CHAPTER 4

THREADS, SMP, AND MICROKERNELS

This chapter examines some more advanced concepts related to process management, which are found in a number of contemporary operating systems. First, 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 one relating to execution. This distinction has led to the development, in many operating systems, of a construct known as the **thread**. After examining threads, we look at **symmetric multiprocessing** (SMP). With SMP, the OS must be able to simultaneously schedule different processes on multiple processors. Finally, we introduce the concept of the **microkernel**, which is an effective means of structuring the OS to support process management and its other tasks.

4.1 PROCESSES AND THREADS

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

- **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.
- **Scheduling/execution:** The execution of a process follows an execution path (trace) through one or more programs (e.g., Figure 1.5 and Figure 1.26). 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 still 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

[1]Alas, even this degree of consistency cannot be 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.

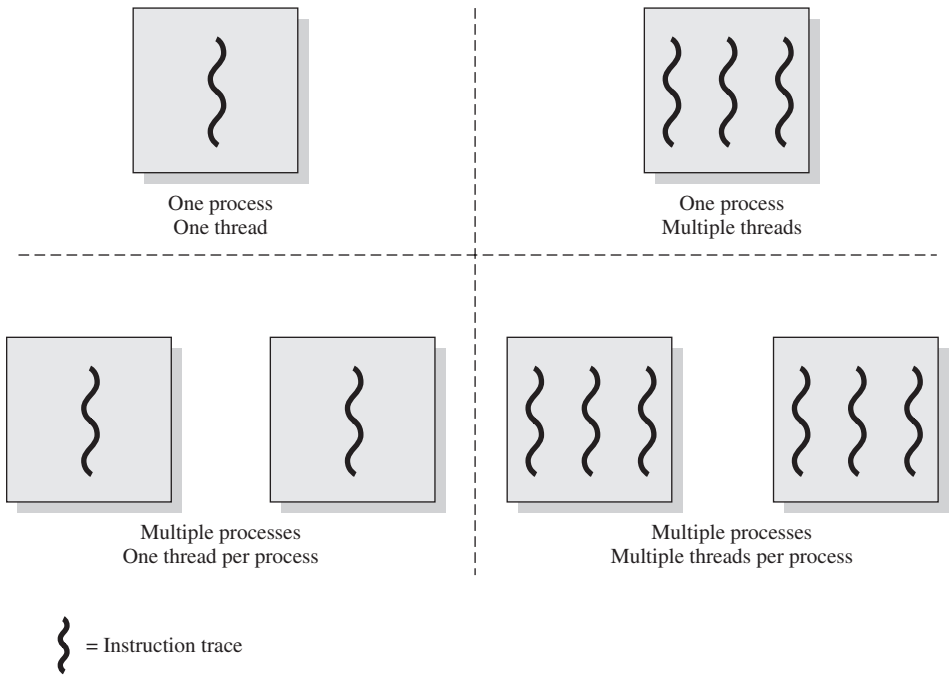

Figure 4.1 Threads and Processes [ANDE97]

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 run-time 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 support multiple threads. This approach is taken in Windows, Solaris, and many 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 are 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.

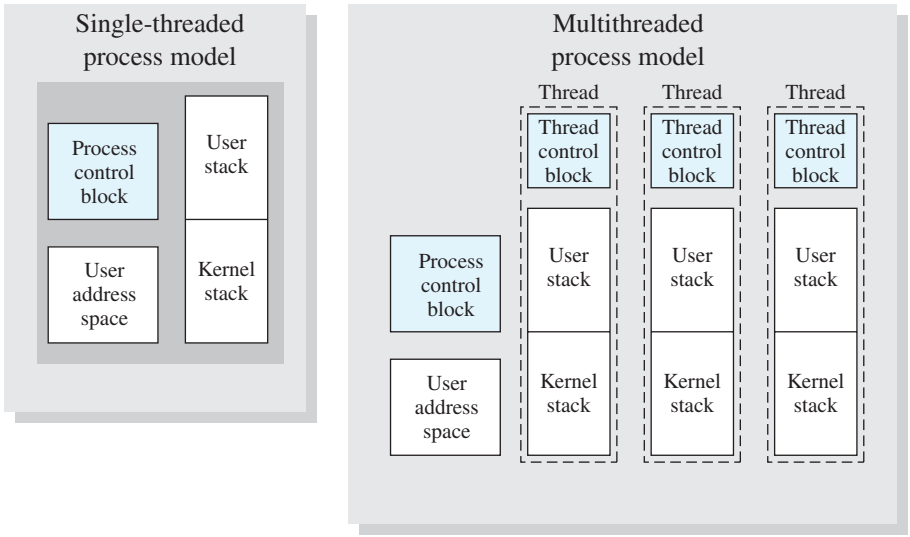

Figure 4.2 **Single Threaded and Multithreaded Process Models**

- 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 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:

- **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.
- **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.
- **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.
- **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]:

- **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.
- **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 now turn to the execution of another ready thread in the same or a different process.
- **Unblock:** When the event for which a thread is blocked occurs, the thread is moved to the Ready queue.
- **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 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 that 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

[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 program behave as if the partner program were running on the same machine. RPCs are often used for client/server applications and are discussed in Chapter 16.

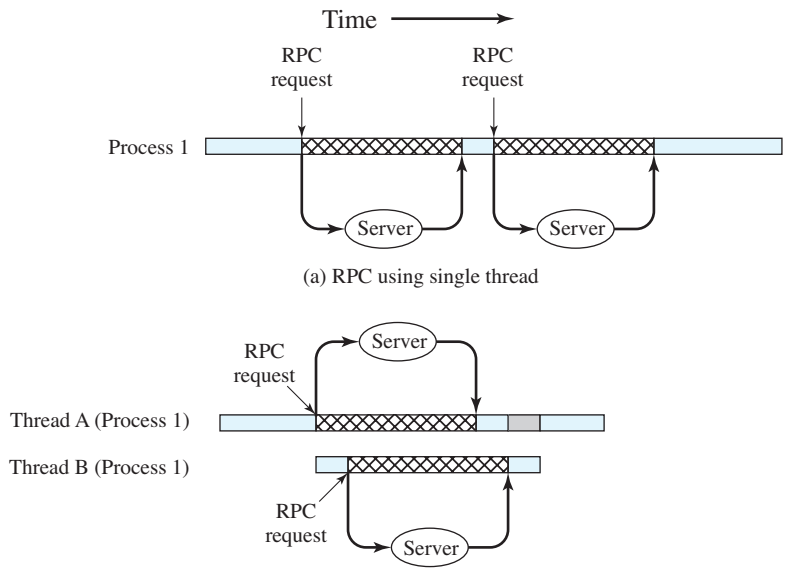

Figure 4.3 **Remote Procedure Call (RPC) Using Threads**

speedup. Note that 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.

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 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 are the subject of Chapters 5 and 6.

[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.

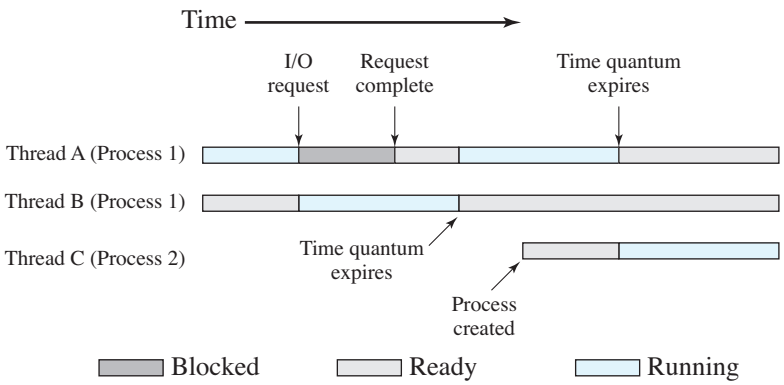

Figure 4.4 Multithreading Example on a Uniprocessor

Example—Adobe PageMaker[4]

An example of the use of threads is the Adobe PageMaker application running under a shared system. PageMaker is a writing, design, and production tool for desktop publishing. The thread structure for PageMaker used in the operating system OS/2, shown in Figure 4.5 [KRON90], was chosen to optimize the responsiveness of the application (similar thread structures would be found on other operating

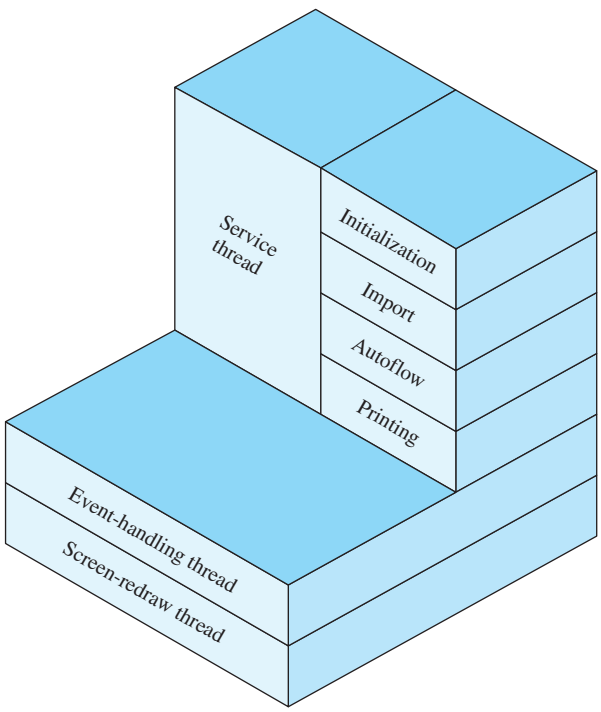

Figure 4.5 Thread Structure for Adobe PageMaker

[4]This example is somewhat dated. However, it illustrates the basic concepts using a well-documented implementation.

systems). Three threads are always active: an event-handling thread, a screen-redraw thread, and a service thread.

Generally, OS/2 is less responsive in managing windows if any input message requires too much processing. The OS/2 guidelines state that no message should require more than 0.1 s processing time. For example, calling a subroutine to print a page while processing a print command would prevent the system from dispatching any further message to any applications, slowing performance. To meet this criterion, time-consuming user operations in PageMaker—printing, importing data, and flowing text—are performed by a service thread. Program initialization is also largely performed by the service thread, which absorbs the idle time while the user invokes the dialogue to create a new document or open an existing document. A separate thread waits on new event messages.

Synchronizing the service thread and event-handling thread is complicated because a user may continue to type or move the mouse, which activates the event-handling thread, while the service thread is still busy. If this conflict occurs, PageMaker filters these messages and accepts only certain basic ones, such as window resize.

The service thread sends a message to the event-handling thread to indicate completion of its task. Until this occurs, user activity in PageMaker is restricted. The program indicates this by disabling menu items and displaying a "busy" cursor. The user is free to switch to other applications, and when the busy cursor is moved to another window, it will change to the appropriate cursor for that application.

The screen redraw function is handled by a separate thread. This is done for two reasons:

1. PageMaker does not limit the number of objects appearing on a page; thus, processing a redraw request can easily exceed the guideline of 0.1 s.
2. Using a separate thread allows the user to abort drawing. In this case, when the user rescales a page, the redraw can proceed immediately. The program is less responsive if it completes an outdated display before commencing with a display at the new scale.

Dynamic scrolling—redrawing the screen as the user drags the scroll indicator—is also possible. The event-handling thread monitors the scroll bar and redraws the margin rulers (which redraw quickly and give immediate positional feedback to the user). Meanwhile, the screen-redraw thread constantly tries to redraw the page and catch up.

Implementing dynamic redraw without the use of multiple threads places a greater burden on the application to poll for messages at various points. Multithreading allows concurrent activities to be separated more naturally in the code.

User-Level and Kernel-Level Threads

There are two broad categories of thread implementation: user-level threads (ULTs) and kernel-level threads (KLTs).[5] The latter are also referred to in the literature as *kernel-supported threads* or *lightweight processes*.

[5]The acronyms ULT and KLT are nor widely used but are introduced for conciseness.

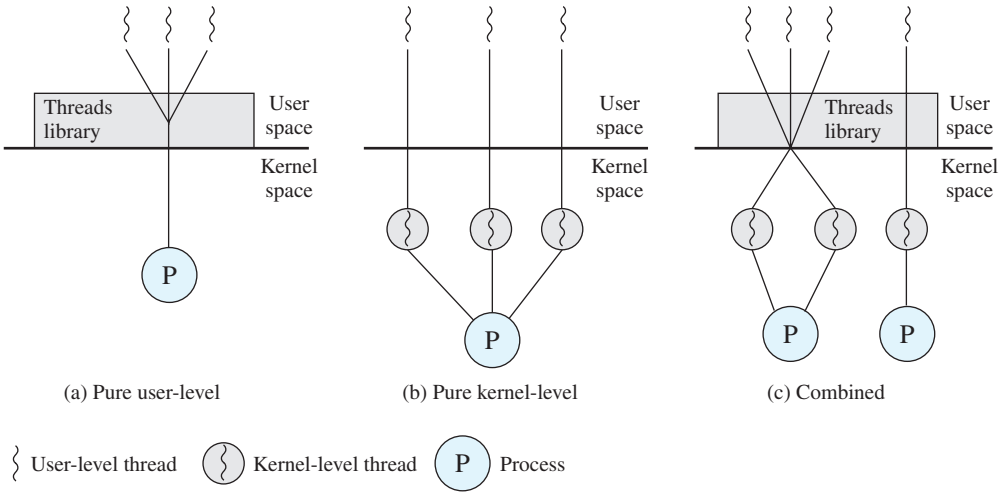

Figure 4.6 **User-Level and Kernel-Level Threads**

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. Figure 4.6a 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 that 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 4.7a. Each of the following is a possible occurrence:

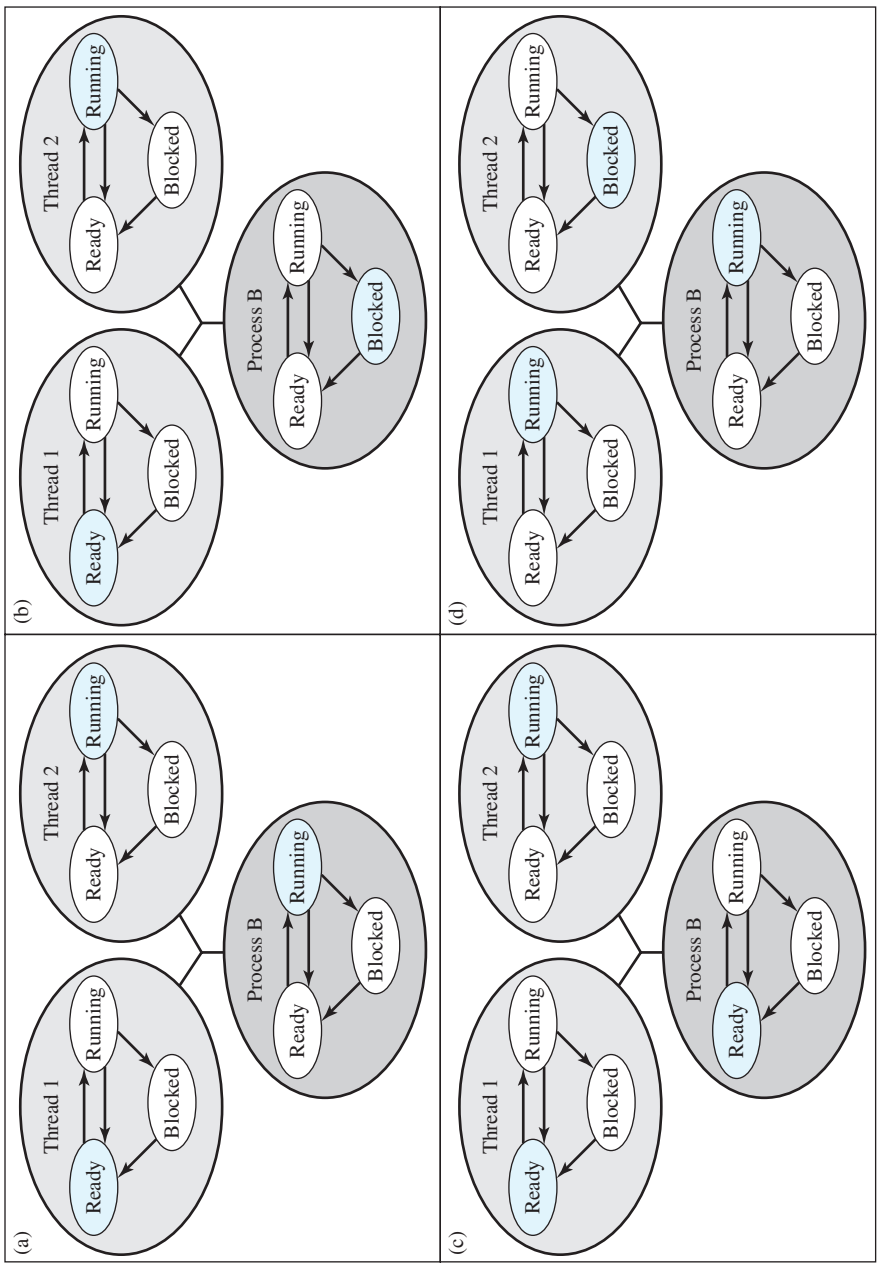

Figure 4.7 Examples of the Relationships between User-Level Thread States and Process States

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 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 4.7b.
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 4.7c.
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 4.7d.

In cases 1 and 2 (Figures 4.7b and 4.7c), 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 also all of the threads within the process are blocked.

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 later is 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 4.6b 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 the null procedure (i.e., the overhead of forking a process/thread); and Signal-Wait, the time for a

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

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 that 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 (Figure 4.6c). 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.

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

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	Combines attributes of M:1 and 1:M cases.	TRIX

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 and 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.[6]

[6]The movement of processes or threads among address spaces, or thread migration, on different machines has become a hot topic in recent years. Chapter 16 explores this topic.

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.

4.2 SYMMETRIC MULTIPROCESSING

Traditionally, the computer has been viewed as a sequential machine. Most computer programming languages require the programmer to specify algorithms as sequences of instructions. A processor executes programs by executing machine instructions in sequence and one at a time. Each instruction is executed in a sequence of operations (fetch instruction, fetch operands, perform operation, store results).

This view of the computer has never been entirely true. At the micro-operation level, multiple control signals are generated at the same time. Instruction pipelining, at least to the extent of overlapping fetch and execute operations, has been around for a long time. Both of these are examples of performing functions in parallel.

As computer technology has evolved and as the cost of computer hardware has dropped, computer designers have sought more and more opportunities for parallelism, usually to improve performance and, in some cases, to improve reliability. In this book, we examine the two most popular approaches to providing parallelism by replicating processors: symmetric multiprocessors (SMPs) and clusters. SMPs are discussed in this section; clusters are examined in Chapter 16.

SMP Architecture

It is useful to see where SMP architectures fit into the overall category of parallel processors. A taxonomy that highlights parallel processor systems first introduced by Flynn [FLYN72] is still the most common way of categorizing such systems. Flynn proposed the following categories of computer systems:

- **Single instruction single data (SISD) stream:** A single processor executes a single instruction stream to operate on data stored in a single memory.
- **Single instruction multiple data (SIMD) stream:** A single machine instruction controls the simultaneous execution of a number of processing elements on a lockstep basis. Each processing element has an associated data memory, so

that each instruction is executed on a different set of data by the different processors. Vector and array processors fall into this category.

- **Multiple instruction single data (MISD) stream:** A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. This structure has never been implemented.
- **Multiple instruction multiple data (MIMD) stream:** A set of processors simultaneously execute different instruction sequences on different data sets.

With the MIMD organization, the processors are general purpose, because they must be able to process all of the instructions necessary to perform the appropriate data transformation. MIMDs can be further subdivided by the means in which the processors communicate (Figure 4.8). If the processors each have a dedicated memory, then each processing element is a self-contained computer. Communication among the computers is either via fixed paths or via some network facility. Such a system is known as a **cluster**, or multicomputer. If the processors share a common memory, then each processor accesses programs and data stored in the shared memory, and processors communicate with each other via that memory; such a system is known as a **shared-memory multiprocessor**.

One general classification of shared-memory multiprocessors is based on how processes are assigned to processors. The two fundamental approaches are master/ slave and symmetric. With a **master/slave** architecture, the OS kernel always runs on a particular processor. The other processors may only execute user programs and perhaps OS utilities. The master is responsible for scheduling processes or threads. Once a process/thread is active, if the slave needs service (e.g., an I/O call), it must send a request to the master and wait for the service to be performed. This approach is quite simple and requires little enhancement to a uniprocessor multiprogramming OS. Conflict resolution is simplified because one processor has control of all memory and I/O resources. The disadvantages of this approach are as follows:

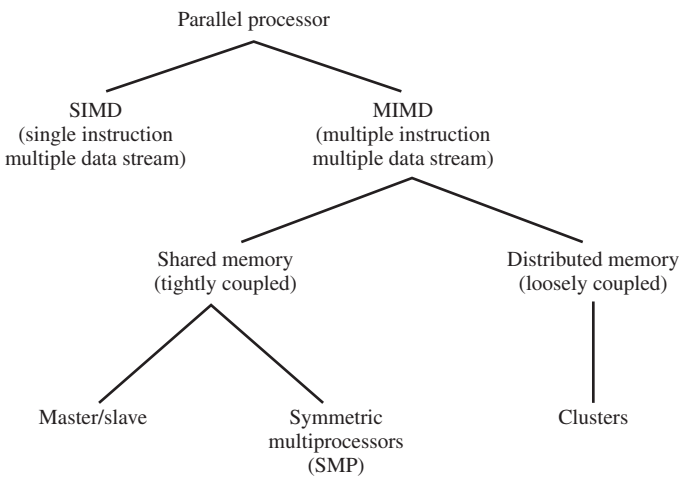

Figure 4.8 Parallel Processor Architectures

- A failure of the master brings down the whole system.
- The master can become a performance bottleneck, because it alone must do all scheduling and process management.

In a **symmetric multiprocessor (SMP)**, the kernel can execute on any processor, and typically each processor does self-scheduling from the pool of available processes or threads. The kernel can be constructed as multiple processes or multiple threads, allowing portions of the kernel to execute in parallel. The SMP approach complicates the OS. It must ensure that two processors do not choose the same process and that processes are not somehow lost from the queue. Techniques must be employed to resolve and synchronize claims to resources.

The design of both SMPs and clusters is complex, involving issues relating to physical organization, interconnection structures, interprocessor communication, OS design, and application software techniques. Our concern here, and later in our discussion of clusters (Chapter 16), is primarily with OS design issues, although in both cases we touch briefly on organization.

SMP Organization

Figure 4.9 illustrates the general organization of an SMP. There are multiple processors, each of which contains its own control unit, arithmetic-logic unit, and registers. Each processor has access to a shared main memory and the I/O devices through

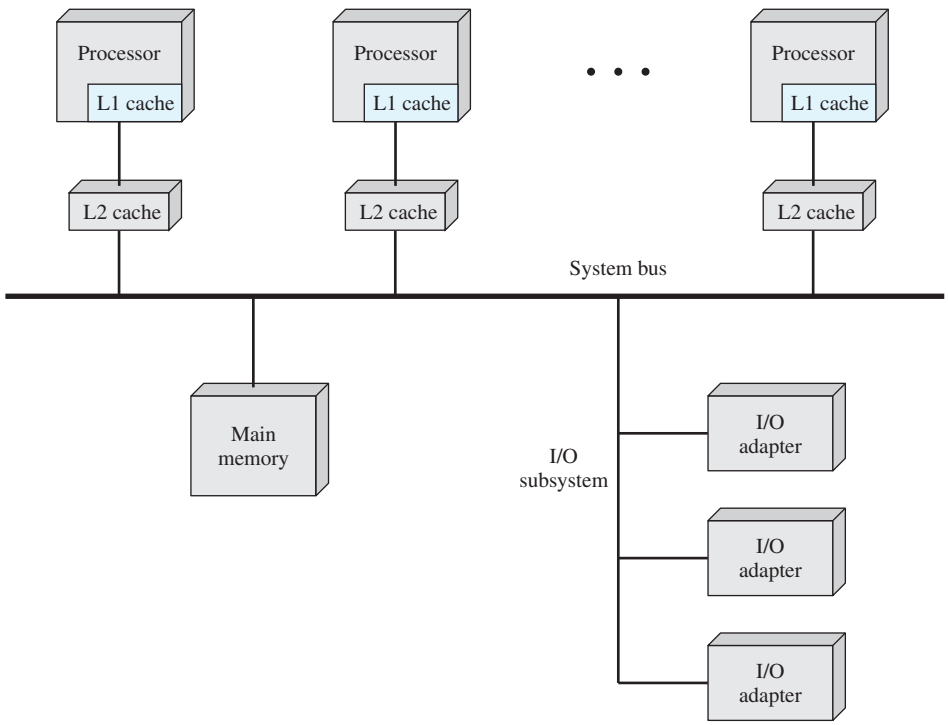

Figure 4.9 **Symmetric Multiprocessor Organization**

some form of interconnection mechanism; a shared bus is a common facility. The processors can communicate with each other through memory (messages and status information left in shared address spaces). It may also be possible for processors to exchange signals directly. The memory is often organized so that multiple simultaneous accesses to separate blocks of memory are possible.

In modern computers, processors generally have at least one level of cache memory that is private to the processor. This use of cache introduces some new design considerations. Because each local cache contains an image of a portion of main memory, if a word is altered in one cache, it could conceivably invalidate a word in another cache. To prevent this, the other processors must be alerted that an update has taken place. This problem is known as the cache coherence problem and is typically addressed in hardware rather than by the OS.[7]

Multiprocessor Operating System Design Considerations

An SMP operating system manages processor and other computer resources so that the user may view the system in the same fashion as a multiprogramming uniprocessor system. A user may construct applications that use multiple processes or multiple threads within processes without regard to whether a single processor or multiple processors will be available. Thus a multiprocessor OS must provide all the functionality of a multiprogramming system plus additional features to accommodate multiple processors. The key design issues include the following:

- **Simultaneous concurrent processes or threads:** Kernel routines need to be reentrant to allow several processors to execute the same kernel code simultaneously. With multiple processors executing the same or different parts of the kernel, kernel tables and management structures must be managed properly to avoid deadlock or invalid operations.
- **Scheduling:** Scheduling may be performed by any processor, so conflicts must be avoided. If kernel-level multithreading is used, then the opportunity exists to schedule multiple threads from the same process simultaneously on multiple processors. Multiprocessor scheduling is examined in Chapter 10.
- **Synchronization:** With multiple active processes having potential access to shared address spaces or shared I/O resources, care must be taken to provide effective synchronization. Synchronization is a facility that enforces mutual exclusion and event ordering. A common synchronization mechanism used in multiprocessor operating systems is locks, described in Chapter 5.
- **Memory management:** Memory management on a multiprocessor must deal with all of the issues found on uniprocessor computers and is discussed in Part Three. In addition, the OS needs to exploit the available hardware parallelism, such as multiported memories, to achieve the best performance. The paging mechanisms on different processors must be coordinated to enforce consistency when several processors share a page or segment and to decide on page replacement.

[7] A description of hardware-based cache coherency schemes is provided in [STAL06a].

- **Reliability and fault tolerance:** The OS should provide graceful degradation in the face of processor failure. The scheduler and other portions of the OS must recognize the loss of a processor and restructure management tables accordingly.

Because multiprocessor OS design issues generally involve extensions to solutions to multiprogramming uniprocessor design problems, we do not treat multiprocessor operating systems separately. Rather, specific multiprocessor issues are addressed in the proper context throughout this book.

4.3 MICROKERNELS

A microkernel is a small OS core that provides the foundation for modular extensions. The term is somewhat fuzzy, however, and there are a number of questions about microkernels that are answered differently by different OS design teams. These questions include how small a kernel must be to qualify as a microkernel, how to design device drivers to get the best performance while abstracting their functions from the hardware, whether to run nonkernel operations in kernel or user space, and whether to keep existing subsystem code (e.g., a version of UNIX) or start from scratch.

The microkernel approach was popularized by its use in the Mach OS, which is now the core of the Macintosh Mac OS X operating system. In theory, this approach provides a high degree of flexibility and modularity. A number of products now boast microkernel implementations, and this general design approach is likely to be seen in most of the personal computer, workstation, and server operating systems developed in the near future.

Microkernel Architecture

Operating systems developed in the mid to late 1950s were designed with little concern about structure. No one had experience in building truly large software systems, and the problems caused by mutual dependence and interaction were grossly underestimated. In these **monolithic operating systems**, virtually any procedure can call any other procedure. Such lack of structure was unsustainable as operating systems grew to massive proportions. For example, the first version of OS/360 contained over a million lines of code; Multics, developed later, grew to 20 million lines of code [DENN84]. As we discussed in Section 2.3, modular programming techniques were needed to handle this scale of software development. Specifically, **layered operating systems**[8] (Figure 4.10a) were developed in which functions are organized hierarchically and interaction only takes place between adjacent layers. With the layered approach, most or all of the layers execute in kernel mode.

Problems remain even with the layered approach. Each layer possesses considerable functionality. Major changes in one layer can have numerous effects, many difficult to trace, on code in adjacent layers (above and below). As a result, it is

[8]As usual, the terminology in this area is not consistently applied in the literature. The term *monolithic operating system* is often used to refer to both of the two types of operating systems that I have referred to as *monolithic* and *layered*.

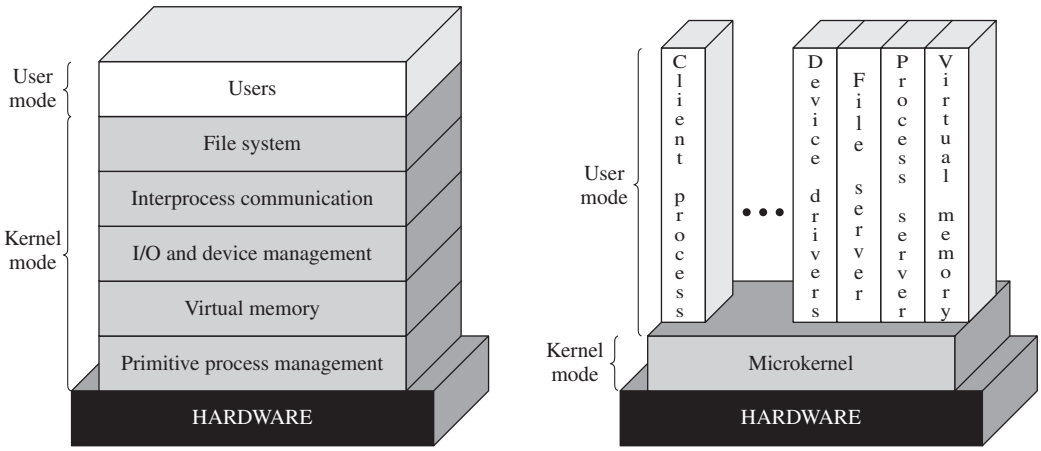

Figure 4.10 **Kernel Architecture**

difficult to implement tailored versions of a base OS with a few functions added or subtracted. And security is difficult to build in because of the many interactions between adjacent layers.

The philosophy underlying the **microkernel** is that only absolutely essential core OS functions should be in the kernel. Less essential services and applications are built on the microkernel and execute in user mode. Although the dividing line between what is in and what is outside the microkernel varies from one design to the next, the common characteristic is that many services that traditionally have been part of the OS are now external subsystems that interact with the kernel and with each other; these include device drivers, file systems, virtual memory manager, windowing system, and security services.

A microkernel architecture replaces the traditional vertical, layered stratification of an OS with a horizontal one (Figure 4.10b). OS components external to the microkernel are implemented as server processes; these interact with each other on a peer basis, typically by means of messages passed through the microkernel. Thus, the microkernel functions as a message exchange: It validates messages, passes them between components, and grants access to hardware. The microkernel also performs a protection function; it prevents message passing unless exchange is allowed.

For example, if an application wishes to open a file, it sends a message to the file system server. If it wishes to create a process or thread, it sends a message to the process server. Each of the servers can send messages to other servers and can invoke the primitive functions in the microkernel. This is a client/server architecture within a single computer.

Benefits of a Microkernel Organization

A number of advantages for the use of microkernels have been reported in the literature (e.g., [FINK04], [LIED96a], [WAYN94a]). These include

- Uniform interfaces
- Extensibility
- Flexibility
- Portability
- Reliability
- Distributed system support
- Support for object-oriented operating systems (OOOSS)

Microkernel design imposes a **uniform interface** on requests made by a process. Processes need not distinguish between kernel-level and user-level services because all such services are provided by means of message passing.

Any OS will inevitably need to acquire features not in its current design, as new hardware devices and new software techniques are developed. The microkernel architecture facilitates **extensibility**, allowing the addition of new services as well as the provision of multiple services in the same functional area. For example, there may be multiple file organizations for diskettes; each organization can be implemented as a user-level process rather than having multiple file services available in the kernel. Thus, users can choose from a variety of services the one that provides the best fit to the user's needs. With the microkernel architecture, when a new feature is added, only selected servers need to be modified or added. The impact of new or modified servers is restricted to a subset of the system. Further, modifications do not require building a new kernel.

Related to the extensibility of the microkernel architecture is its **flexibility**. Not only can new features be added to the OS, but also existing features can be subtracted to produce a smaller, more efficient implementation. A microkernel-based OS is not necessarily a small system. Indeed, the structure lends itself to adding a wide range of features. But not everyone needs, for example, a high level of security or the ability to do distributed computing. If substantial (in terms of memory requirements) features are made optional, the base product will appeal to a wider variety of users.

Intel's near monopoly of many segments of the computer platform market is unlikely to be sustained indefinitely. Thus, **portability** becomes an attractive feature of an OS. In the microkernel architecture, all or at least much of the processor-specific code is in the microkernel. Thus, changes needed to port the system to a new processor are fewer and tend to be arranged in logical groupings.

The larger the size of a software product, the more difficult it is to ensure its reliability. Although modular design helps to enhance **reliability**, even greater gains can be achieved with a microkernel architecture. A small microkernel can be rigorously tested. Its use of a small number of application programming interfaces (APIs) improves the chance of producing quality code for the OS services outside the kernel. The system programmer has a limited number of APIs to master and limited means of interacting with and therefore adversely affecting other system components.

The microkernel lends itself to **distributed system support**, including clusters controlled by a distributed OS. When a message is sent from a client to a server process, the message must include an identifier of the requested service. If a distributed

system (e.g., a cluster) is configured so that all processes and services have unique identifiers, then in effect there is a single system image at the microkernel level. A process can send a message without knowing on which computer the target service resides. We return to this point in our discussion of distributed systems in Part Six.

A microkernel architecture works well in the context of an **object-oriented operating system**. An object-oriented approach can lend discipline to the design of the microkernel and to the development of modular extensions to the OS. As a result, a number of microkernel design efforts are moving in the direction of object orientation [WAYN94b]. One promising approach to marrying the microkernel architecture with OOOS principles is the use of components [MESS96]. Components are objects with clearly defined interfaces that can be interconnected to form software in a building block fashion. All interaction between components uses the component interface. Other systems, such as Windows, do not rely exclusively or fully on object-oriented methods but have incorporated object-oriented principles into the microkernel design.

Microkernel Performance

A potential disadvantage of microkernels that is often cited is that of performance. It takes longer to build and send a message via the microkernel, and accept and decode the reply, than to make a single service call. However, other factors come into play so that it is difficult to generalize about the performance penalty, if any.

Much depends on the size and functionality of the microkernel. [LIED96a] summarizes a number of studies that reveal a substantial performance penalty for what might be called first-generation microkernels. These penalties persisted despite efforts to optimize the microkernel code. One response to this problem was to enlarge the microkernel by reintegrating critical servers and drivers back into the OS. Prime examples of this approach are Mach and Chorus. Selectively increasing the functionality of the microkernel reduces the number of user-kernel mode switches and the number of address-space process switches. However, this workaround reduces the performance penalty at the expense of the strengths of microkernel design: minimal interfaces, flexibility, and so on.

Another approach is to make the microkernel not larger but smaller. [LIED96b] argues that, properly designed, a very small microkernel eliminates the performance penalty and improves flexibility and reliability. To give an idea of the sizes involved, a typical first-generation microkernel consists of 300 Kbytes of code and 140 system call interfaces. An example of a small second-generation microkernel is L4 [HART97, LIED95], which consists of 12 Kbytes of code and 7 system calls. Experience with these systems indicates that they can perform as well or better than a layered OS such as UNIX.

Microkernel Design

Because different microkernels exhibit a range of functionality and size, no hard-and-fast rules can be stated concerning what functions are provided by the microkernel and what structure is implemented. In this section, we present a minimal set of microkernel functions and services, to give a feel for microkernel design.

The microkernel must include those functions that depend directly on the hardware and those functions needed to support the servers and applications

operating in user mode. These functions fall into the general categories of low-level memory management, interprocess communication (IPC), and I/O and interrupt management.

Low-Level Memory Management The microkernel has to control the hardware concept of address space to make it possible to implement protection at the process level. As long as the microkernel is responsible for mapping each virtual page to a physical frame, the bulk of memory management, including the protection of the address space of one process from another and the page replacement algorithm and other paging logic, can be implemented outside the kernel. For example, a virtual memory module outside the microkernel decides when to bring a page into memory and which page already in memory is to be replaced; the microkernel maps these page references into a physical address in main memory.

The concept that paging and virtual memory management can be performed external to the kernel was introduced with Mach's external pager [YOUN87]. Figure 4.11 illustrates the operation of an external pager. When a thread in the application references a page not in main memory, a page fault occurs and execution traps to the kernel. The kernel then sends a message to the pager process indicating which page has been referenced. The pager can decide to load that page and allocate a page frame for that purpose. The pager and the kernel must interact to map the pager's logical operations onto physical memory. Once the page is available, the pager sends a resume message to the application.

This technique enables a nonkernel process to map files and databases into user address spaces without invoking the kernel. Application-specific memory sharing policies can be implemented outside the kernel.

[LIED95] suggests a set of just three microkernel operations that can support external paging and virtual memory management:

- **Grant:** The owner of an address space (a process) can grant a number of its pages to another process. The kernel removes these pages from the grantor's address space and assigns them to the designated process.
- **Map:** A process can map any of its pages into the address space of another process, so that both processes have access to the pages. This creates shared memory between the two processes. The kernel maintains the assignment of these pages to the original owner but provides a mapping to permit access by other processes.

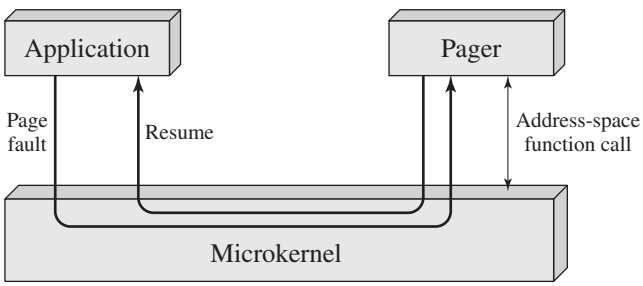

Figure 4.11 Page Fault Processing

- **Flush:** A process can reclaim any pages that were granted or mapped to other processes.

To begin, the kernel allocates all available physical memory as resources to a base system process. As new processes are created, pages from the original total address space can be granted or mapped to the new process. Such a scheme could support multiple virtual memory schemes simultaneously.

Interprocess Communication The basic form of communication between processes or threads in a microkernel OS is messages. A **message** includes a header that identifies the sending and receiving process and a body that contains direct data, a pointer to a block of data, or some control information about the process. Typically, we can think of IPC as being based on ports associated with processes. A **port** is, in essence, a queue of messages destined for a particular process; a process may have multiple ports. Associated with the port is a list of capabilities indicating what processes may communicate with this process. Port identities and capabilities are maintained by the kernel. A process can grant new access to itself by sending a message to the kernel indicating the new port capability.

A note about message passing is appropriate here. Message passing between separate processes with nonoverlapping address spaces involves memory-to-memory copying and thus is bounded by memory speeds and does not scale with processor speeds. Thus, current OS research reflects an interest in thread-based IPC and memory-sharing schemes such as page remapping (a single page shared by multiple processes).

I/O and Interrupt Management With a microkernel architecture, it is possible to handle hardware interrupts as messages and to include I/O ports in address spaces. The microkernel can recognize interrupts but does not handle them. Rather, it generates a message for the user-level process currently associated with that interrupt. Thus, when an interrupt is enabled, a particular user-level process is assigned to the interrupt and the kernel maintains the mapping. Transforming interrupts into messages must be done by the microkernel, but the microkernel is not involved in device-specific interrupt handling.

[LIED96a] suggests viewing hardware as a set of threads that have unique thread identifiers and send messages (consisting simply of the thread ID) to associated software threads in user space. A receiving thread determines whether the message comes from an interrupt and determines the specific interrupt. The general structure of such user-level code is the following:

```
driver thread:
  do
        waitFor (msg, sender);
        if (sender == my_hardware_interrupt) {
              read/write I/O ports;
              reset hardware interrupt;
        }
        else  • • •;
  while (true);
```

4.4 WINDOWS THREAD AND SMP MANAGEMENT

Windows process design is driven by the need to provide support for a variety of OS environments. Processes supported by different OS environments differ in a number of ways, including the following:

- How processes are named
- Whether threads are provided within processes
- How processes are represented
- How process resources are protected
- What mechanisms are used for interprocess communication and synchronization
- How processes are related to each other

Accordingly, the native process structures and services provided by the Windows Kernel are relatively simple and general purpose, allowing each OS subsystem to emulate a particular process structure and functionality. Important characteristics of Windows processes are the following:

- Windows processes are implemented as objects.
- An executable process may contain one or more threads.
- Both process and thread objects have built-in synchronization capabilities.

Figure 4.12, based on one in [RUSS05], illustrates the way in which a process relates to the resources it controls or uses. Each process is assigned a security

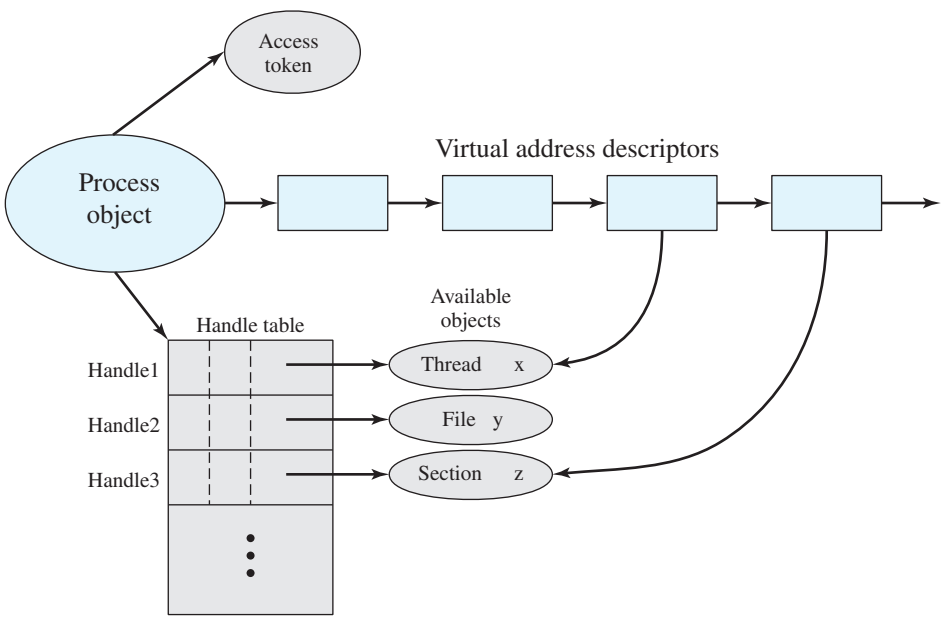

Figure 4.12 A Windows Process and Its Resources

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. One handle exists for each thread contained in this object. Figure 4.12 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.

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 opens files. A thread is a dispatchable unit of work that executes sequentially and is interruptible, so that the processor can turn to another thread.

Each Windows process is represented by an object whose general structure is shown in Figure 4.13a. Each process is defined by 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. Figure 4.13b depicts the object structure for a thread object, and Table 4.4 defines the thread object attributes. Note that 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*.

Note that one of the attributes of a thread object is context. This information enables threads to be suspended and resumed. Furthermore, it is possible to alter the behavior of a thread by altering its context when it is suspended.

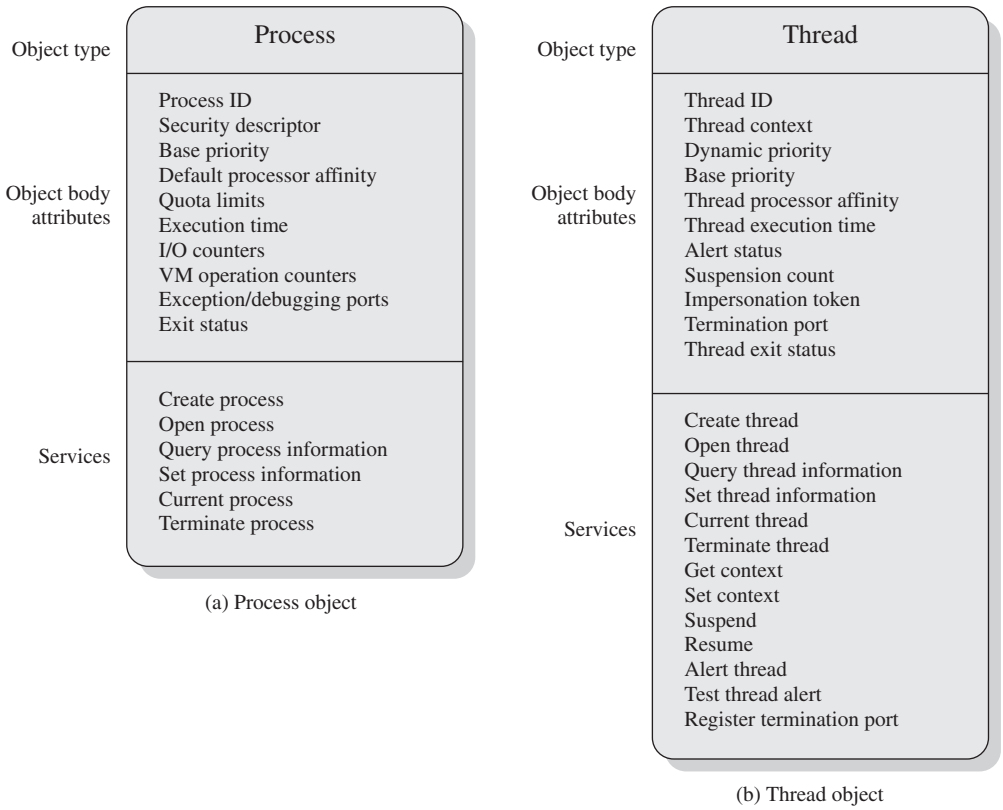

Figure 4.13 Windows Process and Thread Objects

Multithreading

Windows supports concurrency among processes because threads in different processes may execute concurrently. Moreover, multiple threads within the same process may be allocated to separate processors and execute simultaneously. 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.

Thread States

An existing Windows thread is in one of six states (Figure 4.14):

- **Ready:** May be scheduled for execution. The Kernel dispatcher keeps track of all ready threads and schedules them in priority order.

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.

- **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.

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.

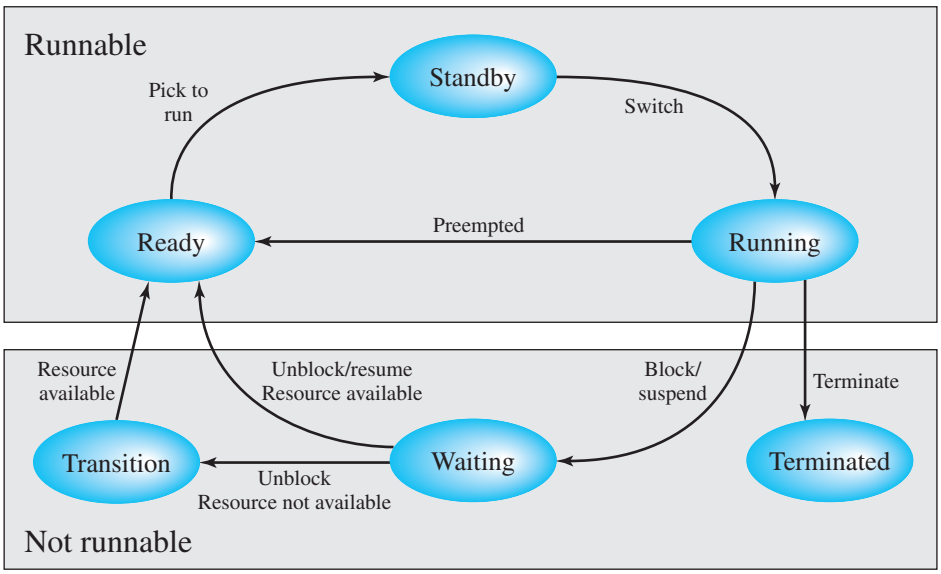

Figure 4.14 **Windows Thread States**

- **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.
- **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.
- **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.
- **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[9] 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 clients. It is the responsibility of each OS

[9]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 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 Windows executive. The executive creates a process object and returns a handle to 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 is always created with a thread. Therefore, for these operating systems, the 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 16-bit Windows and POSIX, threads are not supported. Therefore, for these operating systems, the 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 application process is implemented using a thread is not visible to the application.

When a new process is created in Win32, the new process inherits many of its attributes from the creating process. However, in the Windows environment, this process creation is done indirectly. An application client process issues its process creation request to the OS subsystem; then a process in 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 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.

Symmetric Multiprocessing Support

Windows supports an SMP hardware configuration. The threads of any process, including those of the executive, can run on any processor. In the absence of affinity restrictions, explained in the next paragraph, the microkernel assigns a ready thread to the next available processor. This assures that no processor is idle or is executing a lower-priority thread when a higher-priority thread is ready. Multiple threads from the same process can be executing simultaneously on multiple processors.

As a default, the microkernel uses the policy of **soft affinity** in assigning threads to processors: The dispatcher tries to assign a ready thread to the same processor it last ran on. This helps reuse data still in that processor's memory caches from the previous execution of the thread. It is possible for an application to restrict its thread execution to certain processors (**hard 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:

- **Process:** This is the normal UNIX process and includes the user's address space, stack, and process control block.
- **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)[10] is a user-created unit of execution within a process.
- **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.
- **Kernel threads:** These are the fundamental entities that can be scheduled and dispatched to run on one of the system processors.

Figure 4.15 illustrates the relationship among these four entities. Note that 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 consists 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.

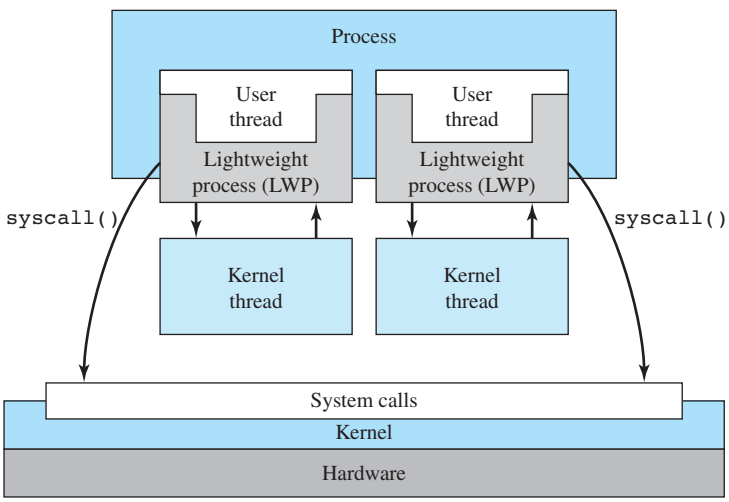

Figure 4.15 Processes and Threads in Solaris [MCDO07]

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

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

In addition, an application has access to hardware through an application programming interface (API) 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 4.16 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 the process ID; the user IDs; a 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; a memory map, which defines the address space for this process; and a 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
- 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 4.17 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

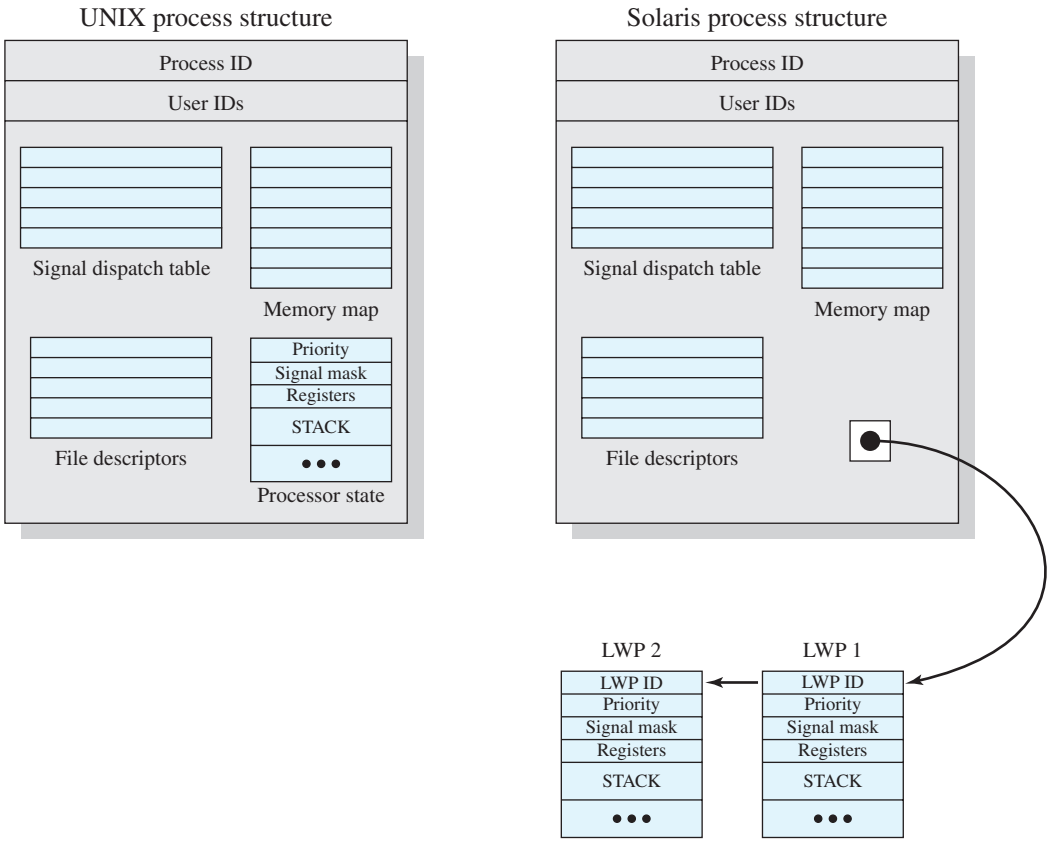

Figure 4.16 **Process Structure in Traditional UNIX and Solaris [LEW196]**

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.
- **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.

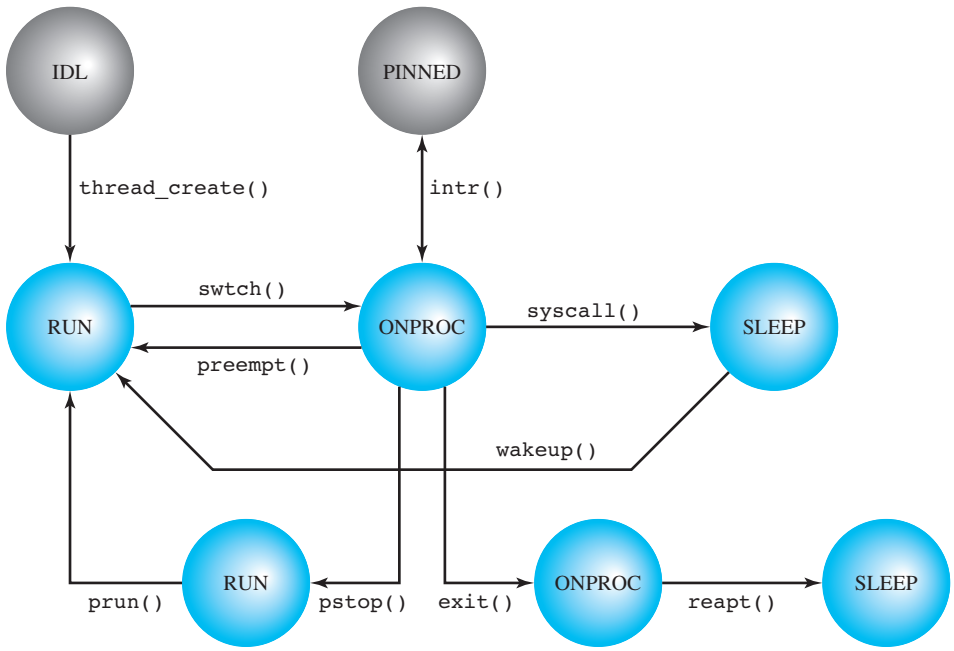

Figure 4.17 **Solaris Thread States [MCDO07]**

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 and 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 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 that 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.

WINDOWS/LINUX COMPARISON

Windows	Linux
Processes are containers for the user-mode address space, a general handle mechanism for referencing kernel objects, and threads; Threads run in a process, and are the schedulable entities	Processes are both containers and the schedulable entities; processes can share address space and system resources, making processes effectively usable as threads
Processes are created by discrete steps which construct the container for a new program and the first thread; a fork() like native API exists, but only used for POSIX compatibility	Processes created by making virtual copies with fork() and then over-writing with exec() to run a new program
Process handle table used to uniformly reference kernel objects (representing processes, threads, memory sections, synchronization, I/O devices, drivers, open files, network connections, timers, kernel transactions, etc)	Kernel objects referenced by ad hoc collection of APIs, and mechanisms – including file descriptors for open files and sockets and PIDs for processes and process groups
Up to 16 million handles on kernel objects are supported per process	Up to 64 open files/sockets are supported per process
Kernel is fully multi-threaded, with kernel preemption enabled on all systems in the original design	Few kernel processes used, and kernel preemption is a recent feature
Many system services implemented using a client/server computing, including the OS personality subsystems that run in user-mode and communicate using remote-procedure calls	Most services are implemented in the kernel, with the exception of many networking functions

- **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 counter keeps track of the amount of time a process is allowed to execute.
- **Identifiers:** Each process has a unique process identifier 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 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 4.18 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.
- **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

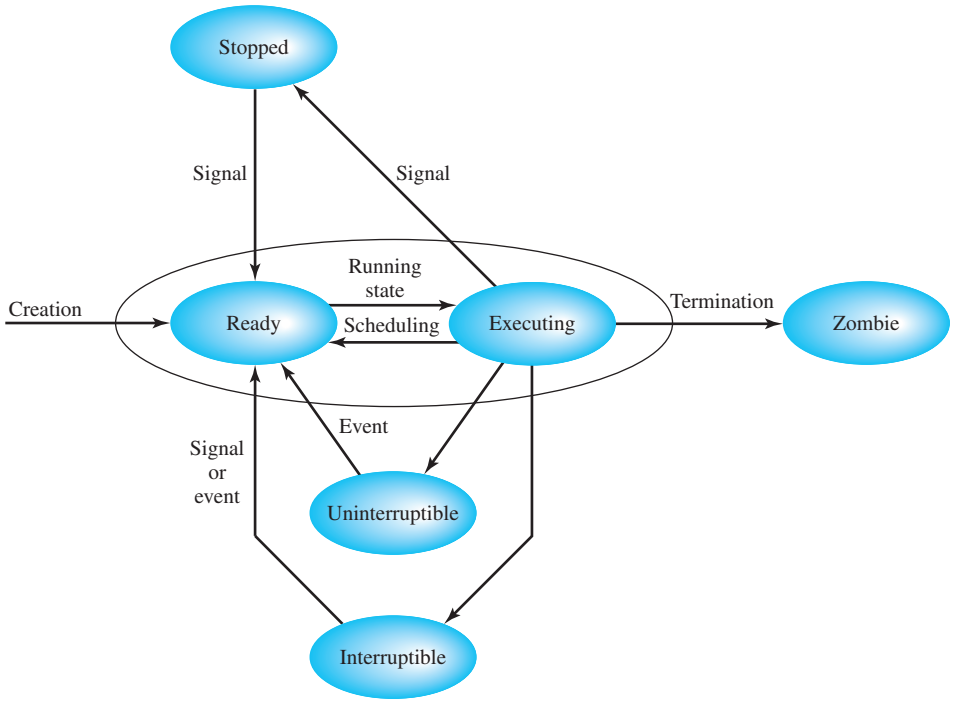

Figure 4.18 **Linux Process/Thread Model**

known as *pthread (POSIX thread) libraries*, with all of the threads mapping into a single kernel-level process.[11] 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 that 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, defined in Table 4.5. The traditional fork() system call is implemented by Linux as a clone() system call with all of the clone flags cleared.

[11]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.

Table 4.5 Linux clone () flags

CLONE_CLEARID	Clear the task ID.
CLONE_DETACHED	The parent does not want a SIGCHLD signal sent on exit.
CLONE_FILES	Shares the table that identifies the open files.
CLONE_FS	Shares the table that identifies the root directory and the current working directory, as well as the value of the bit mask used to mask the initial file permissions of a new file.
CLONE_IDLETASK	Set PID to zero, which refers to an idle task. The idle task is employed when all available tasks are blocked waiting for resources.
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Caller and new task share the same parent process.
CLONE_PTRACE	If the parent process is being traced, the child process will also be traced.
CLONE_SETTID	Write the TID back to user space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Shares the table that identifies the signal handlers.
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_VFORK	If set, the parent does not get scheduled for execution until the child invokes the *execve()* system call.
CLONE_VM	Shares the address space (memory descriptor and all page tables):

When the Linux kernel performs a 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 that 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.

4.7 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.

Symmetric multiprocessing is a method of organizing a multiprocessor system such that any process (or thread) can run on any processor; this includes kernel code and processes. An SMP architecture raises new OS design issues and provides greater performance than a uniprocessor system under similar conditions.

In its pure form, a microkernel OS consists of a very small microkernel that runs in kernel mode and that contains only the most essential and critical OS functions. Other OS functions are implemented to execute in user mode and to use the microkernel for critical services. The microkernel design leads to a flexible and highly modular implementation. However, questions remain about the performance of such an architecture.

4.8 RECOMMENDED READING

[LEWI96] and [KLEI96] provide good overviews of thread concepts and a discussion of programming strategies; the former focuses more on concepts and the latter more on programming, but both provide useful coverage of both topics. [PHAM96] discusses the Windows NT thread facility in depth. Good coverage of UNIX threads concepts is found in [ROBB04].

[MUKH96] provides a good discussion of OS design issues for SMPs. [CHAP97] contains five articles on recent design directions for multiprocessor operating systems. Worthwhile discussions of the principles of microkernel design are contained in [LIED95] and [LIED96]; the latter focuses on performance issues.

CHAP97 Chapin, S., and Maccabe, A., eds. "Multiprocessor Operating Systems: Harnessing the Power." special issue of *IEEE Concurrency*, April–June 1997.

KLEI96 Kleiman, S.; Shah, D.; and Smallders, B. *Programming with Threads*. Upper Saddle River, NJ: Prentice Hall, 1996.

LEWI96 Lewis, B., and Berg, D. *Threads Primer*. Upper Saddle River, NJ: Prentice Hall, 1996.

LIED95 Liedtke, J. "On μ-Kernel Construction." *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December 1995.

LIED96 Liedtke, J. "Toward Real Microkernels." *Communications of the ACM*, September 1996.

MUKH96 Mukherjee, B., and Karsten, S. "Operating Systems for Parallel Machines." In *Parallel Computers: Theory and Practice*. Edited by T. Casavant, P. Tvrkik, and F. Plasil. Los Alamitos, CA: IEEE Computer Society Press, 1996.

PHAM96 Pham, T., and Garg, P. *Multithreaded Programming with Windows NT*. Upper Saddle River, NJ: Prentice Hall, 1996.

ROBB04 Robbins, K., and Robbins, S. *UNIX Systems Programming: Communication, Concurrency, and Threads*. Upper Saddle River, NJ: Prentice Hall, 2004.

4.9 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

kernel-level thread (KLT)	multithreading	task
lightweight process	port	thread
message	process	user-level thread (ULT)
microkernel	symmetric multiprocessor (SMP)	
monolithic operating system		

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 What resources are typically shared by all of the threads of a process?

4.6 List three advantages of ULTs over KLTs.

4.7 List two disadvantages of ULTs compared to KLTs.

4.8 Define jacketing.

4.9 Briefly define the various architectures named in Figure 4.8.

4.10 List the key design issues for an SMP operating system.

4.11 Give examples of services and functions found in a typical monolithic OS that may be external subsystems to a microkernel OS.

4.12 List and briefly explain seven potential advantages of a microkernel design compared to a monolithic design.

4.13 Explain the potential performance disadvantage of a microkernel OS.

4.14 List three functions you would expect to find even in a minimal microkernel OS.

4.15 What is the basic form of communications between processes or threads in a microkernel OS?

Problems

4.1 It was pointed out that two advantages of using multiple threads within a process are that (1) less work is involved in creating a new thread within an existing process than in creating a new process, and (2) communication among threads within the same process is simplified. Is it also the case that a mode switch between two threads within the same process involves less work than a mode switch between two threads in different processes?

4.2 In the discussion of ULTs versus KLTs, it was pointed out that a disadvantage of ULTs is that when a ULT executes a system call, not only is that thread blocked, but also all of the threads within the process are blocked. Why is that so?

4.3 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, 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 thence to the user's screen. 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, screen) 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. LEWI96-42

4.5 If a process exits and there are still threads of that process running, will they continue to run? LEWI96-42

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 execute 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? OS2e-155

4.7 A multiprocessor with eight processors has 20 attached tape drives. There is a large number of jobs submitted to the system that each require a maximum of four tape drives to complete execution. Assume that each job starts running with only three tape drives for a long period before requiring the fourth tape drive for a short period toward the end of its operation. Also assume an endless supply of such jobs.

a. Assume the scheduler in the OS will not start a job unless there are four tape drives available. When a job is started, four drives are assigned immediately and are not released until the job finishes. What is the maximum number of jobs that can be in progress at once? What are the maximum and minimum number of tape drives that may be left idle as a result of this policy?

b. Suggest an alternative policy to improve tape drive utilization and at the same time avoid system deadlock. What is the maximum number of jobs that can be in progress at once? What are the bounds on the number of idling tape drives?

4.8 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 the parallel threads, create any problems or potential problems? login-0207

4.9 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?

4.10 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("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 essential 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 non-zero 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 that 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? POSIXthreadsexplained.pdf

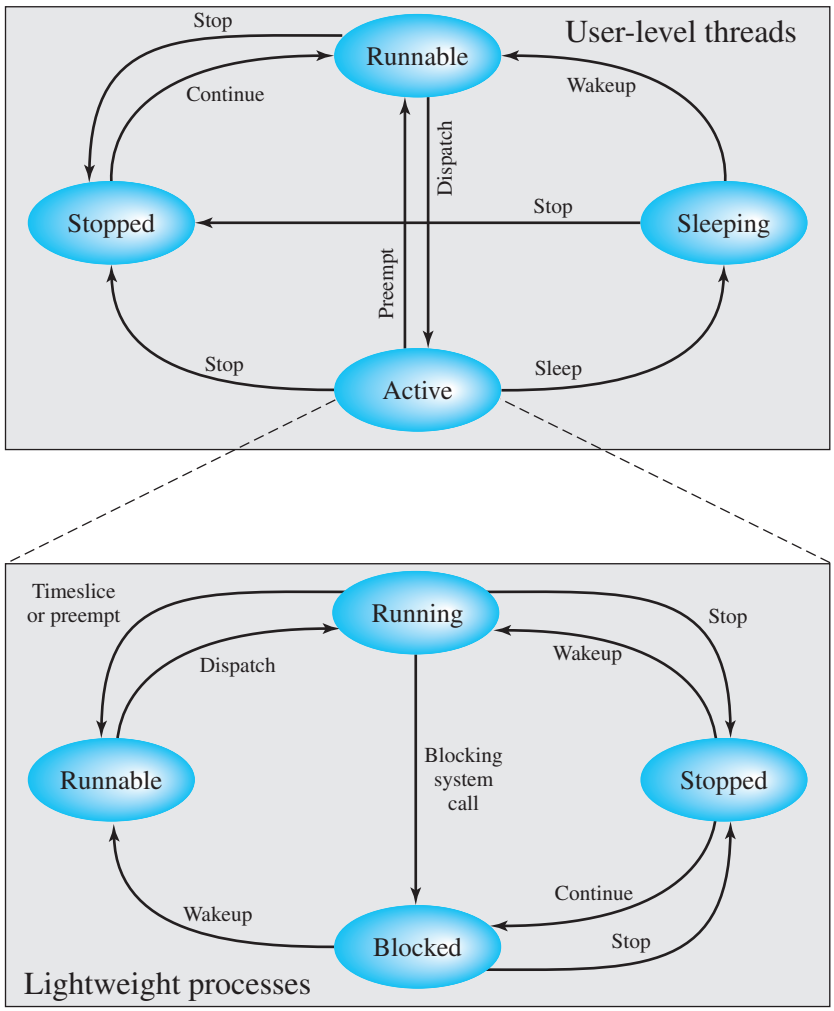

Figure 4.19 **Solaris User-Level Thread and LWP States**

4.11 The Solaris documentation states, that a ULT may yield to another thread of the same priority. Isn't it possible that there will be a runnable thread of higher priority and that therefore the yield function should result in yielding to a thread of the same or higher priority?

4.12 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 4.19 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.

4.13 Explain the rationale for the Uninterruptible state in Linux.