Unit 1

Introduction

1. What Operating System do

The computer system can be divided roughly into four components: the *hardware,* the *operating system,* the *application programs,* and the *users.*
The hardware—the central processing unit (CPU), the memory and the input/output (I/O) devices—provides the basic computing resources for the system. The application programs—such as word processors, spreadsheets and Web browsers—define the ways in which these resources are used to solve users' computing problems. The operating system controls the hardware and coordinates its use among the various application programs for the various users.

Examples of Operating Systems:

- **Windows:** Popular OS developed by Microsoft, known for its user-friendly GUI.
- **macOS:** Apple's OS designed for its Mac computers, known for its integration with Apple's ecosystem.
- **Linux:** An open-source OS used widely in servers, desktops, and embedded systems.
- **Android:** A Linux-based OS designed for mobile devices, particularly smartphones and tablets.
- **iOS:** Apple's mobile operating system for iPhones and iPads, known for its security and smooth user experience.

Operating systems are essential for the functioning of computers and mobile devices, providing a foundation for all applications and services running on these devices.

Definition
An **operating system (OS)** is a critical software layer that manages computer hardware and software resources, providing a stable and consistent environment for applications to run. It acts as an intermediary between users and the computer hardware, ensuring that applications function smoothly and efficiently by handling tasks such as process management, memory management, file systems and input/output operations.

The operating system is the one program running at all times on the computer—usually called the **kernel**. (Along with the kernel, there are two other types of programs: **system programs**, which are associated with the operating system but are not necessarily part of the kernel, and **application programs**, which include all programs not associated with the operation of the system.) Mobile operating systems often include not only a core kernel but also middleware—a set of software frameworks that provide additional services to application developers. For example, each of the two most prominent mobile operating systems—Apple's iOS and Google's Android—features a core kernel along with middleware that supports databases, multimedia and graphics.

User View
Most computer **users sit in front of a PC**, consisting of a monitor, keyboard, mouse and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize

the work (or play) that the user is performing. In this case, the operating system is designed mostly for **ease of use**, with some attention paid to performance and none paid to **resource utilization**—how various hardware and software resources are shared. Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.

In other cases, a **user sits at a terminal connected to a mainframe or a minicomputer**. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization— to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than her fair share.

In still other cases, users sit at **workstations connected to networks of other workstations and servers**. These users have dedicated resources at their disposal, but they also share resources such as networking and servers, including file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer.

A **slightly different view** of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a **control program**. A **control program** manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

2. Operating System structure

One of the most important aspects of operating systems is the ability to multiprogram. A single program cannot, in general, keep either the CPU or the I/O devices busy at all times. Single users frequently have multiple programs running. Multiprogramming increases CPU utilization by organizing jobs (code and data) so that the CPU always has one to execute.

The idea is as follows: The operating system keeps several jobs in memory simultaneously. Since, in general, main memory is too small to accommodate all jobs, the jobs are kept initially on the disk in the job pool. This pool consists of all processes residing on disk awaiting allocation of main memory. The set of jobs in memory can be a subset of the jobs kept in the job pool. The operating system picks and begins to execute one of the jobs in memory. Eventually, the job may have to wait for some task, such as an I/O operation, to complete. In a non-multi-programmed system, the CPU would sit idle. In a multi-programmed system, The OS simply switches to and executes, another job. When ***that*** job needs to wait, the CPU switches to ***another*** job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as at least one job needs to execute, the CPU is never idle. This idea is common in other life situations. A **lawyer** does not work for only one client at a time, for example. While one case is waiting to go to trial or have papers typed, the lawyer can work on another case. If he has enough clients, the lawyer will never be idle for lack of work. **Time sharing** (or **multitasking**) is a logical extension of multiprogramming. In time-sharing systems, the CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running.

Time sharing requires an **interactive** computer system, which provides direct communication between the user and the system. The user gives instructions to the operating system or to a program directly, using an input device such as a keyboard, mouse, touch pad, or touch screen, and waits for immediate results on an output device. Accordingly, the **response time** should be short—typically less than one second.

A time-shared operating system allows many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that the entire computer system is dedicated to his use, even though it is being shared among many users.

A time-shared operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program loaded into memory and executing is called a **process**. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. Time sharing and multiprogramming require that several jobs be kept simultaneously in memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must choose among them. Making this decision involves **job scheduling**.

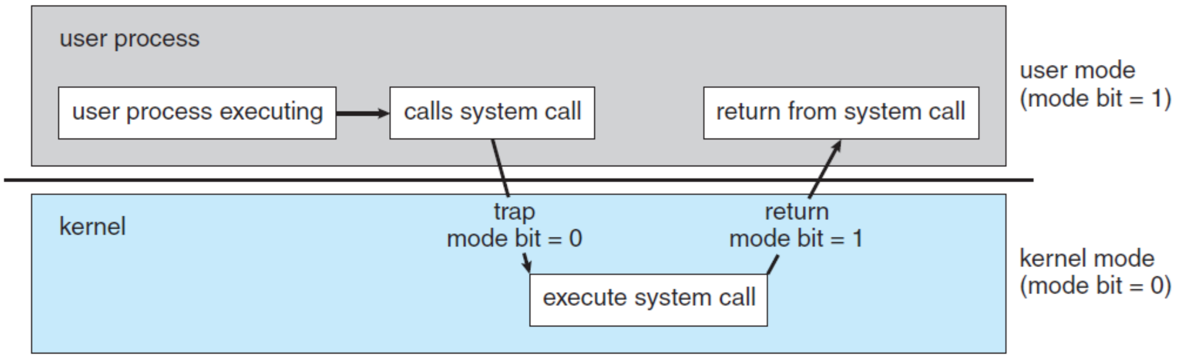
In a time-sharing system, the operating system must ensure reasonable response time. This goal is sometimes accomplished through **swapping**, whereby processes are swapped in and out of main memory to the disk. A time-sharing system must also provide a file system. The file system resides on a collection of disks; hence, disk management must be provided. In addition, a time-sharing system provides a mechanism for protecting resources from inappropriate use.

To ensure orderly execution, the system must provide mechanisms for job synchronization and communication and it may ensure that jobs do not get stuck in a deadlock, forever waiting for one another.

3. Operating System Operations

The modern operating systems are **interrupt driven**. If there are no processes to execute, no I/O devices to service, and no users to whom to respond, an operating system will sit quietly, waiting for something to happen. Events are almost always signalled by the occurrence of an interrupt or a trap. A **trap** (or an **exception**) is a software-generated interrupt caused either by an error (for example, division by zero or invalid memory access) or by a specific request from a user program that an operating-system service be performed. The interrupt-driven nature of an operating system defines that system's general structure. For each type of interrupt, separate segments of code in the operating system determine what action should be taken. An interrupt service routine is provided to deal with the interrupt.

Dual-Mode and Multimode Operation



Transition from user to kernel mode.

Two separate *modes* of operation: user mode and kernel mode (also called supervisor mode, system mode, or privileged mode). A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfil the request. At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (that is, changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.
The dual mode of operation provides us with the means for protecting the operating system from errant users—and errant users from one another. We accomplish this protection by designating some of the machine instructions that may cause harm as **privileged instructions**. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.
The instruction to switch to kernel mode is an example of a privileged instruction. Some other examples include I/O control, timer management and interrupt management.
In **virtual machine manager (VMM)** mode, the VMM has more privileges than user processes but fewer than the kernel. Initial control resides in the operating system, where instructions are executed in kernel mode. When control is given to a user application, the mode is set to user mode. Eventually, control is switched back to the operating system via an interrupt, a trap, or a system call.
System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. System call is a method used by a process to request action by the operating system. When a system call is executed, it is typically

treated by the hardware as a software interrupt. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred; a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory (with pointers to the memory locations
passed in registers). The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

Timer

We must ensure that the operating system maintains control over the CPU. We cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a timer. A timer can be set to interrupt the computer after a specified period. The period may be fixed (for example, 1/60 second) or variable (for example, from 1 millisecond to 1 second). A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs. For instance, a 10-bit counter with a 1-millisecond clock allows interrupts at intervals from 1 millisecond to 1,024 milliseconds, in steps of 1 millisecond.
Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Clearly, instructions that modify the content of the timer are privileged. We can use the timer to prevent a user program from running too long.

System Structures

1. Operating system services

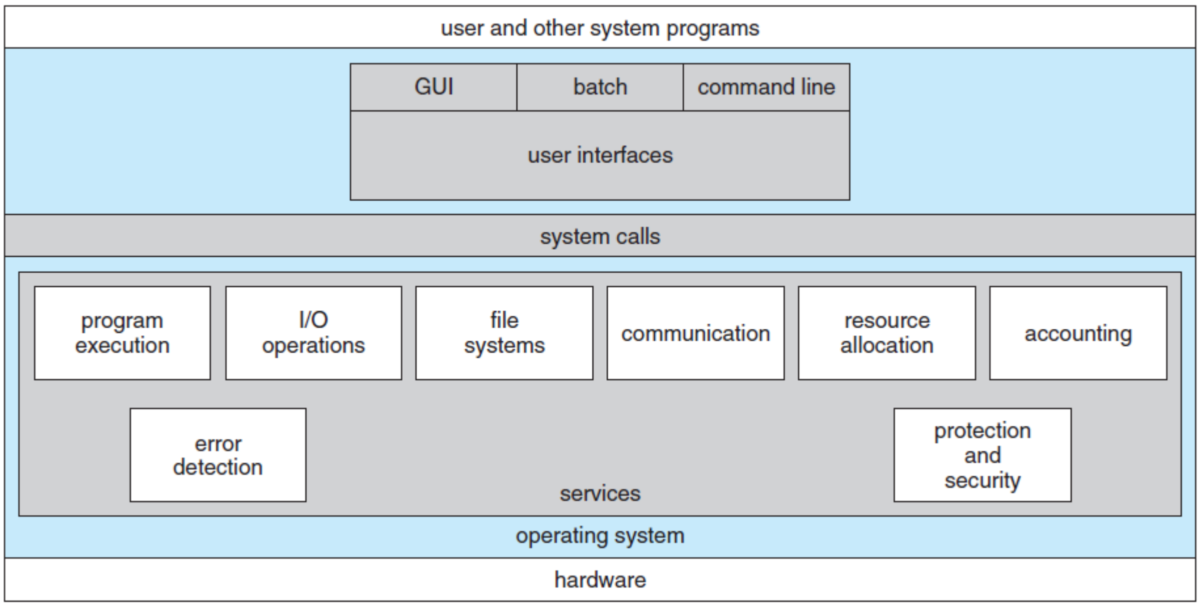
An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided differ from one operating system to another, but we can identify common classes. These operating system services are provided for the convenience of the programmer, to make the programming task easier. The below figure shows one view of the various operating-system services and how they interrelate. One set of operating system services provides functions that are helpful to the user.

User interface. Almost all operating systems have a **user interface** (**UI**). This interface can take several forms. One is a **command-line interface** (**CLI**), which uses text commands and a method for entering them (say, a keyboard for typing in commands in a specific format with specific options). Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a **graphical user interface** (**GUI**) is used.

Program execution. The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally.

I/O operations. A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as recording to a CD or DVD drive or blanking a display screen). For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the operating system must provide a means to do I/O.

File-system manipulation. The file system is of particular interest. Obviously, programs need to read and write files and directories. They also need to create and delete them by name, search for a given file, and list file information. Finally, some operating systems include **permissions management** to allow or deny access to files or directories based on file ownership.



A view of operating system services.

Communications. There are many circumstances in which one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

• **Error detection**. The operating system needs to be detecting and correcting errors constantly. Errors may occur in the **CPU and memory hardware** (such as a memory error or a power failure), in **I/O devices** (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the **user program** (such as an arithmetic overflow, an attempt to access an illegal memory location, or a too-great use of CPU time). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

Accounting : It is responsible for tracking and recording the resource usage of various users and processes running on the system. It monitors how system resources such as CPU time, memory usage, disk I/O, and network bandwidth are being utilized, and provides detailed reports for system administrators or users.

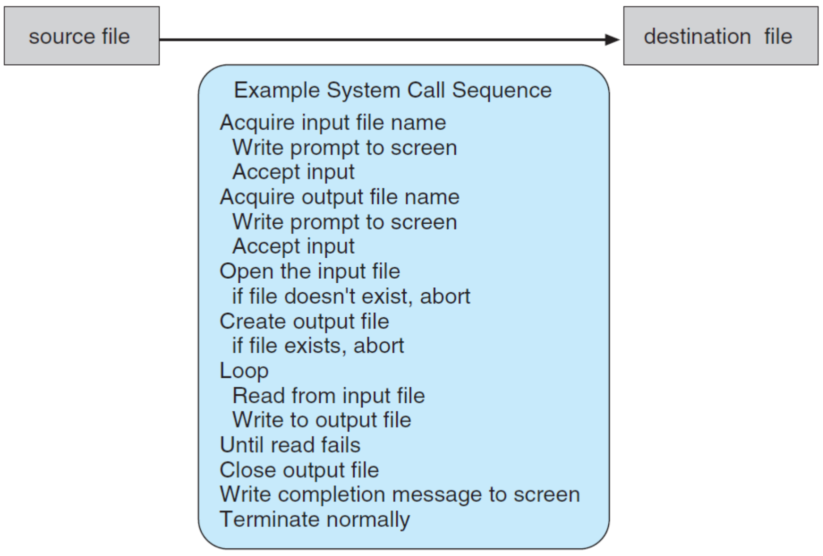
Another set of operating system functions exists not for helping the user but rather for ensuring the efficient operation of the system itself. Systems with multiple users can gain efficiency by sharing the computer resources among the users.

• **Resource allocation**. When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation code, whereas others (such as I/O devices) may have much more general request and release code.

• **Protection and security**. The owners of information stored in a multiuser or networked computer system may want to control use of that information. When several separate processes execute concurrently, it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security starts with requiring each user to authenticate. himself or herself to the system, usually by means of a password, to gain access to system resources.

2. System Calls

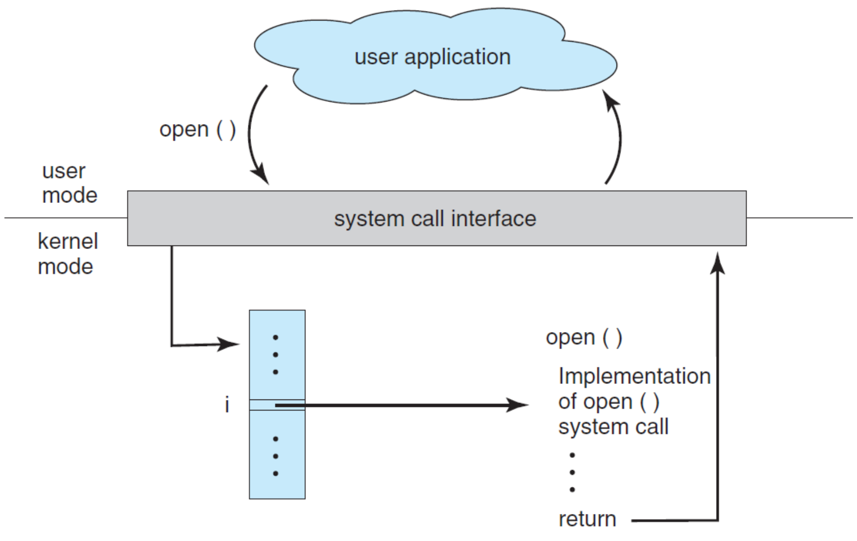
A system call is a mechanism that allows a user-level application to request services or resources from the operating system's kernel. It acts as an interface between user programs and the operating system, enabling programs to interact with hardware or perform tasks that require privileges not available in user mode. System calls are crucial for the functioning of an operating system, as they provide controlled access to the system's hardware and resources, ensuring that applications can perform necessary tasks while maintaining security and stability.



```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

Example of how system calls are used.

For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a **system call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call and any return values. The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library. The relationship between an API, the system-call interface, and the operating system is shown in the below Figure.



The handling of a user application invoking the `open()` system call.

3. Types of System calls [PDFPIC]

System calls can be grouped roughly into six major categories: process control, file manipulation, device manipulation, information maintenance, communications and protection.

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory

- File management
  - create file, delete file
  - open, close
  - read, write, reposition
  - get file attributes, set file attributes

- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get process, file, or device attributes
  - set process, file, or device attributes

- Communications
  - create, delete communication connection
  - send, receive messages
  - transfer status information
  - attach or detach remote devices

i) Process Control

A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()). If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a **debugger**—a system program designed to aid the programmer in finding and correcting errors, or **bugs**—to determine the cause of the problem.

Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command. In an ***interactive system***, the command interpreter simply continues with the next command; it is assumed that the user will issue an appropriate command to respond to any error. In a ***GUI system***, a pop-up window might alert the user to the error and ask for guidance. In a ***batch system,*** the command interpreter usually terminates the entire job and continues with the next job. ***Some systems*** may allow for special recovery actions in case an error occurs.

Quite often, two or more processes may share data. To ensure the integrity of the data being shared, operating systems often provide system calls allowing a process to lock shared data. Then, no other process can access the data until the lock is released. Typically, such system calls include acquire lock() and release lock().

EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix

Process Control	`CreateProcess()`
	`ExitProcess()`
	`WaitForSingleObject()`
		`fork()`
		`exit()`
		`wait()`

File Manipulation	`CreateFile()`
	`ReadFile()`
	`WriteFile()`
	`CloseHandle()`
		`open()`
		`read()`
		`write()`
		`close()`

Device Manipulation	`SetConsoleMode()`
	`ReadConsole()`
	`WriteConsole()`
		`ioctl()`
		`read()`
		write()

Information Maintenance	`GetCurrentProcessID()`
	`SetTimer()`
	`Sleep()`
		`getpid()`
		`alarm()`
		`sleep()`

Communication	`CreatePipe()`
	`CreateFileMapping()`
	`MapViewOfFile()`
		`pipe()`
		`shm_open()`
		`mmap()`

Protection	`SetFileSecurity()`
	`InitlializeSecurityDescriptor()`
	`SetSecurityDescriptorGroup()`
		`chmod()`
		`umask()`
		`chown()`

ii) File Management

We first need to be able to create() and delete() files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open() it and to use it. We may also read(), write(), or reposition() (rewind or skip to the end of the file, for example). Finally, we need to close() the file, indicating that we are no longer using it. We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, get file attributes() and set file attributes(), are required for this function. Some operating systems provide many more calls, such as calls for file move() and copy().

iii) Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.
The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files). A system with multiple users may require us to first request() a device, to ensure exclusive use of it. After we are finished with the device, we release() it. These functions are similar to the open() and close() system calls for files.

iv) Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time() and date(). Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.
Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump() memory. This provision is useful for debugging. The operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes() and set process attributes()).

v) Communication

There are two common models of inter-process communication: the message passing model and the shared-memory model. In the **message-passing model**, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox.In the **shared-memory model**, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes.

vi) Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Typically, system calls providing protection include set permission() and get permission(), which manipulate the permission settings of resources such as files and disks. The allow user() and deny user() system calls specify whether particular users can—or cannot—be allowed access to certain resources.

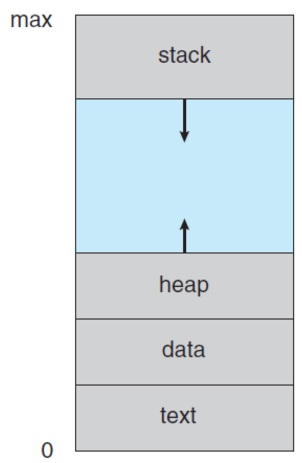
Process Management

Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a process, which is a program in execution. A process is the unit of work in a modern time-sharing system. A batch system executes **jobs**, whereas a time-shared system has **user programs**, or **tasks**. Even on a single-user system, a user may be able to run several programs at one time: a word processor, a Web browser, and an e-mail package.

1. Process concept

i) **The Process -** 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. The structure of a process in memory is shown in the below Figure.



Process in memory.

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. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog.exe or a.out). 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. It is also common to have a process that spawns many processes as it runs.

ii) Process State

As a process executes, it changes **state**. The state of a process is defined in part by the current activity of that process. A process may be in one of the following states:

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

The state diagram corresponding to these states is presented in the below figure.

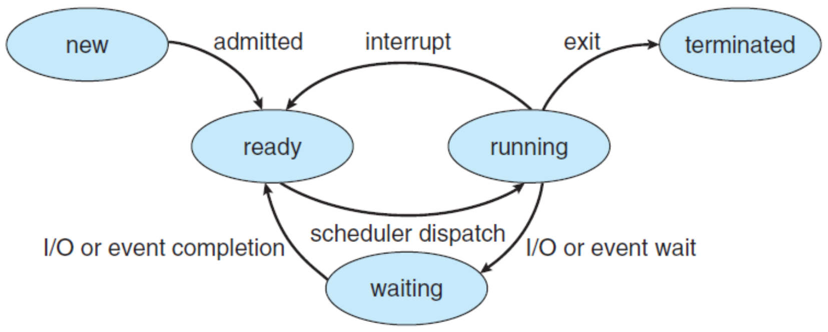
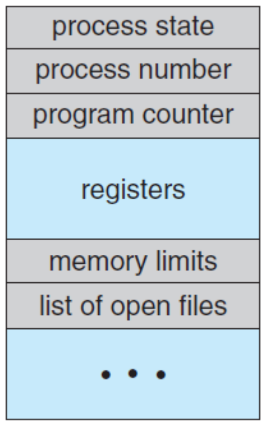


Diagram of process state.

iii) Process Control Block

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

- **Process state**. The state may be new, ready, running, waiting, halted, and so on.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward.
- **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.



Process control block (PCB).

- **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.

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

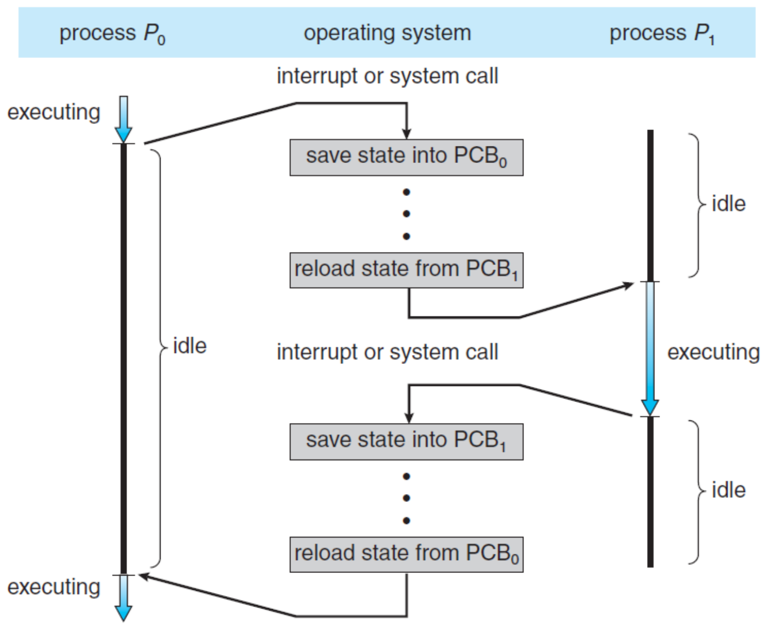


Diagram showing CPU switch from process to process.

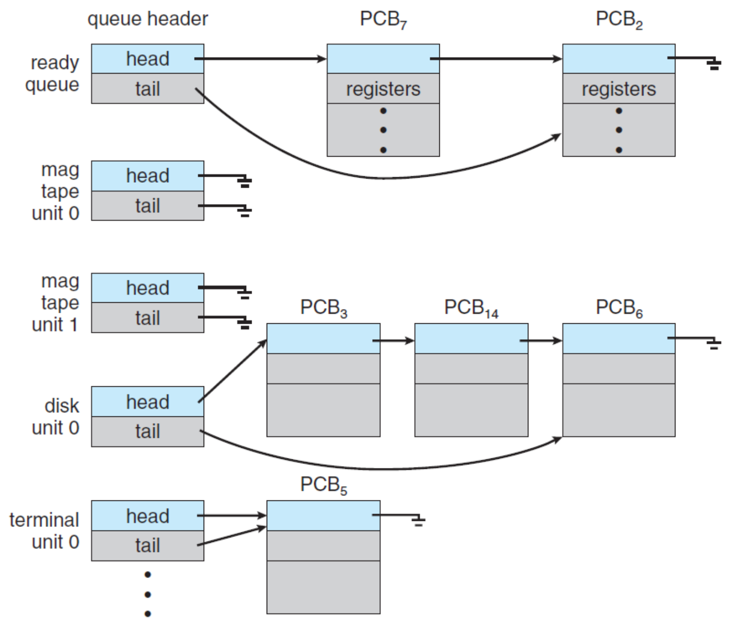
iv) Threads

When a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at a time.

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.

2. Process scheduling

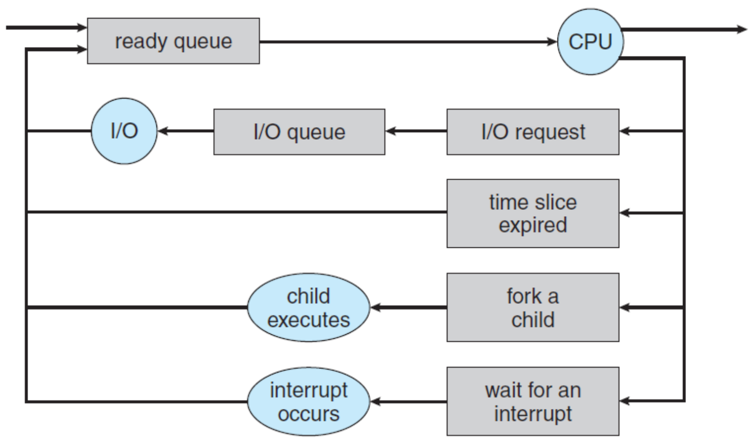
The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program.



The ready queue and various I/O device queues.

i) 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**. This queue is generally stored as a **linked list**. 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. The system also includes other queues. 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**. Each device has its own device queue.



Queueing-diagram representation of process scheduling.

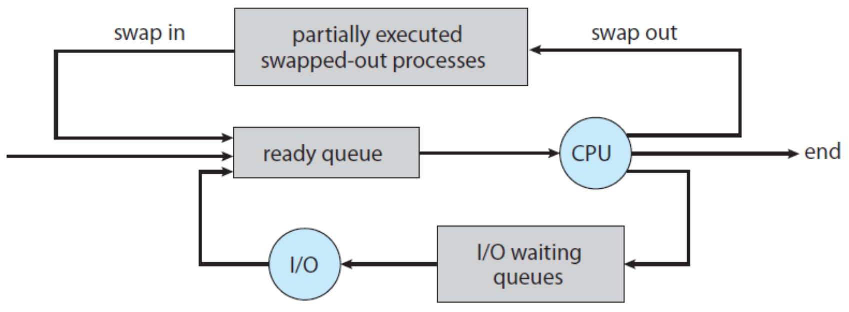
A common representation of process scheduling is a **queueing diagram**, such as that in Figure above. Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate 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:

- The process could issue an I/O request and then be placed in an I/O queue.
- The process could create a new child process and wait for the child's termination.
- 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.

ii) Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler**.
Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The **long-term scheduler**, or **job scheduler**, selects processes from this pool and loads them into memory for execution.

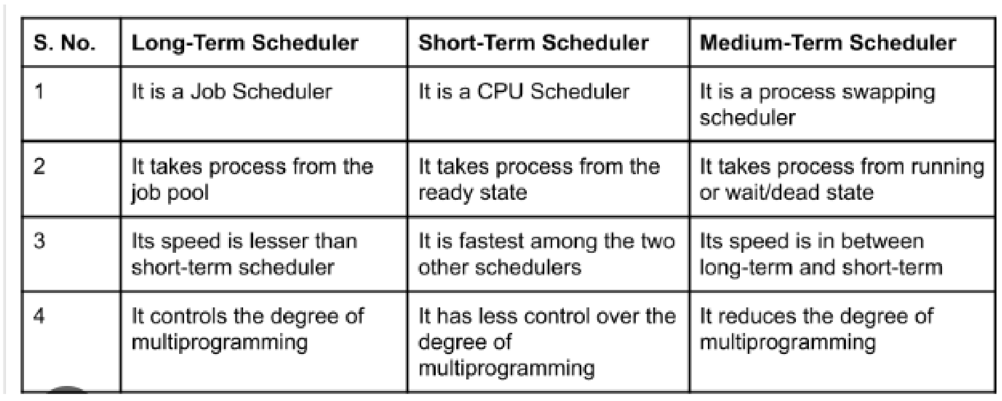


Addition of medium-term scheduling to the queueing diagram.

A **medium-term scheduler** is a type of process scheduler in operating systems that temporarily removes processes from main memory and places them in secondary storage (like a hard disk) to reduce the degree of multiprogramming. This process is called **swapping**. The medium-term scheduler may later bring these processes back into memory when sufficient resources are available, allowing them to resume execution.
The main function of the medium-term scheduler is to manage the processes in a way that balances the load on the CPU, manages memory usage efficiently, and ensures that high-priority processes

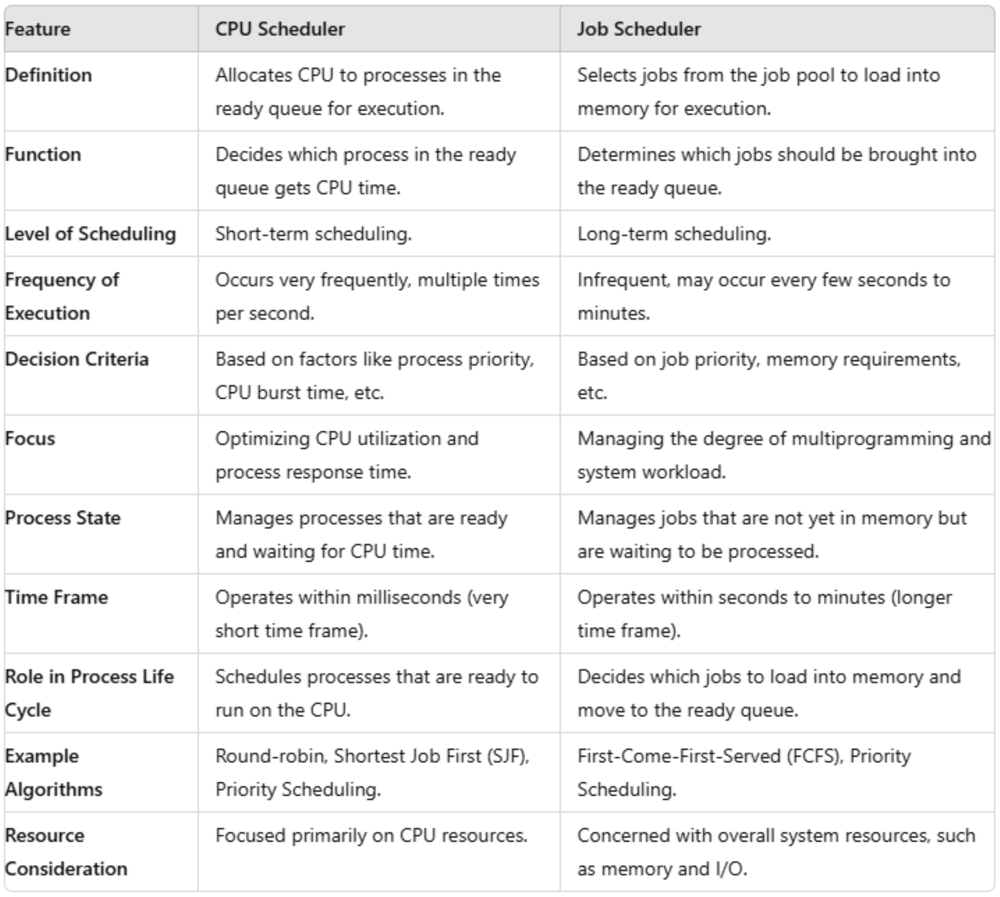
get more CPU time. It helps in optimizing the system's performance by controlling which processes are in memory and ready to execute.

The **degree of multiprogramming** refers to the number of processes that are loaded into memory and are ready to execute simultaneously in a computer system. It is an indicator of the system's capability to handle multiple tasks at once, maximizing CPU utilization by ensuring that there is always at least one process ready to execute while others might be waiting for I/O operations or other resources. A higher degree of multiprogramming typically improves system throughput but requires effective memory management to avoid excessive swapping or thrashing.



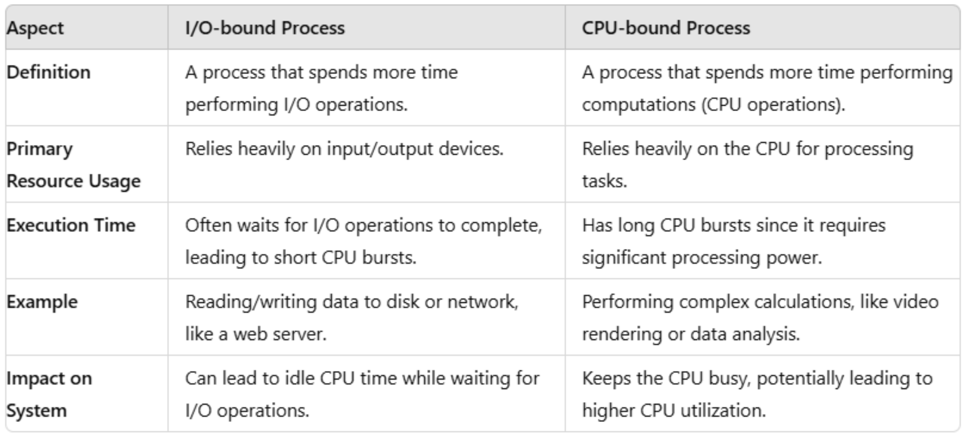
S. No.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a Job Scheduler	It is a CPU Scheduler	It is a process swapping scheduler
2	It takes process from the job pool	It takes process from the ready state	It takes process from running or wait/dead state
3	Its speed is lesser than short-term scheduler	It is fastest among the two other schedulers	Its speed is in between long-term and short-term
4	It controls the degree of multiprogramming	It has less control over the degree of multiprogramming	It reduces the degree of multiprogramming

Difference between CPU scheduler and job scheduler



Feature	CPU Scheduler	Job Scheduler
Definition	Allocates CPU to processes in the ready queue for execution.	Selects jobs from the job pool to load into memory for execution.
Function	Decides which process in the ready queue gets CPU time.	Determines which jobs should be brought into the ready queue.
Level of Scheduling	Short-term scheduling.	Long-term scheduling.
Frequency of Execution	Occurs very frequently, multiple times per second.	Infrequent, may occur every few seconds to minutes.
Decision Criteria	Based on factors like process priority, CPU burst time, etc.	Based on job priority, memory requirements, etc.
Focus	Optimizing CPU utilization and process response time.	Managing the degree of multiprogramming and system workload.
Process State	Manages processes that are ready and waiting for CPU time.	Manages jobs that are not yet in memory but are waiting to be processed.
Time Frame	Operates within milliseconds (very short time frame).	Operates within seconds to minutes (longer time frame).
Role in Process Life Cycle	Schedules processes that are ready to run on the CPU.	Decides which jobs to load into memory and move to the ready queue.
Example Algorithms	Round-robin, Shortest Job First (SJF), Priority Scheduling.	First-Come-First-Served (FCFS), Priority Scheduling.
Resource Consideration	Focused primarily on CPU resources.	Concerned with overall system resources, such as memory and I/O.

Difference between I/O-bound process and CPU-bound process



Aspect	I/O-bound Process	CPU-bound Process
Definition	A process that spends more time performing I/O operations.	A process that spends more time performing computations (CPU operations).
Primary Resource Usage	Relies heavily on input/output devices.	Relies heavily on the CPU for processing tasks.
Execution Time	Often waits for I/O operations to complete, leading to short CPU bursts.	Has long CPU bursts since it requires significant processing power.
Example	Reading/writing data to disk or network, like a web server.	Performing complex calculations, like video rendering or data analysis.
Impact on System	Can lead to idle CPU time while waiting for I/O operations.	Keeps the CPU busy, potentially leading to higher CPU utilization.

iii) Context Switch

A **context switch** is the process by which an operating system saves the state (context) of a currently running process or thread and restores the state of a different process or thread, allowing the CPU to switch from executing one process to another. The state includes information like the program counter, CPU registers, and memory management information.

Advantages of Context Switching:

1. **Efficient CPU Utilization:**
   - Allows multiple processes to share the CPU efficiently, maximizing the use of the CPU by reducing idle time.
2. **Multitasking:**
   - Enables the operating system to manage and execute multiple processes concurrently, giving the illusion that processes are running simultaneously.
3. **Responsiveness:**
   - Improves system responsiveness, especially in time-sharing systems, by allowing high-priority processes to get CPU time more quickly.
4. **Fairness:**
   - Helps in fairly allocating CPU time among processes, preventing any single process from monopolizing the CPU.

Demerits of Context Switching:

1. **Overhead:**
   - Context switching introduces overhead, as saving and loading the state of processes consumes CPU cycles, which could otherwise be used for executing instructions.
2. **Increased Latency:**
   - Frequent context switches can increase the latency of processes, particularly in real-time systems where timely task execution is crucial.

3. **Cache Misses:**
   - When the CPU switches contexts, the contents of the cache may not be relevant to the new process, leading to cache misses and reduced performance.
4. **Complexity:**
   - Managing context switches adds complexity to the operating system, particularly in maintaining the integrity and consistency of process states.

The processes in most systems can execute concurrently and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

3. Operations on processes

i) Process Creation

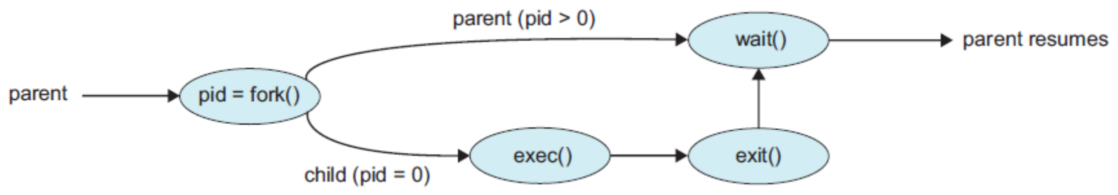
During the course of execution, a process may create several new processes. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.
Most operating systems (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.
In general, 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 prevent any process from overloading the system by creating too many child processes.
In addition to supplying various physical and logical resources, 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. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, image.jpg and the terminal device, and may simply transfer the datum between the two. 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.(India & foreign culture)

To illustrate these differences, let's first consider the UNIX operating system. In UNIX, each process is identified by its process identifier, which is a unique integer. A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. 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. When a process calls exec(), the operating system loads a new program into the process's memory, replacing the existing code, data, and stack of the calling process. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child. Because the call to exec() overlays the process's address space with a new program, the call to exec() does not return control unless an error occurs. The C program shown below illustrates the UNIX system calls.

```
#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;
}
```

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



Process creation using the fork() system call.

ii)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. Termination can occur in other circumstances as well. A process can cause the termination of another process via an appropriate system call (for example, TerminateProcess() in Windows). 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. Note that 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:

• The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)

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

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

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. To illustrate process execution and termination, consider that, in Linux and UNIX systems, we can terminate a process by using the exit() system call, providing an exit status as a parameter:

```
exit(1);          // exit with status 1
```

In fact, under normal termination, exit() may be called either directly (as shown above) or indirectly (by a return statement in main()). A parent process may wait for the termination of a child process by using the wait() system call. 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);
```

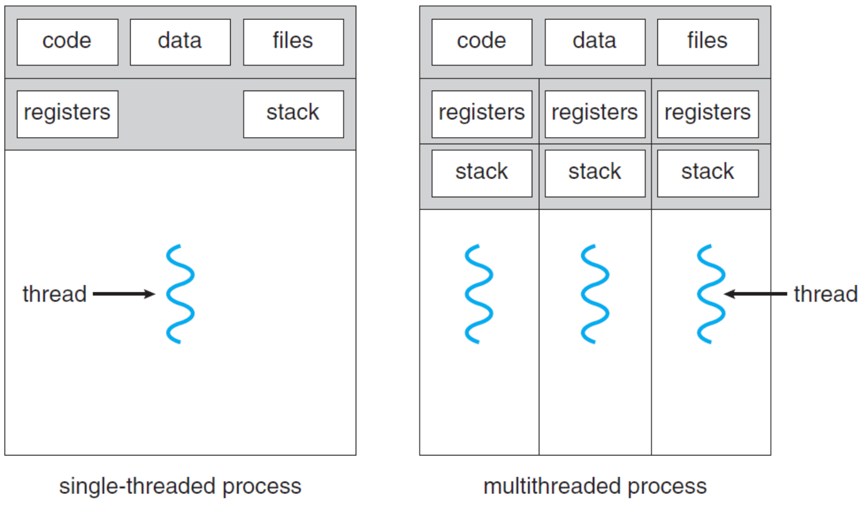
When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status. A process that has terminated, but whose parent has not yet called wait(), 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. Now consider what would happen if a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphans**. 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.

Unit 2

Multithreaded programming

1. Overview

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

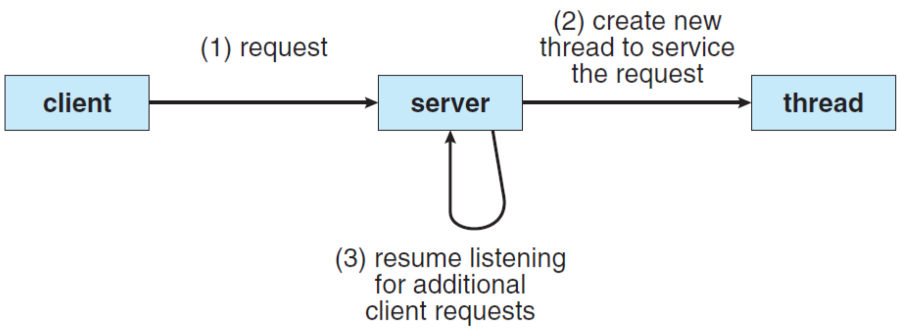


Single-threaded and multithreaded processes.

i) Motivation

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network, for example. 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. In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

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 resume listening for additional requests. This is illustrated in the below figure.


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Multithreaded server architecture.

ii) 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 completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.

2. **Resource sharing**. Processes can only share resources 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

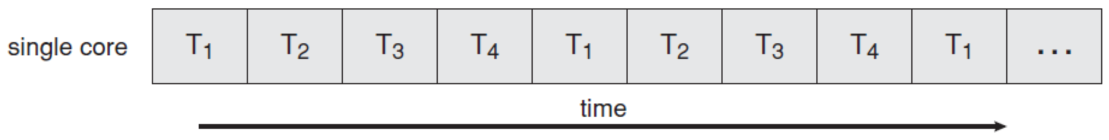
general it is significantly more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

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.

2. Multicore programming

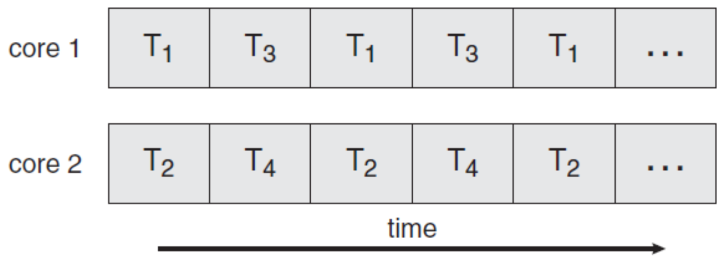
Multicore programming refers to the process of writing software that can run on systems with multiple cores in a single CPU. Each core in a multicore processor can execute instructions independently, allowing for parallel execution of tasks, which enhances performance and efficiency.

On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time (below figure).



Concurrent execution on a single-core system.

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 (below figure).



Parallel execution on a multicore system.

Key concepts of multicore programming:

1. **Concurrency and Parallelism**:
   - Concurrency: Executing multiple tasks in overlapping time frames.
   - Parallelism: Executing multiple tasks simultaneously on different cores.
2. **Multithreading**:
   - Using threads to break down tasks and run them concurrently on multiple cores.

   - Thread libraries like **pthreads** (POSIX threads) are commonly used for this purpose.
3. **Synchronization**:
   - Managing access to shared resources among multiple threads using tools like mutexes, locks, and semaphores to prevent race conditions and ensure consistency.
4. **Load Balancing**:
   - Distributing tasks evenly across cores to optimize performance and avoid overburdening any single core.
5. **Data Sharing and Memory Access**:
   - Ensuring efficient and safe access to shared data among threads to prevent data corruption or inconsistent states.
6. **Task Decomposition**:
   - Breaking down a program into smaller tasks or processes that can be executed in parallel.
7. **Thread Safety**:
   - Writing code that ensures proper functioning when accessed by multiple threads concurrently.
8. **Scalability**:
   - Designing programs that can take advantage of increasing numbers of cores as hardware evolves.
9. **Cache Coherence**:
   - Ensuring that all cores have consistent views of memory when accessing shared data, typically handled by the hardware.
10. **Performance Optimization**:

- Minimizing the overhead of thread creation, synchronization, and data sharing to achieve maximum speed-up.

i) Programming Challenges

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.

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 is inherently more difficult than testing and debugging single-threaded applications. Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future.

ii) Types of Parallelism

In general, there are two types of parallelism: data parallelism and task parallelism. **Data parallelism** focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. **Task parallelism** involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

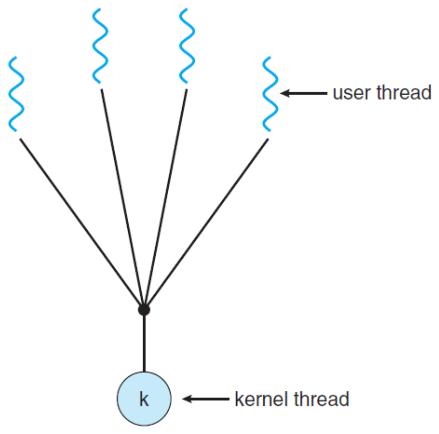
- **Task-based parallelism**: Dividing the program into independent tasks that can run concurrently on different cores.
- **Data parallelism**: Distributing data across multiple cores and performing the same operation on each partition of the data set in parallel.

3. 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.

(i) Many-to-One Model

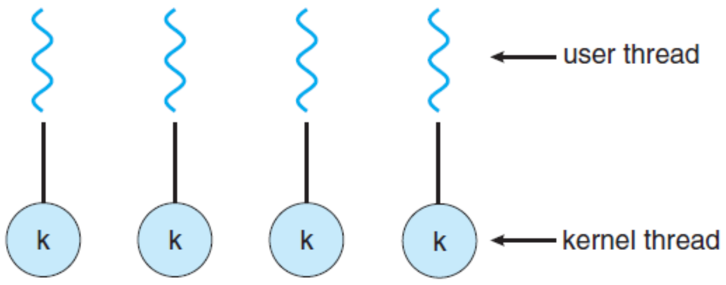
The many-to-one model (Below Figure) maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel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



Many-to-one model.

(ii) *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. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

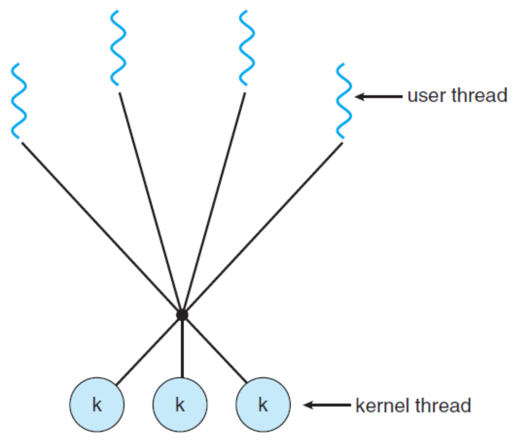


One-to-one model.

(iii) *Many-to-Many Model*

The many-to-many model, 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 multiprocessor than on a single processor).

Let's consider the effect of this design on concurrency. Whereas the many to- one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time. The one-to-one model allows greater concurrency, but the developer has to be careful not to create too many threads within an application (and in some instances may be limited in the number of threads she can create). The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many-to-many model.

4. Thread libraries – pthreads

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 theWindows API; UNIX and Linux systems often use Pthreads.

POSIX Pthreads:

Pthreads (POSIX threads) is a widely used multithreading library in UNIX-like operating systems such as Linux, macOS, and FreeBSD. It provides a standardized API that enables developers to write concurrent programs by creating and managing threads within a single process. The core functions include pthread_create() for spawning threads, pthread_join() for waiting for threads to complete, and pthread_exit() for terminating threads. Pthreads allows fine-grained control over thread behavior and attributes, including scheduling, detachability, and concurrency levels. Thread safety is managed through synchronization mechanisms such as mutexes, condition variables, and semaphores, which ensure that shared resources are accessed in a thread-safe manner.

Pthreads is highly portable across UNIX-based systems and provides low-level control, making it ideal for high-performance applications that require efficient multitasking. However, with this power comes complexity, as developers must explicitly manage synchronization and communication between threads. Deadlocks and race conditions can arise if shared resources are not handled carefully. Despite these challenges, Pthreads remains a go-to solution for developers building multithreaded applications in performance-critical environments, such as servers, real-time systems, and parallel processing applications.

Windows Threads:

Windows threads are a core part of the Windows operating system, allowing developers to create multithreaded applications that take advantage of modern multicore processors. The Windows API provides several functions for thread management, including CreateThread() to launch new threads and WaitForSingleObject() to wait for thread completion. Each thread is assigned its own stack and register set, and synchronization between threads is achieved using Windows-specific mechanisms such as critical sections, mutexes, semaphores, and events. Windows also supports advanced features like thread pools, which optimize thread usage by reusing threads for multiple tasks, improving system performance and resource management.

One of the key benefits of Windows threads is the ability to prioritize threads, allowing more critical tasks to execute with higher CPU time. However, like other threading libraries, developers must carefully manage synchronization to avoid issues like deadlocks or priority inversion. Windows also offers more sophisticated thread management through the use of thread affinity, which allows developers to bind threads to specific CPU cores, enabling better performance tuning for multi-core systems. Overall, Windows threads offer extensive flexibility and control for building responsive, high-performance applications on the Windows platform.

Java Threads:

Java threads are an integral part of the Java programming language, providing a platform-independent way to implement multithreading. Threads in Java can be created either by extending the Thread class or implementing the Runnable interface. This allows developers to run multiple threads concurrently, sharing system resources while maintaining separate execution paths. Java provides built-in synchronization mechanisms such as the synchronized keyword, which ensures that only one thread can access a critical section of code at a time. In addition to these low-level controls, Java also offers higher-level concurrency utilities in the java.util.concurrent package, such as Executors and ThreadPools, simplifying the management of thread execution.

The portability of Java threads across different platforms is one of their biggest strengths, making it easier to write cross-platform, multithreaded applications. Java abstracts away the complexities of native thread management and provides robust error-handling mechanisms. Java threads are often used in server-side applications, game development, and desktop software where concurrency is critical. However, developers must still handle issues like race conditions and deadlocks when multiple threads interact with shared resources. Java's thread management tools, combined with its automatic memory management (via the garbage collector), make it a versatile option for building scalable, concurrent applications.

CPU scheduling and Process Synchronization

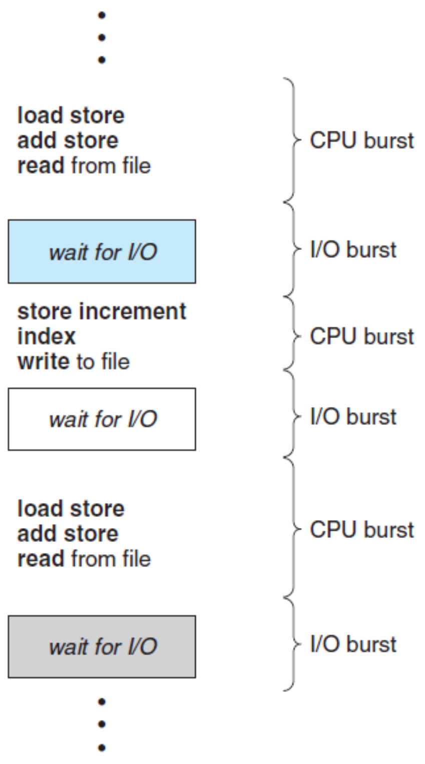
1. Basic concepts

In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. The objective of multiprogramming is to have some process running at all times, 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. All this waiting time is wasted; no useful work is accomplished. With multiprogramming, we try to use this time productively. Several processes are kept in memory at one time. When one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process. This pattern continues. Every time one process has to wait, another process can take over use of the CPU.

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.

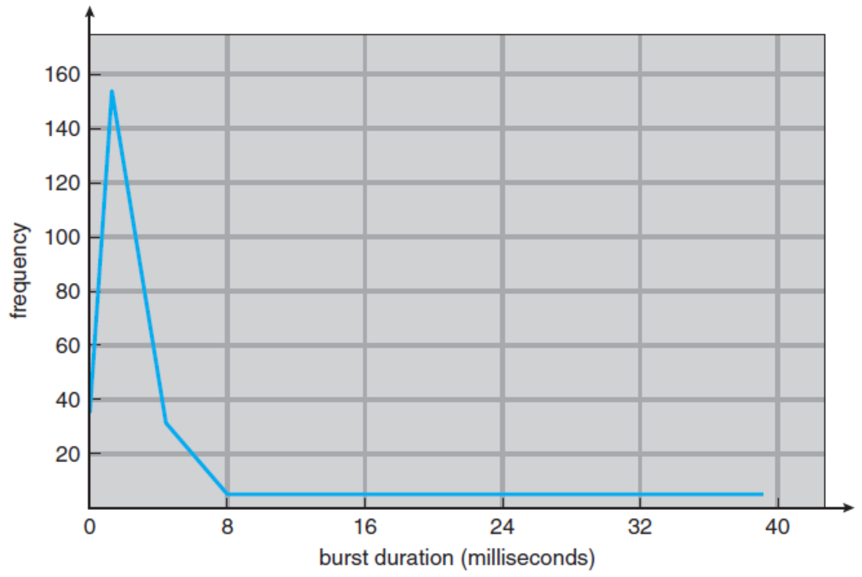
i) CPU–I/O Burst Cycle

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 as shown in below figure.



Alternating sequence of CPU and I/O bursts.

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 similar to that as shown in below figure. The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and a small number of long CPU bursts.



Histogram of CPU-burst durations.

ii) 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 short-term scheduler, or CPU scheduler. The scheduler 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. 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.

iii) Preemptive 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) 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 the CPU either by terminating or by switching to the waiting state. 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 maybe 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. Certain operating systems, including most versions of UNIX, deal with this problem by waiting either for a system call to complete or for an I/O block to take place before doing a context switch. This scheme ensures that the kernel structure is simple, since the kernel will not preempt a process while the kernel data structures are in an inconsistent state. Unfortunately, this kernel-execution model is a poor one for supporting real-time computing where tasks must complete execution within a given time frame.

Another component involved in the CPU-scheduling function is the **dispatcher.** The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

• Switching context

• Switching to user mode

• Jumping to the proper location in the user program to restart that program

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

2. Scheduling criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

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).

• ***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 per hour; for short transactions, it may be ten 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 to get into memory, 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 fairly 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. The turnaround time is generally limited by the speed of the output device.

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. However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example, to guarantee that all users get good service, we may want to minimize the maximum response time.

Benchmark	Turnaround Time	Waiting Time
Definition	The time it takes for a process to complete from the moment it is submitted to the moment it is finished.	The total time a process spends waiting in a ready queue before reaching the CPU.
Calculation	Turnaround Time = Completion Time – Arrival Time/ Burst Time + Waiting Time	Waiting Time = Turnaround Time – Burst Time
Importance	Helps in understanding the efficiency of the CPU in processing tasks.	Can cause delays in the execution of other processes, leading to lower system performance.
Optimization	By optimizing the CPU scheduling algorithm, turnaround time can be reduced.	By minimizing the waiting time, more processes can be executed in less time, improving system efficiency.
Units	Usually measured in milliseconds, seconds or minutes.	Usually measured in the same unit as the burst time, such as milliseconds, seconds or minutes.
It is limited by the speed of the output device?	Yes.	No.

3. Scheduling algorithms

Scheduling algorithms are used by the operating system to determine which process or thread should run on the CPU at a given time. These algorithms are critical for managing system performance, responsiveness, and fairness. There are several types of scheduling algorithms, each designed for different system requirements, such as maximizing CPU utilization, reducing response time, or ensuring fairness between processes.

1. First-Come, First-Served (FCFS):

- **Description**: In FCFS, the process that arrives first is executed first. Processes are handled in the order they appear in the ready queue.
- **Advantages**: Simple to implement and understand.
- **Disadvantages**: It can lead to **convoy effect**, where shorter tasks have to wait for long tasks to finish, increasing the average waiting time.

2. Shortest Job First (SJF):

- **Description**: In SJF, the process with the shortest estimated running time is executed first.
- **Advantages**: Minimizes the average waiting time compared to FCFS.
- **Disadvantages**: It can cause **starvation** of longer processes if shorter processes keep arriving.

3. Round Robin (RR):

- **Description**: Each process is assigned a fixed **time quantum**. After each quantum, the process is moved to the end of the ready queue if it has not finished.
- **Advantages**: Fair to all processes, especially in time-sharing systems. Every process gets a chance to run within a fixed amount of time.

- **Disadvantages**: Performance depends on the choice of the time quantum. Too small quantum leads to too many context switches, while too large a quantum degenerates to FCFS.

4. Priority Scheduling:

- **Description**: Each process is assigned a priority, and the CPU is allocated to the process with the highest priority.
- **Advantages**: Important tasks are given preference, ensuring critical processes are executed first.
- **Disadvantages**: Can lead to **starvation** of low-priority processes, but can be resolved using techniques like **aging** (gradually increasing the priority of waiting processes).

5. Multilevel Queue Scheduling:

- **Description**: The ready queue is divided into multiple queues, each with its own scheduling algorithm (e.g., system processes in one queue, user processes in another). Processes are permanently assigned to one queue based on their type or priority.
- **Advantages**: Provides a good balance between different types of processes, like interactive and batch processes.
- **Disadvantages**: Once assigned to a queue, processes cannot move between queues, which may lead to inefficiencies.

6. Multilevel Feedback Queue:

- **Description**: Similar to multilevel queue scheduling, but processes can move between queues based on their behavior (e.g., a process that uses too much CPU may be moved to a lower-priority queue).
- **Advantages**: Flexible and allows for better control over process behavior. Avoids starvation by dynamically adjusting priorities.
- **Disadvantages**: Complex to implement and fine-tune.

7. Shortest Remaining Time First (SRTF):

- **Description**: A preemptive version of SJF, where the process with the shortest remaining execution time is selected for CPU.
- **Advantages**: Optimal for minimizing the average waiting time.
- **Disadvantages**: Similar to SJF, longer processes can experience starvation.

8. Real-Time Scheduling:

- **Description**: Used in real-time systems, where tasks must be completed within a strict time deadline. Algorithms include **Rate Monotonic Scheduling (RMS)** and **Earliest Deadline First (EDF)**.
- **Advantages**: Ensures that time-critical tasks are completed on time.

- **Disadvantages**: Complex to implement, and not all tasks can always be guaranteed to meet deadlines.

Each scheduling algorithm is suited for different types of systems. **FCFS** is simple but inefficient in time-sharing systems. **SJF** and **SRTF** are good for minimizing wait times but can cause starvation. **Round Robin** ensures fairness in time-sharing environments. **Priority Scheduling** and **Multilevel Feedback Queue** provide more nuanced control, balancing different types of tasks efficiently.

Real-Time CPU Scheduling

Real-Time CPU Scheduling is used in systems where tasks or processes need to be executed within strict time constraints, commonly seen in **real-time operating systems (RTOS)**. These systems are typically found in environments where timely and deterministic task execution is critical, such as embedded systems, medical devices, robotics, telecommunications, and industrial control systems.

Real-time systems are classified into two types:

- **Hard Real-Time Systems**: Missing a deadline results in a critical failure (e.g., pacemaker systems or automotive airbag systems).
- **Soft Real-Time Systems**: Missing a deadline results in degraded performance but not critical failure (e.g., video streaming or gaming systems).

Real-time tasks or processes have specific deadlines that must be met. These tasks are associated with **periodic** or **aperiodic** characteristics.

- **Periodic tasks**: Tasks that repeat at regular intervals (e.g., sensor readings).
- **Aperiodic tasks**: Tasks that occur unpredictably (e.g., emergency signals).

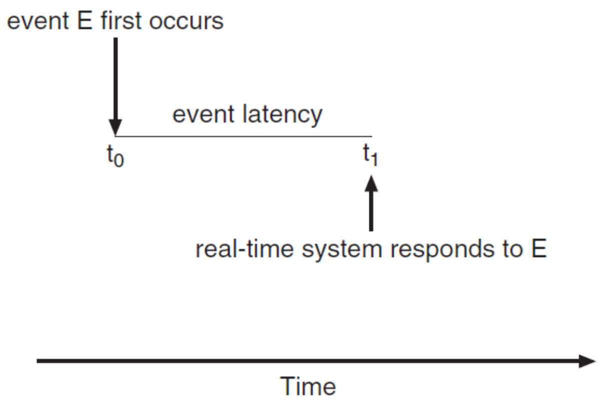
Key Metrics for Real-Time Scheduling:

- **Deadline**: The time by which a task must be completed.
- **Response Time**: The time it takes for a system to respond to an event or request.
- **Jitter**: Variation in the time it takes to complete a task, which real-time systems aim to minimize.

Several issues related to process scheduling in both soft and hard real-time operating systems are as below.

❖ ***Minimizing Latency***

Consider the event-driven nature of a real-time system. The system is typically waiting for an event in real time to occur. Events may arise either in software —as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. **Event latency** is the amount of time that elapses from when an event occurs to when it is serviced (Figure below). Usually, different events have different latency requirements.



Event latency.

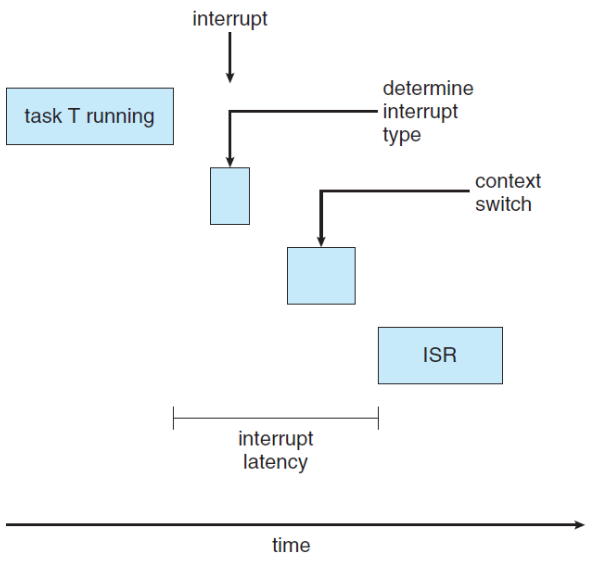
Two types of latencies affect the performance of real-time systems:

1. Interrupt latency

2. Dispatch latency

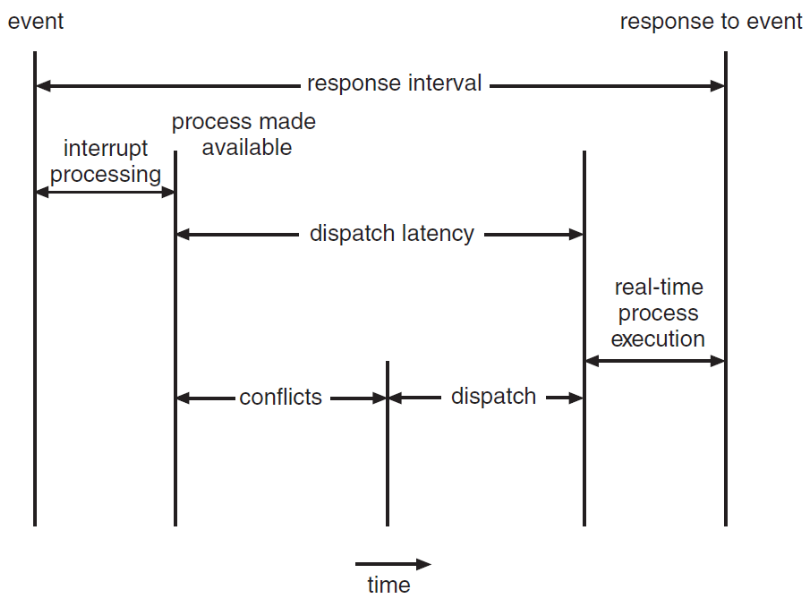
Interrupt latency refers to the period of time fromthe arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the interrupt latency (Figure below).

Obviously, it is crucial for real time operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention. Indeed, for hard real-time systems, interrupt latency must not simply be minimized, it must be bounded to meet the strict requirements of these systems.



Interrupt latency.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts be disabled for only very short periods of time.



Dispatch latency.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as dispatch latency. Providing real-time tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency as well. The most effective technique for keeping dispatch latency low is to provide preemptive kernels.

The **conflict phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel.

2. Release by low-priority processes of resources needed by a high-priority Process.

❖ *Priority-Based Scheduling*

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Recall that priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run. Providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees requires additional scheduling features.

❖ *Rate-Monotonic Scheduling*

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process.Uponentering the system, each periodic task is assigned a priority inversely based on its period. The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

❖ *Earliest-Deadline-First Scheduling*

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

❖ *Proportional Share Scheduling*

Proportional share schedulers operate by allocating *T* shares among all applications. An application can receive *N* shares of time, thus ensuring that the application will have *N/T* of the total processor time. As an example, assume that a total of *T* = 100 shares is to be divided among three processes, *A*, *B*, and *C*. *A*is assigned 50 shares, *B* is assigned 15 shares, and *C* is assigned 20 shares. This scheme ensures that *A* will have 50 percent of total processor time, *B* will have 15 percent, and *C* will have 20 percent.

❖ *POSIX Real-Time Scheduling*

The POSIX standard also provides extensions for real-time computing— POSIX.1b. POSIX defines two scheduling classes for real-time threads:

• SCHED FIFO

• SCHED RR

SCHED FIFO schedules threads according to a first-come, first-served policy using a FIFO queue. However, there is no time slicing amongthreads of equal priority.Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED RR uses a round-robin policy. It is similar to SCHED FIFO except that it provides time slicing among threads of equal priority. POSIX provides an additional scheduling class—SCHED OTHER—but its implementation is undefined and system specific; it may behave differently on different systems.

The POSIX API specifies the following two functions for getting and setting the scheduling policy:

• pthread attr getsched policy(pthread attr t *attr, int *policy)

• pthread attr setsched policy(pthread attr t *attr, int policy)

The first parameter to both functions is a pointer to the set of attributes for the thread. The second parameter is either (1) a pointer to an integer that is set to the current scheduling policy (for pthread attr getsched policy()) or (2) an integer value (SCHED FIFO, SCHED RR, or SCHED OTHER) for the pthread attr setsched policy() function. Both functions return nonzero values if an error occurs.