've now covered the basic concept of threads at the user level. As noted, the concepts and most of the implementational aspects are valid for all thread models. What's missing is the definition of the relationship between threads and the operating systems. How do system calls work? How are threads scheduled on CPUs?

It is at this level that the various implementations differ significantly. The operating systems provide different system calls, and even identical system calls can differ widely in efficiency and robustness. The kernels are constructed differently and provide different resources and services.

Keep in mind as we go through this implementation aspect that 99% of your threads programming will be done above this level, and the major distinctions will be in the area of efficiency.

Concurrency vs. Parallelism

Concurrency means that two or more threads (or traditional processes) can be in the middle of executing code at the same time; it could be the same code or it could be different code (see Figure 2–6). The threads may or may not actually be executing at the same time, but rather, in the middle of it (i.e., one started executing, it was interrupted, and the other one started). Every

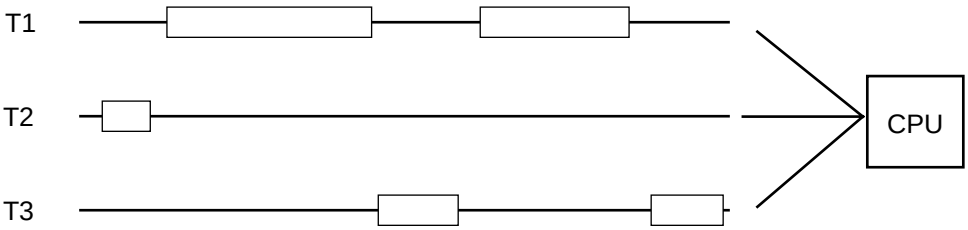


Figure 2–6 *Three Threads Running Concurrently on One CPU*

multitasking operating system has always had numerous concurrent processes, even though only one could be on the CPU at any given time.

Parallelism means that two or more threads actually run at the same time on different CPUs (see Figure 2–7). On a multiprocessor machine, many different threads can run in parallel. They are, of course, also running concurrently.

The vast majority of timing and synchronization issues in multithreading (MT) are those of concurrency, not parallelism. Indeed, the threads model was designed to avoid your ever having to be concerned with the details of parallelism. Running an MT program on a uniprocessor (UP) does not simplify your programming problems at all. Running on a multiprocessor (MP) doesn't complicate them. This is a good thing.

Let us repeat this point. If your program is written correctly on a uniprocessor, it will run correctly on a multiprocessor. The probability of running into a race condition is the same on both a UP and an MP. If it deadlocks on one, it will deadlock on the other. (There are lots of weird little exceptions to the probability part, but you'd have to try hard to make them appear.) There is a small set of bugs, however, which may cause a program to run as (naively) expected on a UP, and show its problems only on an MP (see *Bus Architectures* on page 346).

System Calls

A system call is basically a function that ends up trapping to routines in the kernel. These routines may do things as simple as looking up the user ID for the owner of the current process, or as

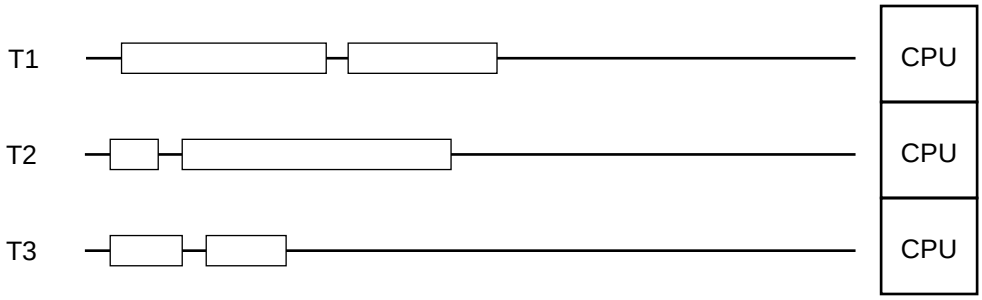


Figure 2–7 *Three Threads Running in Parallel on Three CPUs*

complex as redefining the system's scheduling algorithm. For multithreaded programs, there is a serious issue surrounding how many threads can make system calls concurrently. For some operating systems, the answer is "one"; for others, it's "many." The most important point is that system calls run exactly as they did before, so all your old programs continue to run as they did before, with (almost) no degradation.

Signals

Signals are the UNIX kernel's way of interrupting a running process and letting it know that something of interest has happened. (NT has something similar but doesn't expose it in the Win32 interface.) It could be that a timer has expired, or that some I/O has completed, or that some other process wants to communicate something.

Happily, Java does not use UNIX signals, so we may conveniently ignore them entirely! The role that signals play in UNIX programs is handled in Java either by having a thread respond to a synchronous request or by the use of exceptions.

Synchronization

Synchronization is the method of ensuring that multiple threads coordinate their activities so that one thread doesn't accidentally change data that another thread is working on. This is done by

providing function calls that can limit the number of threads that can access some data concurrently.

In the simplest case (a *mutual exclusion lock*—a *mutex*), only one thread at a time can execute a given piece of code. This code presumably alters some global data or performs reads or writes to a device. For example, thread T1 obtains a lock and starts to work on some global data. Thread T2 must now wait (typically, it goes to sleep) until thread T1 is done before T2 can execute the same code. By using the same lock around all code that changes the data, we can ensure that the data remains consistent.

Scheduling

Scheduling is the act of placing threads onto CPUs so that they can execute, and of taking them off those CPUs so that others can run instead. In practice, scheduling is not generally an issue because "it all works" just about the way you'd expect.

The Value of Using Threads

There is really only one reason for writing MT programs—to get better programs more quickly. If you're an Independent Software Vendor (ISV), you sell more software. If you're developing software for your own in-house use, you simply have better programs to use. The reason you can write better programs is that MT gives your programs and your programmers a number of significant advantages over nonthreaded programs and programming paradigms.

A point to keep in mind here is that you are not replacing simple, nonthreaded programs with fancy, complex, threaded programs. You are using threads only when you need them to replace complex or slow nonthreaded programs. Threads are just one more way to make your programming tasks easier.

The main benefits of writing multithreaded programs are:

- Performance gains from multiprocessing hardware (parallelism)
- Increased application throughput
- Increased application responsiveness

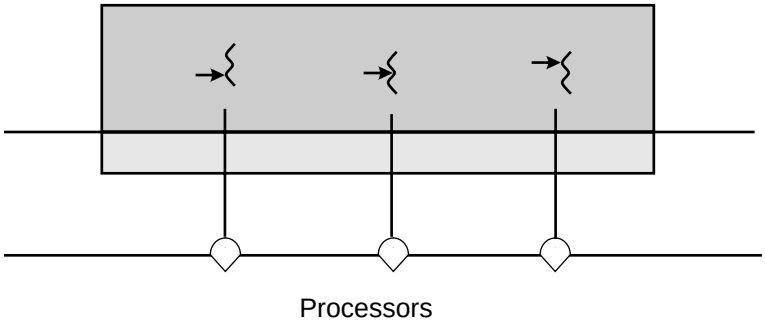


Figure 2–8 *Different Threads Running on Different Processors*

- Replacing process-to-process communications
- Efficient use of system resources
- One binary that runs well on both uniprocessors and multiprocessors
- The ability to create well-structured programs

The following sections elaborate further on these benefits.

Parallelism

Computers with more than one processor offer the potential for enormous application speedups (Figure 2–8). MT is an efficient way for application developers to exploit the parallelism of the hardware. Different threads can run on different processors simultaneously with no special input from the user and no effort on the part of the programmer.

A good example is a process that does matrix multiplication. A thread can be created for each available processor, allowing the program to use the entire machine. The threads can then compute distinct elements of the resulting matrix by performing the appropriate vector multiplication.

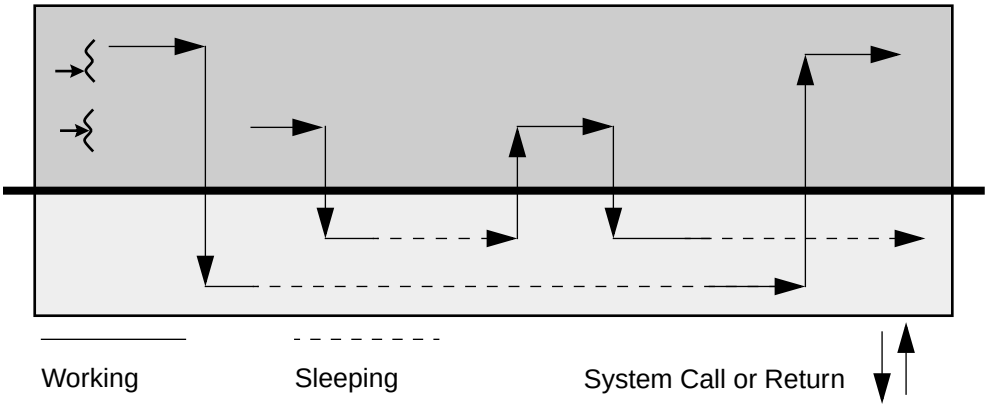


Figure 2–9 *Two Threads Making Overlapping System Calls*

Throughput

When a traditional, single-threaded program requests a service from the operating system, it must wait for that service to complete, often leaving the CPU idle. Even on a uniprocessor, multithreading allows a process to overlap computation with one or more blocking system calls (Figure 2–9). Threads provide this overlap even though each request is coded in the usual synchronous style. The thread making the request must wait, but another thread in the process can continue. Thus, a process can have numerous blocking requests outstanding, giving you the beneficial effects of doing asynchronous I/O while still writing code in the simpler synchronous fashion.

Responsiveness

Blocking one part of a process need not block the entire process. Single-threaded applications that do something lengthy when a button is pressed typically display a “please wait” cursor and freeze while the operation is in progress. If such applications were multithreaded, long operations could be done by independent threads, allowing the application to remain active and making the application more responsive to the user. In

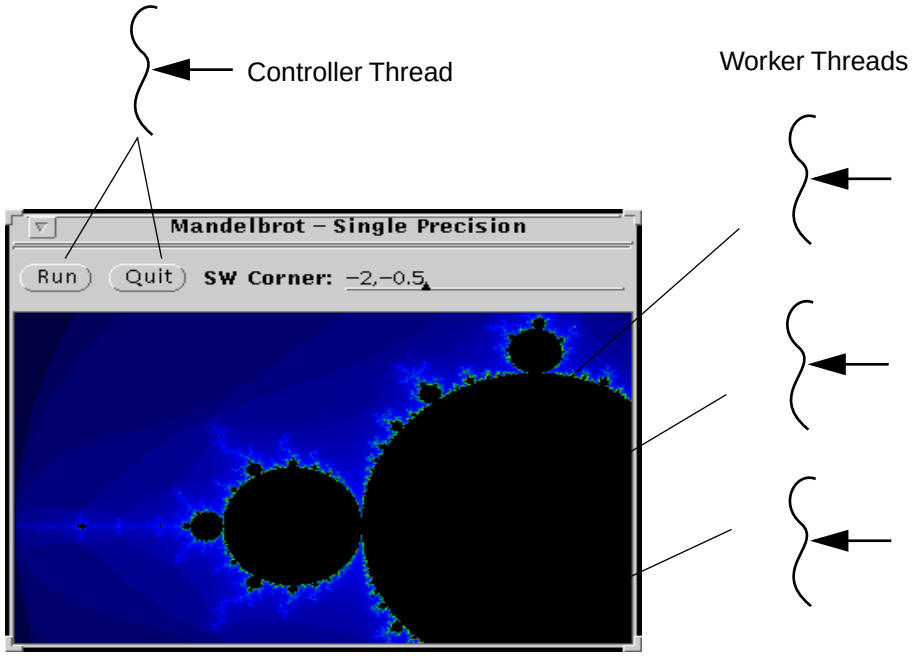


Figure 2–10 *Threads Overlapping Calculation and I/O*

Figure 2–10, one thread is waiting for I/O from the buttons, and several threads are working on the calculations.

Communications

An application that uses multiple processes to accomplish its tasks can be replaced by an application that uses multiple threads to accomplish those same tasks. Where the old program communicated among its processes through traditional interprocess communications facilities (e.g., pipes or sockets), the threaded application can communicate via the inherently shared memory of the process. The threads in the MT process can maintain separate connections while sharing data in the same address space. A classic example is a server program, which can maintain one thread for each client connection, such as in Figure 2–11. This program

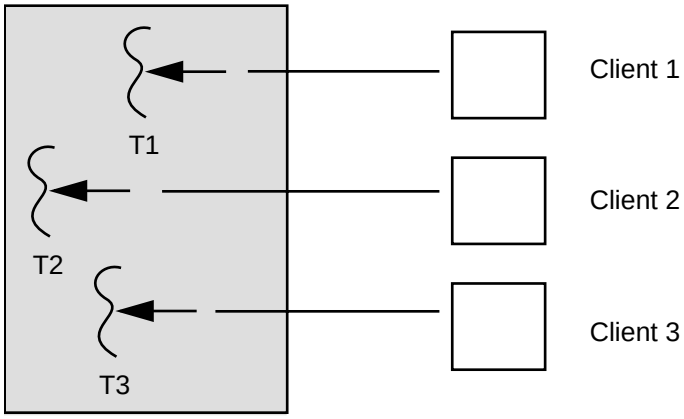


Figure 2–11 *Different Clients Being Handled by Different Threads*

provides excellent performance, simpler programming, and effortless scalability.

System Resources

Programs that use two or more processes to access common data through shared memory are effectively applying more than one thread of control. However, each such process must maintain a complete process structure, including a full virtual memory space and kernel state. The cost of creating and maintaining this large amount of state makes each process much more expensive, in both time and space, than a thread. In addition, the inherent separation between processes may require a major effort by the programmer to communicate among the different processes or to synchronize their actions. By using threads for this communication instead of processes, the program will be easier to debug and can run much faster.

An application can create hundreds or even thousands of threads, one for each synchronous task, with only minor impact on system resources. Threads use a fraction of the system resources needed by processes.

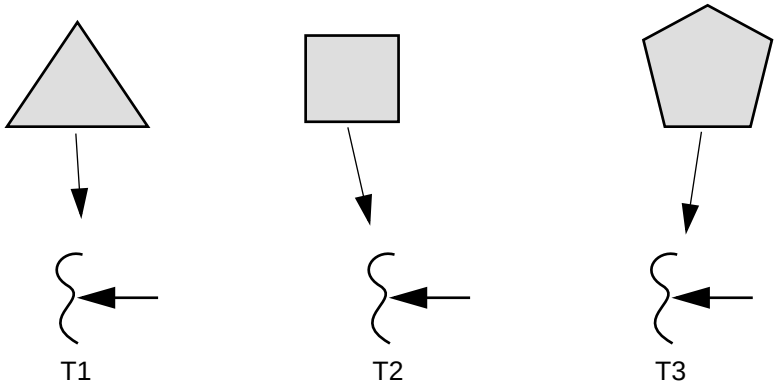


Figure 2–12 *Distributed Objects Running on Distinct Threads*

Distributed Objects

With the first releases of standardized distributed objects and object request brokers, your ability to make use of these will become increasingly important. Distributed objects are inherently multithreaded. Each time you request an object to perform some action, it executes that action in a separate thread (Figure 2–12). Object servers are an absolutely fundamental element in distributed object paradigm, and those servers are inherently multithreaded.

Although you can make a great deal of use of distributed objects without doing any MT programming, knowing what they are doing and being able to create objects that are threaded will increase the usefulness of the objects you do write.

Same Binary for Uniprocessors and Multiprocessors

In most older parallel processing schemes, it was necessary to tailor a program for the individual hardware configuration. With threads, this customization isn't required because the MT paradigm works well irrespective of the number of CPUs. A

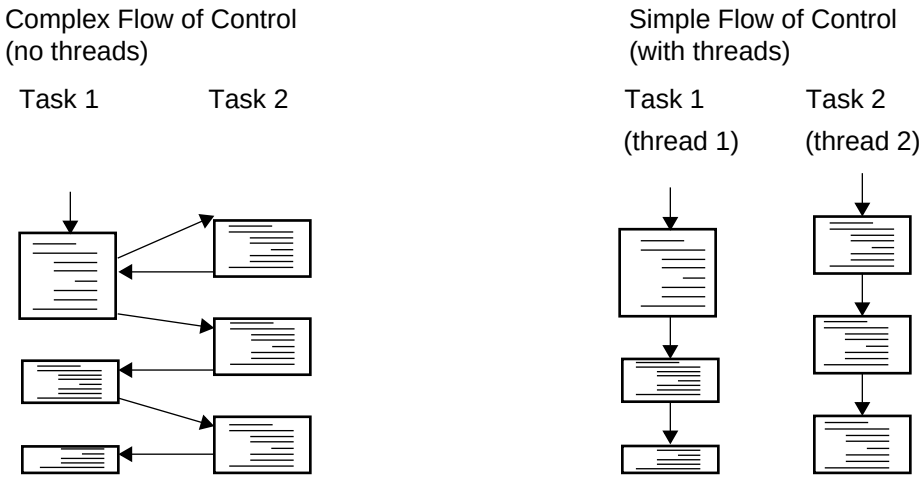


Figure 2–13 *Simplified Flow of Control in Complex Applications*

program can be compiled once, and it will run acceptably on a uniprocessor, whereas on a multiprocessor it will just run faster.

Program Structure

Many programs are structured more efficiently with threads because they are inherently concurrent. A traditional program that tries to do many different tasks is crowded with lots of complicated code to coordinate these tasks. A threaded program can do the same tasks with much less, far simpler code, as in Figure 2–13. Multithreaded programs can be more adaptive to variations in user demands than single-threaded programs can.

This is quite some set of claims, and a bit of healthy skepticism is called for. Sure, it sounds good when we say it, but what about when you try to use it? We cannot guarantee that you will experience the same wonderful results, but we can point out a number of cases where other folks have found MT programming to be of great advantage (see *Performance* on page 28).

What Kinds of Programs to Thread

There is a spectrum of programs that one might wish to thread. On one end, there are those that are inherently "MT-ish"—you look at the work to be done, and you think of it as several independent tasks. In the middle, there are programs where the division of work isn't obvious, but possible. On the far other end, there are those that cannot reasonably be threaded at all.

Inherently MT Programs

Inherently MT programs are those that are easily expressed as numerous threads doing numerous things. Such programs are easier to write using threads, because they are doing different things concurrently anyway. They are generally simpler to write and understand when threaded, easier to maintain, and more robust. The fact that they may run faster is a mere pleasant side effect. For these programs, the general rule is that the more complex the application, the greater the value of threading.

Typical programs that are inherently MT include:

Independent tasks A debugger needs to run and monitor a program, keep its GUI active, and display an interactive data inspector, dynamic call grapher, and performance monitor—all in the same address space, all at the same time.

Servers A server needs to handle numerous overlapping requests simultaneously. NFS®, NIS, DBMSs, stock quotation servers, etc., all receive large numbers of requests that require the server to do some I/O, then process the results and return answers. Completing one request at a time would be very slow.

Repetitive tasks A simulator needs to simulate the interactions of numerous different elements that operate simultaneously. CAD, structural analysis, weather prediction, etc., all model tiny pieces first, then combine the results to produce an overall picture.

Not Obviously MT Programs

Not obviously MT programs are those not inherently MT but for which threading is reasonable. Here you impose threads upon an algorithm that does not have an obvious decomposition, in order to achieve a speedup on an MP machine. Such a program is somewhat harder to write, a bit more difficult to maintain, etc., than its nonthreaded counterpart, but it runs faster. Because of these drawbacks, the (portions of) programs chosen are generally quite simple.

Typical programs in this class include:

Numerical programs Many numerical programs (e.g., matrix operations) are made up of huge numbers of tiny, identical, and independent operations. They are most easily (well, most commonly) expressed as loops inside loops. Slicing these loops into appropriate-sized chunks for threads is slightly more complicated, and there would be no reason to do so, save for the order-*N* speedup that can be obtained on an *N*-way SMP machine.

Old code These are the "slightly modified existing systems." This is existing code that makes you think to yourself: "If I just change a few bits here and there, add a few locks, then I can thread it and double my performance."

It's true, it is possible to do this, and there are lots of examples. However, this is a tough situation because you will constantly be finding new interactions that you didn't realize existed before. In such cases (which, due to the nature of the modern software industry, are far too common), you should concentrate on the bottlenecks and look for absolutely minimal submodules that can be rewritten. It's *always* best to take the time to do it right: re-architect and write the program correctly from the beginning.

Automatic Threading

In a subset of cases, it is possible for a compiler to do the threading for you. If you have a program written in such a way that a compiler can analyze its structure, analyze the interdependencies of the data, and determine that parts of your program can

run simultaneously without data conflicts, then the compiler can build the threads.

With current technology, the capabilities above are limited largely to Fortran programs that have time-consuming loops in which the individual computations in those loops are obviously independent. The primary reason for this limitation is that Fortran programs tend to have very simple structuring, both for code and data, making the analysis viable. Languages like C, which have constructs such as pointers, make the analysis enormously more difficult. There are MP compilers for C, but far fewer programs can take advantage of such compiling techniques.

With the different Fortran MP compilers,[3] it is possible to take vanilla Fortran 77 or 90 code, make no changes to it whatsoever, and have the compiler turn out threaded code. In some cases it works very well; in others, not. The cost of trying it out is very small, of course. A number of Ada compilers will map Ada tasks directly on top of threads, allowing existing Ada programs to take advantage of parallel machines with no changes to the code.

Programs Not to Thread

Then there is a large set of programs that it doesn't make any sense to thread. Probably 99% of all programs either do not lend themselves easily to threading or run just fine the way they are. Some programs simply require separate processes in which to run. Perhaps they need to execute one task as root but need to avoid having any other code running as root. Perhaps the program needs to be able to control its global environment closely, changing working directories, etc. Most programs run quite fast enough as they are and don't have any inherent multitasking, such as an icon editor or a calculator application.

In all truth, multithreaded programming is more difficult than regular programming. There are a host of new problems that must be dealt with, many of which are difficult. Threads are of value primarily when the task at hand is complex.

---

[3]Digital's Fortran compiler, Sun® Fortran MP, Kuck and Associates' Fortran compiler, EPC's Fortran compiler, SGI's MP Fortran compiler, probably more.

What About Shared Memory?

At this time, you may be asking yourself, "What can threads do that can't be done by processes sharing memory?" The first answer is, "nothing." Most anything that you can do with threads, you can do with processes sharing memory. Indeed, a number of vendors implement a significant portion of their threads library in roughly this fashion. There are a few details, such as managing shared file descriptors, which are not supported on all systems. Nonetheless, the additional expense and complication of using multiple processes restricts the usefulness of this method. Java is defined in such a way that sharing memory between processes is not an option, so we will skip over this technique, which is sometimes interesting to C/C++ programmers.

Threads Standards

There are three different definitions for native thread libraries competing for attention today: Win32, OS/2, and POSIX. The first two are proprietary and limited to their individual platforms (Win32 threads run only under NT and Win95, OS/2 threads only on OS/2). The POSIX specification (IEEE 1003.1c, a.k.a. *Pthreads*) is intended for all computing platforms, and implementations are available or in development for almost all major UNIX systems (including Linux), along with VMS and AS/400—not to mention a freeware library for Win32.

By contrast, Java threads are implemented in the JVM, which in turn is built on top of the native threads library for the specific platform.[4] Java does not expose the native threads' APIs, only its own, very small set of functions. This allows Java threads to be easier to use than the native libraries and more portable, but there are still some significant issues in making programs run uniformly across all platforms.

[4]Actually, the JVM is allowed to implement threads any way it feels like. Indeed, the first implementations of Java used *green threads*, which were not native. Today, most JVMs are built on native threads.

POSIX Threads

The POSIX standard defines the API and behavior that all Pthreads libraries must meet. It is part of the extended portion of POSIX, so it is not a requirement for meeting XPG4, but it is required for X/Open UNIX 98, and all major UNIX vendors have implemented this standard. In addition, UNIX98 includes a small set of extensions to Pthreads.

Win32 and OS/2 Threads

Both the NT and OS/2 implementations contain some fairly radical differences from the POSIX standard—to the degree that even porting from one or the other to POSIX will prove moderately challenging. Microsoft has not announced any plans to adopt POSIX. There are freeware POSIX libraries for Win32, and OS/2 also has an optional POSIX library.

DCE Threads

Before POSIX completed work on the standard, it produced a number of drafts that it published for comment. Draft 4 was used as the basis for the threads library in DCE. It is similar to the final spec, but it does contain a number of significant differences. Presumably, no one is writing any new threaded DCE code.

Solaris Threads

Also known as *UI threads*, this is the library that SunSoft used in developing Solaris 2 before the POSIX committee completed its work. It will be available on Solaris 2 for the foreseeable future, although we expect most applications writers will opt for Pthreads. The vast majority of the two libraries are virtually identical.

Performance

Even after reading all these wonderful things about threads, there's always someone who insists on asking that ever-so-bothersome question: "Does it work?" For an answer, we turn to

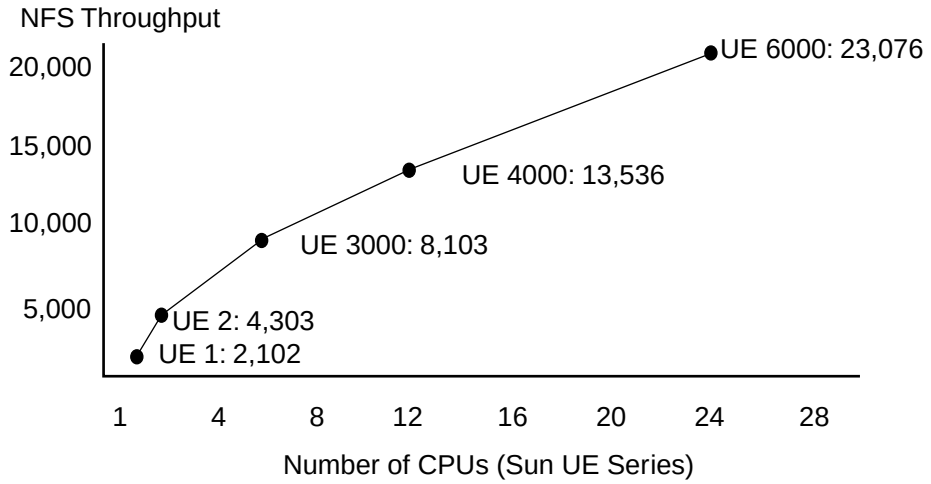


Figure 2–14 *NFS Performance on MP Machines (SPEC '96)*

some real, live shipping programs. Some of these are described in greater detail in the MT "Case Studies" (see *Threads Newsgroup* on page 410).

Operating Systems

OSs are large, complex, yet still highly efficient and robust programs. The various OSs have been in daily use by millions of users over the past couple of years and have endured the stress put on them by hundreds of thousands of programmers who are not known for their generosity toward operating system quirks. Mach, Windows NT, Windows 95, Solaris, IRIX, AIX, OS/2, and Digital UNIX are all threaded, and many of the other UNIX vendors are also moving toward a threaded kernel.

NFS

Under most UNIX systems, both the NFS client and server are threaded (Figure 2–14). There aren't any standardized benchmarks for the client side, so you'll have to take our word for it that it's faster. On the server side, however, there is the LADDIS

benchmark from SPEC. A great deal of time has been spent optimizing NFS for multiple CPUs, quite successfully.

SPECfp 95

The rule for the SPECfp benchmark is that a compiler is allowed to do pretty much anything it wants to, as long as that compiler is available to customers and nobody changes the source code at all. The various Fortran 77/90 MP compilers automatically multithread a program with no user intervention, so they are legal. You give the compiler the code, completely unchanged, and it looks to see if there is any possibility of threading it. It is possible to thread 6 of the 14 SPECfp programs automatically. The results are *very* impressive (Table 2–1).

Table 2–1 *SPECfp95 Results for Alpha 4100 5/466 (SPEC '97)*

Number of CPUs	Tomcatv	Swim	Su2cor	Hydro2d	Mgrid	Turb3d
1	23.8	25.4	10.1	10.0	17.5	19.1
2	33.1	46.2	18.0	15.4	24.5	33.4
4	40.3	83.8	30.3	21.3	34.6	54.9

SPECint_rate95

SPECfp 95 is a reasonable set of benchmarks for single-CPU machines, but it does not give a good picture of the overall performance potential of multiprocessor machines (Figure 2–15). The SPECrate is intended to demonstrate this potential by allowing the vendor to run as many copies of the program as desired (e.g., in one test with 30 CPUs, Sun ran 37 copies of each program). This benchmark does not use the MP compiler.

Java Benchmarks

There are currently no Java benchmarks of interest to parallel processing.

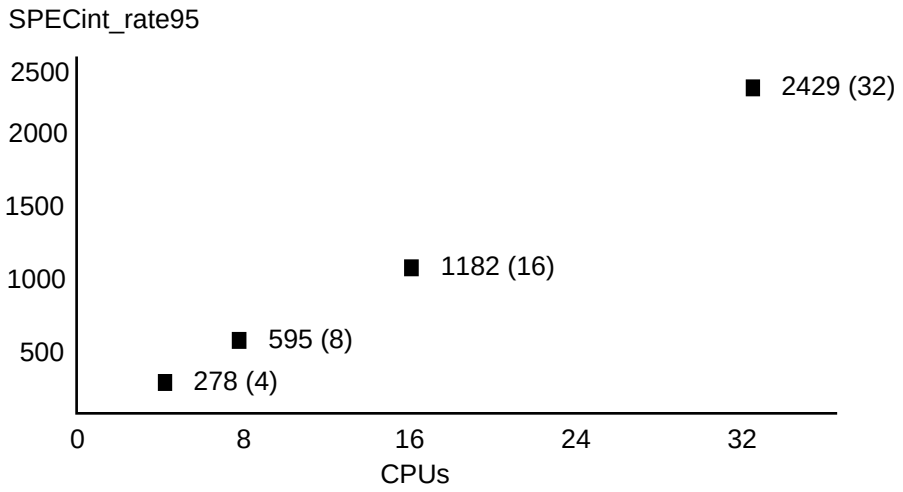


Figure 2–15 *Running SPECrate_fp95 on an SGI Origin/200, 2000 (SPEC '96)*

Summary

Threads allow both concurrent execution in a single address space and parallel execution on multiple-processor machines, and they also make many complex programs easier to write. Most programs are simple and fast enough that they don't need threads, but for those programs that do need them, threads are wonderful.