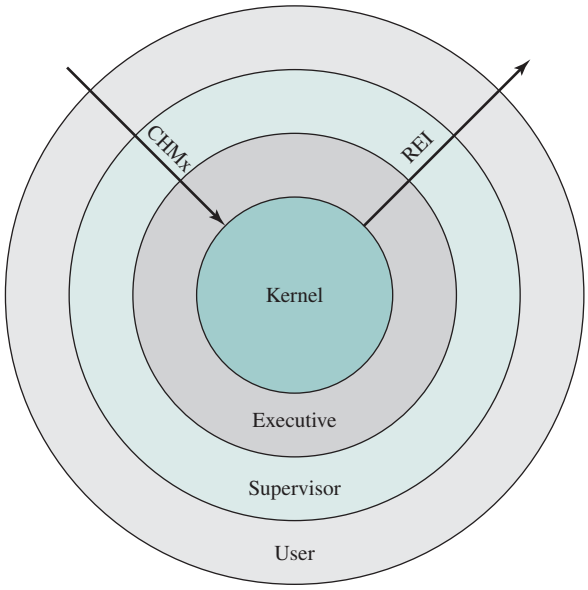


Figure 3.18 VAX/VMS Access Modes

be accessible in domain Dj but not accessible in domain Di, then we must have $j < i$. But this means that every segment accessible in Di is also accessible in Dj.

Explain clearly what the problem is that is referred to in the preceding quote.

3.9 Figure 3.8b suggests that a process can only be in one event queue at a time.

a. Is it possible that you would want to allow a process to wait on more than one event at the same time? Provide an example.

b. In that case, how would you modify the queueing structure of the figure to support this new feature?

3.10 In a number of early computers, an interrupt caused the register values to be stored in fixed locations associated with the given interrupt signal. Under what circumstances is this a practical technique? Explain why it is inconvenient in general.

3.11 In Section 3.4, it was stated that UNIX is unsuitable for real-time applications because a process executing in kernel mode may not be preempted. Elaborate.

3.12 You have executed the following C program:

```
main ()
{  int pid;
pid = fork ();
printf ("%d \n", pid);
}
```

What are the possible outputs, assuming the fork succeeded?

CHAPTER 4

THREADS

The basic idea is that the several components in any complex system will perform particular subfunctions that contribute to the overall function.

—*THE SCIENCES OF THE ARTIFICIAL,* HERBERT SIMON

LEARNING OBJECTIVES

After studying this chapter, you should be able to:

- Understand the distinction between process and thread.
- Describe the basic design issues for threads.
- Explain the difference between user-level threads and kernel-level threads.
- Describe the thread management facility in Windows 7.
- Describe the thread management facility in Solaris.
- Describe the thread management facility in Linux.

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

4.1 PROCESSES AND THREADS

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

- **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). This execution may be interleaved with that of other processes. Thus, a process has an execution state (Running, Ready, etc.) and a dispatching priority and is the entity that is scheduled and dispatched by the OS.

Some thought should convince the reader that these two characteristics are independent and could be treated independently by the OS. This is done in a number of operating systems, particularly recently developed systems. To

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

Multithreading

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

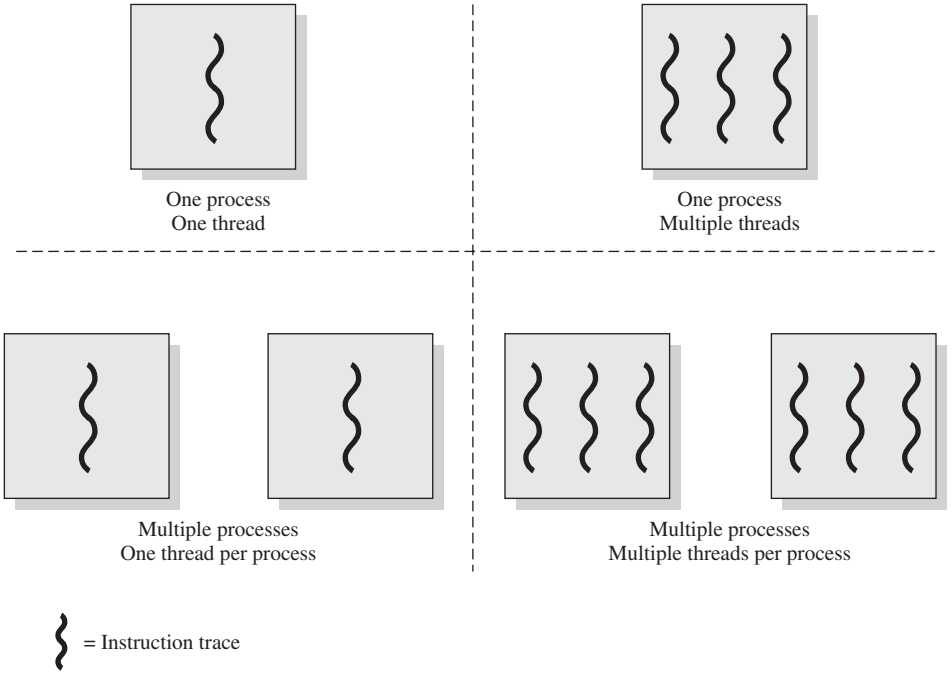


Figure 4.1 Threads and Processes [ANDE97]

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

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
- 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

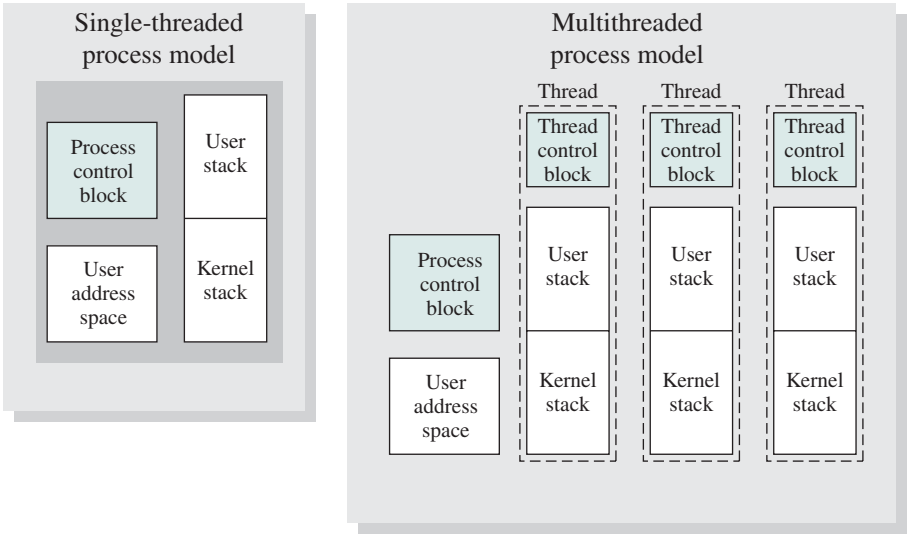


Figure 4.2 Single-Threaded and Multithreaded Process Models

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.

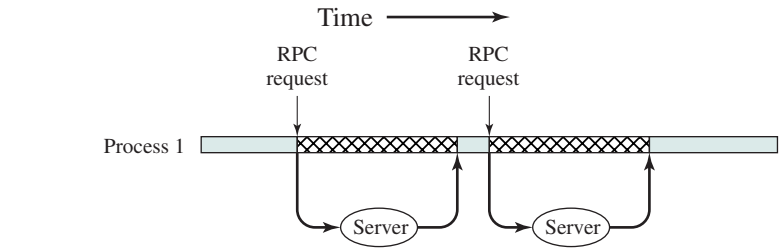
THREAD STATES As with processes, the key states for a thread are Running, Ready, and Blocked. Generally, it does not make sense to associate suspend states with threads because such states are process-level concepts. In particular, if a process is swapped out, all of its threads are necessarily swapped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

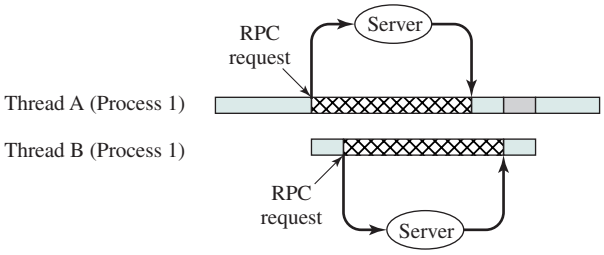
- **Spawn:** Typically, when a new process is spawned, a thread for that process is also spawned. Subsequently, a thread within a process may spawn another thread within the same process, providing an instruction pointer and arguments for the new thread. The new thread is provided with its own register 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 the program has to wait for a response from each server in turn. Rewriting the program to use a separate thread for each RPC results in a substantial speedup. Note 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.



(a) RPC using single thread



(b) RPC using one thread per server (on a uniprocessor)

Blocked, waiting for response to RPC

Blocked, waiting for processor, which is in use by Thread B

Running

Figure 4.3 Remote Procedure Call (RPC) Using Threads

[2]An RPC is a technique by which two programs, which may execute on different machines, interact using procedure call/return syntax and semantics. Both the called and calling 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.

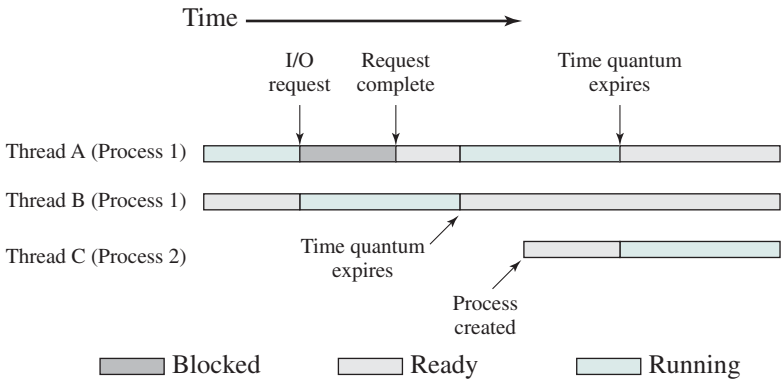


Figure 4.4 Multithreading Example on a Uniprocessor

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

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

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

4.2 TYPES OF THREADS

User-Level and Kernel-Level Threads

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

USER-LEVEL THREADS In a pure ULT facility, all of the work of thread management is done by the application and the kernel is not aware of the existence of threads. Figure 4.5a illustrates the pure ULT approach. Any application can be

[3]In this example, thread C begins to run after thread A exhausts its time quantum, even though thread B is also ready to run. The choice between B and C is a scheduling decision, a topic covered in Part Four.
[4]The acronyms ULT and KLT are not widely used but are introduced for conciseness.

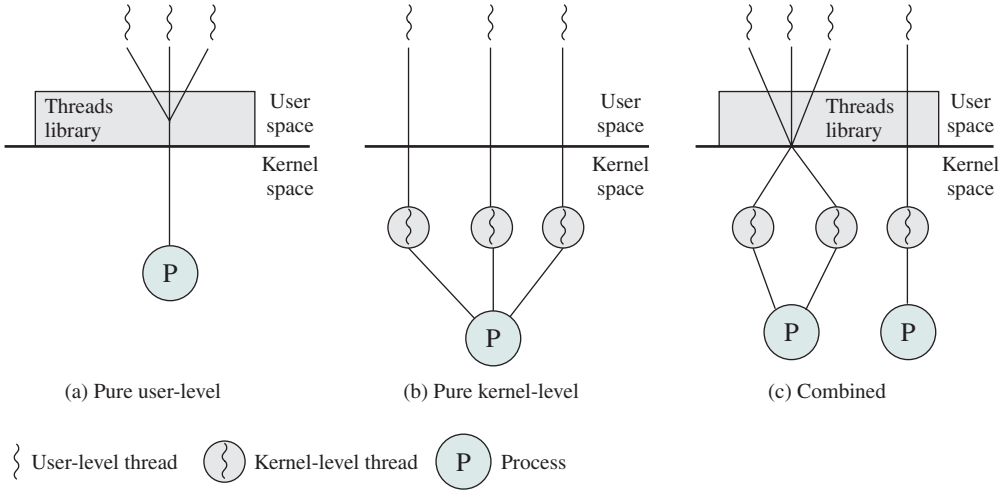


Figure 4.5 **User-Level and Kernel-Level Threads**

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 4.6a. Each of the following is a possible occurrence:

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

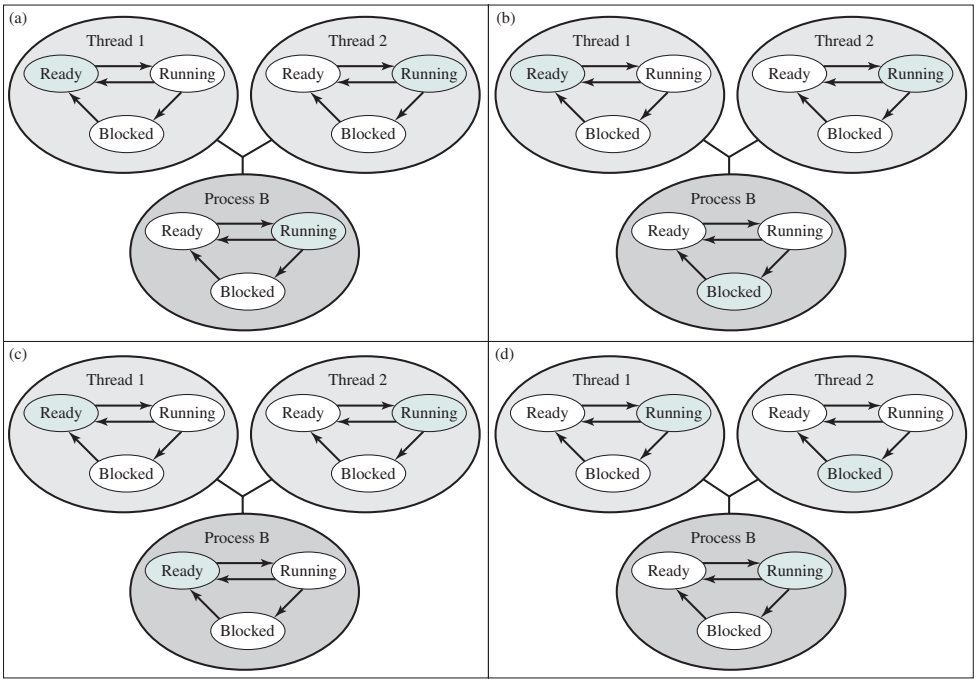


Figure 4.6 **Examples of the Relationships between User-Level Thread States and Process States**

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

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

In cases 1 and 2 (Figures 4.6b and 4.6c), when the kernel switches control back to process B, execution resumes in thread 2. Also note that a process can be interrupted, either by exhausting its time slice or by being preempted by a higher-priority process, while it is executing code in the threads library. Thus, a process may be in the midst of a thread switch from one thread to another when interrupted. When that process is resumed, execution continues within the threads library, which completes the thread switch and transfers control to another thread within that process.

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

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

There are two distinct disadvantages of ULTs compared to KLTs:

1. In a typical OS, many system calls are blocking. As a result, when a ULT executes a system call, not only is that thread blocked, but 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 4.5b depicts the pure KLT approach. The kernel maintains context information for the process as a whole and for individual threads within the process. Scheduling by the kernel is done on a thread basis. This approach overcomes the two principal drawbacks of the ULT approach. First, the kernel can simultaneously schedule multiple threads from the same process on multiple processors. Second, if one thread in a process is blocked, the kernel can schedule another thread of the same process. Another advantage of the KLT approach is that kernel routines themselves can be multithreaded.

The principal disadvantage of the KLT approach compared to the ULT approach is that the transfer of control from one thread to another within the same process requires a mode switch to the kernel. To illustrate the differences, Table 4.1 shows the results of measurements taken on a uniprocessor VAX computer running a UNIX-like OS. The two benchmarks are as follows: Null Fork, the time to create, schedule, execute, and complete a process/thread that invokes the null procedure (i.e., the overhead of forking a process/thread); and Signal-Wait, the time for a process/thread to signal a waiting process/thread and then wait on a condition (i.e., the overhead of synchronizing two processes/threads together). We see that there is an order of magnitude or more of difference between ULTs and KLTs and similarly between KLTs and processes.

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

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

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

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

Other Arrangements

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

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.[5] A notable example of this research is the Clouds operating system, and especially its kernel, known as Ra [DASG92]. Another example is the Emerald system [STEE95].

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

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

4.3 MULTICORE AND MULTITHREADING

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

Performance of Software on Multicore

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

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

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

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

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

1 Processes and Threads

A process has two characteristics:

- Resource ownership
- scheduling/execution

These two things should be kept separate and independent. A unit of scheduling is called a thread or lightweight process and a unit of resource ownership is called a process or task.

Multithreading is allowing multiple converrent paths of execution within a single process. Within a process threads share the process control block and the user address space, but each get their own thread control block, user stack, and kernel stack. All threads share the same state and resources since these are owned by the process. When a thread alters data it can fuck with all the other threads since they share memory. Similarly if a thread uses a resource (say reads a file) all threads can use it.

Benifits of theads:

- creating and terminating threads is much easier than createing and terminating processes (no memory management required)
- it takes less time to switch between threads
- it is easier to communicate between threads than it is between processes (no OS intervention required)

The most common uses of threads:

- foreground/background work: threads that render and threads that compute make a more responsive UI
- asynchronous processing
- speed of execution: threads can be blocked while the process is not
- modular program structure: group tasks by similar actions and shared resources

Threads have a similar state system as processes but they cannot be suspended. Actions that can change a thread state are:

- spawn: a new process is spanded so the threads for that process are also created, threads can also spawn threads
- block: a thread is waiting for an event (same as for processes)
- unblock: the event occurs (same as for process)
- finish: the thread completes

We need to be careful that a blocked thread doesn't block the process and thus all threads. We also need to be careful about multiple threads accessing the same resource simultaneously since they share everything.

2 Types of Threads

User level threads are handled by the user application and the kernel is not aware of them. These tend to come as libraries and are self managed.

Advantages:

- thread switching doesnt need kernel level privileges, no mode switch required
- scheduling can be application specific, can be optimized based on the situation
- can run on any OS

Disadvantages:

- when a system call is made, all threads are blocked
- cannot take advantage of multiprocessing

We can fix the thread blocking problem with jacketing, instead of a system call we make a application level call so that the application can manage blocked threads.

Kernel level threads are handled by the kernel so there is no thread management code in the user level. Here the kernel can take advantage of multiprocessing when scheduling threads and can easily deal with a single blocked thread. The main disadvantage is that switching between threads is more costly since it requires a mode switch.

The best approach is the combined approach where we have as many user level threads as we'd like and we group them so that we have a number of kernel level threads equal to the number of processors.

Thread to Process configurations:

- one to one: one devoted thread in the process
- many to one: a process owns multiple threads (most common)
- one to many: a thread can jump between multiple processes
- many to many: craziness

3 Multicore and Mulithreading