Threads & Concurrency:

Overview:

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 4.1 illustrates the difference between a traditional single-threaded process and a multithreaded process.

Motivation:

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.

Below we highlight a few examples of multithreaded applications:

- An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.
- A web browser might have one thread display images or text while another thread retrieves data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

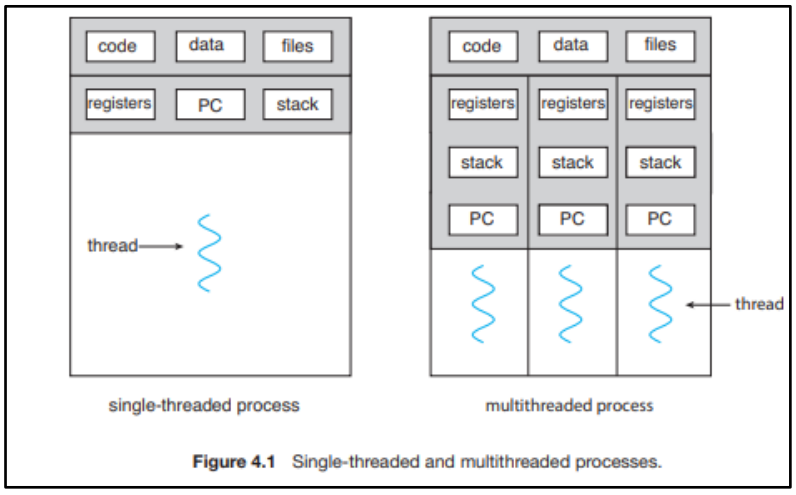


Figure 4.1 Single-threaded and multithreaded processes.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resumes listening for additional requests. This is illustrated in Figure 4.2.

Most operating system kernels are also typically multithreaded. Many applications can also take advantage of multiple threads, including basic sorting, trees, and graph algorithms. In addition, programmers who must solve contemporary CPU-intensive problems in data mining, graphics, and artificial intelligence can leverage the power of modern multicore systems by designing solutions that run in parallel.

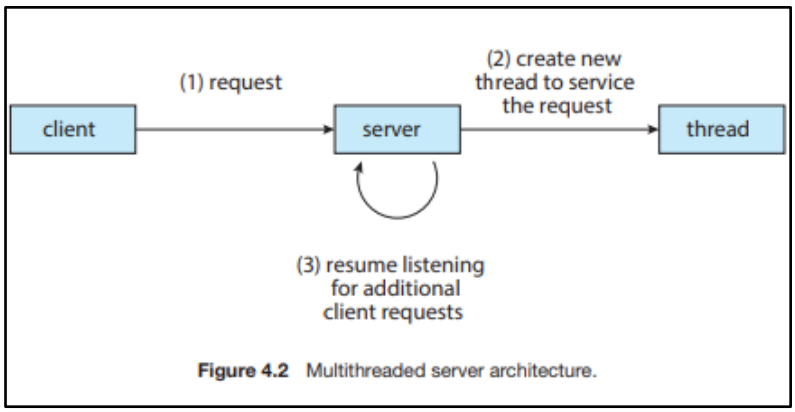


Figure 4.2 Multithreaded server architecture.

Benefits:

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

1. 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. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.

2. Resource sharing: Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. 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.

3. 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. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.

4. 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. We explore this issue further in the following section.

Multicore Programming:

Earlier in the history of computer design, in response to the need for more computing performance, single-CPU systems evolved into multi-CPU systems. A later, yet similar, trend in system design is to place multiple computing cores on a single processing chip where each core appears as a separate CPU to the operating system (Section 1.3.2). We refer to such systems as multicore, and multithreaded programming provides a mechanism for more efficient use of these multiple computing cores and improved concurrency. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (Figure 4.3), because the processing core can execute only one thread at a time. On a system with multiple cores, however, concurrency means that some threads can run in parallel, because the system can assign a separate thread to each core (Figure 4.4).

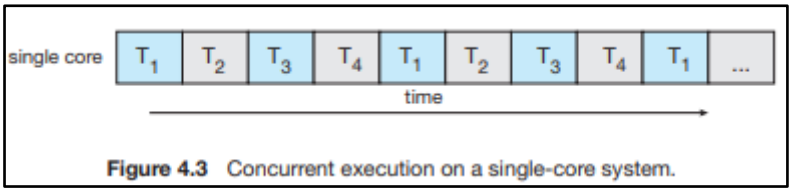


Figure 4.3 Concurrent execution on a single-core system.

Programming Challenges:

In general, five areas present challenges in programming for multicore systems:

1. Identifying tasks. This involves examining applications to find areas that can be divided into separate, concurrent tasks. Ideally, tasks are independent of one another and thus can run in parallel on individual cores.

2. Balance. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks. Using a separate execution core to run that task may not be worth the cost.

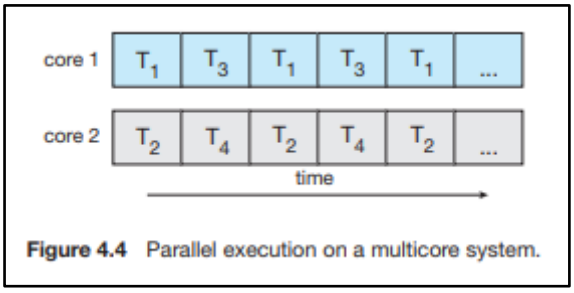


Figure 4.4 Parallel execution on a multicore system.

3. Data splitting. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.

4. Data dependency. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.

5. Testing and debugging. When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs are inherently more difficult than testing and debugging single-threaded applications.

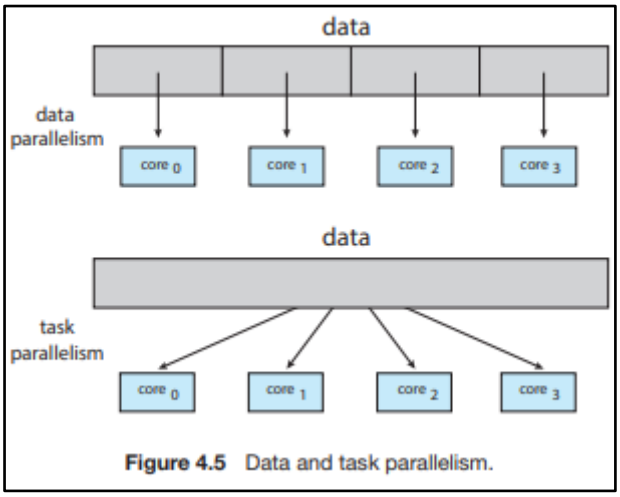


Figure 4.5 Data and task parallelism.

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. Consider, for example, summing the contents of an array of size N. On a single-core system, one thread would simply sum the elements [0] ...[N − 1]. On a dual-core system, however, thread A, running on core 0, could sum the elements [0] . . . [N⁄2 − 1] while thread B, running on core 1, could sum the elements [N⁄2] . . . [N − 1]. The two threads would be running in parallel on separate computing cores.

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. Consider again our example above. In contrast to that situation, an example of task parallelism might involve two threads, each performing a unique statistical operation on the array of elements. The threads again are operating in parallel on separate computing cores, but each is performing a unique operation.

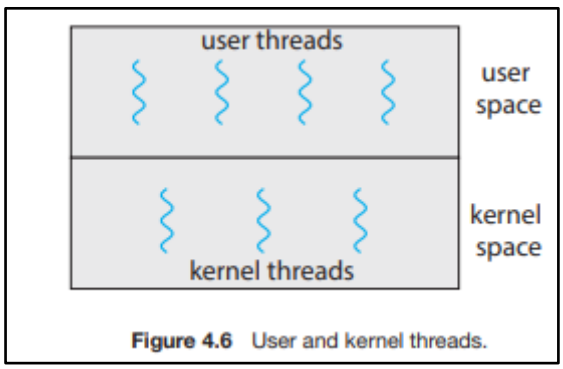


Figure 4.6 User and kernel threads.

Multithreading Models:

User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows, Linux, and macOS— support kernel threads.

Ultimately, a relationship must exist between user threads and kernel threads, as illustrated in Figure 4.6. In this section, we look at three common ways of establishing such a relationship: the many-to-one model, the one-toone model, and the many-to-many model.

Many-to-One Model:

The many-to-one model (Figure 4.7) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient (we discuss thread libraries in Section 4.4). However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. Green threads—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to one mode.

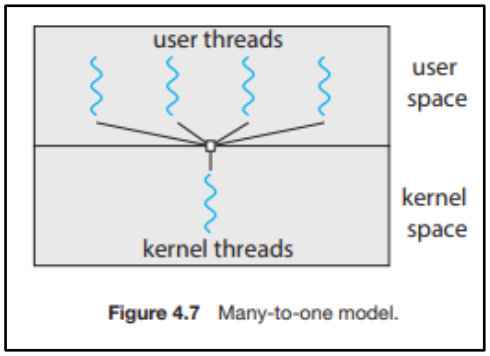


Figure 4.7 Many-to-one model.

One-to-One Model:

The one-to-one model (Figure 4.8) maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, and many kernel threads may burden the performance of a system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

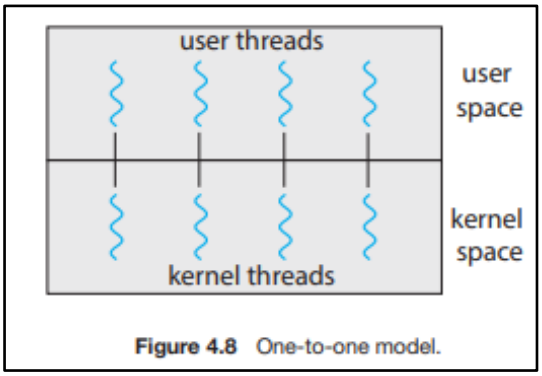


Figure 4.8 One-to-one model.

Many-to-Many Model:

The many-to-many model (Figure 4.9) multiplexes many user-level threads to 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).

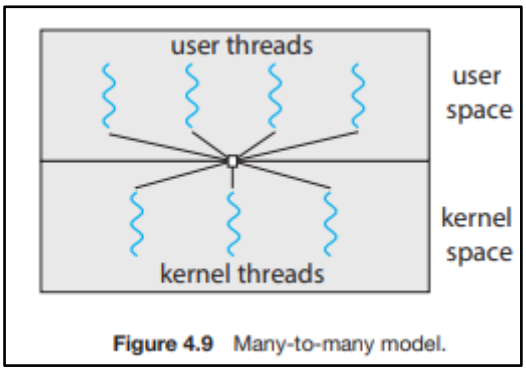


Figure 4.9 Many-to-many model.

One variation on the many-to-many model still multiplexes many userlevel 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 (Figure 4.10).

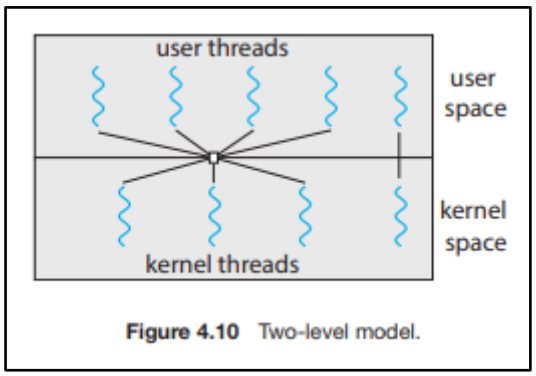


Figure 4.10 Two-level model.

Thread Libraries:

A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.

The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

Three main thread libraries are in use today: POSIX Pthreads, Windows, and Java. Pthreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library. The Windows thread library is a kernel-level library available on Windows systems. The Java thread API allows threads to be created and managed directly in Java programs. However, because in most instances the JVM is running on top of a host operating system, the Java thread API is generally

implemented using a thread library available on the host system. This means that on Windows systems, Java threads are typically implemented using the Windows API; UNIX, Linux, and macOS systems typically use Pthreads.

As an illustrative example, we design a multithreaded program that performs the summation of a non-negative integer in a separate thread using the well-known summation function:

$$sum = \sum_{i=1}^{N} i$$

For example, if N were 5, this function would represent the summation of integers from 1 to 5, which is 15.

Before we proceed with our examples of thread creation, we introduce two general strategies for creating multiple threads: asynchronous threading and synchronous threading. With asynchronous threading, once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently and independently of one another. Because the threads are independent, there is typically little data sharing between them. Asynchronous threading is the strategy used in the multithreaded server illustrated in Figure 4.2 and is also commonly used for designing responsive user interfaces.

Synchronous threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes.

Pthreads:

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization. This is a specification for thread behaviour, not an implementation. Operating-system designers may implement the specification in any way they wish. Numerous systems implement the Pthreads specification; most are UNIX-type systems, including Linux and macOS. Although Windows doesn't support Pthreads natively, some third-party implementations for Windows are available.

```
#include <pthread.h>
#include <stdio.h>

#include <stdlib.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   /* set the default attributes of the thread */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid, &attr, runner, argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

   printf("sum = %d\n",sum);
}

/* The thread will execute in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Figure 4.11 Multithreaded C program using the Pthreads API.

Windows Threads:

The technique for creating threads using the Windows thread library is like the Pthreads technique in several ways. We illustrate the Windows thread API in the C program shown in Figure 4.13. Notice that we must include the windows.h header file when using the Windows API.

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

    /* now wait for the thread to finish */
   WaitForSingleObject(ThreadHandle,INFINITE);

   /* close the thread handle */
   CloseHandle(ThreadHandle);

   printf("sum = %d\n",Sum);
}
```

Figure 4.13 Multithreaded C program using the Windows API.

In situations that require waiting for multiple threads to complete, the WaitForMultipleObjects() function is used. This function is passed four parameters: 1. The number of objects to wait for 2. A pointer to the array of objects 3. A flag indicating whether all objects have been signaled 4. A timeout duration (or INFINITE) For example, if THandles is an array of thread HANDLE objects of size N, the parent thread can wait for all its child threads to complete with this statement: WaitForMultipleObjects(N, THandles, TRUE, INFINITE);

Java Threads:

Threads are the fundamental model of program execution in a Java program, and the Java language and its API provide a rich set of features for the creation and management of threads. All Java programs comprise at least a single thread of control—even a simple Java program consisting of only a main() method runs as a single thread in the JVM. Java threads are available on any system that provides a JVM including Windows, Linux, and macOS. The Java thread API is available for Android applications as well.

There are two techniques for explicitly creating threads in a Java program. One approach is to create a new class that is derived from the Thread class and to override its run() method. An alternative—and more commonly used — technique is to define a class that implements the Runnable interface. This interface defines a single abstract method with the signature public void run(). The code in the run() method of a class that implements Runnable is what executes in a separate thread.

An example is shown below:

```
class Task implements Runnable

{
        public void run()
        {
                System.out.println("I am a thread.");
        }
}
```

Thread creation in Java involves creating a Thread object and passing it an instance of a class that implements Runnable, followed by invoking the start() method on the Thread object. This appears in the following example:

```
Thread worker = new Thread(new Task());
worker.start();
```

Invoking the start() method for the new Thread object does two things:

1. It allocates memory and initializes a new thread in the JVM.
2. It calls the run() method, making the thread eligible to be run by the JVM. (Note again that we never call the run() method directly. Rather, we call the start() method, and it calls the run() method on our behalf.)

Recall that the parent threads in the Pthreads and Windows libraries use pthread join() and WaitForSingleObject() (respectively) to wait for the summation threads to finish before proceeding. The join() method in Java provides similar functionality. (Notice that join() can throw an InterruptedException, which we choose to ignore.)

```
try {
        worker.join();
    } catch (InterruptedException ie) { }
```

If the parent must wait for several threads to finish, the join() method can be enclosed in a for loop similar to that shown for Pthreads in Figure 4.12.

LAMBDA EXPRESSIONS IN JAVA

Beginning with Version 1.8 of the language, Java introduced Lambda expressions, which allow a much cleaner syntax for creating threads. Rather than defining a separate class that implements Runnable, a Lambda expression can be used instead:

```
Runnable task = () -> {
   System.out.println("I am a thread.");
};

Thread worker = new Thread(task);
worker.start();
```

Lambda expressions—as well as similar functions known as closures—are a prominent feature of functional programming languages and have been available in several nonfunctional languages as well including Python, C++, and C#. As we shall see in later examples in this chapter, Lamdba expressions often provide a simple syntax for developing parallel applications.

Threading Issues:

1. The fork() and exec() System Calls:

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.

The exec() system call typically works in the same way as described in Chapter 3. That is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.

Which of the two versions of fork() to use depends on the application. If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate. If, however, the separate process does not call exec() after forking, the separate process should duplicate all threads.

2. Signal Handling:

A signal is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signalled.

All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled

A signal may be handled by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a default signal handler that the kernel runs when handling that signal. This default action can be overridden by a user-define signal handler that is called to handle the signal. Signals are handled in different ways. Some signals may be ignored, while others (for example, an illegal memory access) are handled by terminating the program.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

The standard UNIX function for delivering a signal is:

kill(pid t pid, int signal)

POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid):

pthread kill(pthread t tid, int signal)

Although Windows does not explicitly provide support for signals, it allows us to emulate them using asynchronous procedure calls (APCs).

Thread Cancellation:

Thread cancellation involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be cancelled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are cancelled.

A thread that is to be cancelled is often referred to as the target thread. Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation. One thread immediately terminates the target thread.

2. Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

In Pthreads, thread cancellation is initiated using the pthread cancel () function. The identifier of the target thread is passed as a parameter to the function. The following code illustrates creating—and then canceling—a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

 . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Each mode is defined as a state and a type, as illustrated in the table below. A thread may set its cancellation state and type using an API.

Each mode is defined as a state and a type, as illustrated in the table below. A thread may set its cancellation state and type using an API

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

The default cancellation type is deferred cancellation. However, cancellation occurs only when a thread reaches a cancellation point. Most of the blocking system calls in the POSIX and standard C library are defined as cancellation points, and these are listed when invoking the command man pthreads on a Linux system. Additionally, Pthreads allows a function known as a clean-up handler to be invoked if a thread is cancelled. This function allows any resources a thread may have acquired to be released before the thread is terminated.

The following code illustrates how a thread may respond to a cancellation request using deferred cancellation:

```
while (1) {
   /* do some work for awhile */

    . . .

   /* check if there is a cancellation request */
   pthread_testcancel();
}
```

Thread cancellation in Java uses a policy like deferred cancellation in Pthreads. To cancel a Java thread, you invoke the interrupt () method, which sets the interruption status of the target thread to true:

Thread worker;

...

/* set the interruption status of the thread */ worker.interrupt()

A thread can check its interruption status by invoking the isInterrupted() method, which returns a boolean value of a thread's interruption status:

```
while (!Thread.currentThread().isInterrupted()) {
        ...
}
```

3. Thread-Local Storage:

Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data thread-local storage (or TLS). For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique transaction identifier, we could use thread-local storage.

It is easy to confuse TLS with local variables. However, local variables are visible only during a single function invocation, whereas TLS data are visible across function invocations. Additionally, when the developer has no control over the thread creation process— for example, when using an implicit technique such as a thread pool— then an alternative approach is necessary.

For example, if we wished to assign a unique identifier for each thread, we would declare it as follows:

```
static thread int threadID;
```

Scheduler Activations:

A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models discussed in Section 4.3.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads. This data structure— typically known as a lightweight process, or LWP—is shown in Figure 4.20.

One scheme for communication between the user-thread library and the kernel is known as scheduler activation. It works as follows: The kernel provides an application with a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an upcall. Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.

One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying

the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running.

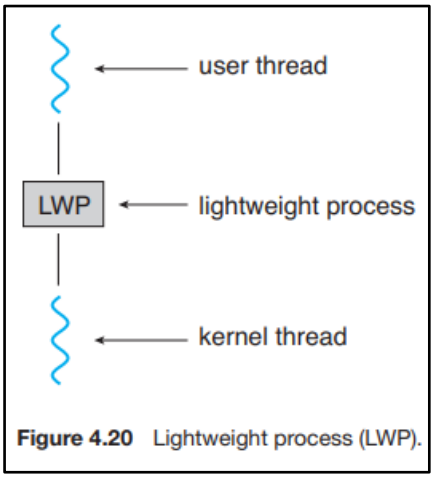


Figure 4.20 Lightweight process (LWP).

Operating-System Examples:

Windows Threads:

A Windows application runs as a separate process, and each process may contain one or more threads. The Windows API for creating threads is covered in Section 4.4.2. Additionally, Windows uses the one-to-one mapping described in Section 4.3.2, where each user-level thread maps to an associated kernel thread.

The general components of a thread include:

- A thread ID uniquely identifying the thread
- A register set representing the status of the processor
- A program counters
- A user stack, employed when the thread is running in user mode, and a kernel stack, employed when the thread is running in kernel mode
- A private storage area used by various run-time libraries and dynamic link libraries (DLLs)

The register set, stacks, and private storage area are known as the context of the thread.

The primary data structures of a thread include:

- ETHREAD—executive thread block
- KTHREAD—kernel thread block
- TEB— thread environment block

The key components of the ETHREAD include a pointer to the process to which the thread belongs and the address of the routine in which the thread starts control. The ETHREAD also contains a pointer to the corresponding KTHREAD.

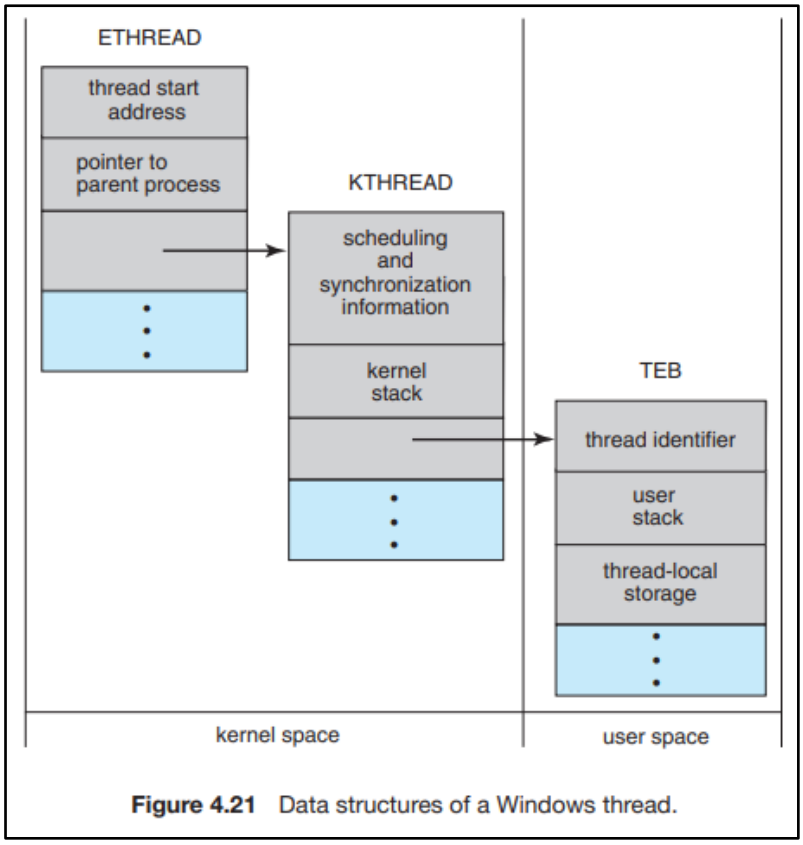


Figure 4.21 Data structures of a Windows thread.

The KTHREAD includes scheduling and synchronization information for the thread. In addition, the KTHREAD includes the kernel stack (used when the thread is running in kernel mode) and a pointer to the TEB.

Linux Threads:

Linux provides the fork() system call with the traditional functionality of duplicating a process, as described in Chapter 3. Linux also provides the ability to create threads using the clone() system call. However, Linux does not distinguish between processes and threads. In fact, Linux uses the term task —rather than process or thread— when referring to a flow of control within a program.

When clone() is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed in Figure 4.22. For example, suppose that clone() is passed the flags CLONE FS, CLONE VM, CLONE SIGHAND, and CLONE FILES. The parent and child tasks will then share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files.

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Figure 4.22 Some of the flags passed when `clone()` is invoked.

Using clone() in this fashion is equivalent to creating a thread as described in this chapter, since the parent task shares most of its resources with its child task. However, if none of these flags is set when clone() is invoked, no sharing takes place, resulting in functionality similar to that provided by the fork() system call.

The varying level of sharing is possible because of the way a task is represented in the Linux kernel. A unique kernel data structure (specifically, struct task struct) exists for each task in the system. This data structure, instead of storing data for the task, contains pointers to other data structures where these data are stored— for example, data structures that represent the list of open files, signal-handling information, and virtual memory. When fork () is invoked, a new task is created, along with a copy of all the associated data structures of the parent process. A new task is also created when the clone () system call is made. However, rather than copying all data structures, the new task points to the data structures of the parent task, depending on the set of flags passed to clone().

Finally, the flexibility of the clone() system call can be extended to the concept of containers, a virtualization topic which was introduced in Chapter 1. Recall from that chapter that a container is a virtualization technique provided by the operating system that allows creating multiple Linux systems (containers) under a single Linux kernel that run-in isolation to one another. Just as certain flags passed to clone () can distinguish between creating a task that behaves more like a process or a thread based upon the amount of sharing between the parent and child tasks, there are other flags that can be passed to clone() that allow a Linux container to be created.

CPU Scheduling:

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive. In this chapter, we introduce basic CPU-scheduling concepts and present several CPU-scheduling algorithms, including real-time systems. We also consider the problem of selecting an algorithm for a particular system.

Basic Concepts:

The objective of multiprogramming is to have some process always running, to maximize CPU utilization. The idea is relatively simple. A process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. Every time one process must wait, another process can take overuse of the CPU. On a multicore system, this concept of keeping the CPU busy is extended to all processing cores on the system.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use. The CPU is, of course, one of the primary computer resources. Thus, its scheduling is central to operating-system design.

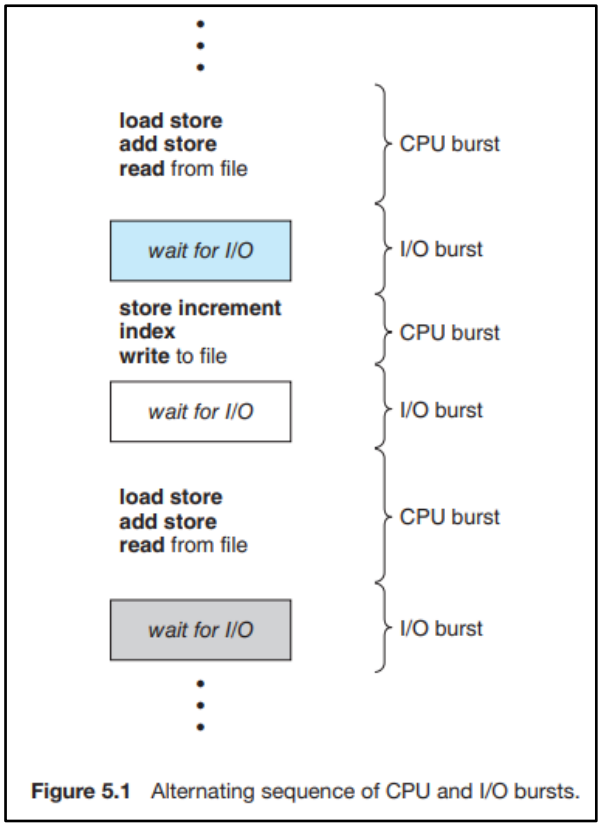


Figure 5.1 Alternating sequence of CPU and I/O bursts.

CPU – I/O Burst Cycle:

The success of CPU scheduling depends on an observed property of processes: process execution consists of a cycle of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a CPU burst. That is followed by an I/O burst, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution (Figure 5.1).

The durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve like that shown in Figure 5.2. The curve is generally characterized as exponential or hyper exponential, with many short CPU bursts and a small number of long CPU bursts. An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important when implementing a CPU-scheduling algorithm.

CPU Scheduler:

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the CPU scheduler, which selects a process from the processes in memory that are ready to execute and allocates the CPU to that process. Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in

the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

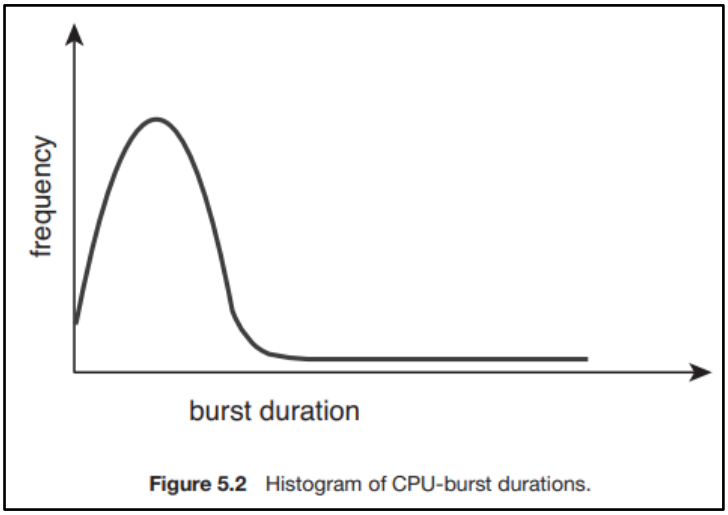


Figure 5.2 Histogram of CPU-burst durations.

Preemptive and Nonpreemptive Scheduling:

CPU-scheduling decisions may take place under the following four circumstances: 1. When a process switches from the running state to the waiting state (for example, as the result of an I/O request or an invocation of wait() for the termination of a child process) 2. When a process switches from the running state to the ready state (for example, when an interrupt occurs) 3. When a process switches from the waiting state to the ready state (for example, at completion of I/O) 4. When a process terminates For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is nonpreemptive or cooperative. Otherwise, it is preemptive. Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state. Virtually all modern operating systems including Windows, macOS, Linux, and UNIX use preemptive scheduling algorithms

Unfortunately, preemptive scheduling can result in race conditions when data are shared among several processes. Consider the case of two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data, which are in an inconsistent state.

Preemption also affects the design of the operating-system kernel. During the processing of a system call, the kernel may be busy with an activity on behalf of a process. Such activities may involve changing important kernel data (for instance, I/O queues). What happens if the process is preempted in the middle of these changes and the kernel (or the device driver) needs to read or modify the same structure? Chaos ensues.

Dispatcher:

Another component involved in the CPU-scheduling function is the dispatcher. The dispatcher is the module that gives control of the CPU's core to the process selected by the CPU scheduler. This function

involves the following:

- Switching context from one process to another
- Switching to user mode
- Jumping to the proper location in the user program to resume that program

The dispatcher should be as fast as possible since it is invoked during every context switch. The time it takes for the dispatcher to stop one process and start another running is known as the dispatch latency and is illustrated in Figure 5.3.

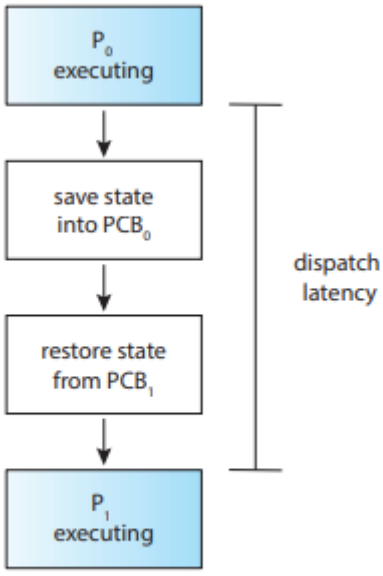


Figure 5.3 The role of the dispatcher.

Scheduling Criteria:

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best.

The criteria include the following:

• **CPU utilization**. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system). (CPU utilization can be obtained by using the top command on Linux, macOS, and UNIX systems.)

• **Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process over several seconds; for short transactions, it may be tens of processes per second.

• **Turnaround time.** From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting in the ready queue, executing on the CPU, and doing I/O.

• **Waiting time.** The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

• **Response time**. In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. In most cases, we optimize the average measure.

Scheduling Algorithms:

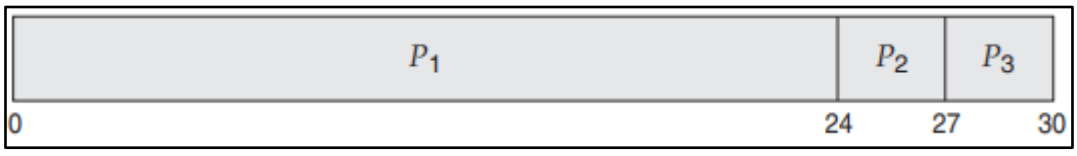
CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU's core. There are many different CPUscheduling algorithms.

First-Come, First-Served Scheduling:

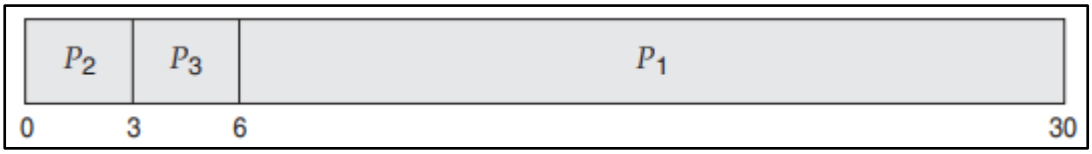
By far the simplest CPU-scheduling algorithm is the first-come first-serve (FCFS) scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following Gantt chart, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



The waiting time is 0 milliseconds for process P1, 24 milliseconds for process P2, and 27 milliseconds for process P3. Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds. If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



The average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds. This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes' CPU burst times vary greatly.

Note also that the FCFS scheduling algorithm is nonpreemptive. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for interactive systems, where it is important that each process get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

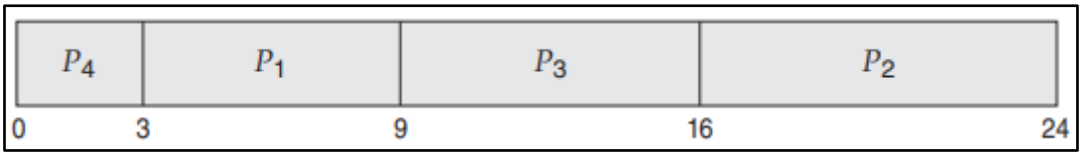
Shortest-Job-First Scheduling:

A different approach to CPU scheduling is the shortest-job-firs (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm because scheduling depends on the length of the next CPU burst of a process, rather than its total length. We use the term SJF because most people and textbooks use this term to refer to this type of scheduling.

As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



The waiting time is 3 milliseconds for process P1, 16 milliseconds for process P2, 9 milliseconds for process P3, and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds. By comparison, if we were using the FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

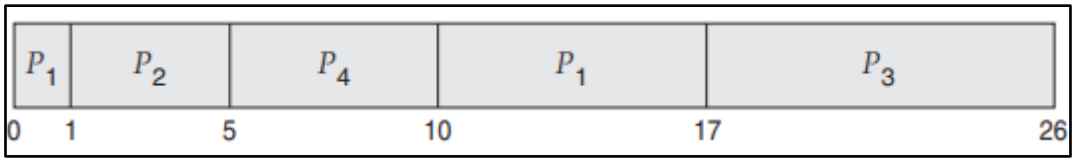
Although the SJF algorithm is optimal, it cannot be implemented at the level of CPU scheduling, as there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. We expect that the next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

The SJF algorithm can be either preemptive or nonpreemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a nonpreemptive SJF algorithm will allow the currently running process to finish its CPU burst. Preemptive SJF scheduling is sometimes called shortest-remainingtime-firs scheduling.

As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:



Process P1 is started at time 0 since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled. The average waiting time for this example is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5 milliseconds. Nonpreemptive SJF scheduling would result in an average waiting time of 7.75 milliseconds.

Round-Robin Scheduling:

The round-robin (RR) scheduling algorithm is like FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

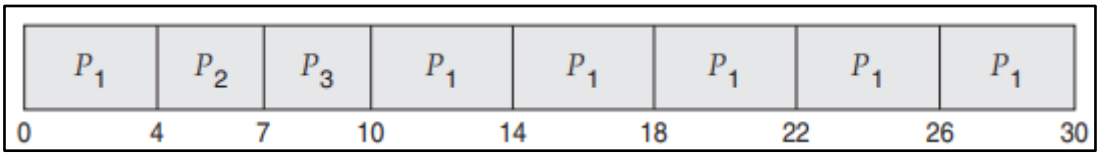
To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

If we use a time quantum of 4 milliseconds, then process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first-time quantum, and the CPU is given to the next process in the queue, process P2. Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:



Let's calculate the average waiting time for this schedule. P1 waits for 6 milliseconds (10 − 4), P2 waits for 4 milliseconds, and P3 waits for 7 milliseconds. Thus, the average waiting time is 17/3 = 5.66 milliseconds.

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

If there are n processes in the ready queue and the time quantum is q, then each process gets 1/n of the CPU time in chunks of at most q time units. Each process must wait no longer than (n − 1) × q time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy is the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in many context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12-time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6-time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 5.5).

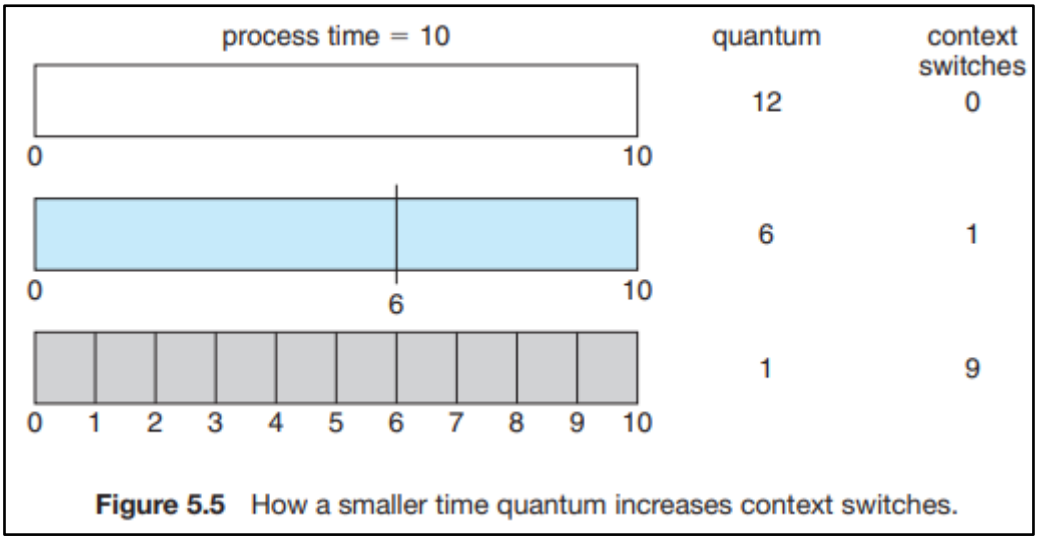


Figure 5.5 How a smaller time quantum increases context switches.

Turnaround time also depends on the size of the time quantum. As we can see from Figure 5.6, the average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

Priority Scheduling:

The SJF algorithm is a special case of the general priority-scheduling algorithm. A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

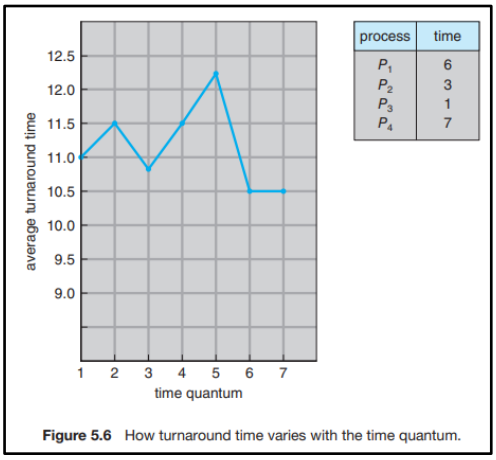
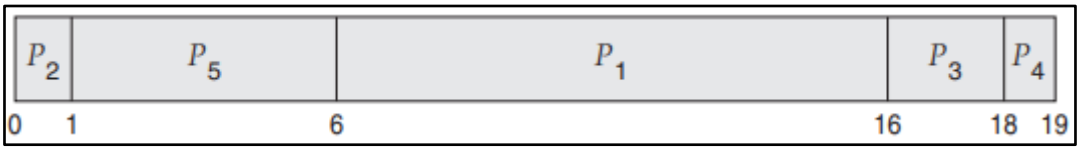


Figure 5.6 How turnaround time varies with the time quantum.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, ···, P5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Using priority scheduling, we would schedule these processes according to the following Gantt chart:



The average waiting time is 8.2 milliseconds.

Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

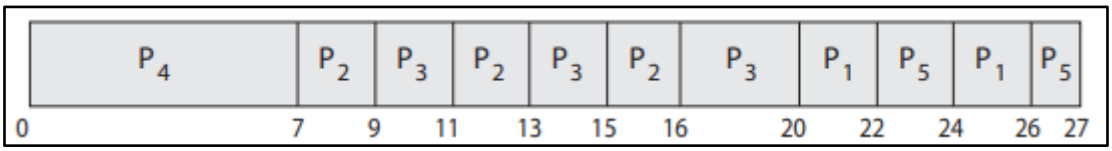
A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely.

A solution to the problem of indefinite blockage of low-priority processes is aging. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could periodically (say, every second) increase the priority of a waiting process by 1. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take a little over 2 minutes for a priority-127 process to age to a priority-0 process.

Another option is to combine round-robin and priority scheduling in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling. Let's illustrate with an example using the following set of processes, with the burst time in milliseconds:

Process	Burst Time	Priority
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Using priority scheduling with round-robin for processes with equal priority, we would schedule these processes according to the following Gantt chart using a time quantum of 2 milliseconds:



In this example, process P4 has the highest priority, so it will run to completion. Processes P2 and P3 have the next-highest priority, and they will execute in a round-robin fashion. Notice that when process P2 finishes at time 16, process P3 is the highest-priority process, so it will run until it completes execution. Now, only processes P1 and P5 remain, and as they have equal priority, they will execute in round-robin order until they complete.

Shortest Remaining Time First Scheduling:

The Preemptive version of Shortest Job First (SJF) scheduling is known as Shortest Remaining Time First (SRTF). With the help of the SRTF algorithm, the process having the smallest amount of time remaining until completion is selected first to execute.

So basically, in SRTF, the processes are scheduled according to the shortest remaining time.

However, the SRTF algorithm involves more overheads than the shortest job first (SJF)scheduling, because in SRTF OS is required frequently to monitor the CPU time of the jobs in the READY queue and to perform context switching.

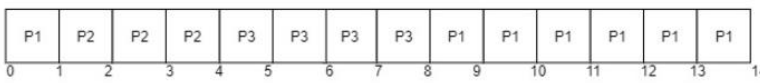
In the SRTF scheduling algorithm, the execution of any process can be stopped after a certain amount of time. On arrival of every process, the short-term scheduler schedules those processes from the list of available processes & running processes that have the least remaining burst time.

After all the processes are available in the ready queue, then, no preemption will be done and then the algorithm will work the same as SJF scheduling. In the Process Control Block, the context of the process is saved, when the process is removed from the execution and when the next process is scheduled. The PCB is accessed on the next execution of this process.

Example

Process	Burst Time	Arrival Time
P1	7	0
P2	3	1
P3	4	3

The Gantt Chart for SRTF will be:



Explanation

At the 0th unit of the CPU, there is only one process that is P1, so P1 gets executed for the 1-time unit.

- At the 1st unit of the CPU, Process P2 arrives. Now, the P1 needs 6 more units more to be executed, and the P2 needs only 3 units. So, P2 is executed first by preempting P1.
- At the 3rd unit of time, the process P3 arrives, and the burst time of P3 is 4 units which is more than the completion time of P2 that is 1 unit, so P2 continues its execution.
- Now after the completion of P2, the burst time of P3 is 4 units that means it needs only 4 units for completion while P1 needs 6 units for completion.
- So, this algorithm picks P3 above P1 due to the reason that the completion time of P3 is less than that of P1
- P3 gets completed at time unit 8, there are no new processes arrived.
- So again, P1 is sent for execution, and it gets completed at the 14th unit.

As Arrival Time and Burst time for three processes P1, P2, P3 are given in the above diagram. Let us calculate Turnaround time, completion time, and waiting time.

Process	Arrival Time	Burst Time	Completion time	Turn around Time Turn Around Time = Completion Time – Arrival Time	Waiting Time Waiting Time = Turn Around Time – Burst Time
P1	0	7	14	14-0=14	14-7=7
P2	1	3	4	4-1=3	3-3=0
P3	3	4	8	8-3=5	5-4=1

Average waiting time is calculated by adding the waiting time of all processes and then dividing them by no. of processes.

average waiting time = waiting for time of all processes/ no.of processes

average waiting time=7+0+1=8/3 = **2.66ms**

Highest Response Ratio Next (HRRN):

HRRN(Highest Response Ratio Next)Scheduling is a non-preemptive scheduling algorithm in the operating system. It is one of the optimal algorithms used for scheduling.

As HRRN is a non-preemptive scheduling algorithm so in case if there is any process that is currently in execution with the CPU and during its execution, if any new process arrives in the memory with burst time smaller than the currently running process then at that time the currently running process will not be put in the ready queue & complete its execution without any interruption.

HRRN is basically the modification of Shortest Job Next(SJN) in order to reduce the problem of starvation. In the HRRN scheduling algorithm, the CPU is assigned to the next process that has the highest response ratio and not to the process having less burst time. Now, let us first take a look at how to calculate the Response ratio.

Response Ratio = (W+S)/S

Where, W=It indicates the Waiting Time.

S=It indicates the Service time that is Burst Time.

Algorithm of HRNN

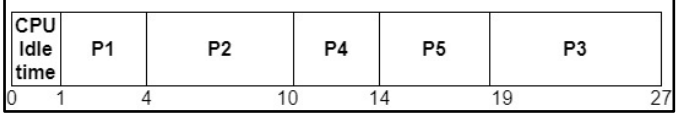
1. In the HRNN scheduling algorithm, once a process is selected for execution will run until its completion.
2. The first step is to calculate the waiting time for all the processes. Waiting time simply means the sum of the time spent waiting in the ready queue by processes.
3. Processes get scheduled each time for execution in order to find the response ratio for each available process.
4. Then after the process having the highest response ratio is executed first by the processor.
5. In a case, if two processes have the same response ratio then the tie is broken using the FCFS scheduling algorithm.

HRRN Scheduling Example:

Here, we have several processes with different burst and arrival times, and a Gantt chart to represent CPU allocation time.

Process	Burst Time	Arrival Time
P1	3	1
P2	6	3
P3	8	5
P4	4	7
P5	5	8

The Gantt Chart according to HRRN will be:



Explanation
Given below is the explanation of the above example

At time=0 there is no process available in the ready queue, so from 0 to 1 CPU is idle. Thus 0 to 1 is considered as CPU idle time.

At time=1, only the process P1 is available in the ready queue. So, process P1 executes till its completion.

After process P1, at time=4 only process P2 arrived, so the process P2 gets executed because the operating system did not have any other option.

At time=10, the processes P3, P4, and P5 were in the ready queue. So in order to schedule the next process after P2, we need to calculate the response ratio.

In this step, we are going to calculate the response ratio for P3, P4, and P5.

Response Ratio = W+S/S

RR(P3) = [(10-5) +8]/8

= 1.625

RR(P4) = [(10-7) +4]/4

= 1.75

RR(P5) = [(10-8) +5]/5

= 1.4

From the above results, it is clear that Process P4 has the Highest Response ratio, so the Process P4 is schedule after P2.

At time t=10, execute process P4 due to its large value of Response ratio.

Now in the ready queue, we have two processes P3 and P5, after the execution of P4 let us calculate the response ratio of P3 and P5

RR (P3) = [(14-5) +8]/8

=2.125

RR (P5) = [(14-8) +5]/5

=2.2

From the above results,it is clear that Process P5 has the Highest Response ratio, so the Process P5 is schedule after P4

At t=14, process P5 is executed.

After the complete execution of P5, P3 is in the ready queue so at time t=19 P3 gets executed.

In the table given below, we will calculate turnaround time, waiting time, and completion time for all the Processes.

Process	Arrival Time	Burst Time	Completion time	Turn around Time **Turn Around Time = Completion Time – Arrival Time**	Waiting Time **Waiting Time = Turn Around Time – Burst Time**
P1	1	3	4	4-1=3	3-3=0
P2	3	6	10	10-3=7	7-6=1
P3	5	8	27	27-5=22	22-8=14
P4	7	4	14	14-7=7	7-4=3
P5	8	5	19	19-8=11	11-5=6

Average waiting time is calculated by adding the waiting time of all processes and then dividing them by no. of processes.

average waiting time = waiting time of all processes/ no.of processes

average waiting time=0+1+14+3+6/5=24/5 = **4.8ms**

Multilevel Feedback Queue Scheduling:

The multilevel feedback queue scheduling algorithm, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes—which are typically characterized by short CPU bursts —in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.

For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (Figure 5.9). The scheduler first executes all processes in queue 0. Only when queue 0 is empty will it execute processes in queue 1. Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty. A process that arrives for queue 1 will preempt a process in queue 2. A process in queue 1 will in turn be preempted by a process arriving for queue 0.

An entering process is put in queue 0. A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2. Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty. To prevent starvation, a process that waits too long in a lower-priority queue may gradually be moved to a higher-priority queue.

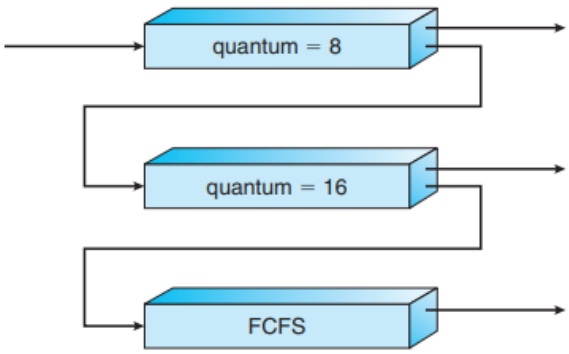


Figure 5.9 Multilevel feedback queues.

In general, a multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue
- The method used to determine which queue a process will enter when that process needs service

The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design. Unfortunately, it is also the most complex algorithm, since defining the best scheduler requires some means by which to select values for all the parameters.

Multi-Processor Scheduling:

Traditionally, the term multiprocessor referred to systems that provided multiple physical processors, where each processor contained one single-core CPU. However, the definition of multiprocessor has evolved significantly, and on modern computing systems, multiprocessor now applies to the following system architectures:

- Multicore CPUs
- Multithreaded cores
- NUMA systems
- Heterogeneous multiprocessing.

Approaches to Multiple-Processor Scheduling:

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor — the master server. The other processors execute only user code. This asymmetric multiprocessing is simple because only one core accesses the system data structures, reducing the need for data sharing. The downfall of this approach

is the master server becoming a potential bottleneck where overall system performance may be reduced. The standard approach for supporting

multiprocessors is symmetric multiprocessing (SMP), where each processor is self-scheduling. Scheduling proceeds by having the scheduler for each processor examine the ready queue and select a thread to run. Note that this provides two possible strategies for organizing the threads eligible to be scheduled:

1. All threads may be in a common ready queue.
2. Each processor may have its own private queue of threads.

These two strategies are contrasted in Figure 5.11. If we select the first option, we have a possible race condition on the shared ready queue and therefore must ensure that two separate processors do not choose to schedule the same thread and that threads are not lost from the queue.

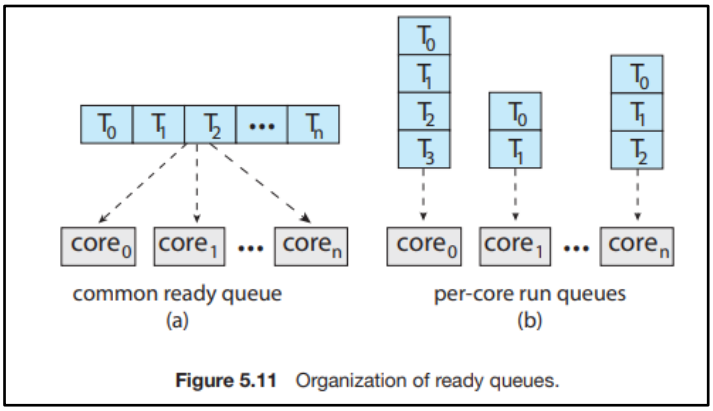


Figure 5.11 Organization of ready queues.

Multicore Processors:

Traditionally, SMP systems have allowed several processes to run in parallel by providing multiple physical processors. However, most contemporary computer hardware now places multiple computing cores on the same physical chip, resulting in a multicore processor. Each core maintains its architectural state and thus appears to the operating system to be a separate logical CPU. SMP systems that use multicore processors are faster and consume less power than systems in which each CPU has its own physical chip. Multicore processors may complicate scheduling issues. Let's consider how this can happen. Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a memory stall, occurs primarily because modern processors operate at much faster speeds than memory.

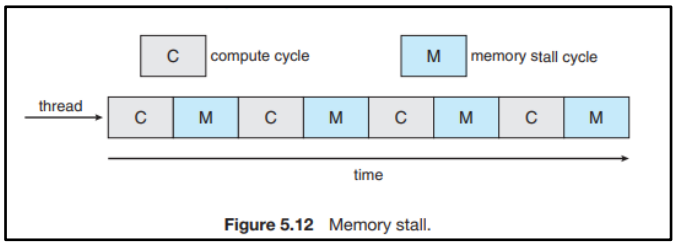


Figure 5.12 Memory stall.

To remedy this situation, many recent hardware designs have implemented multithreaded processing cores in which two (or more) hardware threads are assigned to each core. That way, if one hardware thread stalls while waiting for memory, the core can switch to another thread. Figure 5.13 illustrates

a dual-threaded processing core on which the execution of thread 0 and the execution of thread 1 are interleaved. From an operating system perspective, each hardware thread maintains its architectural state, such as instruction pointer and register set, and thus appears as a logical CPU that is available to run a software thread. This technique—known as chip multithreading (CMT) —is illustrated in Figure 5.14. Here, the processor contains four computing cores, with each core containing two hardware threads. From the perspective of the operating system, there are eight logical CPUs.

Intel processors use the term hyper-threading (also known as simultaneous multithreading or SMT) to describe assigning multiple hardware threads to a single processing core. Contemporary Intel processors—such as the i7—support two threads per core, while the Oracle Sparc M7 processor supports eight threads per core, with eight cores per processor, thus providing the operating system with 64 logical CPUs.

In general, there are two ways to multithread a processing core: coarse grained and fine-graine multithreading. With coarse-grained multithreading, a thread executes on a core until a long-latency event such as a memory stall occurs. Because of the delay caused by the long-latency event, the core must switch to another thread to begin execution.

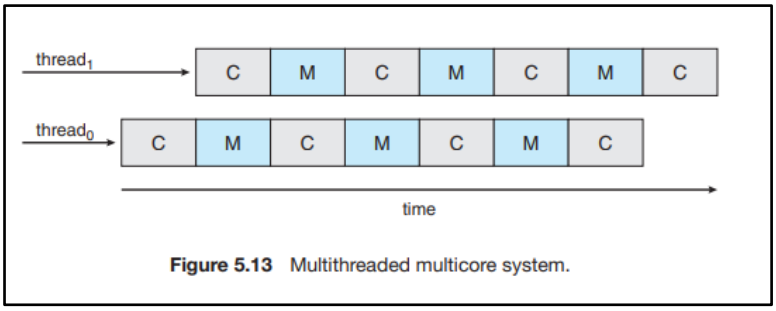


Figure 5.13 Multithreaded multicore system.

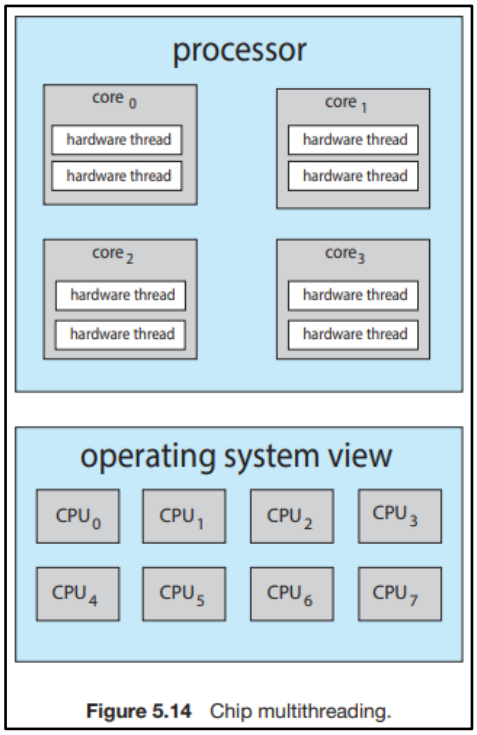


Figure 5.14 Chip multithreading.

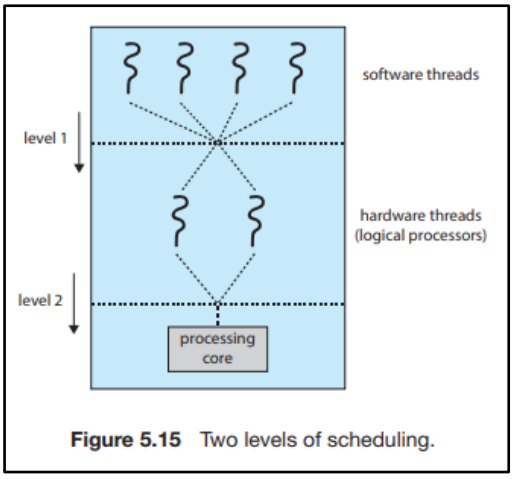


Figure 5.15 Two levels of scheduling.

Load Balancing:

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private ready queue of eligible threads to execute. On systems with a common run queue, load balancing is unnecessary, because once a processor becomes idle, it immediately extracts a runnable thread from the common ready queue.

There are two general approaches to load balancing: push migration and pull migration. With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) threads from overloaded to idle or less-busy processors. Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are, in fact, often implemented in parallel on load-balancing systems. For example, the Linux CFS scheduler (described in Section 5.7.1) and the ULE scheduler available for FreeBSD systems implement both techniques.

Processor Affinity:

The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most operating systems with SMP support try to avoid migrating a thread from one processor to another and instead attempt to keep a thread running on the same processor and take advantage of a warm cache. This is known as processor affinity — that is, a process has an affinity for the processor on which it is currently running.

The two strategies described in Section 5.5.1 for organizing the queue of threads available for scheduling have implications for processor affinity. If we adopt the approach of a common ready queue, a thread may be selected for execution by any processor.

Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as soft affinity. Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors during load balancing. In contrast, some systems provide system calls that support hard affinity, thereby allowing a process to specify a subset of processors on which it can run. Many systems provide both soft and hard affinity. For example,

Linux implements soft affinity, but it also provides the sched_setaffinity() system call, which supports hard affinity by allowing a thread to specify the set of CPUs on which it is eligible to run.

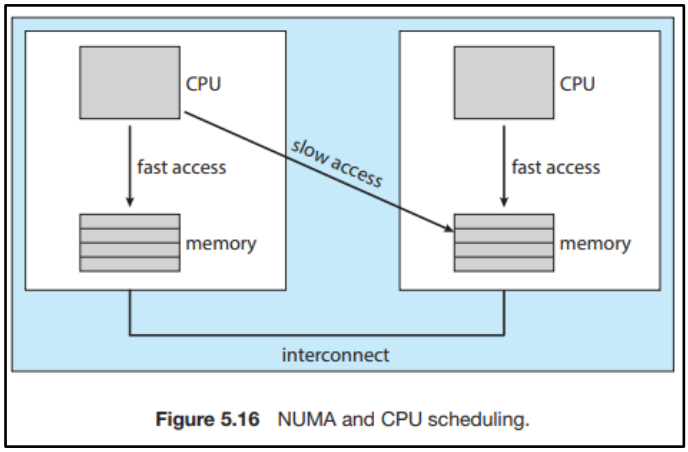


Figure 5.16 NUMA and CPU scheduling.

Heterogeneous Multiprocessing:

Although mobile systems now include multicore architectures, some systems are now designed using cores that run the same instruction set, yet vary in terms of their clock speed and power management, including the ability to adjust the power consumption of a core to the point of idling the core. Such systems are known as heterogeneous multiprocessing (HMP). Note this is not a form of asymmetric multiprocessing as described in Section 5.5.1 as both system and user tasks can run on any core. Rather, the intention behind HMP is to better manage power consumption by assigning tasks to certain cores based upon the specific demands of the task.

For ARM processors that support it, this type of architecture is known as big. LITTLE where higher-performance big cores are combined with energy efficient LITTLE cores. Big cores consume greater energy and therefore should only be used for short periods of time. Likewise, little cores use less energy and can therefore be used for longer periods.