Operating System Structure

1. Simple Structure

2. Layered Approach

3. Microkernels

4. Modules

1. Simple Structure

- Such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was written to provide the most functionality in the least space, so it was not carefully divided into modules.

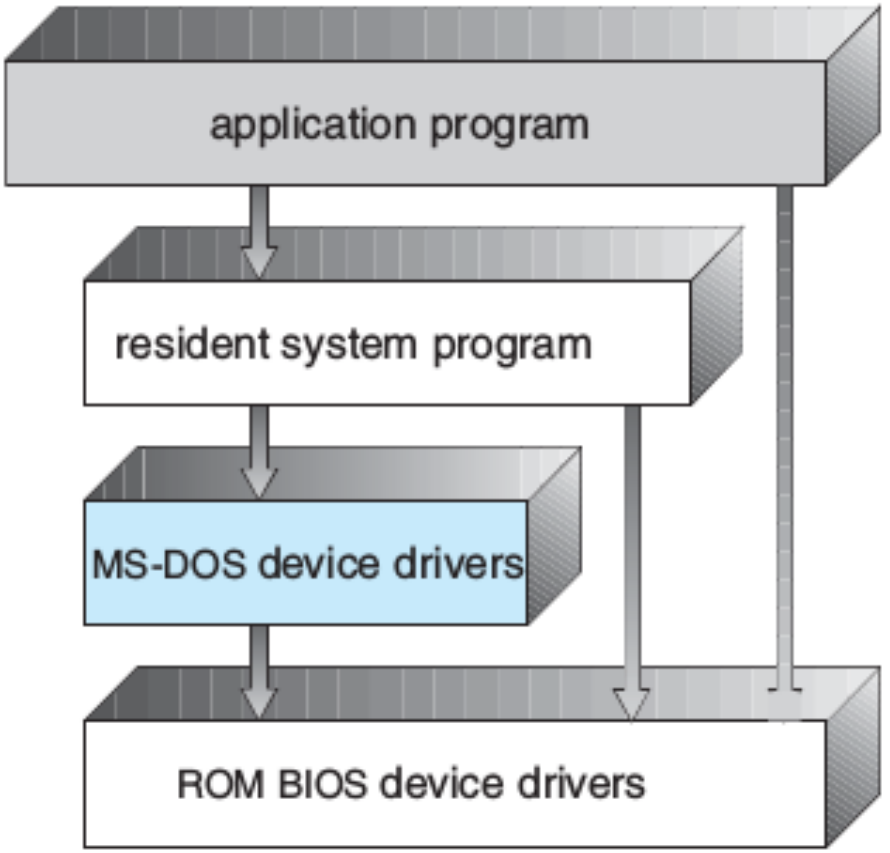

Figure 2.11 MS-DOS layer structure.

- In MS-DOS , the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.
- Another example of limited structuring is the original UNIX operating system. Like MS-DOS , UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs.
- The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. That is an enormous amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain.

2. Layered Approach

- The operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.
- A typical operating-system layer—say, layer M —consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers.

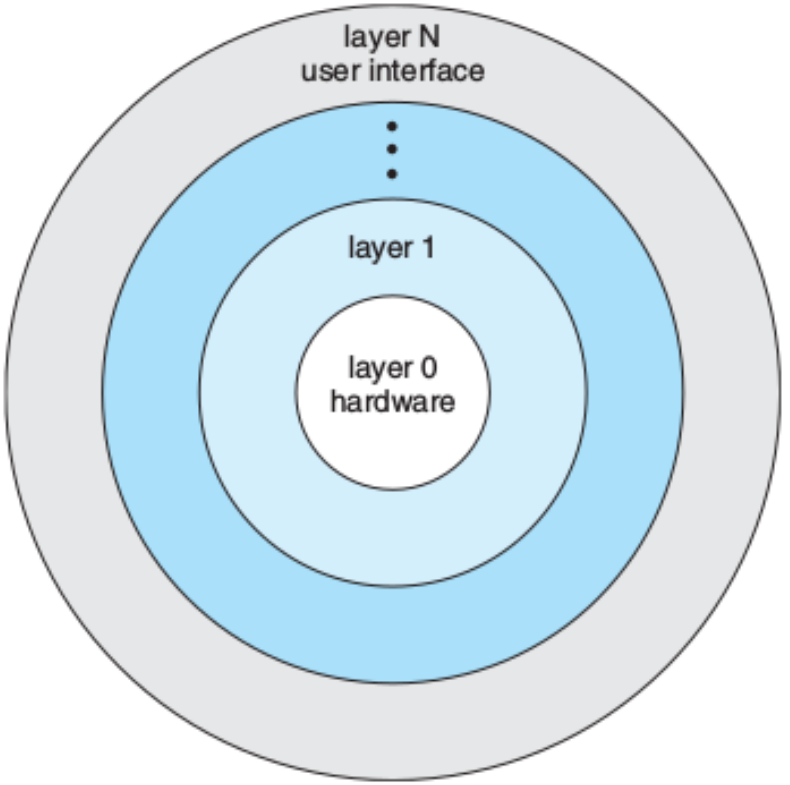

Figure 2.13 A layered operating system.

- The layers are selected so that each uses functions (operations) and services of only lower-level layers. This approach simplifies debugging and system verification.
- The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions.
- Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged.
- Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

- The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary.
- Layered implementations tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU -scheduling layer, which is then passed to the hardware.
- At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call. The net result is a system call that takes longer than does one on a nonlayered system.

3. Microkernels

- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.
- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through message passing.
- For example, if the client program wishes to access a file, it must interact with the file server. The client program and service never interact directly. Rather, they communicate indirectly by exchanging messages with the microkernel.
- One benefit of the microkernel approach is that it makes extending the operating system easier. All new services are added to user space and consequently do not require modification of the kernel.

- When the kernel does have to be modified, the changes tend to be fewer, because the microkernel is a smaller kernel.
- The resulting operating system is easier to port from one hardware design to another.
- The microkernel also provides more security and reliability, since most services are running as user—rather than kernel—processes. If a service fails, the rest of the operating system remains untouched.

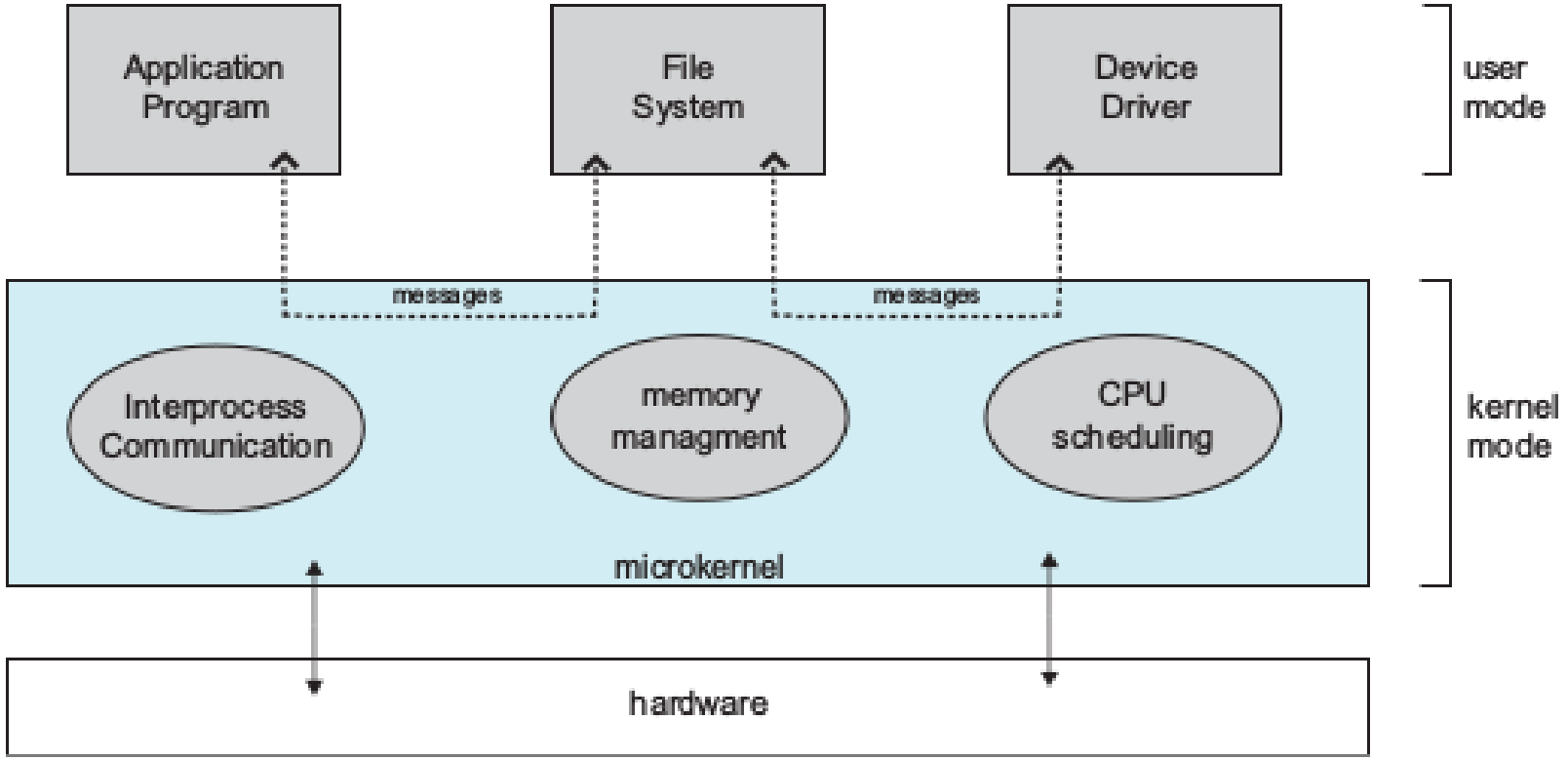

Figure 2.14 Architecture of a typical microkernel.

4. Modules

- The kernel has a set of core components and links in additional services via modules, either at boot time or during run time.
- This type of design is common in modern implementations of UNIX , such as Solaris, Linux, and Mac OS X , as well as Windows.
- Kernel provides core services while other services are implemented dynamically, as the kernel is running.
- Linking services dynamically is preferable to adding new features directly to the kernel, which would require recompiling the kernel every time a change was made.
- The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system, because any module can call any other module.
- The approach is also similar to the microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

System Boot

- After an operating system is generated, it must be made available for use by the hardware. The procedure of starting a computer by loading the kernel is known as **booting** the system.
- On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution.
- When a CPU receives a reset event—for instance, when it is powered up or rebooted —the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program.
- Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be running.

The Process

- A process is a program in execution. A process is more than the 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 registers.
- A process generally also includes the **process stack**, which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section**, which contains global variables.
- A process may also include a **heap**, which is memory that is dynamically allocated during process run time.
- 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.

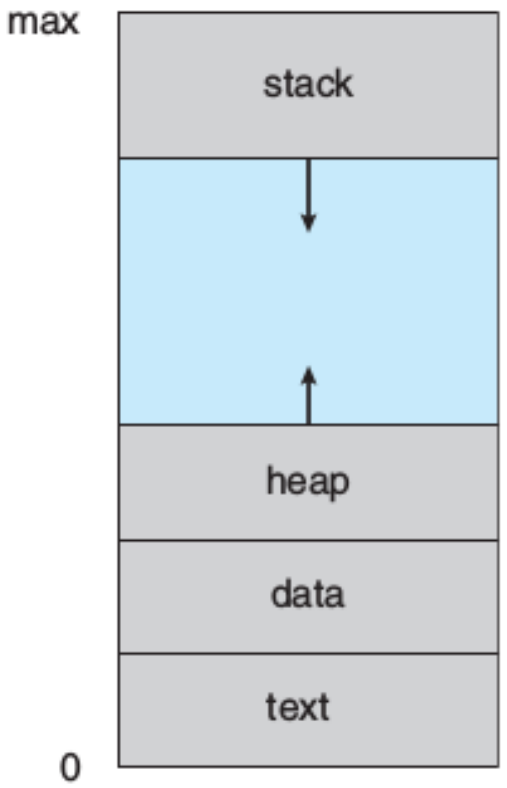

Figure 3.1 Process in memory.

- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences. For instance, several users may be running different copies of the mail program, or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

- A process itself can be an execution environment for other code. The Java programming environment provides a good example.
- An executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions.
- For example, to run the compiled Java program Program.class, we would enter

  java Program
- The command **java** runs the JVM as an ordinary process, which in turns executes the Java program Program in the virtual machine.

Process State

- As a process executes, it changes state. A process may be in one of the following states:

  1. **New**. The process is being created.

  2. **Running**. Instructions are being executed.

  3. **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).

  4. **Ready**. The process is waiting to be assigned to a processor.

  5. **Terminated**. The process has finished execution.

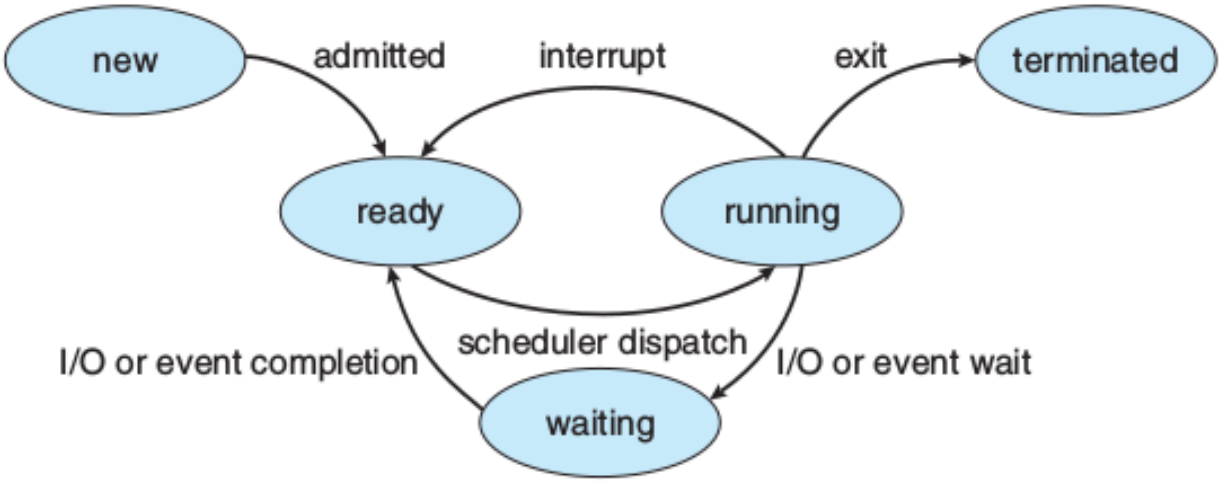

Figure 3.2 Diagram of process state.

Process Control Block

- Each process is represented in the operating system by a process control block **(PCB)**—also called a task control block. It contains many pieces of information associated with a specific process, including these:

  1. **Process state**. The state may be new, ready, running, waiting, halted, and so on.

  2. **Program counter**. The counter indicates the address of the next instruction to be executed for this process.

  3. **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. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.

  4. **CPU -scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

5. **Memory-management information**. This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

6. **Accounting information**. This information includes the amount of CPU and real time used, time limits, job or process numbers, and so on.

7. **I/O status information**. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

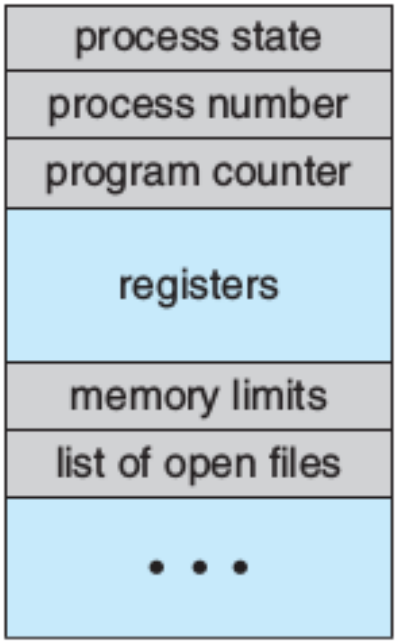

Figure 3.3 Process control block (PCB).

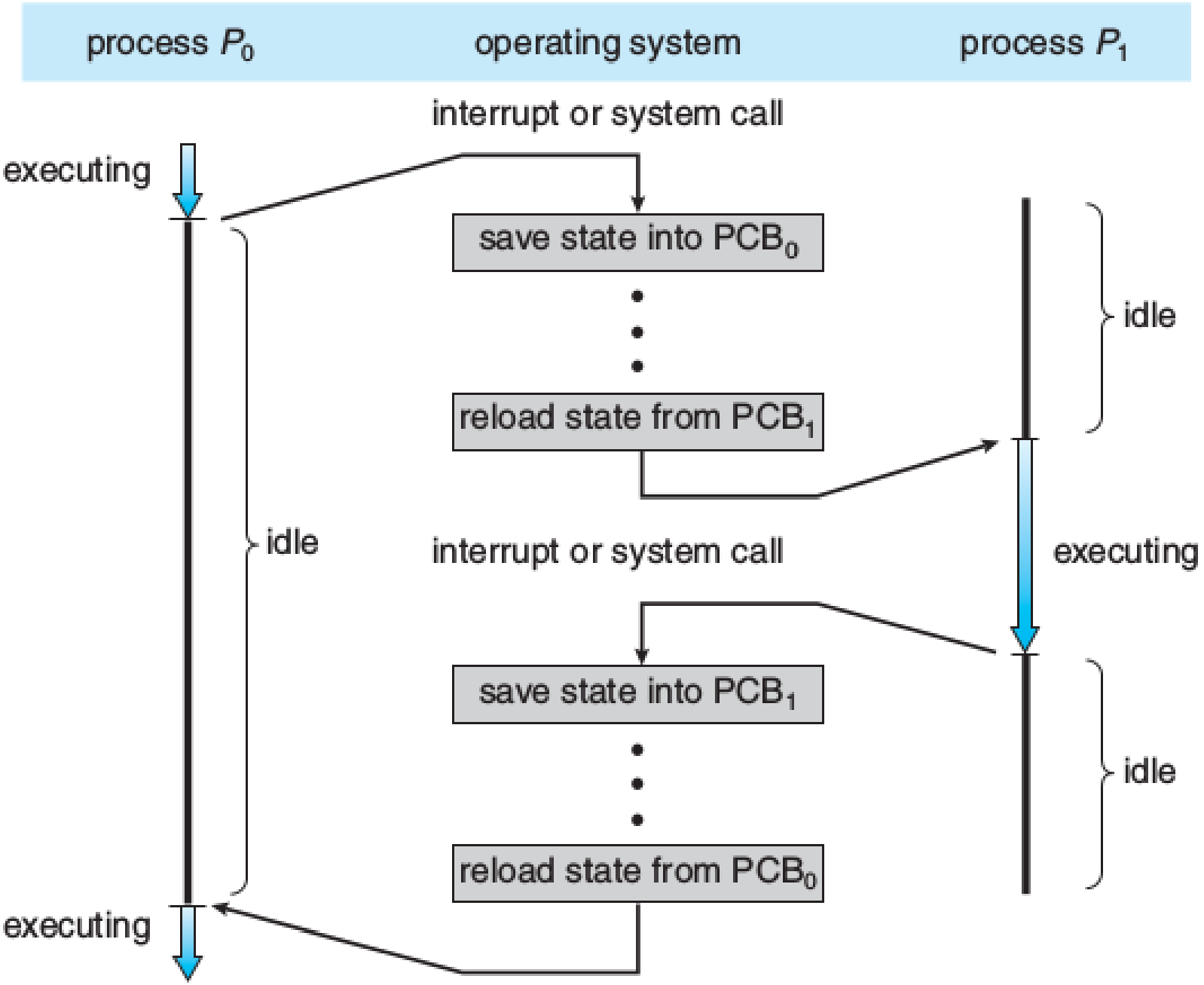

Figure 3.4 Diagram showing CPU switch from process to process.

Threads

- A process is a program that performs a single thread of execution.
- 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. On a system that supports threads, the PCB is expanded to include information for each thread.
- Simultaneously type in characters and run the spell checker within the same process, for example

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.
- For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling Queues

- As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- When a process is allocated the CPU , it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request.
- Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a **device queue**.

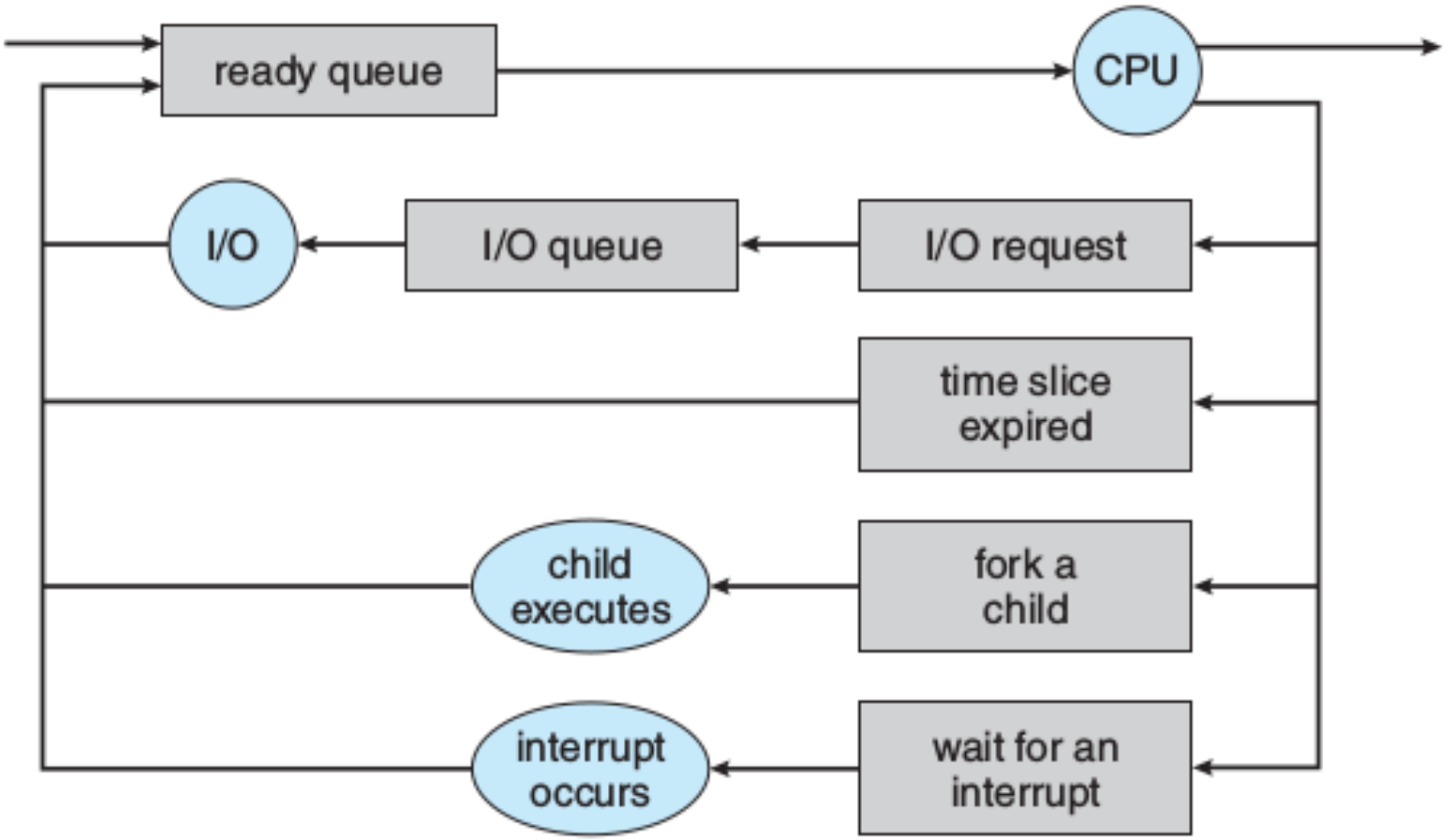

Figure 3.6 Queueing-diagram representation of process scheduling.

- Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

- A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

  1. The process could issue an I/O request and then be placed in an I/O queue.

  2. The process could create a new child process and wait for the child's termination.

  3. The process could be removed forcibly from the CPU , as a result of an interrupt, and be put back in the ready queue.

- In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue.
- A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Schedulers

- A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues. The selection process is carried out by the appropriate scheduler.
- The **long-term scheduler**, or **job scheduler**, selects processes from job pool and loads them into memory for execution.
- The **short-term scheduler**, or **CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.
- The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request.
- Because of the short time between executions, the short-term scheduler must be fast.

- The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next.
- The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). 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.
- Thus, the long-term scheduler may need to be invoked only when a process leaves the system.
- Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

- An **I/O -bound process** is one that spends more of its time doing I/O than it spends doing computations.
- A **CPU -bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.
- It is important that the long-term scheduler select a good process mix of I/O -bound and CPU -bound processes.
- If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do.
- If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced.
- The system with the best performance will thus have a combination of CPU -bound and I/O -bound processes.

- A **medium-term scheduler** is removes a process from memory and thus reduces the degree of multiprogramming.
- Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping**.
- The process is swapped out, and is later swapped in, by the medium-term scheduler.
- Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

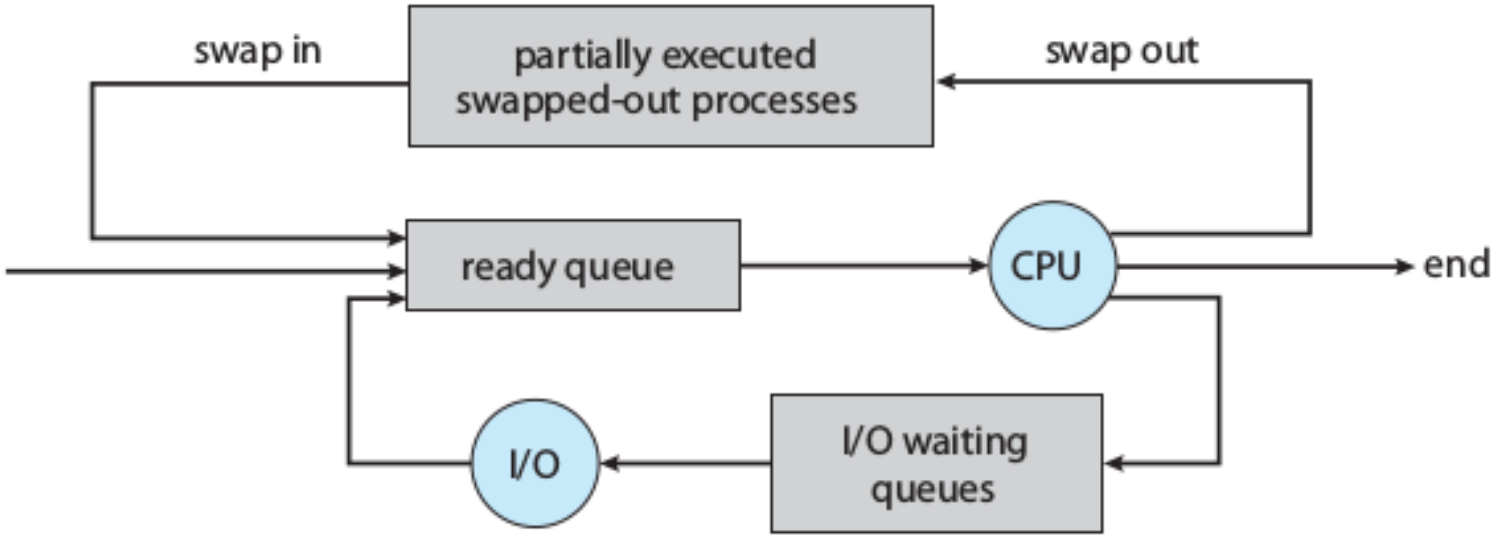

Figure 3.7 Addition of medium-term scheduling to the queueing diagram.

Context Switch

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory-management information.
- Switching the CPU to another process requires performing a **state save** of the current process and a **state restore** of a different process. This task is known as a **context switch**.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure overhead, because the system does no useful work while switching.

Process Creation

- During the course of execution, a process may create several new processes. The creating process is called a **parent** process, and the new processes are called the **children** of that process.
- Most operating systems (including UNIX , Linux, and Windows) identify processes according to a unique **process identifier** (or pid), which is typically an integer number.
- The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel.
- When a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.

- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.
- The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.
- Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.
- The parent process may pass along initialization data (input) to the child process. For example, consider a process whose function is to display the contents of a file —say, image.jpg —on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file image.jpg. Using that file name, it will open the file and write the contents out. It may also get the name of the output device.

- When a process creates a new process, two possibilities for execution exist:

  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 address-space possibilities for 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 new program loaded into it.

- In UNIX, 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.
- This mechanism allows the parent process to communicate easily with its child process.
- Both processes (the parent and the child) continue execution at the instruction after the fork() , with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
- After a fork() system call, one of the two processes typically uses the **exec()** system call to replace the process's memory space with a new program.
- The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution.
- 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.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX `fork()` system call.

- In the example, the child process overlays its address space with the UNIX command /bin/ls (used to get a directory listing) using the execlp() system call (execlp() is a version of the exec() system call).
- The parent waits for the child process to complete with the wait() system call.
- When the child process completes (by either implicitly or explicitly invoking exit()), the parent process resumes from the call to wait() , where it completes using the exit() system call.

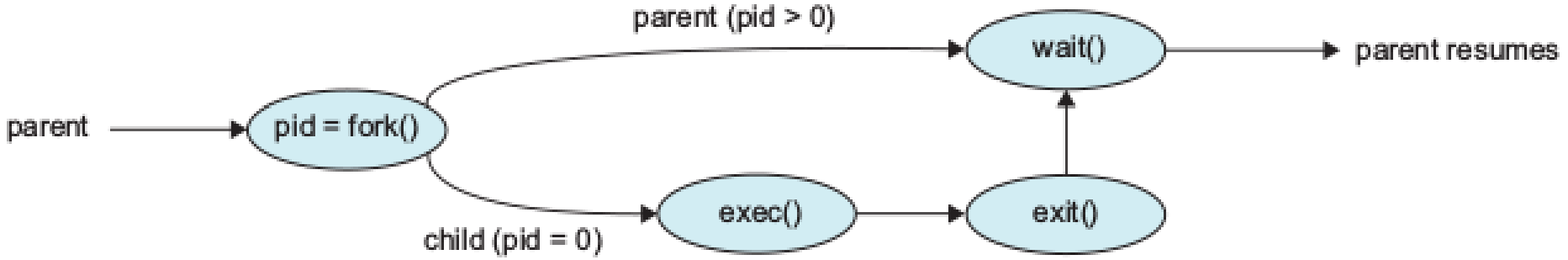

Figure 3.10 Process creation using the `fork()` system call.

Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
- At that point, the process may return a status value (typically an integer) 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 by the operating system.
- A process can cause the termination of another process via an appropriate system call. Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.
- A parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

- A parent may terminate the execution of one of its children for a variety of reasons, such as these:

  1. The child has exceeded its usage of some of the resources that it has been allocated.

  2. The task assigned to the child is no longer required.

  3. The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

- Some systems do not allow a child to exist if its parent has terminated. In such systems, 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.
- The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

```
pid t pid;

int status;

pid = wait(&status);
```

- When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait() , because the process table contains the process's exit status.
- A process that has terminated, but whose parent has not yet called wait() , is known as a **zombie** process. All processes transition to this state when they terminate, but generally they exist as zombies only briefly.
- Once the parent calls wait() , the process identifier of the zombie process and its entry in the process table are released.

- If a parent did not invoke wait() and instead terminated, leaving its child processes as **orphans**. Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes.
- The init process periodically invokes wait() , thereby allowing the exit status of any orphaned process to be collected and releasing the orphan’s process identifier and process-table entry.

Shell Programming

- Cat command

  cat [OPTION] [FILE]

  1. cat //listen to standard input

     This is a new line

     This is a new line

  2. cat > readme.txt //writes contents

     This is a readme file.

     This is a new line.

  3. cat readme.txt //prints contents

     This is a readme file.

     This is a new line.

  4. cat >> readme.txt //appends contents

     This is an appended line.

Awk command

- awk options 'selection _criteria {action }' input-file > output-file

  Options:

- -f program-file : Reads the AWK program source from the file program-file, instead of from the first command line argument.
- -F fs : Use fs for the input field separator

  1. awk '{print}' employee.txt //print every line of file

  2. awk '/manager/ {print}' employee.txt //print lines which contain pattern

  3. Splitting a Line Into Fields : For each record i.e line, the awk command splits the record delimited by whitespace character by default and stores it in the $n variables. If the line has 4 words, it will be stored in $1, $2, $3 and $4 respectively. Also, $0 represents the whole line.

  awk '{print $1,$4}' employee.txt

Built-In Variables In Awk

- Awk's built-in variables include the field variables—$1, $2, $3, and so on ($0 is the entire line) — that break a line of text into individual words or pieces called fields.

  1. NR: NR command keeps a current count of the number of input records. Remember that records are usually lines. Awk command performs the pattern/action statements once for each record in a file.

  2. NF: NF command keeps a count of the number of fields within the current input record.

  3. FS: FS command contains the field separator character which is used to divide fields on the input line. The default is "white space", meaning space and tab characters. FS can be reassigned to another character (typically in BEGIN) to change the field separator.

- awk '{print NR,$0}' employee.txt

  prints line no. and the line
- awk '{print $1,$NF}' employee.txt

  pritns first and last word of each line
- awk 'NR==3, NR==6 {print NR,$0}' employee.txt

  prints line 3 to 6
- awk '{print NR "- " $1 }' employee.txt

  prints line no., then ‘–‘ and then first word

Threads

- A thread is a basic unit of CPU utilization; it comprises a thread ID , a program counter, 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 (or heavyweight) 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.
- A web browser might have one thread display images or text while another thread retrieves data from the network, for example.

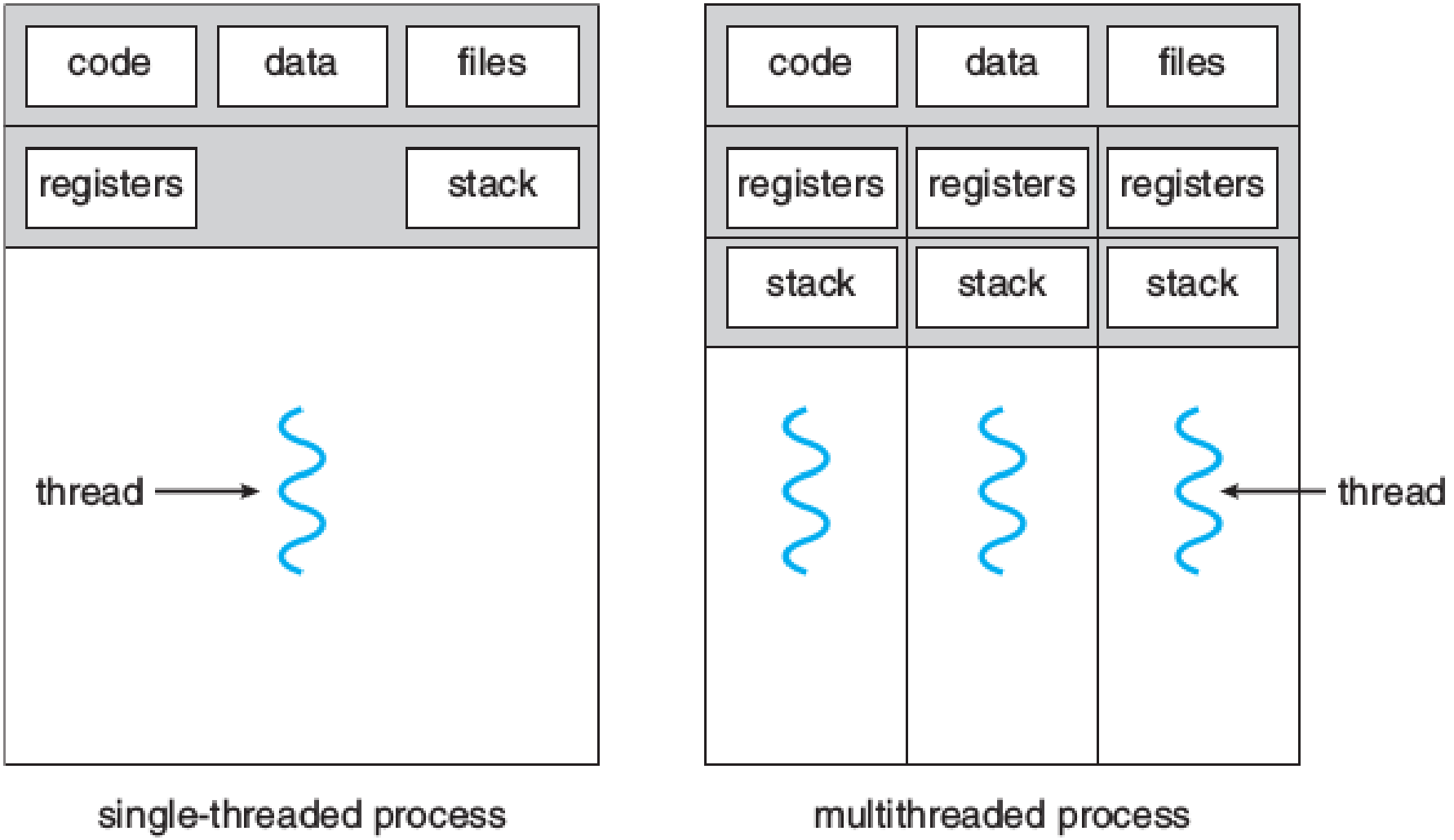

Figure 4.1 Single-threaded and multithreaded processes.

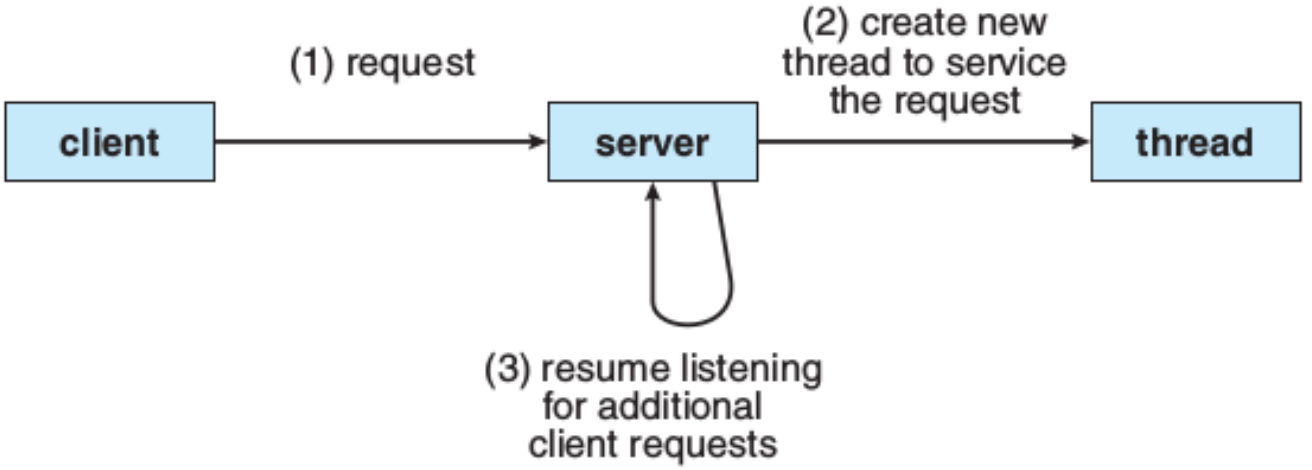

Figure 4.2 Multithreaded server architecture.

- A busy web server may have several (perhaps thousands of) clients concurrently accessing it.
- 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. Process creation is time consuming and resource intensive.
- 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 resume listening for additional requests.

Benefits

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.

2. Resource sharing. Processes can only share resources 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.

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.

4. **Scalability**. In a multiprocessor architecture, threads may run in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

Multicore Programming

- Multiple computing cores are placed on a single chip. Each core appears as a separate processor to the operating system.
- 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, because the processing core is capable of executing only one thread at a time.
- On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core.
- A system is **parallel** if it can perform more than one task simultaneously. In contrast, a **concurrent** system supports more than one task by allowing all the tasks to make progress.

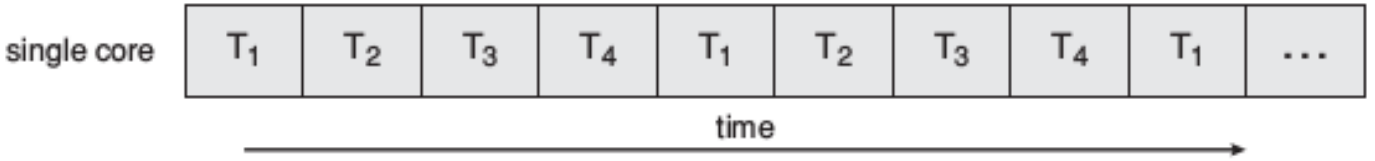

Figure 4.3 Concurrent execution on a single-core system.

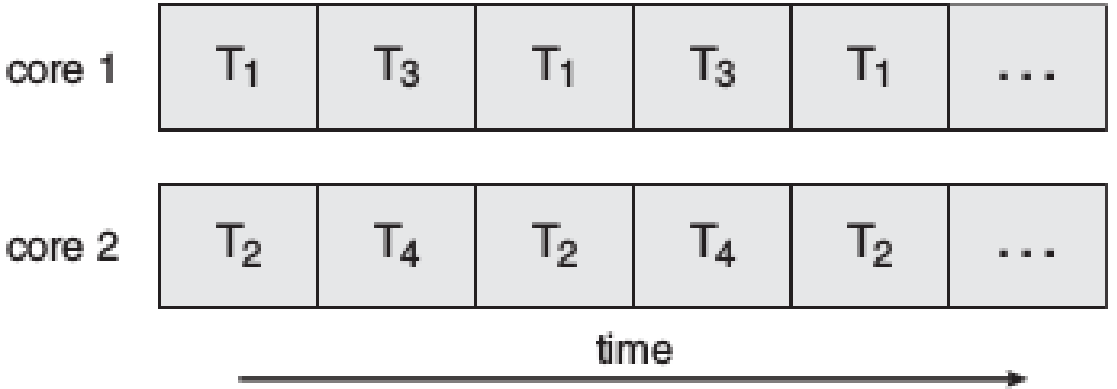

Figure 4.4 Parallel execution on a multicore system.

- Modern Intel CPUs frequently support two threads per core, while the Oracle T4 CPU supports eight threads per core.
- This support means that multiple threads can be loaded into the core for fast switching.

Programming challenges for multicore systems

1. **Identifying tasks**. This involves examining applications to find area 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. 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.

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**. 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 is inherently more difficult than testing and debugging single-threaded applications.

Types of 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. 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.

Multithreading models

- Support for threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads**.
- 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.
- A relationship must exist between user threads and kernel threads. There are three common ways of establishing such a relationship: the **many-to-one** model, the **one-to-one** model, and the **many-to-many** model.

1. **Many to one model**

- The many-to-one model maps many user-level threads to one kernel thread.
- Thread management is done by the thread library in user space, so it is efficient
- 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.

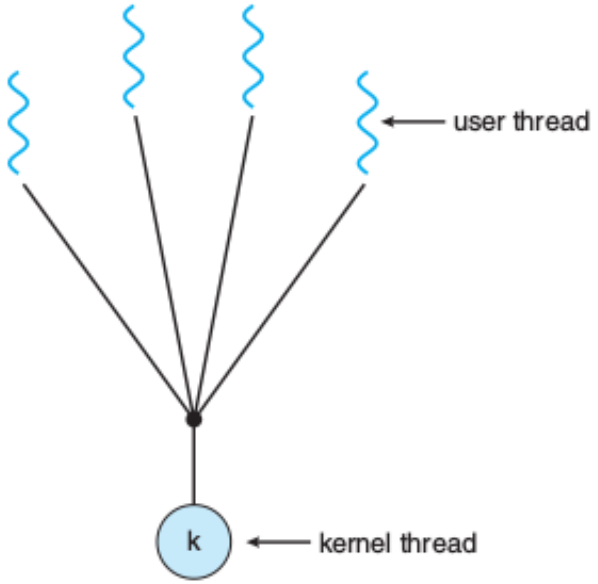

Figure 4.5 Many-to-one model.

2. **One to One model**

- The one-to-one model 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. The overhead of creating kernel threads can burden the performance of an application

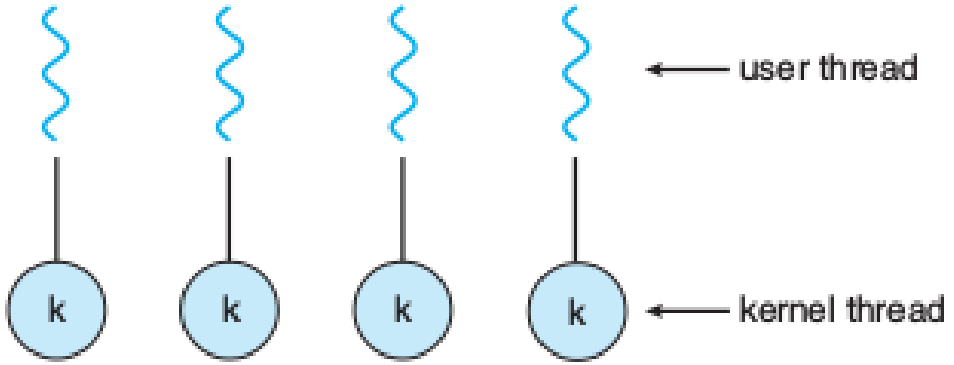

Figure 4.6 One-to-one model.

3. Many to Many model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

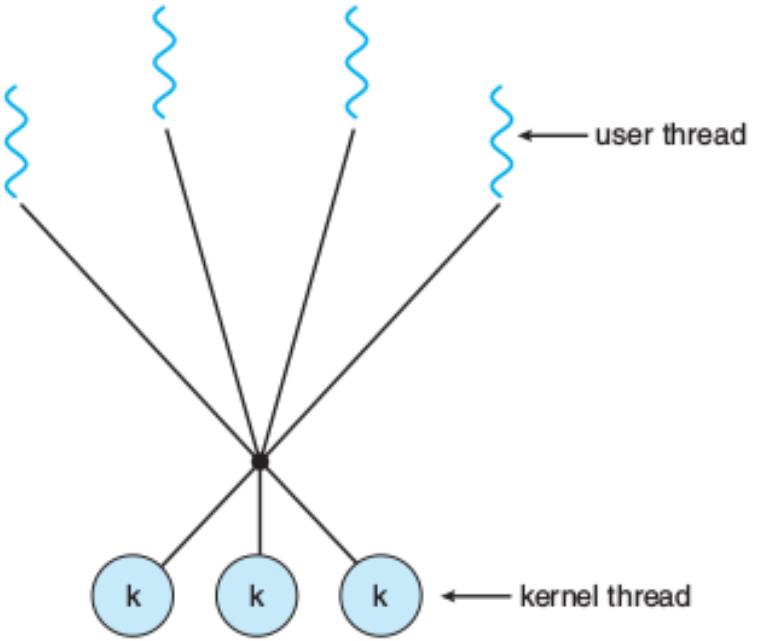

Figure 4.7 Many-to-many model.

- 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.**

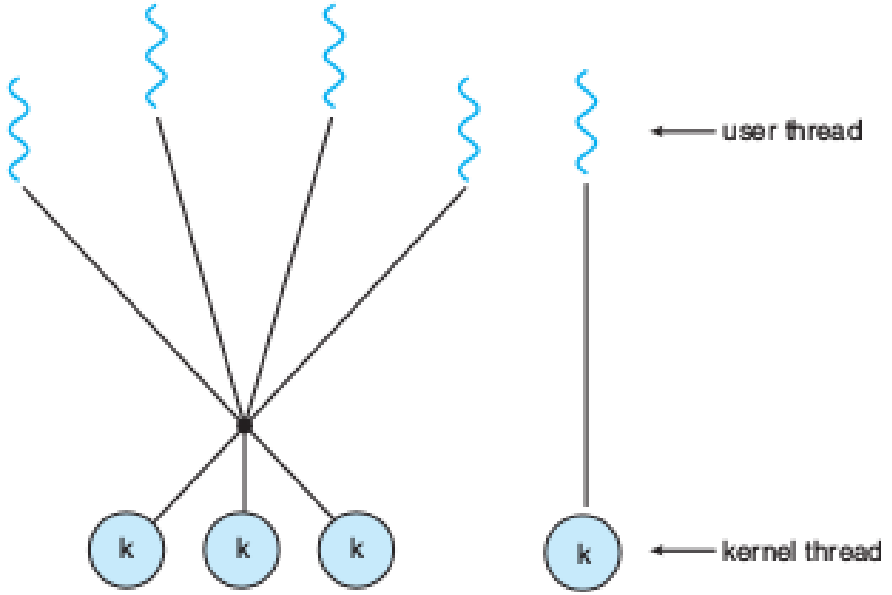

Figure 4.8 Two-level model.

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 (Portable OS Interface based on unix) Pthreads, Windows, and Java.
- 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.
- For POSIX and Windows threading, any data declared globally—that is, declared outside of any function—are shared among all threads belonging to the same process.
- Because Java has no notion of global data, access to shared data must be explicitly arranged between threads.
- Data declared local to a function are typically stored on the stack. Since each thread has its own stack, each thread has its own copy of local data.

- 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.
- Each thread runs independently of every other thread, and the parent thread need not know when its child terminates.
- Because the threads are independent, there is typically little data sharing between threads.

- **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 —the so-called **fork-join** strategy.
- Here, the threads created by the parent perform work concurrently, but the parent cannot continue until this work has been completed.
- Once each thread has finished its work, it terminates and joins with its parent. Only after all of the children have joined can the parent resume execution.
- Typically, synchronous threading involves significant data sharing among threads.

Pthreads

- Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.
- The following C program demonstrates the basic Pthreads API for constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread.
- In a Pthreads program, separate threads begin execution in a specified function. In the example, this is the runner() function.
- When this program begins, a single thread of control begins in main() . After some initialization, main() creates a second thread that begins control in the runner() function. Both threads share the global data sum .

```c
#include <pthread.h>
#include <stdio.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 */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   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 begin control 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);
}
```

- Each thread has a set of attributes, including stack size and scheduling information. The pthread_attr_t attr declaration represents the attributes for the thread. We set the attributes in the function call pthread_attr_init(&attr).
- pthread_join() function takes two parameters –

  1. **thread-id** Is the thread to wait for.

  2. **status** Is the location where the exit status of the joined thread is stored. This can be set to NULL if the exit status is not required.

- A simple method for waiting on several threads using the pthread join() function is to enclose the operation within a simple for loop.

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.