CHAPTER 4

THREADS

Learning Objectives

After studying this chapter, you should be able to:

- Understand the distinction between process and thread.
- Describe the basic design issues for threads.
- Explain the difference between user-level threads and kernel-level threads.
- Describe the thread management facility in Windows.
- Describe the thread management facility in Solaris.
- Describe the thread management facility in Linux.

This chapter examines some more advanced concepts related to process management, which are found in a number of contemporary operating systems. We show that the concept of process is more complex and subtle than presented so far and in fact embodies two separate and potentially independent concepts: one relating to resource ownership, and another relating to execution. This distinction has led to the development, in many operating systems, of a construct known as the **thread**.

4.1 PROCESSES AND THREADS

The discussion so far has presented the concept of a process as embodying two characteristics:

1. **Resource ownership:** A process includes a virtual address space to hold the process image; recall from Chapter 3 that the process image is the collection of program, data, stack, and attributes defined in the process control block. From time to time, a process may be allocated control or ownership of resources, such as main memory, I/O channels, I/O devices, and files. The OS performs a protection function to prevent unwanted interference between processes with respect to resources.
2. **Scheduling/execution:** The execution of a process follows an execution path (trace) through one or more programs (e.g., Figure 1.5). This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority, and is the entity that is scheduled and dispatched by the OS.

Some thought should convince the reader that these two characteristics are independent and could be treated independently by the OS. This is done in a number of operating systems, particularly recently developed systems. To distinguish the two characteristics, the unit of dispatching is usually referred to as a thread or

lightweight process, while the unit of resource ownership is usually referred to as a **process** or **task**.[1]

Multithreading

Multithreading refers to the ability of an OS to support multiple, concurrent paths of execution within a single process. The traditional approach of a single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach. The two arrangements shown in the left half of Figure 4.1 are single-threaded approaches. MS-DOS is an example of an OS that supports a single-user process and a single thread. Other operating systems, such as some variants of UNIX, support multiple user processes, but only support one thread per process. The right half of Figure 4.1 depicts multithreaded approaches. A Java runtime environment is an example of a system of one process with multiple threads. Of interest in this section is the use of multiple processes, each of which supports multiple threads. This approach is taken in Windows, Solaris, and many

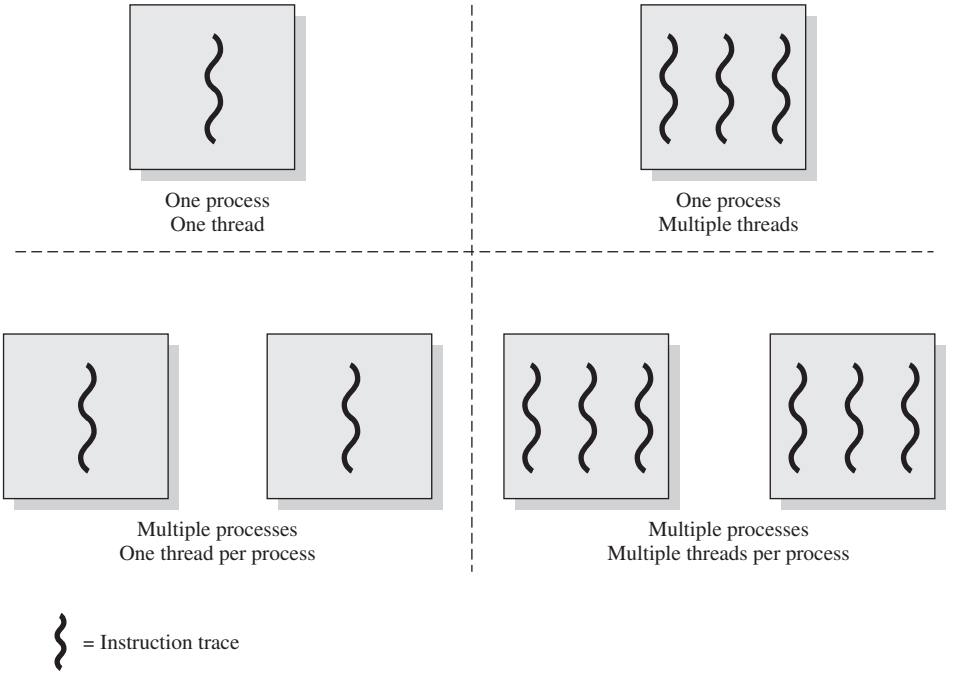


Figure 4.1 Threads and Processes

[1]Alas, even this degree of consistency is not maintained. In IBM's mainframe operating systems, the concepts of address space and task, respectively, correspond roughly to the concepts of process and thread that we describe in this section. Also, in the literature, the term *lightweight process* is used as either (1) equivalent to the term *thread*, (2) a particular type of thread known as a kernel-level thread, or (3) in the case of Solaris, an entity that maps user-level threads to kernel-level threads.

modern versions of UNIX, among others. In this section, we give a general description of multithreading; the details of the Windows, Solaris, and Linux approaches will be discussed later in this chapter.

In a multithreaded environment, a process is defined as the unit of resource allocation and a unit of protection. The following are associated with processes:

- A virtual address space that holds the process image
- Protected access to processors, other processes (for interprocess communication), files, and I/O resources (devices and channels)

Within a process, there may be one or more threads, each with the following:

- A thread execution state (Running, Ready, etc.)
- A saved thread context when not running; one way to view a thread is as an independent program counter operating within a process
- An execution stack
- Some per-thread static storage for local variables
- Access to the memory and resources of its process, shared with all other threads in that process

Figure 4.2 illustrates the distinction between threads and processes from the point of view of process management. In a single-threaded process model (i.e., there is no distinct concept of thread), the representation of a process includes its process control block and user address space, as well as user and kernel stacks to manage the call/return behavior of the execution of the process. While the process is running, it controls the processor registers. The contents of these registers are saved when the process is not running. In a multithreaded environment, there is still a single process

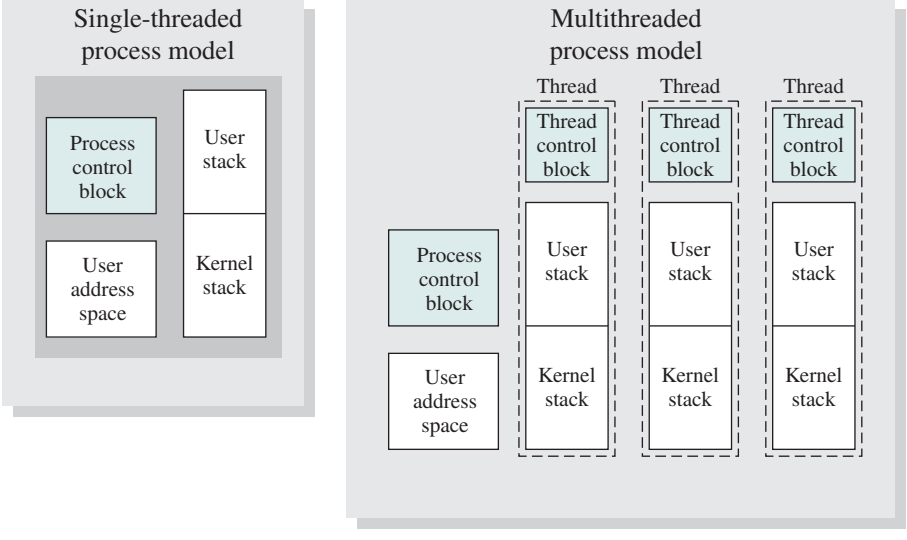


Figure 4.2 Single-Threaded and Multithreaded Process Models

control block and user address space associated with the process, but now there are separate stacks for each thread, as well as a separate control block for each thread containing register values, priority, and other thread-related state information.

Thus, all of the threads of a process share the state and resources of that process. They reside in the same address space and have access to the same data. When one thread alters an item of data in memory, other threads see the results if and when they access that item. If one thread opens a file with read privileges, other threads in the same process can also read from that file.

The key benefits of threads derive from the performance implications:

1. It takes far less time to create a new thread in an existing process, than to create a brand-new process. Studies done by the Mach developers show that thread creation is ten times faster than process creation in UNIX [TEVA87].
2. It takes less time to terminate a thread than a process.
3. It takes less time to switch between two threads within the same process than to switch between processes.
4. Threads enhance efficiency in communication between different executing programs. In most operating systems, communication between independent processes requires the intervention of the kernel to provide protection and the mechanisms needed for communication. However, because threads within the same process share memory and files, they can communicate with each other without invoking the kernel.

Thus, if there is an application or function that should be implemented as a set of related units of execution, it is far more efficient to do so as a collection of threads, rather than a collection of separate processes.

An example of an application that could make use of threads is a file server. As each new file request comes in, a new thread can be spawned for the file management program. Because a server will handle many requests, many threads will be created and destroyed in a short period. If the server runs on a multiprocessor computer, then multiple threads within the same process can be executing simultaneously on different processors. Further, because processes or threads in a file server must share file data and therefore coordinate their actions, it is faster to use threads and shared memory than processes and message passing for this coordination.

The thread construct is also useful on a single processor to simplify the structure of a program that is logically doing several different functions.

[LETW88] gives four examples of the uses of threads in a single-user multiprocessing system:

1. **Foreground and background work:** For example, in a spreadsheet program, one thread could display menus and read user input, while another thread executes user commands and updates the spreadsheet. This arrangement often increases the perceived speed of the application by allowing the program to prompt for the next command before the previous command is complete.
2. **Asynchronous processing:** Asynchronous elements in the program can be implemented as threads. For example, as a protection against power failure, one can design a word processor to write its random access memory (RAM)

buffer to disk once every minute. A thread can be created whose sole job is periodic backup and that schedules itself directly with the OS; there is no need for fancy code in the main program to provide for time checks or to coordinate input and output.

3. **Speed of execution:** A multithreaded process can compute one batch of data while reading the next batch from a device. On a multiprocessor system, multiple threads from the same process may be able to execute simultaneously. Thus, even though one thread may be blocked for an I/O operation to read in a batch of data, another thread may be executing.
4. **Modular program structure:** Programs that involve a variety of activities or a variety of sources and destinations of input and output may be easier to design and implement using threads.

In an OS that supports threads, scheduling and dispatching is done on a thread basis; hence, most of the state information dealing with execution is maintained in thread-level data structures. There are, however, several actions that affect all of the threads in a process, and that the OS must manage at the process level. For example, suspension involves swapping the address space of one process out of main memory to make room for the address space of another process. Because all threads in a process share the same address space, all threads are suspended at the same time. Similarly, termination of a process terminates all threads within that process.

Thread Functionality

Like processes, threads have execution states and may synchronize with one another. We look at these two aspects of thread functionality in turn.

Thread States As with processes, the key states for a thread are Running, Ready, and Blocked. Generally, it does not make sense to associate suspend states with threads because such states are process-level concepts. In particular, if a process is swapped out, all of its threads are necessarily swapped out because they all share the address space of the process.

There are four basic thread operations associated with a change in thread state [ANDE04]:

1. **Spawn:** Typically, when a new process is spawned, a thread for that process is also spawned. Subsequently, a thread within a process may spawn another thread within the same process, providing an instruction pointer and arguments for the new thread. The new thread is provided with its own register context and stack space and placed on the Ready queue.
2. **Block:** When a thread needs to wait for an event, it will block (saving its user registers, program counter, and stack pointers). The processor may then turn to the execution of another ready thread in the same or a different process.
3. **Unblock:** When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.
4. **Finish:** When a thread completes, its register context and stacks are deallocated.

A significant issue is whether the blocking of a thread results in the blocking of the entire process. In other words, if one thread in a process is blocked, does this prevent the running of any other thread in the same process, even if that other thread is in a ready state? Clearly, some of the flexibility and power of threads is lost if the one blocked thread blocks an entire process.

We will return to this issue subsequently in our discussion of user-level versus kernel-level threads, but for now, let us consider the performance benefits of threads that do not block an entire process. Figure 4.3 (based on one in [KLEI96]) shows a program that performs two remote procedure calls (RPCs)[2] to two different hosts to obtain a combined result. In a single-threaded program, the results are obtained in sequence, so the program has to wait for a response from each server in turn. Rewriting the program to use a separate thread for each RPC results in a substantial speedup. Note if this program operates on a uniprocessor, the requests must be generated sequentially and the results processed in sequence; however, the program waits concurrently for the two replies.

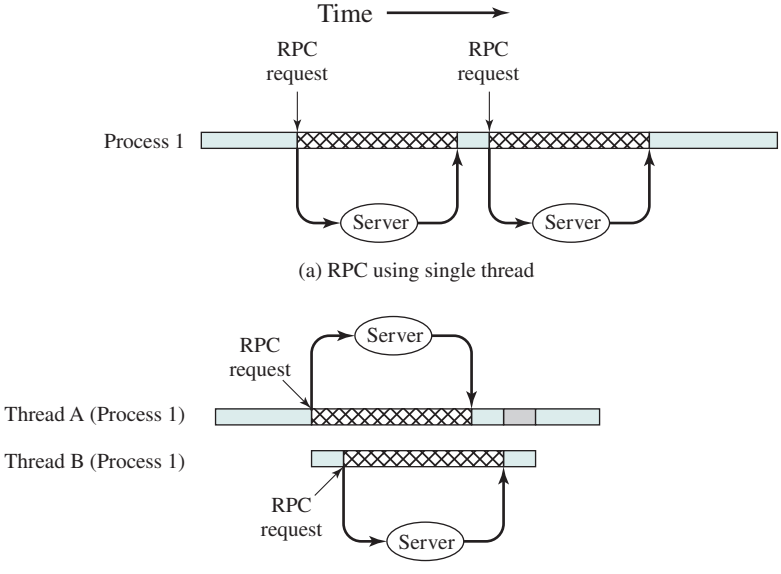


Figure 4.3 Remote Procedure Call (RPC) Using Threads

[2]An RPC is a technique by which two programs, which may execute on different machines, interact using procedure call/return syntax and semantics. Both the called and calling programs behave as if the partner program were running on the same machine. RPCs are often used for client/server applications and will be discussed in Chapter 16.

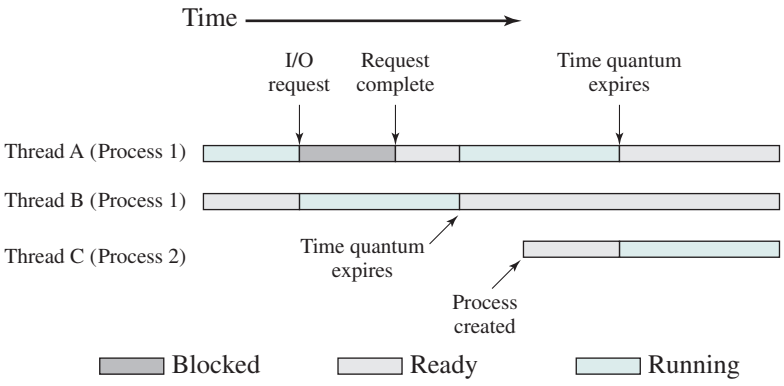


Figure 4.4 **Multithreading Example on a Uniprocessor**

On a uniprocessor, multiprogramming enables the interleaving of multiple threads within multiple processes. In the example of Figure 4.4, three threads in two processes are interleaved on the processor. Execution passes from one thread to another either when the currently running thread is blocked or when its time slice is exhausted.[3]

THREAD SYNCHRONIZATION All of the threads of a process share the same address space and other resources, such as open files. Any alteration of a resource by one thread affects the environment of the other threads in the same process. It is therefore necessary to synchronize the activities of the various threads so that they do not interfere with each other or corrupt data structures. For example, if two threads each try to add an element to a doubly linked list at the same time, one element may be lost or the list may end up malformed.

The issues raised and the techniques used in the synchronization of threads are, in general, the same as for the synchronization of processes. These issues and techniques will be the subject of Chapters 5 and 6.

4.2 TYPES OF THREADS

User-Level and Kernel-Level Threads

There are two broad categories of thread implementation: user-level threads (ULTs) and kernel-level threads (KLTs).[4] The latter are also referred to in the literature as *kernel-supported threads* or *lightweight processes*.

USER-LEVEL THREADS In a pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads.

[3]In this example, thread C begins to run after thread A exhausts its time quantum, even though thread B is also ready to run. The choice between B and C is a scheduling decision, a topic covered in Part Four.

[4]The acronyms ULT and KLT are not widely used, but are introduced for conciseness.

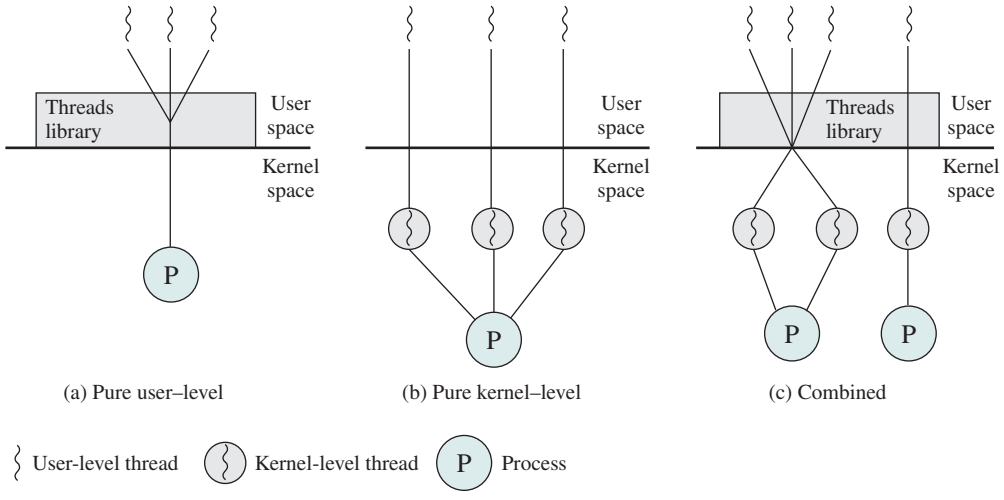


Figure 4.5 User-Level and Kernel-Level Threads

Figure 4.5a illustrates the pure ULT approach. Any application can be programmed to be multithreaded by using a threads library, which is a package of routines for ULT management. The threads library contains code for creating and destroying threads, for passing messages and data between threads, for scheduling thread execution, and for saving and restoring thread contexts.

By default, an application begins with a single thread and begins running in that thread. This application and its thread are allocated to a single process managed by the kernel. At any time that the application is running (the process is in the Running state), the application may spawn a new thread to run within the same process. Spawning is done by invoking the spawn utility in the threads library. Control is passed to that utility by a procedure call. The threads library creates a data structure for the new thread and then passes control to one of the threads within this process that is in the Ready state, using some scheduling algorithm. When control is passed to the library, the context of the current thread is saved, and when control is passed from the library to a thread, the context of that thread is restored. The context essentially consists of the contents of user registers, the program counter, and stack pointers.

All of the activity described in the preceding paragraph takes place in user space and within a single process. The kernel is unaware of this activity. The kernel continues to schedule the process as a unit and assigns a single execution state (Ready, Running, Blocked, etc.) to that process. The following examples should clarify the relationship between thread scheduling and process scheduling. Suppose process B is executing in its thread 2; the states of the process and two ULTs that are part of the process are shown in Figure 4.6a. Each of the following is a possible occurrence:

1. The application executing in thread 2 makes a system call that blocks B. For example, an I/O call is made. This causes control to transfer to the kernel. The kernel invokes the I/O action, places process B in the Blocked state, and

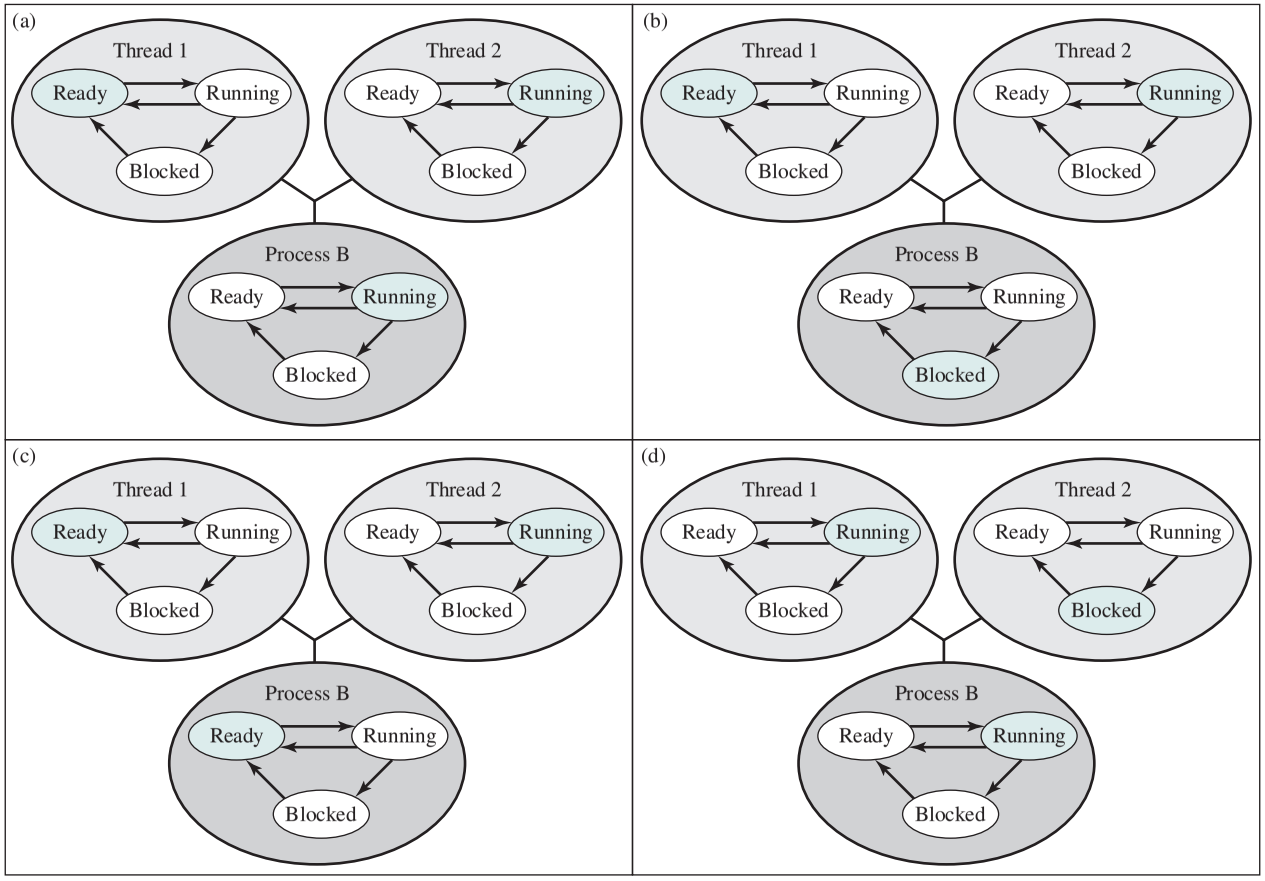


Figure 4.6 **Examples of the Relationships between User-Level Thread States and Process States**

switches to another process. Meanwhile, according to the data structure maintained by the threads library, thread 2 of process B is still in the Running state. It is important to note that thread 2 is not actually running in the sense of being executed on a processor; but it is perceived as being in the Running state by the threads library. The corresponding state diagrams are shown in Figure 4.6b.

2. A clock interrupt passes control to the kernel, and the kernel determines that the currently running process (B) has exhausted its time slice. The kernel places process B in the Ready state and switches to another process. Meanwhile, according to the data structure maintained by the threads library, thread 2 of process B is still in the Running state. The corresponding state diagrams are shown in Figure 4.6c.
3. Thread 2 has reached a point where it needs some action performed by thread 1 of process B. Thread 2 enters a Blocked state and thread 1 transitions from Ready to Running. The process itself remains in the Running state. The corresponding state diagrams are shown in Figure 4.6d.

Note that each of the three preceding items suggests an alternative event starting from diagram (a) of Figure 4.6. So each of the three other diagrams (b, c, d) shows a transition from the situation in (a). In cases 1 and 2 (Figures 4.6b and 4.6c), when the kernel switches control back to process B, execution resumes in thread 2. Also note that a process can be interrupted, either by exhausting its time slice or by being preempted by a higher-priority process, while it is executing code in the threads library. Thus, a process may be in the midst of a thread switch from one thread to another when interrupted. When that process is resumed, execution continues within the threads library, which completes the thread switch and transfers control to another thread within that process.

There are a number of advantages to the use of ULTs instead of KLTs, including the following:

1. Thread switching does not require kernel-mode privileges because all of the thread management data structures are within the user address space of a single process. Therefore, the process does not switch to the kernel mode to do thread management. This saves the overhead of two mode switches (user to kernel; kernel back to user).
2. Scheduling can be application specific. One application may benefit most from a simple round-robin scheduling algorithm, while another might benefit from a priority-based scheduling algorithm. The scheduling algorithm can be tailored to the application without disturbing the underlying OS scheduler.
3. ULTs can run on any OS. No changes are required to the underlying kernel to support ULTs. The threads library is a set of application-level functions shared by all applications.

There are two distinct disadvantages of ULTs compared to KLTs:

1. In a typical OS, many system calls are blocking. As a result, when a ULT executes a system call, not only is that thread blocked, but all of the threads within the process are blocked as well.

2. In a pure ULT strategy, a multithreaded application cannot take advantage of multiprocessing. A kernel assigns one process to only one processor at a time. Therefore, only a single thread within a process can execute at a time. In effect, we have application-level multiprogramming within a single process. While this multiprogramming can result in a significant speedup of the application, there are applications that would benefit from the ability to execute portions of code simultaneously.

There are ways to work around these two problems. For example, both problems can be overcome by writing an application as multiple processes rather than multiple threads. But this approach eliminates the main advantage of threads: Each switch becomes a process switch rather than a thread switch, resulting in much greater overhead.

Another way to overcome the problem of blocking threads is to use a technique referred to as **jacketing**. The purpose of jacketing is to convert a blocking system call into a nonblocking system call. For example, instead of directly calling a system I/O routine, a thread calls an application-level I/O jacket routine. Within this jacket routine is code that checks to determine if the I/O device is busy. If it is, the thread enters the Blocked state and passes control (through the threads library) to another thread. When this thread is later given control again, the jacket routine checks the I/O device again.

Kernel-Level Threads In a pure KLT facility, all of the work of thread management is done by the kernel. There is no thread management code in the application level, simply an application programming interface (API) to the kernel thread facility. Windows is an example of this approach.

Figure 4.5b depicts the pure KLT approach. The kernel maintains context information for the process as a whole and for individual threads within the process. Scheduling by the kernel is done on a thread basis. This approach overcomes the two principal drawbacks of the ULT approach. First, the kernel can simultaneously schedule multiple threads from the same process on multiple processors. Second, if one thread in a process is blocked, the kernel can schedule another thread of the same process. Another advantage of the KLT approach is that kernel routines themselves can be multithreaded.

The principal disadvantage of the KLT approach compared to the ULT approach is that the transfer of control from one thread to another within the same process requires a mode switch to the kernel. To illustrate the differences, Table 4.1 shows the results of measurements taken on a uniprocessor VAX computer running a UNIX-like OS. The two benchmarks are as follows: Null Fork, the time to create, schedule, execute, and complete a process/thread that invokes

Table 4.1 Thread and Process Operation Latencies (μs)

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840

the null procedure (i.e., the overhead of forking a process/thread); and Signal-Wait, the time for a process/thread to signal a waiting process/thread and then wait on a condition (i.e., the overhead of synchronizing two processes/threads together). We see there is an order of magnitude or more of difference between ULTs and KLTs, and similarly between KLTs and processes.

Thus, on the face of it, while there is a significant speedup by using KLT multithreading compared to single-threaded processes, there is an additional significant speedup by using ULTs. However, whether or not the additional speedup is realized depends on the nature of the applications involved. If most of the thread switches in an application require kernel-mode access, then a ULT-based scheme may not perform much better than a KLT-based scheme.

Combined Approaches Some operating systems provide a combined ULT/KLT facility (see Figure 4.5c). In a combined system, thread creation is done completely in user space, as is the bulk of the scheduling and synchronization of threads within an application. The multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs. The programmer may adjust the number of KLTs for a particular application and processor to achieve the best overall results.

In a combined approach, multiple threads within the same application can run in parallel on multiple processors, and a blocking system call need not block the entire process. If properly designed, this approach should combine the advantages of the pure ULT and KLT approaches while minimizing the disadvantages.

Solaris is a good example of an OS using this combined approach. The current Solaris version limits the ULT/KLT relationship to be one-to-one.

Other Arrangements

As we have said, the concepts of resource allocation and dispatching unit have traditionally been embodied in the single concept of the process—that is, as a 1 : 1 relationship between threads and processes. Recently, there has been much interest in providing for multiple threads within a single process, which is a many-to-one relationship. However, as Table 4.2 shows, the other two combinations have also been investigated, namely, a many-to-many relationship and a one-to-many relationship.

Table 4.2 Relationship between Threads and Processes

Threads: Processes	Description	Example Systems
1:1	Each thread of execution is a unique process with its own address space and resources.	Traditional UNIX implementations
M:1	A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.	Windows NT, Solaris, Linux, OS/2, OS/390, MACH
1:M	A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.	Ra (Clouds), Emerald
M:N	It combines attributes of M:1 and 1:M cases.	TRIX

MANY-TO-MANY RELATIONSHIP The idea of having a many-to-many relationship between threads and processes has been explored in the experimental operating system TRIX [PAZZ92, WARD80]. In TRIX, there are the concepts of domain and thread. A domain is a static entity, consisting of an address space and "ports" through which messages may be sent and received. A thread is a single execution path, with an execution stack, processor state, and scheduling information.

As with the multithreading approaches discussed so far, multiple threads may execute in a single domain, providing the efficiency gains discussed earlier. However, it is also possible for a single-user activity, or application, to be performed in multiple domains. In this case, a thread exists that can move from one domain to another.

The use of a single thread in multiple domains seems primarily motivated by a desire to provide structuring tools for the programmer. For example, consider a program that makes use of an I/O subprogram. In a multiprogramming environment that allows user-spawned processes, the main program could generate a new process to handle I/O, then continue to execute. However, if the future progress of the main program depends on the outcome of the I/O operation, then the main program will have to wait for the other I/O program to finish. There are several ways to implement this application:

1. The entire program can be implemented as a single process. This is a reasonable and straightforward solution. There are drawbacks related to memory management. The process as a whole may require considerable main memory to execute efficiently, whereas the I/O subprogram requires a relatively small address space to buffer I/O and to handle the relatively small amount of program code. Because the I/O program executes in the address space of the larger program, either the entire process must remain in main memory during the I/O operation, or the I/O operation is subject to swapping. This memory management effect would also exist if the main program and the I/O subprogram were implemented as two threads in the same address space.
2. The main program and I/O subprogram can be implemented as two separate processes. This incurs the overhead of creating the subordinate process. If the I/O activity is frequent, one must either leave the subordinate process alive, which consumes management resources, or frequently create and destroy the subprogram, which is inefficient.
3. Treat the main program and the I/O subprogram as a single activity that is to be implemented as a single thread. However, one address space (domain) could be created for the main program and one for the I/O subprogram. Thus, the thread can be moved between the two address spaces as execution proceeds. The OS can manage the two address spaces independently, and no process creation overhead is incurred. Furthermore, the address space used by the I/O subprogram could also be shared by other simple I/O programs.

The experiences of the TRIX developers indicate that the third option has merit, and may be the most effective solution for some applications.

ONE-TO-MANY RELATIONSHIP In the field of distributed operating systems (designed to control distributed computer systems), there has been interest in the

concept of a thread as primarily an entity that can move among address spaces.[5] A notable example of this research is the Clouds operating system, and especially its kernel, known as Ra [DASG92]. Another example is the Emerald system [STEE95].

A thread in Clouds is a unit of activity from the user's perspective. A process is a virtual address space with an associated process control block. Upon creation, a thread starts executing in a process by invoking an entry point to a program in that process. Threads may move from one address space to another, and actually span computer boundaries (i.e., move from one computer to another). As a thread moves, it must carry with it certain information, such as the controlling terminal, global parameters, and scheduling guidance (e.g., priority).

The Clouds approach provides an effective way of insulating both users and programmers from the details of the distributed environment. A user's activity may be represented as a single thread, and the movement of that thread among computers may be dictated by the OS for a variety of system-related reasons, such as the need to access a remote resource, and load balancing.

4.3 MULTICORE AND MULTITHREADING

The use of a multicore system to support a single application with multiple threads (such as might occur on a workstation, a video game console, or a personal computer running a processor-intense application) raises issues of performance and application design. In this section, we first look at some of the performance implications of a multithreaded application on a multicore system, then describe a specific example of an application designed to exploit multicore capabilities.

Performance of Software on Multicore

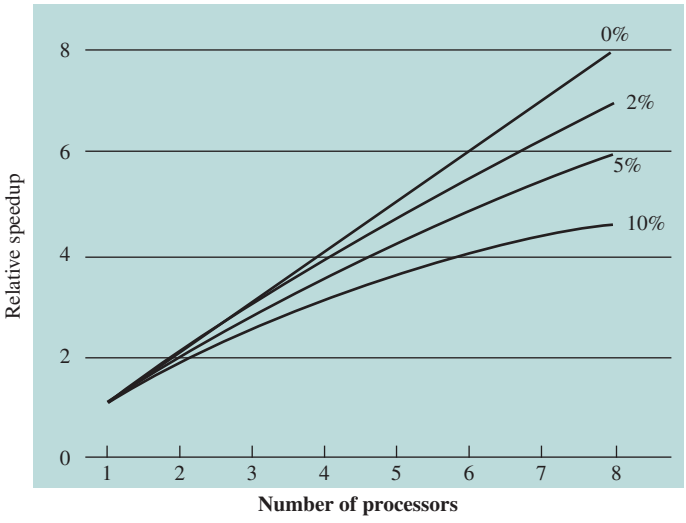
The potential performance benefits of a multicore organization depend on the ability to effectively exploit the parallel resources available to the application. Let us focus first on a single application running on a multicore system. Amdahl's law (see Appendix E) states that:

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{1}{(1 - f) + \frac{f}{N}}$$

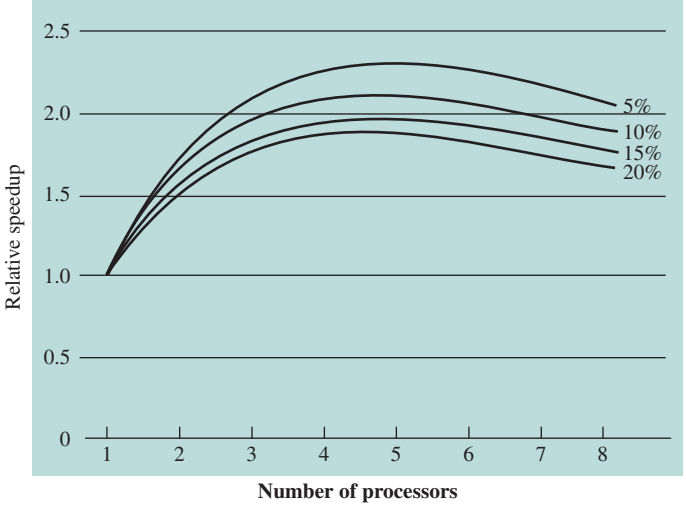
The law assumes a program in which a fraction $(1 - f)$ of the execution time involves code that is inherently serial, and a fraction f that involves code that is infinitely parallelizable with no scheduling overhead.

This law appears to make the prospect of a multicore organization attractive. But as Figure 4.7a shows, even a small amount of serial code has a noticeable impact. If only 10% of the code is inherently serial ($f = 0.9$), running the program on a multicore system with eight processors yields a performance gain of a factor of only 4.7. In

[5]The movement of processes or threads among address spaces, or thread migration, on different machines has become a hot topic in recent years. Chapter 18 will explore this topic.



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions



(b) Speedup with overheads

Figure 4.7 Performance Effect of Multiple Cores

addition, software typically incurs overhead as a result of communication and distribution of work to multiple processors and cache coherence overhead. This results in a curve where performance peaks and then begins to degrade because of the increased burden of the overhead of using multiple processors. Figure 4.7b, from [MCDO07], is a representative example.

However, software engineers have been addressing this problem, and there are numerous applications in which it is possible to effectively exploit a multicore

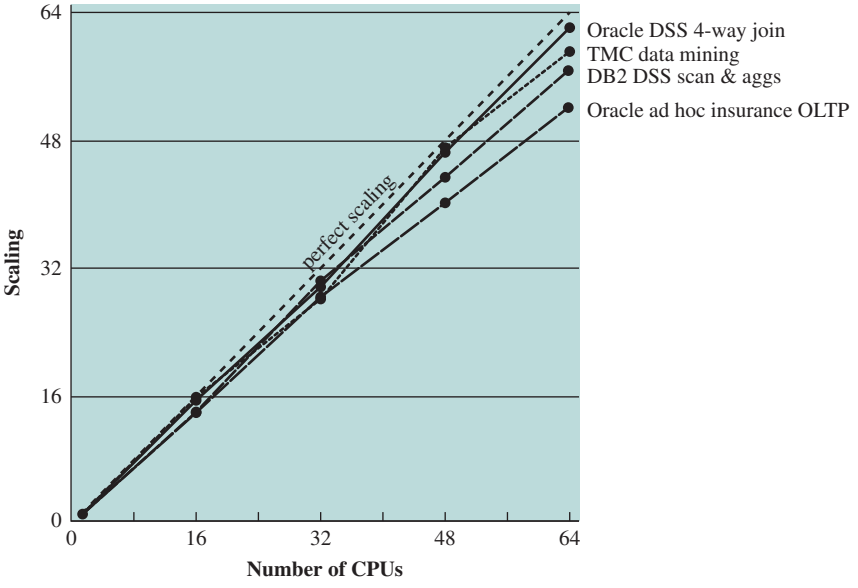


Figure 4.8 Scaling of Database Workloads on Multiprocessor Hardware

system. [MCDO07] reports on a set of database applications, in which great attention was paid to reducing the serial fraction within hardware architectures, operating systems, middleware, and the database application software. Figure 4.8 shows the result. As this example shows, database management systems and database applications are one area in which multicore systems can be used effectively. Many kinds of servers can also effectively use the parallel multicore organization, because servers typically handle numerous relatively independent transactions in parallel.

In addition to general-purpose server software, a number of classes of applications benefit directly from the ability to scale throughput with the number of cores. [MCDO06] lists the following examples:

- **Multithreaded native applications:** Multithreaded applications are characterized by having a small number of highly threaded processes. Examples of threaded applications include Lotus Domino or Siebel CRM (Customer Relationship Manager).
- **Multiprocess applications:** Multiprocess applications are characterized by the presence of many single-threaded processes. Examples of multiprocess applications include the Oracle database, SAP, and PeopleSoft.
- **Java applications:** Java applications embrace threading in a fundamental way. Not only does the Java language greatly facilitate multithreaded applications, but the Java Virtual Machine is a multithreaded process that provides scheduling and memory management for Java applications. Java applications that can benefit directly from multicore resources include application servers such as Oracle's Java Application Server, BEA's Weblogic, IBM's Websphere, and the open-source Tomcat application server. All applications that use a Java 2

Platform, Enterprise Edition (J2EE platform) application server can immediately benefit from multicore technology.

- **Multi-instance applications:** Even if an individual application does not scale to take advantage of a large number of threads, it is still possible to gain from multicore architecture by running multiple instances of the application in parallel. If multiple application instances require some degree of isolation, virtualization technology (for the hardware of the operating system) can be used to provide each of them with its own separate and secure environment.

Application Example: Valve Game Software

Valve is an entertainment and technology company that has developed a number of popular games, as well as the Source engine, one of the most widely played game engines available. Source is an animation engine used by Valve for its games and licensed for other game developers.

In recent years, Valve has reprogrammed the Source engine software to use multithreading to exploit the power of multicore processor chips from Intel and AMD [REIM06]. The revised Source engine code provides more powerful support for Valve games such as *Half Life 2*.

From Valve's perspective, threading granularity options are defined as follows [HARR06]:

- **Coarse threading:** Individual modules, called systems, are assigned to individual processors. In the Source engine case, this would mean putting rendering on one processor, AI (artificial intelligence) on another, physics on another, and so on. This is straightforward. In essence, each major module is single-threaded and the principal coordination involves synchronizing all the threads with a timeline thread.
- **Fine-grained threading:** Many similar or identical tasks are spread across multiple processors. For example, a loop that iterates over an array of data can be split up into a number of smaller parallel loops in individual threads that can be scheduled in parallel.
- **Hybrid threading:** This involves the selective use of fine-grained threading for some systems, and single-threaded for other systems.

Valve found that through coarse threading, it could achieve up to twice the performance across two processors compared to executing on a single processor. But this performance gain could only be achieved with contrived cases. For real-world gameplay, the improvement was on the order of a factor of 1.2. Valve also found that effective use of fine-grained threading was difficult. The time-per-work unit can be variable, and managing the timeline of outcomes and consequences involved complex programming.

Valve found that a hybrid threading approach was the most promising and would scale the best, as multicore systems with 8 or 16 processors became available. Valve identified systems that operate very effectively being permanently assigned to a single processor. An example is sound mixing, which has little user interaction, is not constrained by the frame configuration of windows, and works on its own set

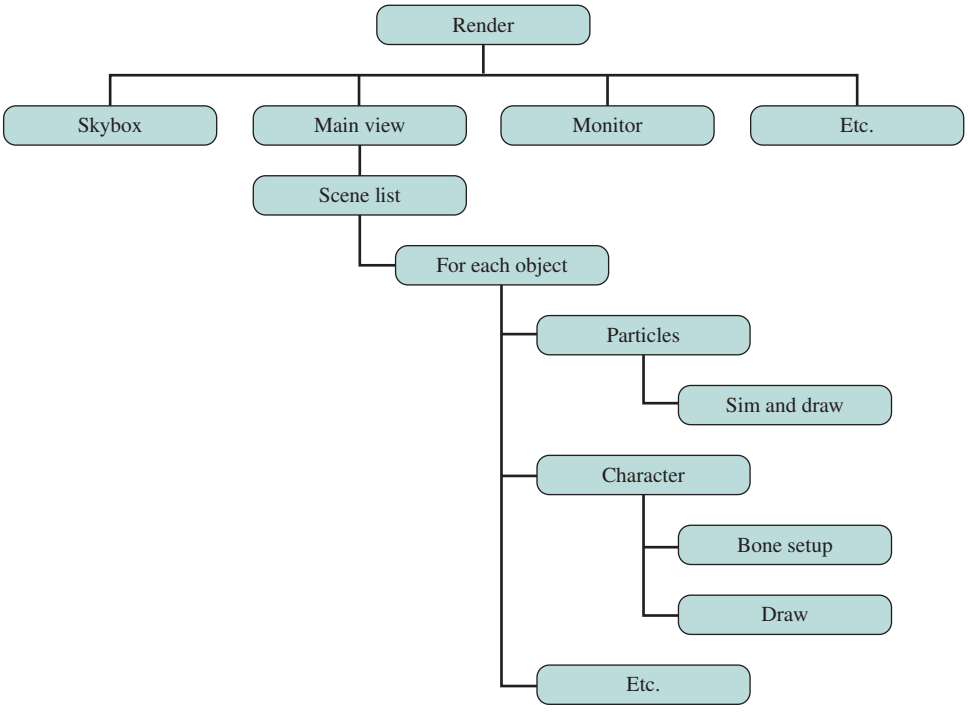


Figure 4.9 Hybrid Threading for Rendering Module

of data. Other modules, such as scene rendering, can be organized into a number of threads so that the module can execute on a single processor but achieve greater performance as it is spread out over more and more processors.

Figure 4.9 illustrates the thread structure for the rendering module. In this hierarchical structure, higher-level threads spawn lower-level threads as needed. The rendering module relies on a critical part of the Source engine, the world list, which is a database representation of the visual elements in the game's world. The first task is to determine what are the areas of the world that need to be rendered. The next task is to determine what objects are in the scene as viewed from multiple angles. Then comes the processor-intensive work. The rendering module has to work out the rendering of each object from multiple points of view, such as the player's view, the view of monitors, and the point of view of reflections in water.

Some of the key elements of the threading strategy for the rendering module are listed in [LEON07] and include the following:

- Construct scene rendering lists for multiple scenes in parallel (e.g., the world and its reflection in water).
- Overlap graphics simulation.
- Compute character bone transformations for all characters in all scenes in parallel.
- Allow multiple threads to draw in parallel.

The designers found that simply locking key databases, such as the world list, for a thread was too inefficient. Over 95% of the time, a thread is trying to read from a data set, and only 5% of the time at most is spent in writing to a data set. Thus, a concurrency mechanism known as the single-writer-multiple-readers model works effectively.

4.4 WINDOWS PROCESS AND THREAD MANAGEMENT

This section begins with an overview of the key objects and mechanisms that support application execution in Windows. The remainder of the section looks in more detail at how processes and threads are managed.

An **application** consists of one or more processes. Each **process** provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

A **thread** is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

A **job object** allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object. Examples include enforcing limits such as working set size and process priority or terminating all processes associated with a job.

A **thread pool** is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads.

A **fiber** is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers. In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads. From a system standpoint, a fiber assumes the identity of the thread that runs it. For example if a fiber accesses thread local storage, it is accessing the thread local storage of the thread that is running it. In addition, if a fiber calls the `ExitThread` function, the thread that is running it exits. However, a fiber does not have all the same state information associated with it as that associated with a thread. The only state information maintained for a fiber is its stack, a subset of its registers, and the fiber data provided during fiber creation. The saved registers are the set of registers typically preserved across

a function call. Fibers are not preemptively scheduled. A thread schedules a fiber by switching to it from another fiber. The system still schedules threads to run. When a thread that is running fibers is preempted, its currently running fiber is preempted but remains selected.

User-mode scheduling (UMS) is a lightweight mechanism that applications can use to schedule their own threads. An application can switch between UMS threads in user mode without involving the system scheduler, and regain control of the processor if a UMS thread blocks in the kernel. Each UMS thread has its own thread context instead of sharing the thread context of a single thread. The ability to switch between threads in user mode makes UMS more efficient than thread pools for short-duration work items that require few system calls. UMS is useful for applications with high performance requirements that need to efficiently run many threads concurrently on multiprocessor or multicore systems. To take advantage of UMS, an application must implement a scheduler component that manages the application's UMS threads and determines when they should run.

Management of Background Tasks and Application Lifecycles

Beginning with Windows 8, and carrying through to Windows 10, developers are responsible for managing the state of their individual applications. Previous versions of Windows always give the user full control of the lifetime of a process. In the classic desktop environment, a user is responsible for closing an application. A dialog box might prompt them to save their work. In the new Metro interface, Windows takes over the process lifecycle of an application. Although a limited number of applications can run alongside the main app in the Metro UI using the SnapView functionality, only one Store application can run at one time. This is a direct consequence of the new design. Windows Live Tiles give the appearance of applications constantly running on the system. In reality, they receive push notifications and do not use system resources to display the dynamic content offered.

The foreground application in the Metro interface has access to all of the processor, network, and disk resources available to the user. All other apps are suspended and have no access to these resources. When an app enters a suspended mode, an event should be triggered to store the state of the user's information. This is the responsibility of the application developer. For a variety of reasons, whether it needs resources or because an application timed out, Windows may terminate a background app. This is a significant departure from the Windows operating systems that precede it. The app needs to retain any data the user entered, settings they changed, and so on. That means you need to save your app's state when it's suspended, in case Windows terminates it, so you can restore its state later. When the app returns to the foreground, another event is triggered to obtain the user state from memory. No event fires to indicate termination of a background app. Rather, the application data will remain resident on the system, as though it is suspended, until the app is launched again. Users expect to find the app as they left it, whether it was suspended or terminated by Windows, or closed by the user. Application developers can use code to determine whether it should restore a saved state.

Some applications, such as news feeds, may look at the date stamp associated with the previous execution of the app and elect to discard the data in favor of newly obtained information. This is a determination made by the developer, not by the operating system. If the user closes an app, unsaved data is not saved. With foreground tasks occupying all of the system resources, starvation of background apps is a reality in Windows. This makes the application development relating to the state changes critical to the success of a Windows app.

To process the needs of background tasks, a background task API is built to allow apps to perform small tasks while not in the foreground. In this restricted environment, apps may receive push notifications from a server or a user may receive a phone call. Push notifications are template XML strings. They are managed through a cloud service known as the Windows Notification Service (WNS). The service will occasionally push updates to the user's background apps. The API will queue those requests and process them when it receives enough processor resources. Background tasks are severely limited in the usage of processor, receiving only one processor second per processor hour. This ensures that critical tasks receive guaranteed application resource quotas. It does not, however, guarantee a background app will ever run.

The Windows Process

Important characteristics of Windows processes are the following:

- Windows processes are implemented as objects.
- A process can be created as a new process or as a copy of an existing process.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.

Figure 4.10, based on one in [RUSS11], illustrates the way in which a process relates to the resources it controls or uses. Each process is assigned a security access token, called the primary token of the process. When a user first logs on, Windows creates an access token that includes the security ID for the user. Every process that is created by or runs on behalf of this user has a copy of this access token. Windows uses the token to validate the user's ability to access secured objects, or to perform restricted functions on the system and on secured objects. The access token controls whether the process can change its own attributes. In this case, the process does not have a handle opened to its access token. If the process attempts to open such a handle, the security system determines whether this is permitted, and therefore whether the process may change its own attributes.

Also related to the process is a series of blocks that define the virtual address space currently assigned to this process. The process cannot directly modify these structures, but must rely on the virtual memory manager, which provides a memory-allocation service for the process.

Finally, the process includes an object table, with handles to other objects known to this process. Figure 4.10 shows a single thread. In addition, the process has access to a file object and to a section object that defines a section of shared memory.

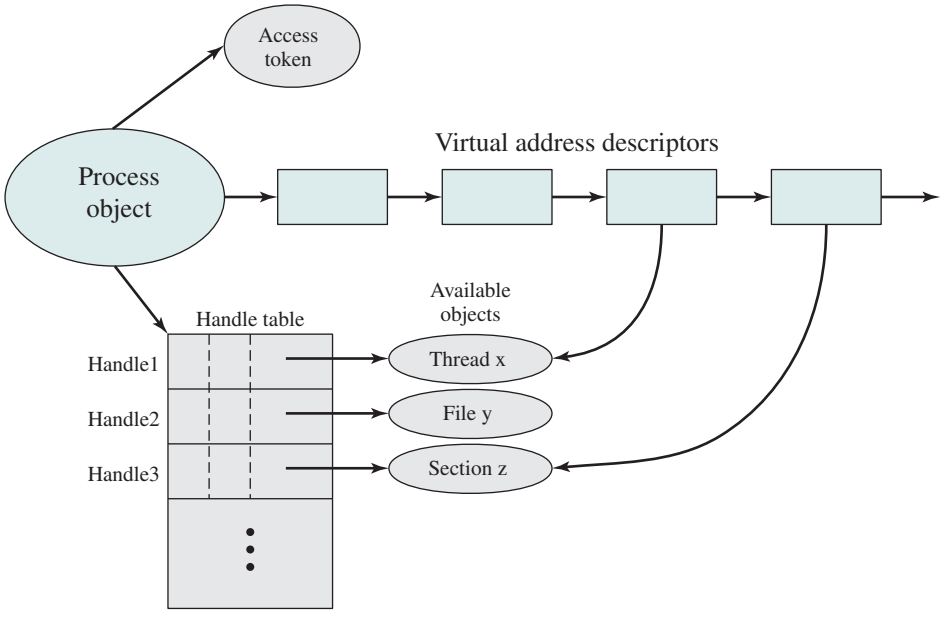


Figure 4.10 **A Windows Process and Its Resources**

Process and Thread Objects

The object-oriented structure of Windows facilitates the development of a general-purpose process facility. Windows makes use of two types of process-related objects: processes and threads. A process is an entity corresponding to a user job or application that owns resources, such as memory and open files. A thread is a dispatchable unit of work that executes sequentially and is interruptible, so the processor can turn to another thread.

Each Windows process is represented by an object. Each process object includes a number of attributes and encapsulates a number of actions, or services, that it may perform. A process will perform a service when called upon through a set of published interface methods. When Windows creates a new process, it uses the object class, or type, defined for the Windows process as a template to generate a new object instance. At the time of creation, attribute values are assigned. Table 4.3 gives a brief definition of each of the object attributes for a process object.

A Windows process must contain at least one thread to execute. That thread may then create other threads. In a multiprocessor system, multiple threads from the same process may execute in parallel. Table 4.4 defines the thread object attributes. Note some of the attributes of a thread resemble those of a process. In those cases, the thread attribute value is derived from the process attribute value. For example, the *thread processor affinity* is the set of processors in a multiprocessor system that may execute this thread; this set is equal to or a subset of the *process processor affinity.*

Table 4.3 Windows Process Object Attributes

Process ID	A unique value that identifies the process to the operating system.
Security descriptor	Describes who created an object, who can gain access to or use the object, and who is denied access to the object.
Base priority	A baseline execution priority for the process's threads.
Default processor affinity	The default set of processors on which the process's threads can run.
Quota limits	The maximum amount of paged and nonpaged system memory, paging file space, and processor time a user's processes can use.
Execution time	The total amount of time all threads in the process have executed.
I/O counters	Variables that record the number and type of I/O operations that the process's threads have performed.
VM operation counters	Variables that record the number and types of virtual memory operations that the process's threads have performed.
Exception/debugging ports	Interprocess communication channels to which the process manager sends a message when one of the process's threads causes an exception. Normally, these are connected to environment subsystem and debugger processes, respectively.
Exit status	The reason for a process's termination.

Note one of the attributes of a thread object is context, which contains the values of the processor registers when the thread last ran. This information enables threads to be suspended and resumed. Furthermore, it is possible to alter the behavior of a thread by altering its context while it is suspended.

Table 4.4 Windows Thread Object Attributes

Thread ID	A unique value that identifies a thread when it calls a server.
Thread context	The set of register values and other volatile data that defines the execution state of a thread.
Dynamic priority	The thread's execution priority at any given moment.
Base priority	The lower limit of the thread's dynamic priority.
Thread processor affinity	The set of processors on which the thread can run, which is a subset or all of the processor affinity of the thread's process.
Thread execution time	The cumulative amount of time a thread has executed in user mode and in kernel mode.
Alert status	A flag that indicates whether a waiting thread may execute an asynchronous procedure call.
Suspension count	The number of times the thread's execution has been suspended without being resumed.
Impersonation token	A temporary access token allowing a thread to perform operations on behalf of another process (used by subsystems).
Termination port	An interprocess communication channel to which the process manager sends a message when the thread terminates (used by subsystems).
Thread exit status	The reason for a thread's termination.

Multithreading

Windows supports concurrency among processes because threads in different processes may execute concurrently (appear to run at the same time). Moreover, multiple threads within the same process may be allocated to separate processors and execute simultaneously (actually run at the same time). A multithreaded process achieves concurrency without the overhead of using multiple processes. Threads within the same process can exchange information through their common address space and have access to the shared resources of the process. Threads in different processes can exchange information through shared memory that has been set up between the two processes.

An object-oriented multithreaded process is an efficient means of implementing a server application. For example, one server process can service a number of clients concurrently.

Thread States

An existing Windows thread is in one of six states (see Figure 4.11):

1. **Ready:** A ready thread may be scheduled for execution. The Kernel dispatcher keeps track of all ready threads and schedules them in priority order.
2. **Standby:** A standby thread has been selected to run next on a particular processor. The thread waits in this state until that processor is made available. If the standby thread's priority is high enough, the running thread on that processor may be preempted in favor of the standby thread. Otherwise, the standby thread waits until the running thread blocks or exhausts its time slice.

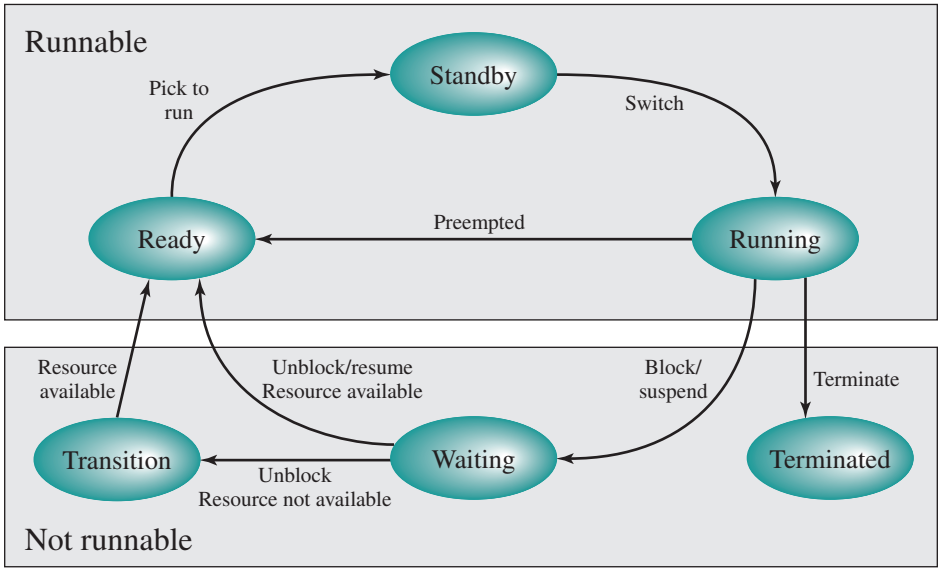


Figure 4.11 Windows Thread States

3. **Running:** Once the Kernel dispatcher performs a thread switch, the standby thread enters the Running state and begins execution and continues execution until it is preempted by a higher-priority thread, exhausts its time slice, blocks, or terminates. In the first two cases, it goes back to the Ready state.
4. **Waiting:** A thread enters the Waiting state when (1) it is blocked on an event (e.g., I/O), (2) it voluntarily waits for synchronization purposes, or (3) an environment subsystem directs the thread to suspend itself. When the waiting condition is satisfied, the thread moves to the Ready state if all of its resources are available.
5. **Transition:** A thread enters this state after waiting if it is ready to run, but the resources are not available. For example, the thread's stack may be paged out of memory. When the resources are available, the thread goes to the Ready state.
6. **Terminated:** A thread can be terminated by itself, by another thread, or when its parent process terminates. Once housekeeping chores are completed, the thread is removed from the system, or it may be retained by the Executive[6] for future reinitialization.

Support for OS Subsystems

The general-purpose process and thread facility must support the particular process and thread structures of the various OS environments. It is the responsibility of each OS subsystem to exploit the Windows process and thread features to emulate the process and thread facilities of its corresponding OS. This area of process/thread management is complicated, and we give only a brief overview here.

Process creation begins with a request for a new process from an application. The application issues a create-process request to the corresponding protected subsystem, which passes the request to the Executive. The Executive creates a process object and returns a handle for that object to the subsystem. When Windows creates a process, it does not automatically create a thread. In the case of Win32, a new process must always be created with an initial thread. Therefore, the Win32 subsystem calls the Windows process manager again to create a thread for the new process, receiving a thread handle back from Windows. The appropriate thread and process information are then returned to the application. In the case of POSIX, threads are not supported. Therefore, the POSIX subsystem obtains a thread for the new process from Windows so that the process may be activated but returns only process information to the application. The fact that the POSIX process is implemented using both a process and a thread from the Windows Executive is not visible to the application.

When a new process is created by the Executive, the new process inherits many of its attributes from the creating process. However, in the Win32 environment, this process creation is done indirectly. An application client process issues its process creation request to the Win32 subsystem; then the subsystem in turn issues a process request to the Windows executive. Because the desired effect is that the new process inherits characteristics of the client process and not of the server process, Windows enables the

[6]The Windows Executive is described in Chapter 2. It contains the base operating system services, such as memory management, process and thread management, security, I/O, and interprocess communication.

subsystem to specify the parent of the new process. The new process then inherits the parent's access token, quota limits, base priority, and default processor affinity.

4.5 SOLARIS THREAD AND SMP MANAGEMENT

Solaris implements multilevel thread support designed to provide considerable flexibility in exploiting processor resources.

Multithreaded Architecture

Solaris makes use of four separate thread-related concepts:

1. **Process:** This is the normal UNIX process and includes the user's address space, stack, and process control block.
2. **User-level threads:** Implemented through a threads library in the address space of a process, these are invisible to the OS. A user-level thread (ULT)[7] is a user-created unit of execution within a process.
3. **Lightweight processes:** A lightweight process (LWP) can be viewed as a mapping between ULTs and kernel threads. Each LWP supports ULT and maps to one kernel thread. LWPs are scheduled by the kernel independently, and may execute in parallel on multiprocessors.
4. **Kernel threads:** These are the fundamental entities that can be scheduled and dispatched to run on one of the system processors.

Figure 4.12 illustrates the relationship among these four entities. Note there is always exactly one kernel thread for each LWP. An LWP is visible within a process to the application. Thus, LWP data structures exist within their respective process address space. At the same time, each LWP is bound to a single dispatchable kernel thread, and the data structure for that kernel thread is maintained within the kernel's address space.

A process may consist of a single ULT bound to a single LWP. In this case, there is a single thread of execution, corresponding to a traditional UNIX process. When concurrency is not required within a single process, an application uses this process structure. If an application requires concurrency, its process contains multiple threads, each bound to a single LWP, which in turn are each bound to a single kernel thread.

In addition, there are kernel threads that are not associated with LWPs. The kernel creates, runs, and destroys these kernel threads to execute specific system functions. The use of kernel threads rather than kernel processes to implement system functions reduces the overhead of switching within the kernel (from a process switch to a thread switch).

Motivation

The three-level thread structure (ULT, LWP, kernel thread) in Solaris is intended to facilitate thread management by the OS and to provide a clean interface to applications. The ULT interface can be a standard thread library. A defined ULT maps onto a LWP, which is managed by the OS and which has defined states of execution,

[7]Again, the acronym ULT is unique to this book and is not found in the Solaris literature.

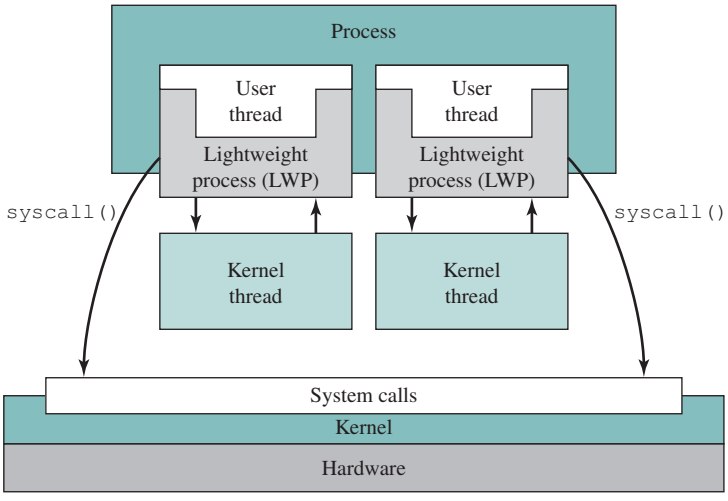


Figure 4.12 Processes and Threads in Solaris [MCDO07]

defined subsequently. An LWP is bound to a kernel thread with a one-to-one correspondence in execution states. Thus, concurrency and execution are managed at the level of the kernel thread.

In addition, an application has access to hardware through an application programming interface consisting of system calls. The API allows the user to invoke kernel services to perform privileged tasks on behalf of the calling process, such as read or write a file, issue a control command to a device, create a new process or thread, allocate memory for the process to use, and so on.

Process Structure

Figure 4.13 compares, in general terms, the process structure of a traditional UNIX system with that of Solaris. On a typical UNIX implementation, the process structure includes:

- Process ID.
- User IDs.
- Signal dispatch table, which the kernel uses to decide what to do when sending a signal to a process.
- File descriptors, which describe the state of files in use by this process.
- Memory map, which defines the address space for this process.
- Processor state structure, which includes the kernel stack for this process.

Solaris retains this basic structure but replaces the processor state block with a list of structures containing one data block for each LWP.

The LWP data structure includes the following elements:

- An LWP identifier
- The priority of this LWP and hence the kernel thread that supports it

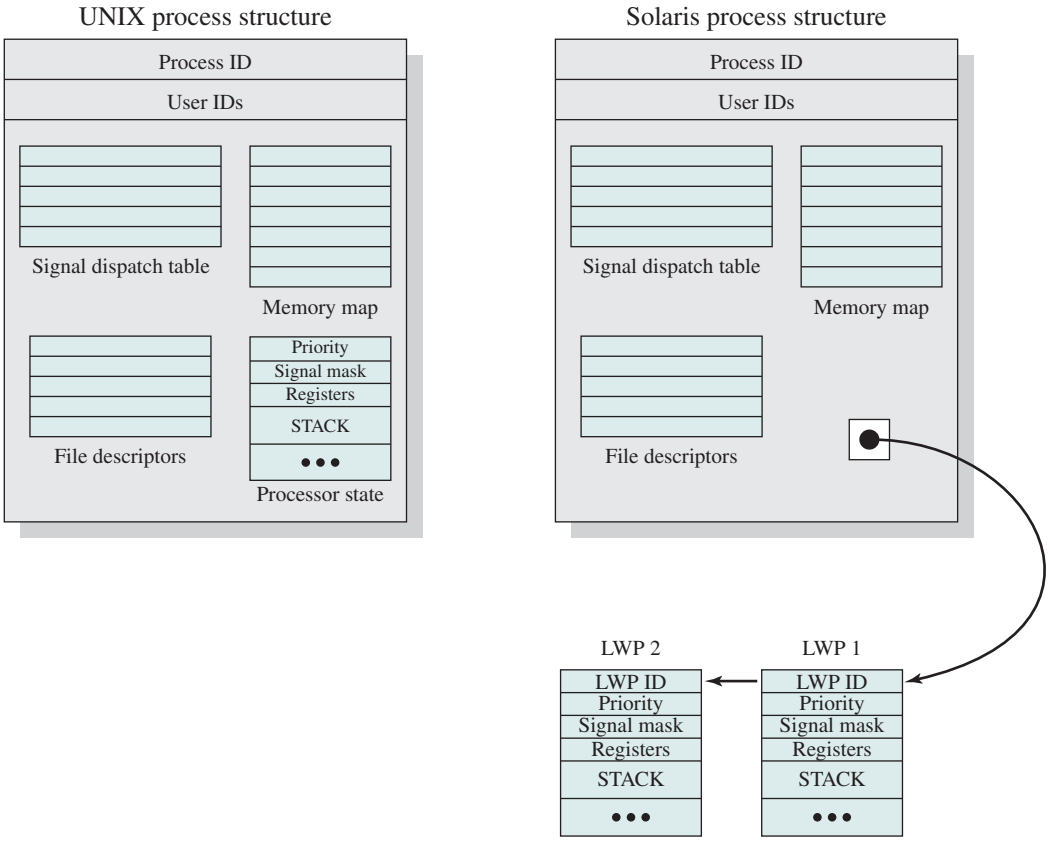


Figure 4.13 Process Structure in Traditional UNIX and Solaris [LEWI96]

- A signal mask that tells the kernel which signals will be accepted
- Saved values of user-level registers (when the LWP is not running)
- The kernel stack for this LWP, which includes system call arguments, results, and error codes for each call level
- Resource usage and profiling data
- Pointer to the corresponding kernel thread
- Pointer to the process structure

Thread Execution

Figure 4.14 shows a simplified view of both thread execution states. These states reflect the execution status of both a kernel thread and the LWP bound to it. As mentioned, some kernel threads are not associated with an LWP; the same execution diagram applies. The states are as follows:

- **RUN:** The thread is runnable; that is, the thread is ready to execute.
- **ONPROC:** The thread is executing on a processor.

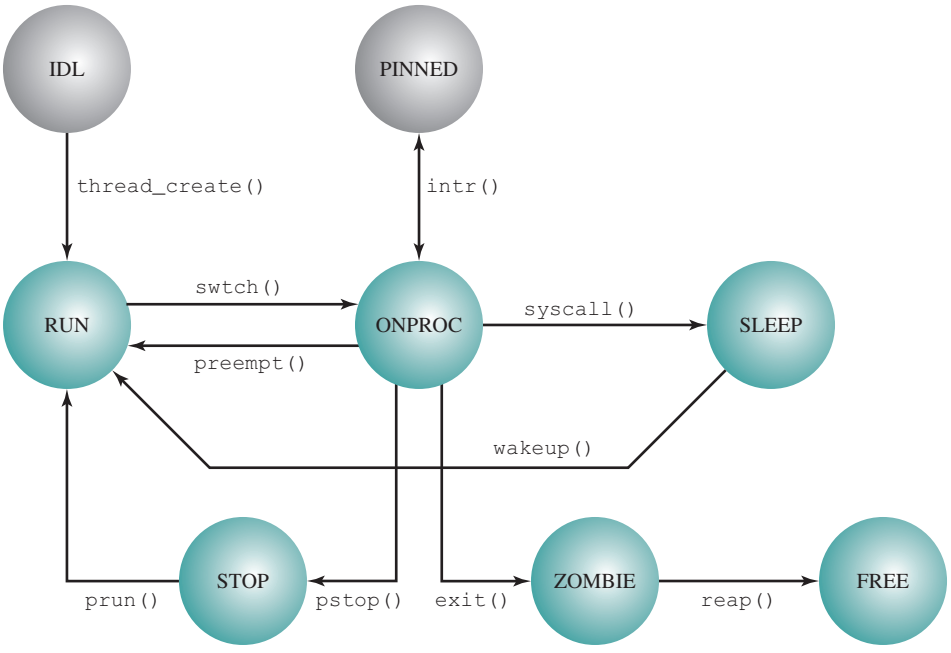


Figure 4.14 **Solaris Thread States**

- **SLEEP:** The thread is blocked.
- **STOP:** The thread is stopped.
- **ZOMBIE:** The thread has terminated.
- **FREE:** Thread resources have been released and the thread is awaiting removal from the OS thread data structure.

A thread moves from ONPROC to RUN if it is preempted by a higher-priority thread or because of time slicing. A thread moves from ONPROC to SLEEP if it is blocked and must await an event to return the RUN state. Blocking occurs if the thread invokes a system call and must wait for the system service to be performed. A thread enters the STOP state if its process is stopped; this might be done for debugging purposes.

Interrupts as Threads

Most operating systems contain two fundamental forms of concurrent activity: processes and interrupts. Processes (or threads) cooperate with each other and manage the use of shared data structures by means of a variety of primitives that enforce mutual exclusion (only one process at a time can execute certain code or access certain data) and that synchronize their execution. Interrupts are synchronized by preventing their handling for a period of time. Solaris unifies these two concepts into a single model, namely kernel threads, and the mechanisms for scheduling and executing kernel threads. To do this, interrupts are converted to kernel threads.

The motivation for converting interrupts to threads is to reduce overhead. Interrupt handlers often manipulate data shared by the rest of the kernel. Therefore, while a kernel routine that accesses such data is executing, interrupts must be blocked, even though most interrupts will not affect that data. Typically, the way this is done is for the routine to set the interrupt priority level higher to block interrupts, then lower the priority level after access is completed. These operations take time. The problem is magnified on a multiprocessor system. The kernel must protect more objects and may need to block interrupts on all processors.

The solution in Solaris can be summarized as follows:

1. Solaris employs a set of kernel threads to handle interrupts. As with any kernel thread, an interrupt thread has its own identifier, priority, context, and stack.
2. The kernel controls access to data structures and synchronizes among interrupt threads using mutual exclusion primitives, of the type to be discussed in Chapter 5. That is, the normal synchronization techniques for threads are used in handling interrupts.
3. Interrupt threads are assigned higher priorities than all other types of kernel threads.

When an interrupt occurs, it is delivered to a particular processor and the thread that was executing on that processor is pinned. A pinned thread cannot move to another processor and its context is preserved; it is simply suspended until the interrupt is processed. The processor then begins executing an interrupt thread. There is a pool of deactivated interrupt threads available, so a new thread creation is not required. The interrupt thread then executes to handle the interrupt. If the handler routine needs access to a data structure that is currently locked in some fashion for use by another executing thread, the interrupt thread must wait for access to that data structure. An interrupt thread can only be preempted by another interrupt thread of higher priority.

Experience with Solaris interrupt threads indicates that this approach provides superior performance to the traditional interrupt-handling strategy [KLEI95].

4.6 LINUX PROCESS AND THREAD MANAGEMENT

Linux Tasks

A process, or task, in Linux is represented by a `task_struct` data structure. The `task_struct` data structure contains information in a number of categories:

- **State:** The execution state of the process (executing, ready, suspended, stopped, zombie). This is described subsequently.
- **Scheduling information:** Information needed by Linux to schedule processes. A process can be normal or real time and has a priority. Real-time processes are scheduled before normal processes, and within each category, relative priorities can be used. A reference counter keeps track of the amount of time a process is allowed to execute.
- **Identifiers:** Each process has a unique process identifier (PID) and also has user and group identifiers. A group identifier is used to assign resource access privileges to a group of processes.

- **Interprocess communication:** Linux supports the IPC mechanisms found in UNIX SVR4, described later in Chapter 6.
- **Links:** Each process includes a link to its parent process, links to its siblings (processes with the same parent), and links to all of its children.
- **Times and timers:** Includes process creation time and the amount of processor time so far consumed by the process. A process may also have associated one or more interval timers. A process defines an interval timer by means of a system call; as a result, a signal is sent to the process when the timer expires. A timer may be single use or periodic.
- **File system:** Includes pointers to any files opened by this process, as well as pointers to the current and the root directories for this process
- **Address space:** Defines the virtual address space assigned to this process
- **Processor-specific context:** The registers and stack information that constitute the context of this process

Figure 4.15 shows the execution states of a process. These are as follows:

- **Running:** This state value corresponds to two states. A Running process is either executing, or it is ready to execute.
- **Interruptible:** This is a blocked state, in which the process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process.

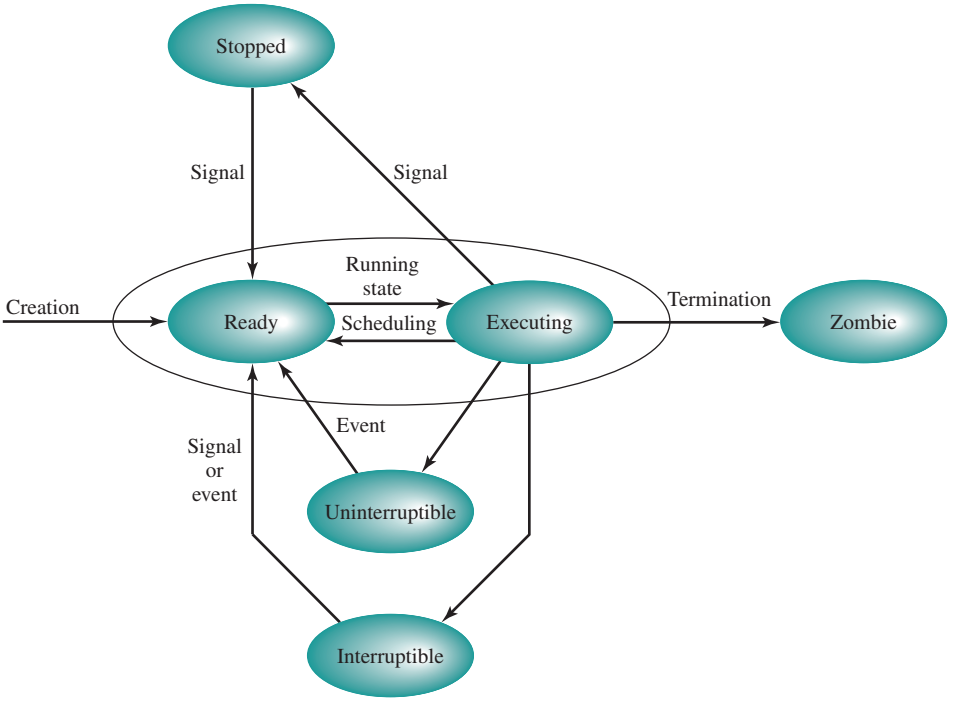


Figure 4.15 Linux Process/Thread Model

- **Uninterruptible:** This is another blocked state. The difference between this and the Interruptible state is that in an Uninterruptible state, a process is waiting directly on hardware conditions and therefore will not handle any signals.
- **Stopped:** The process has been halted and can only resume by positive action from another process. For example, a process that is being debugged can be put into the Stopped state.
- **Zombie:** The process has been terminated but, for some reason, still must have its task structure in the process table.

Linux Threads

Traditional UNIX systems support a single thread of execution per process, while modern UNIX systems typically provide support for multiple kernel-level threads per process. As with traditional UNIX systems, older versions of the Linux kernel offered no support for multithreading. Instead, applications would need to be written with a set of user-level library functions, the most popular of which is known as *pthread (POSIX thread) libraries*, with all of the threads mapping into a single kernel-level process.[8] We have seen that modern versions of UNIX offer kernel-level threads. Linux provides a unique solution in that it does not recognize a distinction between threads and processes. Using a mechanism similar to the lightweight processes of Solaris, user-level threads are mapped into kernel-level processes. Multiple user-level threads that constitute a single user-level process are mapped into Linux kernel-level processes that share the same group ID. This enables these processes to share resources such as files and memory, and to avoid the need for a context switch when the scheduler switches among processes in the same group.

A new process is created in Linux by copying the attributes of the current process. A new process can be *cloned* so it shares resources such as files, signal handlers, and virtual memory. When the two processes share the same virtual memory, they function as threads within a single process. However, no separate type of data structure is defined for a thread. In place of the usual fork() command, processes are created in Linux using the clone() command. This command includes a set of flags as arguments. The traditional fork() system call is implemented by Linux as a clone() system call with all of the clone flags cleared.

Examples of clone flags include the following:

- CLONE_NEWPID: Creates new process ID namespace.
- CLONE_PARENT: Caller and new task share the same parent process.
- CLONE_SYSVSEM: Shares System V SEM_UNDO semantics.
- CLONE_THREAD: Inserts this process into the same thread group of the parent. If this flag is true, it implicitly enforces CLONE_PARENT.
- CLONE_VM: Shares the address space (memory descriptor and all page tables).

[8]POSIX (Portable Operating Systems based on UNIX) is an IEEE API standard that includes a standard for a thread API. Libraries implementing the POSIX Threads standard are often named *Pthreads*. Pthreads are most commonly used on UNIX-like POSIX systems such as Linux and Solaris, but Microsoft Windows implementations also exist.

- CLONE_FS: Shares the same filesystem information (including current working directory, the root of the filesystem, and the umask).
- CLONE_FILES: Shares the same file descriptor table. Creating a file descriptor or closing a file descriptor is propagated to the another process, as well as changing the associated flags of a file descriptor using the fcntl() system call.

When the Linux kernel performs a context switch from one process to another, it checks whether the address of the page directory of the current process is the same as that of the to-be-scheduled process. If they are, then they are sharing the same address space, so a context switch is basically just a jump from one location of code to another location of code.

Although cloned processes that are part of the same process group can share the same memory space, they cannot share the same user stacks. Thus the clone() call creates separate stack spaces for each process.

Linux Namespaces

Associated with each process in Linux are a set of **namespaces**. A namespace enables a process (or multiple processes that share the same namespace) to have a different view of the system than other processes that have other associated namespaces. Namespaces and cgroups (which will be described in the following section) are the basis of Linux lightweight virtualization, which is a feature that provides a process or group of processes with the illusion that they are the only processes on the system. This feature is used widely by Linux Containers projects. There are currently six namespaces in Linux: mnt, pid, net, ipc, uts, and user.

Namespaces are created by the clone() system call, which gets as a parameter one of the six namespaces clone flags (CLONE_NEWNS, CLONE_NEWPID, CLONE_NEWNET, CLONE_NEWIPC, CLONE_NEWUTS, and CLONE_NEWUSER). A process can also create a namespace with the unshare() system call with one of these flags; as opposed to clone(), a new process is not created in such a case; only a new namespace is created, which is attached to the calling process.

Mount Namespace A mount namespace provides the process with a specific view of the filesystem hierarchy, such that two processes with different mount namespaces see different filesystem hierarchies. All of the file operations that a process employs apply only to the filesystem visible to the process.

UTS Namespace The UTS (UNIX timesharing) namespace is related to the uname Linux system call. The uname call returns the name and information about the current kernel, including nodename, which is the system name within some implementation-defined network; and domainname, which is the NIS domain name. NIS (Network Information Service) is a standard scheme used on all major UNIX and UNIX-like systems. It allows a group of machines within an NIS domain to share a common set of configuration files. This permits a system administrator to set up NIS client systems with only minimal configuration data and add, remove, or modify configuration data from a single location. With the UTS namespace, initialization and configuration parameters can vary for different processes on the same system.

IPC Namespace An IPC namespace isolates certain interprocess communication (IPC) resources, such as semaphores, POSIX message queues, and more. Thus, concurrency mechanisms can be employed by the programmer that enable IPC among processes that share the same IPC namespace.

PID Namespace PID namespaces isolate the process ID space, so processes in different PID namespaces can have the same PID. This feature is used for Checkpoint/Restore In Userspace (CRIU), a Linux software tool. Using this tool, you can freeze a running application (or part of it) and checkpoint it to a hard drive as a collection of files. You can then use the files to restore and run the application from the freeze point on that machine or on a different host. A distinctive feature of the CRIU project is that it is mainly implemented in user space, after attempts to implement it mainly in kernel failed.

Network Namespace Network namespaces provide isolation of the system resources associated with networking. Thus, each network namespace has its own network devices, IP addresses, IP routing tables, port numbers, and so on. These namespaces virtualize all access to network resources. This allows each process or a group of processes that belong to this network namespace to have the network access it needs (but no more). At any given time, a network device belongs to only one network namespace. Also, a socket can belong to only one namespace.

User Namespace User namespaces provide a container with its own set of UIDs, completely separate from those in the parent. So when a process clones a new process it can assign it a new user namespace, as well as a new PID namespace, and all the other namespaces. The cloned process can have access to and privileges for all of the resources of the parent process, or a subset of the resources and privileges of the parent. The user namespaces are considered sensitive in terms of security, as they enable creating non-privileged containers (processes which are created by a non-root user).

The Linux cgroup Subsystem The Linux cgroup subsystem, together with the namespace subsystem, are the basis of lightweight process virtualization, and as such they form the basis of Linux containers; almost every Linux containers project nowadays (such as Docker, LXC, Kubernetes, and others) is based on both of them. The Linux cgroups subsystem provides resource management and accounting. It handles resources such as CPU, network, memory, and more; and it is mostly needed in both ends of the spectrum (embedded devices and servers), and much less in desktops. Development of cgroups was started in 2006 by engineers at Google under the name "process containers," which was later changed to "cgroups" to avoid confusion with Linux Containers. In order to implement cgroups, no new system call was added. A new virtual file system (VFS), "cgroups" (also referred to sometimes as cgroupfs) was added, as all the cgroup filesystem operations are filesystem based. A new version of cgroups, called cgroups v2, was released in kernel 4.5 (March 2016). The cgroup v2 subsystem addressed many of the inconsistencies across cgroup v1 controllers, and made cgroup v2 better organized, by establishing strict and consistent interfaces.

Currently, there are 12 cgroup v1 controllers and 3 cgroup v2 controllers (memory, I/O, and PIDs) and there are other v2 controllers that are a work in progress.

In order to use the cgroups filesystem (i.e., browse it, attach tasks to cgroups, and so on), it first must be mounted, like when working with any other filesystem. The cgroup filesystem can be mounted on any path on the filesystem, and many userspace applications and container projects use `/sys/fs/cgroup` as a mounting point. After mounting the cgroups filesystem, you can create subgroups, attach processes and tasks to these groups, set limitations on various system resources, and more. The cgroup v1 implementation will probably coexist with the cgroup v2 implementation as long as there are userspace projects that use it; we have a parallel phenomenon in other kernel subsystems, when a new implementation of existing subsystem replaces the current one; for example, currently both iptables and the new nftables coexist, and in the past, iptables coexisted with ipchains.

4.7 ANDROID PROCESS AND THREAD MANAGEMENT

Before discussing the details of the Android approach to process and thread management, we need to describe the Android concepts of applications and activities.

Android Applications

An Android application is the software that implements an app. Each Android application consists of one or more instance of one or more of four types of application components. Each component performs a distinct role in the overall application behavior, and each component can be activated independently within the application and even by other applications. The following are the four types of components:

1. **Activities:** An activity corresponds to a single screen visible as a user interface. For example, an e-mail application might have one activity that shows a list of new e-mails, another activity to compose an e-mail, and another activity for reading e-mails. Although the activities work together to form a cohesive user experience in the e-mail application, each one is independent of the others. Android makes a distinction between internal and exported activities. Other apps may start exported activities, which generally include the main screen of the app. However, other apps cannot start the internal activities. For example, a camera application can start the activity in the e-mail application that composes new mail, in order for the user to share a picture.
2. **Services:** Services are typically used to perform background operations that take a considerable amount of time to finish. This ensures faster responsiveness, for the main thread (a.k.a. UI thread) of an application, with which the user is directly interacting. For example, a service might create a thread to play music in the background while the user is in a different application, or it might create a thread to fetch data over the network without blocking user interaction with an activity. A service may be invoked by an application. Additionally, there are system services that run for the entire lifetime of the Android system, such as Power Manager, Battery, and Vibrator services. These system services create threads that are part of the System Server process.
3. **Content providers:** A content provider acts as an interface to application data that can be used by the application. One category of managed data is private

data, which is used only by the application containing the content provider. For example the NotePad application uses a content provider to save notes. The other category is shared data, accessible by multiple applications. This category includes data stored in file systems, an SQLite database, on the Web, or any other persistent storage location your application can access.

4. **Broadcast receivers:** A broadcast receiver responds to system-wide broadcast announcements. A broadcast can originate from another application, such as to let other applications know that some data has been downloaded to the device and is available for them to use, or from the system (for example, a low-battery warning).

Each application runs on its own dedicated virtual machine and its own single process that encompasses the application and its virtual machine (see Figure 4.16). This approach, referred to as the sandboxing model, isolates each application. Thus, one application cannot access the resources of the other without permission being granted. Each application is treated as a separate Linux user with its own unique user ID, which is used to set file permissions.

Activities

An Activity is an application component that provides a screen with which users can interact in order to do something, such as make a phone call, take a photo, send an e-mail, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

As was mentioned, an application may include multiple activities. When an application is running, one activity is in the foreground, and it is this activity that

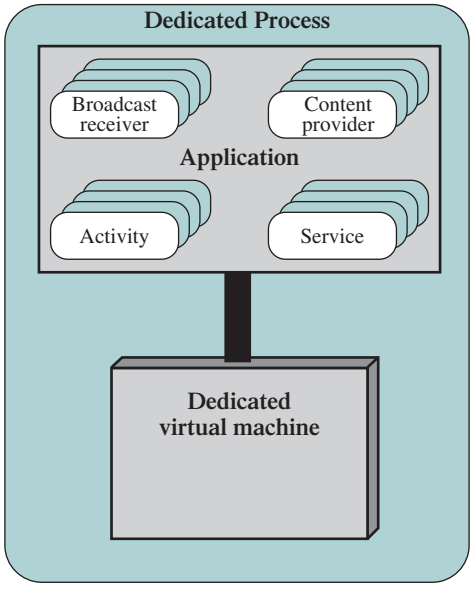


Figure 4.16 Android Application

interacts with the user. The activities are arranged in a last-in-first-out stack (the *back stack*), in the order in which each activity is opened. If the user switches to some other activity within the application, the new activity is created and pushed on to the top of the back stack, while the preceding foreground activity becomes the second item on the stack for this application. This process can be repeated multiple times, adding to the stack. The user can back up to the most recent foreground activity by pressing a Back button or similar interface feature.

Activity States Figure 4.17 provides a simplified view of the state transition diagram of an activity. Keep in mind there may be multiple activities in the application, each one at its own particular point on the state transition diagram. When a new activity is launched, the application software performs a series of API calls to the

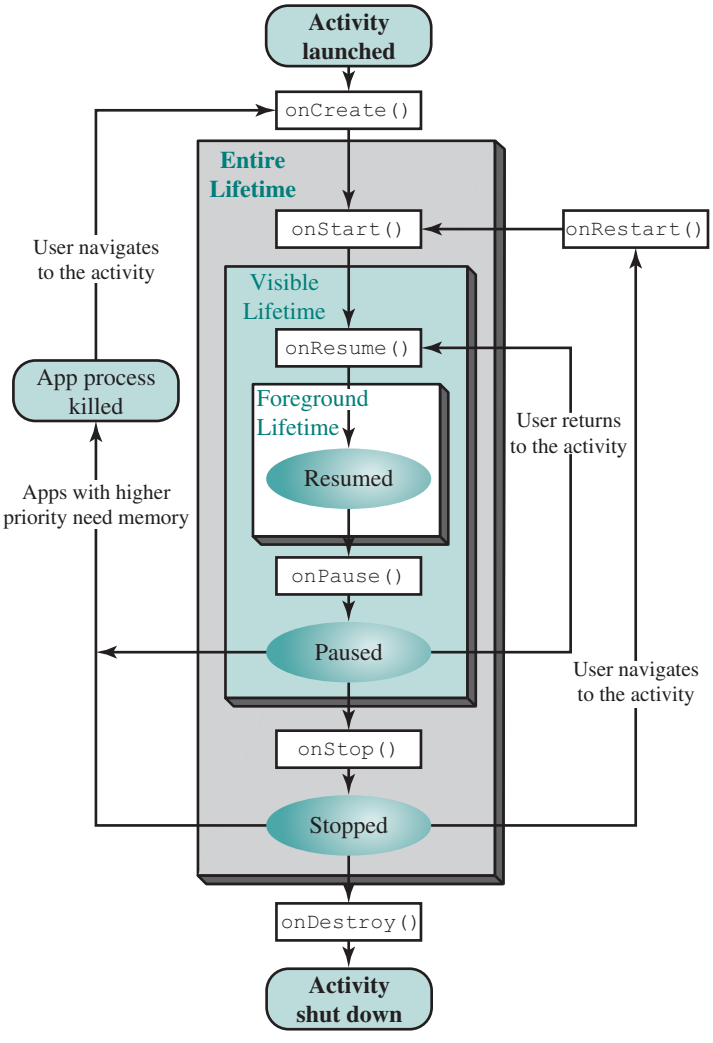


Figure 4.17 Activity State Transition Diagram

Activity Manager (Figure 2.20): `onCreate()` does the static setup of the activity, including any data structure initialization; `onStart()` makes the activity visible to the user on the screen; `onResume()` passes control to the activity so user input goes to the activity. At this point the activity is in the Resumed state. This is referred to as the *foreground lifetime* of the activity. During this time, the activity is in front of all other activities on screen and has user input focus.

A user action may invoke another activity within the application. For example, during the execution of the e-mail application, when the user selects an e-mail, a new activity opens to view that e-mail. The system responds to such an activity with the onPause() system call, which places the currently running activity on the stack, putting it in the Paused state. The application then creates a new activity, which will enter the Resumed state.

At any time, a user may terminate the currently running activity by means of the Back button, closing a window, or some other action relevant to this activity. The application then invokes `onStop(0)` to stop the activity. The application then pops the activity that is on the top of the stack and resumes it. The Resumed and Paused states together constitute the *visible lifetime* of the activity. During this time, the user can see the activity on-screen and interact with it.

If the user leaves one application to go to another, for example, by going to the Home screen, the currently running activity is paused and then stopped. When the user resumes this application, the stopped activity, which is on top of the back stack, is restarted and becomes the foreground activity for the application.

KILLING AN APPLICATION If too many things are going on, the system may need to recover some of main memory to maintain responsiveness. In that case, the system will reclaim memory by killing one or more activities within an application and also terminating the process for that application. This frees up memory used to manage the process as well as memory to manage the activities that were killed. However, the application itself still exists. The user is unaware of its altered status. If the user returns to that application, it is necessary for the system to recreate any killed activities as they are invoked.

The system kills applications in a stack-oriented style: So it will kill least recently used apps first. Apps with foregrounded services are extremely unlikely to be killed.

Processes and Threads

The default allocation of processes and threads to an application is a single process and a single thread. All of the components of the application run on the single thread of the single process for that application. To avoid slowing down the user interface when slow and/or blocking operations occur in a component, the developer can create multiple threads within a process and/or multiple processes within an application. In any case, all processes and their threads for a given application execute within the same virtual machine.

In order to reclaim memory in a system that is becoming heavily loaded, the system may kill one or more processes. As was discussed in the preceding section, when a process is killed, one or more of the activities supported by that process are also killed. A precedence hierarchy is used to determine which process or processes to kill in order

to reclaim needed resources. Every process exists at a particular level of the hierarchy at any given time, and processes are killed beginning with the lowest precedence first. The levels of the hierarchy, in descending order of precedence, are as follows:

- **Foreground process:** A process that is required for what the user is currently doing. More than one process at a time can be a foreground process. For example, both the process that hosts the activity with which the user is interacting (activity in Resumed state), and the process that hosts a service that is bound to the activity with which the user is interacting, are foreground processes.
- **Visible process:** A process that hosts a component that is not in the foreground, but still visible to the user.
- **Service process:** A process running a service that does not fall into either of the higher categories. Examples include playing music in the background or downloading data on the network.
- **Background process:** A process hosting an activity in the Stopped state.
- **Empty process:** A process that doesn't hold any active application components. The only reason to keep this kind of process alive is for caching purposes, to improve startup time the next time a component needs to run in it.

4.8 MAC OS X GRAND CENTRAL DISPATCH

As was mentioned in Chapter 2, Mac OS X Grand Central Dispatch (GCD) provides a pool of available threads. Designers can designate portions of applications, called blocks, that can be dispatched independently and run concurrently. The OS will provide as much concurrency as possible based on the number of cores available and the thread capacity of the system. Although other operating systems have implemented thread pools, GCD provides a qualitative improvement in ease of use and efficiency [LEVI16].

A block is a simple extension to C or other languages, such as C++. The purpose of defining a block is to define a self-contained unit of work, including code plus data. Here is a simple example of a block definition:

```
x = ^{printf("hello world\n");}
```

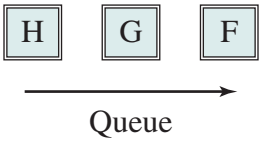
A block is denoted by a caret at the start of the function, which is enclosed in curly brackets. The above block definition defines `x` as a way of calling the function, so that invoking the function `x()` would print the words *hello world.*

Blocks enable the programmer to encapsulate complex functions, together with their arguments and data, so that they can easily be referenced and passed around in a program, much like a variable. Symbolically:

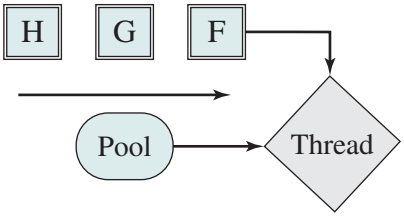
$$\boxed{F} = F + \text{data}$$

Blocks are scheduled and dispatched by means of queues. The application makes use of system queues provided by GCD and may also set up private queues. Blocks are put onto a queue as they are encountered during program execution. GCD then uses those queues to describe concurrency, serialization, and callbacks. Queues are lightweight user-space data structures, which generally makes them far

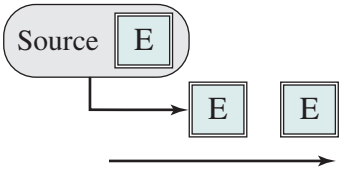
more efficient than manually managing threads and locks. For example, this queue has three blocks:



Depending on the queue and how it is defined, GCD treats these blocks either as potentially concurrent activities, or as serial activities. In either case, blocks are dispatched on a first-in-first-out basis. If this is a concurrent queue, then the dispatcher assigns F to a thread as soon as one is available, then G, then H. If this is a serial queue, the dispatcher assigns F to a thread, then only assigns G to a thread after F has completed. The use of predefined threads saves the cost of creating a new thread for each request, reducing the latency associated with processing a block. Thread pools are automatically sized by the system to maximize the performance of the applications using GCD while minimizing the number of idle or competing threads.



In addition to scheduling blocks directly, the application can associate a single block and queue with an event source, such as a timer, network socket, or file descriptor. Every time the source issues an event, the block is scheduled if it is not already running. This allows rapid response without the expense of polling or "parking a thread" on the event source.



An example from [SIRA09] indicates the ease of using GCD. Consider a document-based application with a button that, when clicked, will analyze the current document and display some interesting statistics about it. In the common case, this analysis should execute in under a second, so the following code is used to connect the button with an action:

```
- (Inaction)analyzeDocument:(NSButton *)sender
{
   NSDictionary *stats = [myDoc analyze];
   [myModel setDict:stats];
```

```
    [myStatsView setNeedsDisplay:YES];
    [stats release];
}
```

The first line of the function body analyzes the document, the second line updates the application's internal state, and the third line tells the application that the statistics view needs to be updated to reflect this new state. This code, which follows a common pattern, is executed in the main thread. The design is acceptable so long as the analysis does not take too long, because after the user clicks the button, the main thread of the application needs to handle that user input as fast as possible so it can get back to the main event loop to process the next user action. But if the user opens a very large or complex document, the analyze step may take an unacceptably long amount of time. A developer may be reluctant to alter the code to meet this unlikely event, which may involve application-global objects, thread management, callbacks, argument marshalling, context objects, new variables, and so on. But with GCD, a modest addition to the code produces the desired result:

```
- (IBAction)analyzeDocument:(NSButton *)sender
    {dispatch_async(dispatch_get_global_queue(0, 0), ^{
       NSDictionary *stats = [myDoc analyze];
       dispatch_async(dispatch_get_main_queue(), ^{
           [myModel setDict:stats];
           [myStatsView setNeedsDisplay:YES];
           [stats release];
         });
     });
}
```

All functions in GCD begin with `dispatch_`. The outer `dispatch_async()` call puts a task on a global concurrent queue. This tells the OS that the block can be assigned to a separate concurrent queue, off the main queue, and executed in parallel. Therefore, the main thread of execution is not delayed. When the analyze function is complete, the inner `dispatch_async()` call is encountered. This directs the OS to put the following block of code at the end of the main queue, to be executed when it reaches the head of the queue. So, with very little work on the part of the programmer, the desired requirement is met.

4.9 SUMMARY

Some operating systems distinguish the concepts of process and thread, the former related to resource ownership, and the latter related to program execution. This approach may lead to improved efficiency and coding convenience. In a multithreaded system, multiple concurrent threads may be defined within a single process. This may be done using either user-level threads or kernel-level threads. User-level

threads are unknown to the OS and are created and managed by a threads library that runs in the user space of a process. User-level threads are very efficient because a mode switch is not required to switch from one thread to another. However, only a single user-level thread within a process can execute at a time, and if one thread blocks, the entire process is blocked. Kernel-level threads are threads within a process that are maintained by the kernel. Because they are recognized by the kernel, multiple threads within the same process can execute in parallel on a multiprocessor and the blocking of a thread does not block the entire process. However, a mode switch is required to switch from one thread to another.

4.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

application	message	thread
fiber	multithreading	thread pool
jacketing	namespaces	user-level thread
job object	port	user-mode scheduling (UMS)
kernel-level thread	process	
lightweight process	task	

Review Questions

4.1. Table 3.5 lists typical elements found in a process control block for an unthreaded OS. Of these, which should belong to a thread control block, and which should belong to a process control block for a multithreaded system?

4.2. List reasons why a mode switch between threads may be cheaper than a mode switch between processes.

4.3. What are the two separate and potentially independent characteristics embodied in the concept of process?

4.4. Give four general examples of the use of threads in a single-user multiprocessing system.

4.5. How is a thread different from a process?

4.6. What are the advantages of using multithreading instead of multiple processes?

4.7. List some advantages and disadvantages of using kernel-level threads.

4.8. Explain the concept of threads in the case of the Clouds operating system.

Problems

4.1. The use of multithreading improves the overall efficiency and performance of the execution of an application or program. However, not all programs are suitable for multithreading. Can you give some examples of programs where a multithreaded solution fails to improve on the performance of a single-threaded solution? Also give some examples where the performance improves when multiple threads are used in place of single threads.

4.2. Suppose a program has a main routine that calls two sub-routines. The sub-routines can be executed in parallel. Give two possible approaches to implement this program, one using threads and the other without.

4.3. OS/2 from IBM is an obsolete OS for PCs. In OS/2, what is commonly embodied in the concept of process in other operating systems is split into three separate types of entities: session, processes, and threads. A session is a collection of one or more processes associated with a user interface (keyboard, display, and mouse). The session represents an interactive user application, such as a word processing program or a spreadsheet. This concept allows the personal computer user to open more than one application, giving each one or more windows on the screen. The OS must keep track of which window, and therefore which session, is active, so that keyboard and mouse input are routed to the appropriate session. At any time, one session is in foreground mode, with other sessions in background mode. All keyboard and mouse input is directed to one of the processes of the foreground session, as dictated by the applications. When a session is in foreground mode, a process performing video output sends it directly to the hardware video buffer and then to the user's display. When the session is moved to the background, the hardware video buffer is saved to a logical video buffer for that session. While a session is in background, if any of the threads of any of the processes of that session executes and produces screen output, that output is directed to the logical video buffer. When the session returns to foreground, the screen is updated to reflect the current contents of the logical video buffer for the new foreground session.

There is a way to reduce the number of process-related concepts in OS/2 from three to two. Eliminate sessions, and associate the user interface (keyboard, mouse, and display) with processes. Thus, one process at a time is in foreground mode. For further structuring, processes can be broken up into threads.

a. What benefits are lost with this approach?

b. If you go ahead with this modification, where do you assign resources (memory, files, etc.): at the process or thread level?

4.4. Consider an environment in which there is a one-to-one mapping between user-level threads and kernel-level threads that allows one or more threads within a process to issue blocking system calls while other threads continue to run. Explain why this model can make multithreaded programs run faster than their single-threaded counterparts on a uniprocessor computer.

4.5. An application has 20% of code that is inherently serial. Theoretically, what will its maximum speedup be if it is run on a multicore system with four processors?

4.6. The OS/390 mainframe operating system is structured around the concepts of address space and task. Roughly speaking, a single address space corresponds to a single application and corresponds more or less to a process in other operating systems. Within an address space, a number of tasks may be generated and executed concurrently; this corresponds roughly to the concept of multithreading. Two data structures are key to managing this task structure. An address space control block (ASCB) contains information about an address space needed by OS/390 whether or not that address space is executing. Information in the ASCB includes dispatching priority, real and virtual memory allocated to this address space, the number of ready tasks in this address space, and whether each is swapped out. A task control block (TCB) represents a user program in execution. It contains information needed for managing a task within an address space, including processor status information, pointers to programs that are part of this task, and task execution state. ASCBs are global structures maintained in system memory, while TCBs are local structures maintained within their address space. What is the advantage of splitting the control information into global and local portions?

4.7. Many current language specifications, such as for C and C++, are inadequate for multithreaded programs. This can have an impact on compilers and the correctness

of code, as this problem illustrates. Consider the following declarations and function definition:

```
int global_positives = 0;
typedef struct list {
   struct list *next;
   double val;
} * list;
void count_positives(list l)
{
   list p;
   for (p = l; p; p = p -> next)
      if (p -> val > 0.0)
         ++global_positives;
}
```

Now consider the case in which thread A performs

```
count_positives(<list containing only negative values>);
```

while thread B performs

```
++global_positives;
```

a. What does the function do?
b. The C language only addresses single-threaded execution. Does the use of two parallel threads create any problems or potential problems?

4.8. But some existing optimizing compilers (including gcc, which tends to be relatively conservative) will "optimize" `count_positives` to something similar to

```
void count_positives(list l)
{
     list p;
     register int r;
r = global_positives;
   for (p = l; p; p = p -> next)
      if (p -> val > 0.0) ++r;
   global_positives = r;
}
```

What problem or potential problem occurs with this compiled version of the program if threads A and B are executed concurrently?

4.9. Consider the following code using the POSIX Pthreads API:

thread2.c

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int myglobal;
    void *thread_function(void *arg) {
       int i,j;
```

```
        for ( i=0; i<20; i++ ) {
           j=myglobal;
           j=j+1;
           printf(".");
           fflush(stdout);
           sleep(1);
           myglobal=j;
        }
        return NULL;
}

int main(void) {
   pthread_t mythread;
   int i;
   if ( pthread_create( &mythread, NULL, thread_function,
       NULL) ) {
      printf(ldquo;error creating thread.");
      abort();
   }
for ( i=0; i<20; i++) {
   myglobal=myglobal+1;
   printf("o");
   fflush(stdout);
   sleep(1);
}
if ( pthread_join ( mythread, NULL ) ) {
   printf("error joining thread.");
abort();
}
printf("\nmyglobal equals %d\n",myglobal);
exit(0);
}
```

In `main()` we first declare a variable called mythread, which has a type of `pthread_t`. This is essentially an ID for a thread. Next, the `if` statement creates a thread associated with `mythread`. The call `pthread_create()` returns zero on success and a nonzero value on failure. The third argument of `pthread_create()` is the name of a function that the new thread will execute when it starts. When this `thread_function()` returns, the thread terminates. Meanwhile, the main program itself defines a thread, so there are two threads executing. The `pthread_join` function enables the main thread to wait until the new thread completes.

a. What does this program accomplish?

b. Here is the output from the executed program:

```
$ ./thread2
..o.o.o.o.oo.o.o.o.o.o.o.o.o.o..o.o.o.o.o
myglobal equals 21
```

Is this the output you would expect? If not, what has gone wrong?

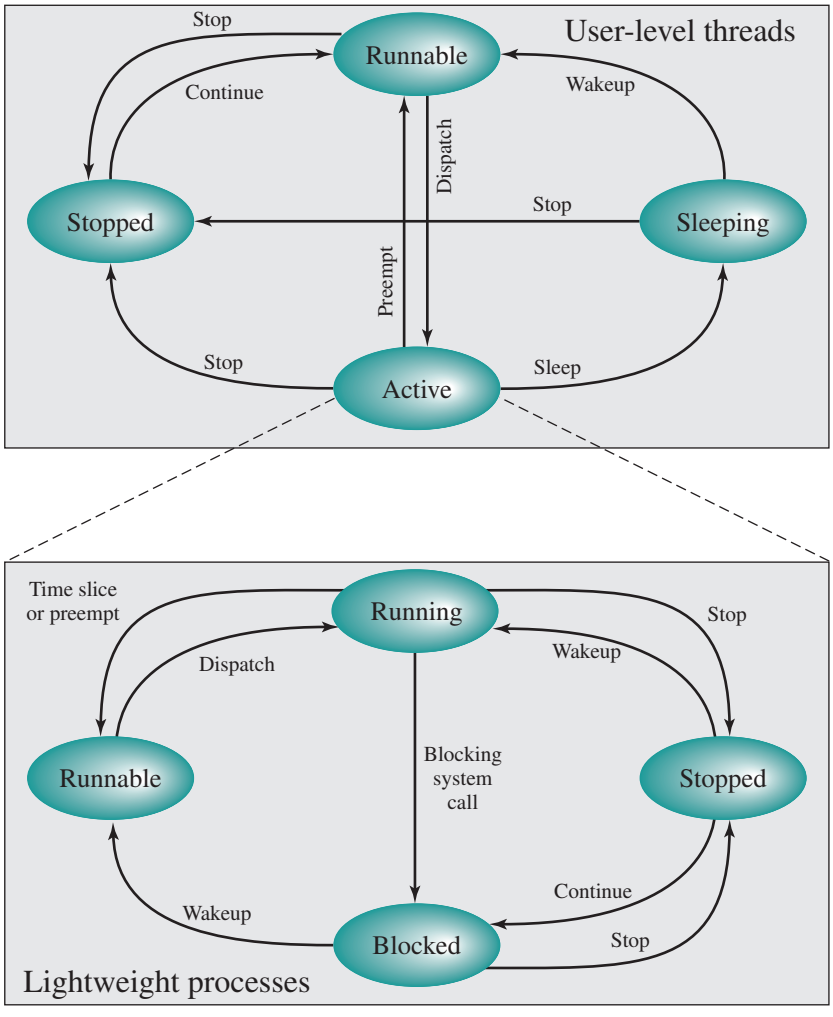


Figure 4.18 Solaris User-Level Thread and LWP States

4.10. It is sometimes required that when two threads are running, one thread should automatically preempt the other. The preempted thread can execute only when the other has run to completion. Implement the stated situation by setting priorities for the threads; use any programming language of your choice.

4.11. In Solaris 9 and Solaris 10, there is a one-to-one mapping between ULTs and LWPs. In Solaris 8, a single LWP supports one or more ULTs.

- **a.** What is the possible benefit of allowing a many-to-one mapping of ULTs to LWPs?
- **b.** In Solaris 8, the thread execution state of a ULT is distinct from that of its LWP. Explain why.
- **c.** Figure 4.18 shows the state transition diagrams for a ULT and its associated LWP in Solaris 8 and 9. Explain the operation of the two diagrams and their relationships.

4.12. Explain the rationale for the Uninterruptible state in Linux.