

Chapter 3

PROCESSES AND THREADS

3.1. Process Concepts

3.1.1. The Process

A process can be thought of as a program in execution. We emphasize that a program by itself is not a process. A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. A process will need certain resources – such as CPU time, memory, files, and I/O devices – to accompany its task. These resources are allocated to the process either when it is created or while it is executing.

A process is the unit of work in most systems. Such a system consists of a collection of processes: operating system processes execute system code and user processes execute user code. All these processes may execute concurrently.

A process is more than program code, which is sometimes known as the *text section*. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's register. The memory layout of a process is typically divided into multiple sections, and is shown in Figure 3.1. These sections include:

- ★ **Text section**— this includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
- ★ **Data section**— this section contains the global and static variables.
- ★ **Heap section**—memory that is dynamically allocated during program run time
- ★ **Stack section**—temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

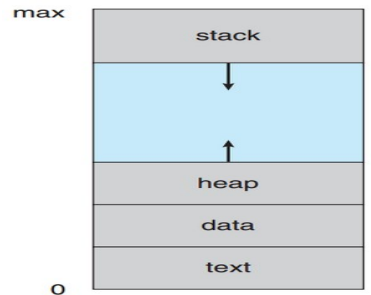


Figure 3.1 Layout of a process in memory

3.1.2. Process States

As a process executes, it changes states. The state of a process is defined in part by the current activity of that process. Each process may be in either of the following states, as shown in Figure 3.2:

- **New:** This is the initial state when a process is first started/created.
- **Ready:** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come

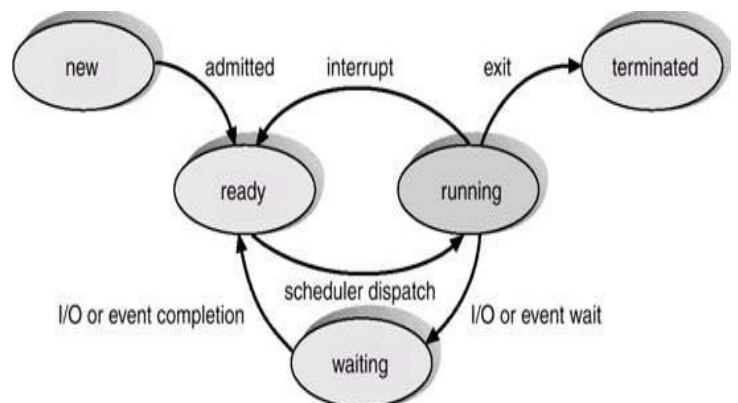


Figure 3.2 Process state diagrams

into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process

- **Running:** Instructions are being executed. Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
- **Waiting:** The process is waiting for some event to occur (such as an I/O completion or reception of a signal). Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
- **Terminated:** The process has finished execution. Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

There are four possible transitions among these states.

- **Running to block:** Occurs when a process finds out that it cannot continue to run, e.g. it tries to read from the input stream but there is no input available. In some systems the process must execute a system call for this transition while in other systems the transition is automatic with no need of system call.
- **Running to ready:** Occurs when the operating system scheduler decides that the running process runs long enough. The scheduler blocks the running process to assign the CPU to another process.
- **Ready to running:** Occurs when all processes get their fair share of the CPU and the scheduler decides that it is now the turn for a process to get the CPU.
- **Blocked to ready:** Occurs when the external event for which the process was waiting happens. If there is no process in a running state, the process is transferred to running state automatically.

3.1.3. Process Control Block

Each process is represented in the operating system by a **process control block (PCB)** – also called a **task control block**, as shown in Figure 3.3 A PCB contains many pieces of information associated with a specific process, including these:

- ★ **Process state:** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
- ★ **Program counter:** The counter indicates the address of the next instruction to be executed for this process.
- ★ **CPU registers:** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers and general-purpose registers, plus any condition code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterwards.
- ★ **CPU Scheduling information:** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- ★ **Memory-management information:** This information may include such information such as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

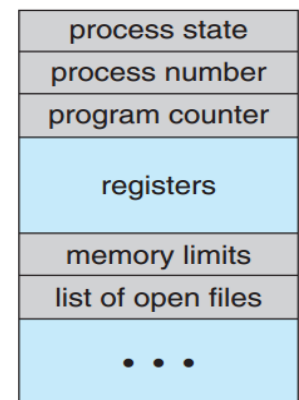


Figure 3.3 Process control block (PCB)

- ★ **Accounting information:** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- ★ **I/O status information:** The information includes the list of I/O devices allocated to the process, a list of open files, and so on.

In brief, the PCB simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data. The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

3.1.4. Threads

The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time. Thus, the user cannot simultaneously type in characters and run the spell checker.

Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. A multithreaded word processor could, for example, assign one thread to manage user input while another thread runs the spell checker. On systems that support threads, the PCB is expanded to include information for each thread. Other changes throughout the system are also needed to support threads.

3.2. Process Scheduling

The objective of **multiprogramming** is to have some process running all the time so as to maximize CPU utilization. The objective of **time-sharing** is to switch the CPU among processors so frequently that users can interact with each program while it is running. A **uniprocessor** system can have only one running process at a given time. If more processes exist, the rest must wait until the CPU is free and can be rescheduled. Switching the CPU from one process to another requires saving of the context of the current process and loading the state of the new process, as shown in Figure 3.4. That is called **context switching**.

The **process scheduling** is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

3.2.1. Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The appropriate scheduler carries out this selection process. Schedulers are special system software, which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide

which process to run. Schedulers are of three types – Long-Term Scheduler, Short-Term Scheduler and Medium-Term Scheduler.

Long Term Scheduler

The **Long-term scheduler** (or job scheduler) selects which processes should be brought into the ready queue, from the job pool that is the list of all jobs in the system. A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling.

The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long-term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

The **Short-term scheduler** (or **CPU scheduler**) selects which process should be executed next and allocates CPU. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them.

Also known as dispatchers, make the decision of which process to execute next. They are faster than long-term schedulers. Its main objective is to increase system performance in accordance with the chosen set of criteria.

Medium Term Scheduler

Medium-term scheduling is a part of **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. It is in charge of handling the swapped out-processes.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, the suspended process is moved to the secondary storage. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

3.2.2. Context Switching

A **context switch** is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the state from the current running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers.

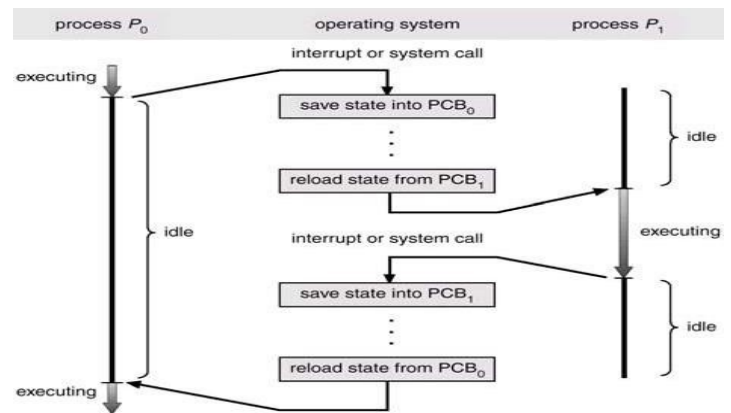


Figure 3.4 Context switching

3.3. Operations on Processes

The processes in the system execute concurrently and they must be created and deleted dynamically, thus the operating system must provide the mechanism for the creation and deletion of processes.

3.3.1. Process Creation

A process may create several new processes via a create-process system call during the course of its execution. The creating process is called a **parent process** while the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes. Figure 3.5 shows partially the process tree in a UNIX/Linux system.

In general, a process will need certain resources (such as CPU time, memory files, I/O devices) to accomplish its task. When a process creates a sub process, also known as a child, that sub process may be able to obtain its resources directly from the operating system or may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among several of its children. Restricting a process to a subset of the parent's resources prevents a process from overloading the system by creating too many sub processes.

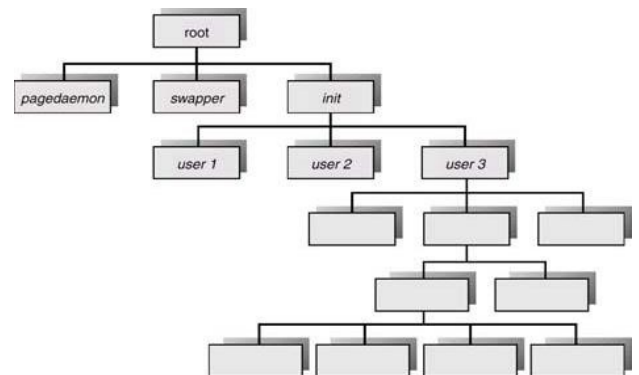


Figure 3.5 Process tree in UNIX/Linux

When a process is created it obtains in addition to various physical and logical resources, initialization data that may be passed along from the parent process to the child process. When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a program loaded into it.

In order to consider these different implementations let us consider the UNIX operating system. In UNIX each process identifier identifies a process, which is a unique integer. A new process is created by the `fork()` system call. The new process consists of a copy of the address space of the original process (Parent). This mechanism allows the parent process to communicate easily with the child process. Both processes continue execution at the instruction after the fork call, with one difference, the return code for the `fork()` is zero for the child process, while the process identifier of the child is returned to the parent process.

Typically, the `exec()` system call is used after a fork system call by one of the two processes to replace the process' memory space with a new program. The `exec()` system call loads a binary file in memory -destroying the memory image of the program containing the `exec()` system call. —and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children, or if it has nothing else to do while the child runs, it can issue a wait system call to move itself off the ready queue until the termination of the child.

3.3.2. Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by calling the `exit()` system call. At that point, the process may return data to its parent process (via the `wait()` system call). All the resources of the process - including physical and virtual memory, open files, and I/O buffers - are deallocated and reclaimed by the operating system

Termination occurs under additional circumstances. Usually only the parent of the process that is to be terminated can invoke this system call. Therefore, parents need to know the identities of its children, and thus, when one process creates another process, the identity of the newly created process is passed the parent. A parent may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such a system, if a process terminates either normally or abnormally, then all its children must also be terminated. This phenomenon referred to as cascading termination, is normally initiated by the operating system.

After a process has been created, run, and completed, it will be terminated. A process can be terminated because of one of the following four reasons.

- **Normal exit (voluntary):** when a process completes its job, it executes a system call that informs the operating system that it has completed and wants to terminate. In UNIX this system call is *exit* and *ExitProcess* in Windows. The process termination could also be user initiated; for example, when the user clicks the close button on a window.
- **Error exit (voluntary):** when a process encounters a problem in executing a program, it will terminate voluntarily. An example can be compiling a nonexistent source code file.
- **Fatal error (involuntary):** is caused by bugs in an executing program. Examples include division by zero, referring nonexistent memory address, and executing an illegal operation. When a fatal error occurs in a process, the operating system terminates the process.
- **Killed by another process (involuntary):** a process executes a system call to terminate another process. In UNIX, this system call is *kill* and the corresponding Windows system call is *TerminateProcess*. The killer process must have the necessary authorization over the killee.

3.4. Interprocess Communication

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by any other process executing in the system. Clearly any process that shares data with other processes is a cooperating process. The advantages of cooperating processes are:

- **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file) we must provide an environment to allow concurrent users to access these types of resources.
- **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks each of which will be running in parallel with the others. Such a speedup can be obtained only if the computer has multiple processing elements (such as CPU's or I/O channels).
- **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience:** Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

3.5. Threads

A **thread** is a basic unit of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

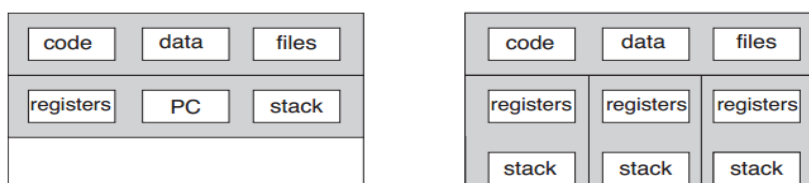


Figure 3.6 illustrates the difference between a traditional **single-threaded process** and a **multithreaded process**.

When multithreading is present, processes normally start with a single thread present. This thread has the ability to create new threads by calling a library procedure, for example, `thread_create`. A parameter to `thread_create` typically specifies the name of a procedure for the new thread to run.

Most software applications that run on modern computers and mobile devices are multithreaded. An application typically is implemented as a separate process with several threads of control. Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

The benefits of multithreaded programming can be broken down into four major categories:

- **Resource sharing:** Processes can share resources only through techniques such as shared memory and message passing. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
- **Responsiveness:** Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Economy:** Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
- **Scalability:** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

3.5.1. Types of Parallelism

In general, there are two types of parallelism: **data parallelism** and **task parallelism**.

Data parallelism focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

Task parallelism involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

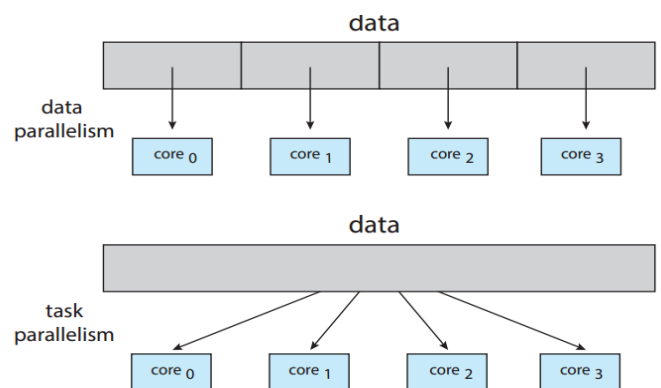


Figure 3.7 Data and task parallelism

Fundamentally, data parallelism involves the distribution of data across multiple cores, & task parallelism involves the distribution of tasks across multiple cores, as shown in Figure 3.7. However, data and task parallelism are not mutually exclusive, and an application may in fact use a hybrid of these two strategies.

3.5.2. User and Kernel Threads

Support for threads may be provided at either user level for **user threads** or by kernel for **kernel threads**.

User threads are supported above kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. Since the kernel is unaware of user-level threads, all thread creation and scheduling are done in the user space without the need for kernel intervention, and therefore are fast to create and manage.

Kernel threads are supported directly by the operating system. The kernel performs the scheduling, creation, and management in kernel space; the kernel level threads are hence slower to create and manage, compared to user level threads.

3.5.3. Multithreading Models

A relationship must ultimately exist between user threads and kernel Threads. There are various models for mapping user-level threads to kernel-level threads. We describe briefly these models, their main characteristics, and examples.

1. **Many-to-One:** In this model, many user-level threads are supported per kernel thread, as shown in Figure 3.8. Since only one kernel-level thread supports many user threads, there is no concurrency. This means that a process blocks when a thread makes a system call. Examples of these threads are Solaris Green threads POSIX Pthreads.

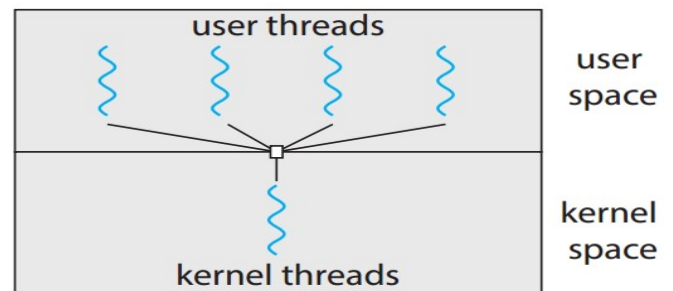


Figure 3.8 *Many-to-one model*.

2. **One-to-One:** In this model, there is a kernel thread for every user thread. This means that a process does not block when a thread makes a system call. It also allows multiple threads to run in parallel on multiprocessors. The main disadvantage of this model is the overhead of creating a kernel thread per user thread. Examples of these threads are WindowsNT, Windows 2000, and OS/2.

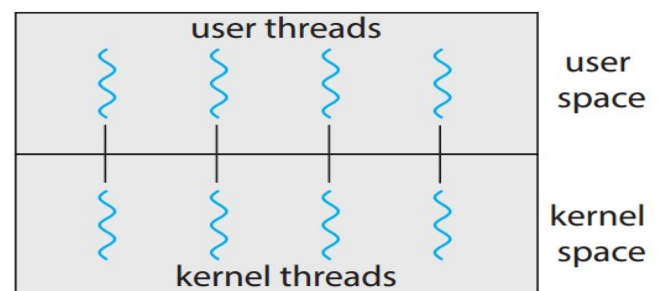
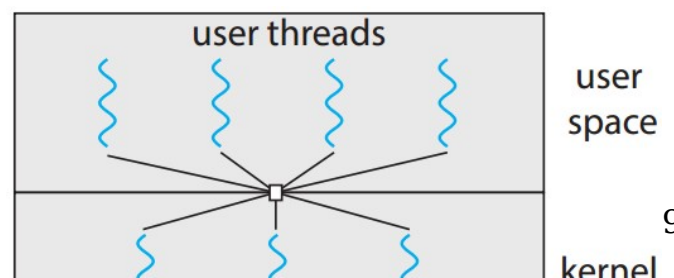


Figure 3.9 *One-to-one model*

3. **Many-to-Many:** In this model, multiple user-level threads are multiplexed over a smaller or equal number of kernel threads.



The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores). Examples of these threads are Solais 2 and HP-UX.

Let us consider the effect of this design on concurrency.

- ✓ The **many-to-one model** allows the developer to create as many user threads as he/she wishes; it does not result in parallelism, because the kernel can schedule only one kernel thread at a time.
- ✓ The **one-to-one model** allows greater concurrency, but the developer has to be careful not to create too many threads within an application. (In fact, on some systems, he/she may be limited in the number of threads she can create.)
- ✓ The **many-to-many model** suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. In addition, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

One variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model**.

3.5.4. Thread Usage

Having described what threads are, it is now time to explain why anyone wants them. The main reason for having threads is that in many applications, multiple activities are going on at once. Some of these may block from time to time. By decomposing such an application into multiple sequential threads that run in quasi-parallel, the programming model becomes simpler.

It is probably easiest to see why threads are useful by giving some concrete examples. As a first example, consider a word processor. Most word processors display the document being created on the screen formatted exactly as it will appear on the printed page. In particular, all the line breaks and page breaks are in their correct and final position so the user can inspect them and change the document if need be (e.g., to eliminate widows and orphans—incomplete top and bottom lines on a page, which are considered esthetically displeasing).

If the program were single-threaded, then whenever a disk backup started, commands from the keyboard and mouse would be ignored until the backup was finished. The user would perceive this as sluggish performance. Alternatively, keyboard and mouse events could interrupt the disk backup, allowing good performance but leading to a complex interrupt-driven programming model. With three threads, the programming model is much simpler. The first thread just interacts with the user. The second thread reformats the document when told to. The third thread writes the contents of RAM to disk periodically.

3.6. Interprocess Communication

Processes frequently need to communicate with other processes. For example, in a shell pipeline, the output of the first process must be passed to the second process, and so on down the line. Thus, there is a need for communication between processes, preferably in a well-structured way not using interrupts. In the following sections, we will look at some of the issues related to this **Interprocess Communication** or **IPC**.

Very briefly, there are three issues here. The first was alluded to above: how one process can pass information to another. The second has to do with making sure two or more processes do not get into each other's way when engaging in critical activities (suppose two processes each try to grab the last 1 MB of memory). The third concerns proper sequencing when dependencies are present: if process A produces data and process B prints them, B has to wait until A has produced some data before starting to print. We will examine all three of these issues starting in the next section.

3.6.1. Race Conditions

In some operating systems, processes that are working together may share some common storage that each one can read and write. To see how Interprocess communication works in practice, let us consider a simple but common example a print spooler.

When a process wants to print a file, it enters the file name in a special **spooler directory**. Another process, the **printer daemon**, periodically checks to see if there are any files to be printed, and if there are, it prints them and then removes their names from the directory.

3.6.2. Critical Regions

The key to preventing trouble here and in many other situations involving shared memory, shared files, and shared everything else is to find some way to prohibit more than one process from reading and writing the shared data at the same time. Put in other words, what we need is **mutual exclusion**, that is, some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from doing the same thing.

The problem of avoiding race conditions can also be formulated in an abstract way. Part of the time, a process is busy doing internal computations and other things that do not lead to race conditions. However, sometimes a process have to access shared memory or files, or doing other critical things that can lead to races. That part of the program where the shared memory is 1. No two processes may be simultaneously inside their critical regions.

- 1.No assumptions may be made about speeds or the number of CPUs.
- 2.No process running outside its critical region may block other processes.
- 3.No process should have to wait forever to enter its critical region.

accessed is called the **critical region** or **critical section** . If we could arrange matters such that no two processes were ever in their critical regions at the same time, we could avoid races. Although this requirement avoids race conditions, this is not sufficient for having parallel processes cooperate correctly and efficiently using shared data. We need four conditions to hold to have a good solution:

3.6.3. Mutual Exclusion with Busy Waiting

In this section we will examine various proposals for achieving mutual exclusion, so that while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

Disabling Interrupts

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupt occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

This approach is generally unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did it and never turned them on again? That could be the end of the system. Furthermore if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

Lock Variables

As a second attempt, let us look for a software solution. Consider having a single, shared (lock) variable, initially 0. When a process wants to enter its critical region, it first tests the lock. If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

The Producer-Consumer Problem

As an example of how these primitives can be used, let us consider the **producer-consumer** problem (also known as the **bounded-buffer** problem). Two processes share a common, fixedsize buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out. (It is also possible to generalize the problem to have m producers and n consumers, but we will only consider the case of one producer and one consumer because this assumption simplifies the solutions).

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove

an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

This approach sounds simple enough, but it leads to the same kinds of race conditions we saw earlier with the spooler directory. To keep track of the number of items in the buffer, we will need a variable, *count*. If the maximum number of items the buffer can hold is N , the producer's code will first test to see if *count* is N . If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*.

3.6.4. Semaphores

Dijkstra proposed having two operations, down and up (generalizations of sleep and wakeup respectively). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it and possibly going to sleep, is all done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.

The up operation increments the value of the semaphore addressed. If one or more processes were sleeping on that semaphore, unable to complete an earlier down operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its down. Thus, after an up on a semaphore with processes sleeping on it, the semaphore will still be 0, but there will be one fewer process sleeping on it. The operation of incrementing the semaphore and waking up one process is also indivisible. No process ever blocks doing an up, just as no process ever blocks doing a wakeup in the earlier model.

3.6.5. Mutexes

When the semaphore's ability to count is not needed, a simplified version of the semaphore, called a mutex, is sometimes used. Mutexes are good only for managing mutual exclusion to some shared resource or piece of code. They are easy and efficient to implement, which makes them especially useful in thread packages that are implemented entirely in user space.

A **mutex** is a variable that can be in one of two states: unlocked or locked. Consequently, only 1 bit is required to represent it, but in practice an integer often is used, with 0 meaning unlocked and all other values meaning locked. Two procedures are used with mutexes. When a thread (or process) needs access to a critical region, it calls *mutex_lock*. If the mutex is current unlocked (meaning that the critical region is available), the call succeeds and the calling thread is free to enter the critical region.

3.6.6. Message Passing

That something else is **message passing**. This method of interprocess communication uses two primitives, send and receive, which, like semaphores and

unlike monitors, are system calls rather than language constructs. As such, they can easily be put into library procedures, such as `send (destination, &message);` and `receive (source, &message);`

The former call sends a message to a given destination and the latter one receives a message from a given source (or from *ANY*, if the receiver does not care). If no message is available, the receiver can block until one arrives. Alternatively, it can return immediately with an error code. Message passing systems have many challenging problems and design issues that do not arise with semaphores or monitors, especially if the communicating processes are on different machines connected by a network. For example, messages can be lost by the network. To guard against lost messages, the sender and receiver can agree that as soon as a message has been received, the receiver will send back a special **acknowledgement** message. If the sender has not received the acknowledgement within a certain time interval, it retransmits the message.

Now consider what happens if the message itself is received correctly, but the acknowledgement is lost. The sender will retransmit the message, so the receiver will get it twice. It is essential that the receiver be able to distinguish a new message from the retransmission of an old one. Usually, this problem is solved by putting consecutive sequence numbers in each original message. If the receiver gets a message bearing the same sequence number as the previous message, it knows that the message is a duplicate that can be ignored. Successfully communicating in the face of unreliable message passing is a major part of the study of computer networks.

The Producer-Consumer Problem with Message Passing

Now let us see how the producer-consumer problem can be solved with message passing and no shared memory. In this solution, a total of N messages is used, analogous to the N slots in a shared memory buffer. The consumer starts out by sending N empty messages to the producer. Whenever the producer has an item to give to the consumer, it takes an empty message and sends back a full one. In this way, the total number of messages in the system remains constant in time, so they can be stored in a given amount of memory known in advance.

If the producer works faster than the consumer, all the messages will end up full, waiting for the consumer: the producer will be blocked, waiting for an empty to come back. If the consumer works faster, then the reverse happens: all the messages will be empties waiting for the producer to fill them up: the consumer will be blocked, waiting for a full message.

3.7. Deadlock

For many applications, a process needs exclusive access to not one resource, but several like devices, files and so forth. Suppose, two processes each want to record a scanned document on a CD. Process *A* requests permission to use the scanner and is granted it. Process *B* is programmed differently and requests the CD recorder first and is also granted it.

Now *A* asks for the CD recorder, but the request is denied until *B* releases it. Unfortunately, instead of releasing the CD recorder *B* asks for the scanner. At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

Deadlocks can also occur across machines. For example, many offices have a local area network with many computers connected to it. Often devices such as scanners, CD recorders, printers, and tape drives are connected to the network as shared resources, available to any user on any machine.

Deadlocks can occur in a variety of situations besides requesting dedicated I/O devices. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions. If process *A* locks record *R1* and process *B* locks record *R2*, and then each process tries to lock the other one's record, we also have a deadlock. In short, a resource is anything that can be used by only a single process at any instant of time.

3.7.1. Conditions for Deadlock

Coffman et al. (1971) showed that four conditions must hold for there to be a deadlock:

1. **Mutual exclusion condition.** Each resource is either currently assigned to exactly one process or is available.
2. **Hold and wait condition.** Processes currently holding resources granted earlier can request new resources.
3. **No preemption condition.** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. **Circular wait condition.** There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

3.7.2. Deadlock Modeling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. An arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig.6-3(a), resource *R* is currently assigned to process *A*.

An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 6-3(b), process *B* is waiting for resource *S*. In Fig.6-3(c) we see a deadlock: process *C* is waiting for resource *T*, which is currently held by

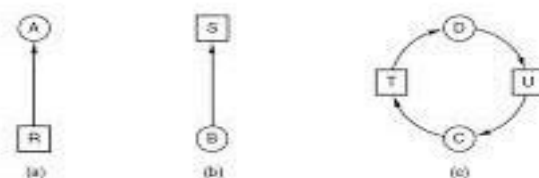


Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

process *D*. Process *D* is not about to release resource *T* because it is waiting for resource *U*, held by *C*. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is *C-T-D-U-C*.

Now let us look at an example of how resource graphs can be used. Imagine that we have three processes, *A*, *B*, and *C*, and three resources *R*, *S*, and *T*. The requests and releases of the three processes are given in Fig. 6-4(a)-(c). The operating system is free to run any unblocked process at any instant, so it could decide to run *A* until *A* finished all its work, then run *B* to completion, and finally run *C*.

This ordering does not lead to any deadlocks (because there is no competition for resources) but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes do any I/O at all, shortest job first is better than round robin, so under some circumstances running all processes sequentially may be the best way.

Let us now suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order of Fig. 6-4(d). If these six requests are carried out in that order, the six resulting resource graphs are shown in Fig. 6-4(e)-(j). After request 4 has been made, *A* blocks waiting for *S*, as shown in Fig. 6-4(h). In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 6-4(j).

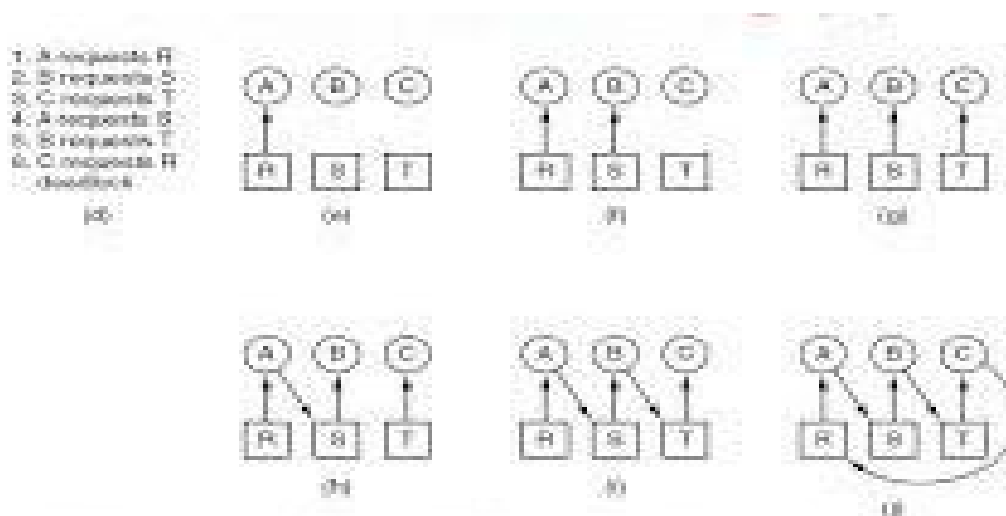


Figure 6-4. An example of how deadlock occurs and how it can be avoided.

Translation: *Operating Systems* (3rd ed. 2005) Prentice Hall, Inc. All rights reserved. 0-13-066000-0

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
7. no deadlock

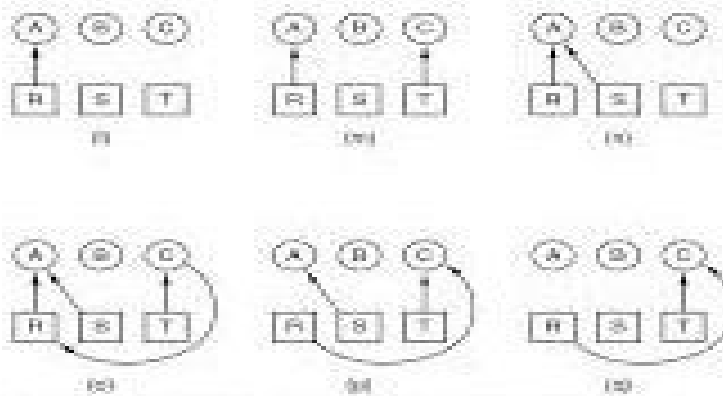


Figure 6-4. An example of how deadlock occurs and how it can be avoided.

Reisner, Modern Operating Systems, 3rd ed. (c) 2005 Pearson Education, Inc. All rights reserved. 8-11-PG0218

In general, four strategies are used for dealing with deadlocks. Just ignore the problem altogether. Maybe if you ignore it, it will ignore you.

1. Detection and recovery. Let deadlocks occur, detect them, and take action.
2. Dynamic avoidance by careful resource allocation.
3. Prevention, by structurally negating one of the four conditions necessary to cause a deadlock.

3.7.3. Deadlock Detection and Recovery

In this section we will look at some of the ways deadlocks can be detected and some of the ways recovery from them can be handled.

Deadlock Detection with One Resource of Each Type

Let us begin with the simplest case: only one resource of each type exists. Such a system might have one scanner, one CD recorder, one plotter, and one tape drive, but no more than one of each class of resource. In other words, we are excluding systems with two printers for the moment. We will treat them later using a different method.

For such a system, we can construct a resource graph of the sort illustrated in Fig. below. If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked.

As an example of a more complex system than the ones we have looked at so far, consider a system with seven processes, A through G, and six resources, R through W. The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process A holds R and wants S.
2. Process B holds nothing but wants T.
3. Process C holds nothing but wants S.
4. Process D holds U and wants S and T.
5. Process E holds T and wants V.
6. Process F holds W and wants S.
7. Process G holds V and wants U.

