I M.C.A.	OPERATING SYSTEMS	L	T	P	C
I Semester		3	0	0	3

UNIT–I: Introduction to OS, Process Management

Introduction to Operating System Concept: Types of operating systems, operating systems concepts, operating systems services, Introduction to System call, System call types.

Process Management: Process concept, The process, Process State Diagram, Process control block, Process Scheduling- Scheduling Queues, Schedulers, Operations on Processes, Interprocess Communication, Threading Issues, Scheduling-Basic Concepts, Scheduling Criteria, Scheduling Algorithms.

Operating System - Overview

An **Operating System** (OS) is an interface between a computer user and computer hardware. An operating system is a software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

An operating system is software that enables applications to interact with a computer's hardware. The software that contains the core components of the operating system is called the **kernel**.

The primary purposes of an **Operating System** are to enable applications (softwares) to interact with a computer's hardware and to manage a system's hardware and software resources.

Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc. Today, Operating systems is found almost in every device like mobile phones, personal computers, mainframe computers, automobiles, TV, Toys etc.

What is an Operating System?

- An operating system is a program (or set of programs) that manages the computer hardware
- It also provides a basis for running application programs and acts as an intermediary between the computer user and the computer hardware
- Some operating systems are designed to be convenient, others are designed to be efficient, and still others are a combination of both
  - Mainframe operating systems are designed primarily to optimize utilization of hardware
  - PC operating systems support a range of software from complex games to business applications
  - Operating systems for handheld computers are designed to provide a portable environment in which a user can easily interface with the computer
- Because an operating system is large and complex, it must be created piece by piece
  - Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions
  - This chapter provides a general overview of the major components of an operating system

Definitions

We can have a number of definitions of an Operating System. Let's go through few of them:

An Operting System is the low-level software that supports a computer's basic functions, such as scheduling tasks and controlling peripherals.

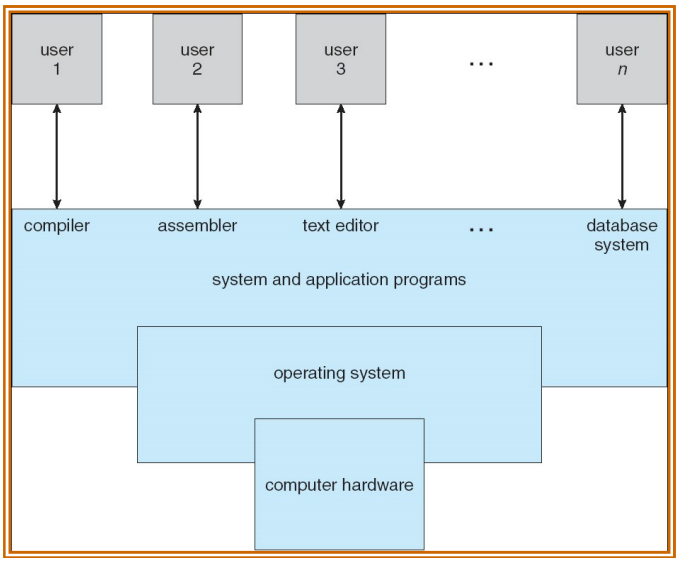
We can refine this definition as follows:

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.

An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs.

Architecture

We can draw a generic architecture diagram of an Operating System which is as follows:



Four Components of a Computer System

Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Network Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management −

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management −

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management −

- Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management −

- Keeps track of information, location, uses, status etc. The collective facilities are often known as **file system**.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Other Important Activities

Following are some of the important activities that an Operating System performs −

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

Operating System Types Year Wise Evolution

Operating Systems evolved over time from almost no OS to AI Powered one. Following list shows the evolution of Operating Systems over time with technological advancements.

- **Batch Processing Systems** − These systems were popular From 1940s to 1950s. The users of a batch operating system did not interact with the computer directly. Each user prepared his job on an off-line device like punch cards and submited it to the computer operator who then batched the similar jobs together to speed up processing and run as a group. The programmers left their programs with the operator and the operator then sorted the programs with similar requirements into batches. In such systems, CPU usage was very low and it was difficult to prioritize jobs over one another.
- **Multiprogramming Systems** − These operating systems emerged from 1950s to 1960s and revolutionalized the computer arena. Now a user could load multile programs into memory and each program could have specific memory allocated. While one program was waiting for I/O operation, CPU was alloted to second program.
- **Time-Sharing Systems** − Such Operating system can be categorized from 1960s to 1970s yearwise. Time-sharing or multitasking is a logical extension of multiprogramming. Processor's time which is shared among multiple users simultaneously is termed as time-sharing. The operating system used CPU scheduling and multiprogramming to provide each user with a small portion of a time. Computer systems that were designed primarily as batch systems have been modified to time-sharing systems.
- **GUI Based Systems** − From 1970s to 1980s, GUI based Operating Systems became popular. These operating systems were more user friendly. In stead of typing commands, a user could click on graphical icons. Microsoft Windows is one of earlier popular GUI based operating system which still dominates the personal computer space.
- **Networked Systems** − As time advances, so as technologies. From 1980s to 1990s, network based system gained momentum. A Network Operating System runs on a server and provides the server the capability to manage data, users, groups, security, applications, and other networking functions. The primary purpose of the network operating system is to allow shared file and printer access among multiple computers in a network, typically a local area network (LAN), a private network or to other networks.
- **Mobile Operating Systems** − From Late 1990s to Early 2000s, Symbian, Java ME based OS were popular for mobile devices. Over the period of time, with the introduction of smart phones, need of more complex operation systems arised. That leads to development of Android

and iOS mobile operating system which are getting more and more powerful and becoming feature rich till date.

- **AI Powered** − From 2010s to Present
  In today's time, Artificial Intelligence is dominating every aspects of computers including Operating Systems. Siri, Google Assistant, Alexa and many other AI based assitant softwares which can even understand the voice commands and can perform any operation that a user needs to perform.

Components of Operating System

There are various components of an Operating System to perform well defined tasks. Though most of the Operating Systems differ in structure but logically they have similar components. Each component must be a well-defined portion of a system that appropriately describes the functions, inputs, and outputs.

There are following 8-components of an Operating System:

1. Process Management
2. I/O Device Management
3. File Management
4. Network Management
5. Main Memory Management
6. Secondary Storage Management
7. Security Management
8. Command Interpreter System

Following section explains all the above components in more detail:

Process Management

A process is program or a fraction of a program that is loaded in main memory. A process needs certain resources including CPU time, Memory, Files, and I/O devices to accomplish its task. The process management component manages the multiple processes running simultaneously on the Operating System.

A program in running state is called a process.

The operating system is responsible for the following activities in connection with process management:

- Create, load, execute, suspend, resume, and terminate processes.
- Switch system among multiple processes in main memory.
- Provides communication mechanisms so that processes can communicate with each others
- Provides synchronization mechanisms to control concurrent access to shared data to keep shared data consistent.
- Allocate/de-allocate resources properly to prevent or avoid deadlock situation.

I/O Device Management

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. I/O Device Management provides an abstract level of H/W devices and keep the details from applications to ensure proper use of devices, to prevent errors, and to provide users with convenient and efficient programming environment.

Following are the tasks of I/O Device Management component:

- Hide the details of H/W devices
- Manage main memory for the devices using cache, buffer, and spooling
- Maintain and provide custom drivers for each device.

File Management

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms; magnetic tape, disk, and drum are the most common forms.

A file is defined as a set of correlated information and it is defined by the creator of the file. Mostly files represent data, source and object forms, and programs. Data files can be of any type like alphabetic, numeric, and alphanumeric.

A files is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user.

The operating system implements the abstract concept of the file by managing mass storage device, such as types and disks. Also files are normally organized into directories to ease their use. These directories may contain files and other directories and so on.
The operating system is responsible for the following activities in connection with file management:

- File creation and deletion
- Directory creation and deletion
- The support of primitives for manipulating files and directories
- Mapping files onto secondary storage
- File backup on stable (nonvolatile) storage media

Network Management
The definition of network management is often broad, as network management involves several different components. Network management is the process of managing and administering a computer network. A computer network is a collection of various types of computers connected with each other. Network management comprises fault analysis, maintaining the quality of service, provisioning of networks, and performance management.
Network management is the process of keeping your network healthy for an efficient communication between different computers.
Following are the features of network management:

- Network administration
- Network maintenance
- Network operation
- Network provisioning
- Network security

Main Memory Management
Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.
Main memory is a volatile storage device which means it loses its contents in the case of system failure or as soon as system power goes down.
The main motivation behind Memory Management is to maximize memory utilization on the computer system.
The operating system is responsible for the following activities in connections with memory management:

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes to load when memory space becomes available.
- Allocate and deallocate memory space as needed.

Secondary Storage Management
The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory.
Most modern computer systems use disks as the principle on-line storage medium, for both programs and data. Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing.
The operating system is responsible for the following activities in connection with disk management:

- Free space management
- Storage allocation
- Disk scheduling

Security Management
The operating system is primarily responsible for all task and activities happen in the computer system. The various processes in an operating system must be protected from each other's activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, cpu and other resources can be operated on only by those processes that have gained proper authorization from the operating system.

Security Management refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement.

For example, memory addressing hardware ensure that a process can only execute within its own address space. The timer ensure that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do it's own I/O, to protect the integrity of the various peripheral devices.

Command Interpreter System

One of the most important component of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system.

Command Interpreter System executes a user command by calling one or more number of underlying system programs or system calls.

Command Interpreter System allows human users to interact with the Operating System and provides convenient programming environment to the users.

Many commands are given to the operating system by control statements. A program which reads and interprets control statements is automatically executed. This program is called the shell and few examples are Windows DOS command window, Bash of Unix/Linux or C-Shell of Unix/Linux.

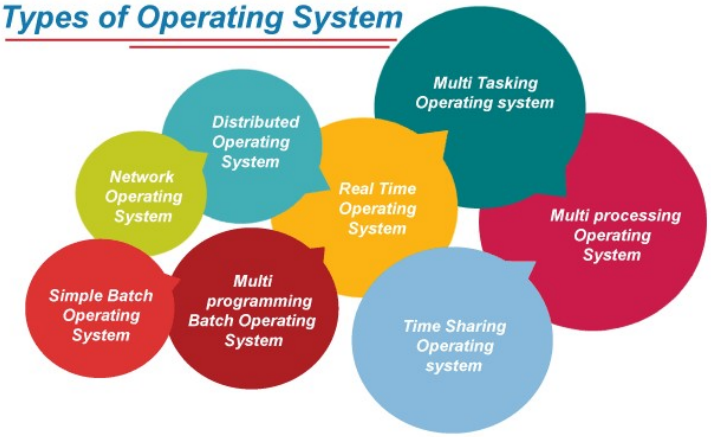
Other Important Activities

An Operating System is a complex Software System. Apart from the above mentioned components and responsibilities, there are many other activities performed by the Operating System. Few of them are listed below:

- **Security** − By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance** − Recording delays between request for a service and response from the system.
- **Job accounting** − Keeping track of time and resources used by various jobs and users.
- **Error detecting aids** − Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other softwares and users** − Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

Types of Operating Systems

An operating system is a well-organized collection of programs that manages the computer hardware. It is a type of system software that is responsible for the smooth functioning of the computer system.

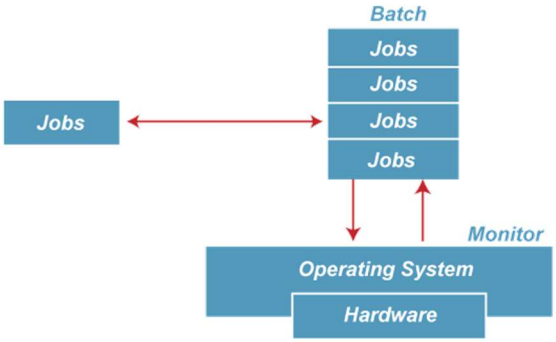


Batch Operating System

In the 1970s, Batch processing was very popular. In this technique, similar types of jobs were batched together and executed in time. People were used to having a single computer which was called a mainframe.

In Batch operating system, access is given to more than one person; they submit their respective jobs to the system for the execution.

The system put all of the jobs in a queue on the basis of first come first serve and then executes the jobs one by one. The users collect their respective output when all the jobs get executed.



The purpose of this operating system was mainly to transfer control from one job to another as soon as the job was completed. It contained a small set of programs called the resident monitor that always resided in one part of the main memory. The remaining part is used for servicing jobs.

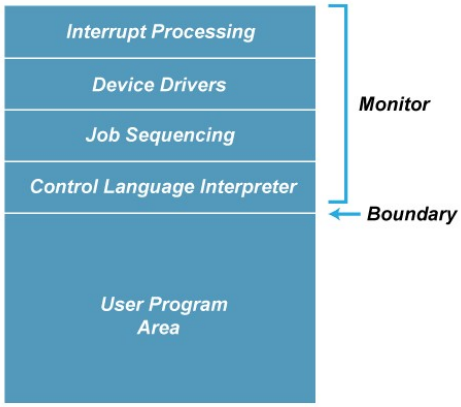


Figure: Memory Layout of the resident monitor

Advantages of Batch OS

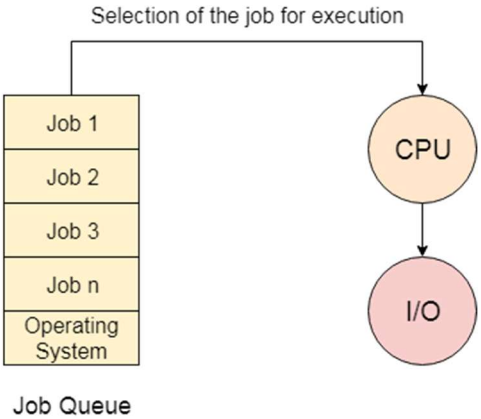
- The use of a resident monitor improves computer efficiency as it eliminates CPU time between two jobs.

Disadvantages of Batch OS

1. Starvation

Batch processing suffers from starvation.

For Example:



There are five jobs J1, J2, J3, J4, and J5, present in the batch. If the execution time of J1 is very high, then the other four jobs will never be executed, or they will have to wait for a very long time. Hence the other processes get starved.

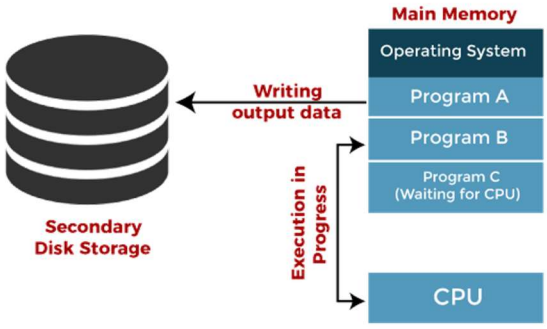
2. Not Interactive

Batch Processing is not suitable for jobs that are dependent on the user's input. If a job requires the input of two numbers from the console, then it will never get it in the batch processing scenario since the user is not present at the time of execution.

Multiprogramming Operating System

Multiprogramming is an extension to batch processing where the CPU is always kept busy. Each process needs two types of system time: CPU time and IO time.

In a multiprogramming environment, when a process does its I/O, The CPU can start the execution of other processes. Therefore, multiprogramming improves the efficiency of the system.



Jobs in multiprogramming system

Advantages of Multiprogramming OS

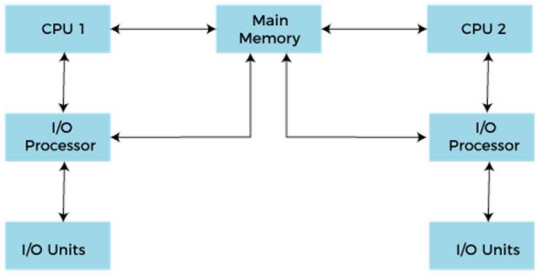
- Throughout the system, it increased as the CPU always had one program to execute.
- Response time can also be reduced.

Disadvantages of Multiprogramming OS

- Multiprogramming systems provide an environment in which various systems resources are used efficiently, but they do not provide any user interaction with the computer system.

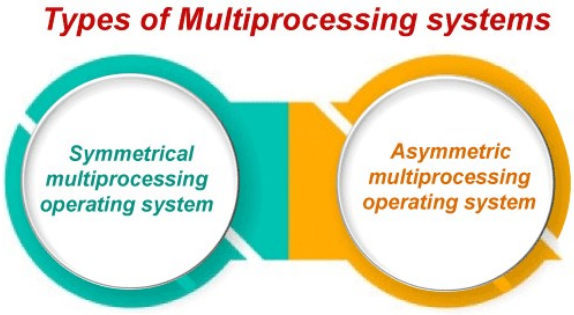
Multiprocessing Operating System

In Multiprocessing, Parallel computing is achieved. There are more than one processors present in the system which can execute more than one process at the same time. This will increase the throughput of the system.



Working of Multiprocessor System

In Multiprocessing, Parallel computing is achieved. More than one processor present in the system can execute more than one process simultaneously, which will increase the throughput of the system.

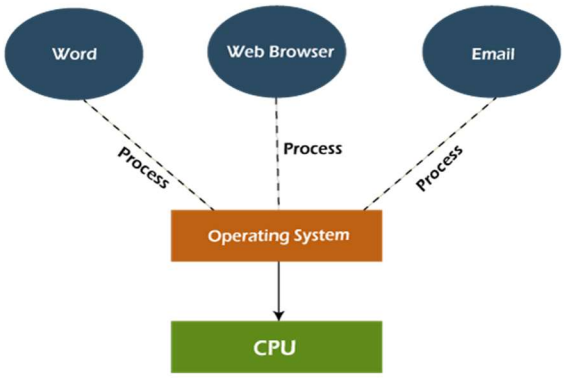


Advantages of Multiprocessing operating system:

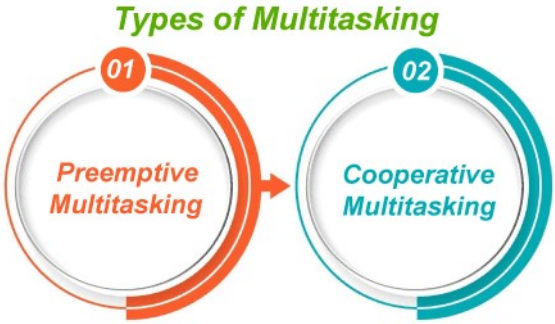
- **Increased reliability:** Due to the multiprocessing system, processing tasks can be distributed among several processors. This increases reliability as if one processor fails, the task can be given to another processor for completion.
- **Increased throughout:** As several processors increase, more work can be done in less.

Disadvantages of Multiprocessing operating System

- Multiprocessing operating system is more complex and sophisticated as it takes care of multiple CPUs simultaneously.



The multitasking operating system is a logical extension of a multiprogramming system that enables **multiple** programs simultaneously. It allows a user to perform more than one computer task at the same time.



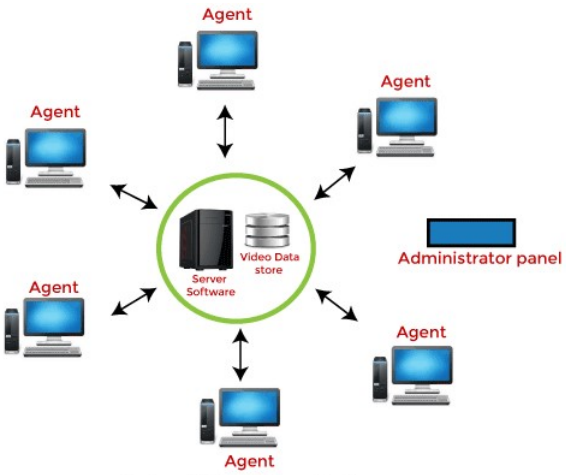
Advantages of Multitasking operating system

- This operating system is more suited to supporting multiple users simultaneously.
- The multitasking operating systems have well-defined memory management.

Disadvantages of Multitasking operating system

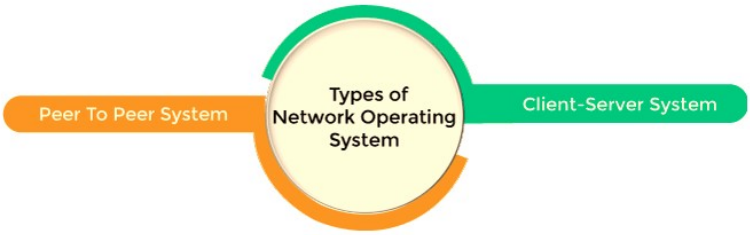
- The multiple processors are busier at the same time to complete any task in a multitasking environment, so the CPU generates more heat.

Network Operating System



Network Operating Systems

An Operating system, which includes software and associated protocols to communicate with other computers via a network conveniently and cost-effectively, is called Network Operating System.



Advantages of Network Operating System

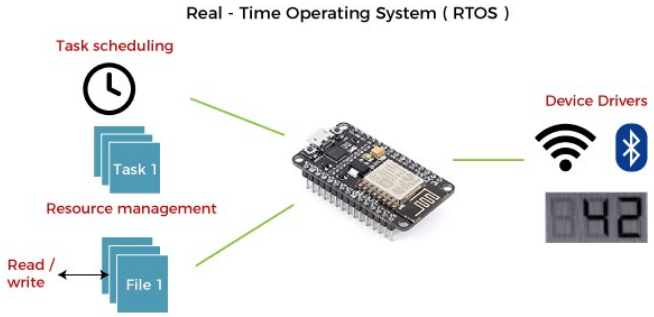
- In this type of operating system, network traffic reduces due to the division between clients and the server.
- This type of system is less expensive to set up and maintain.

Disadvantages of Network Operating System

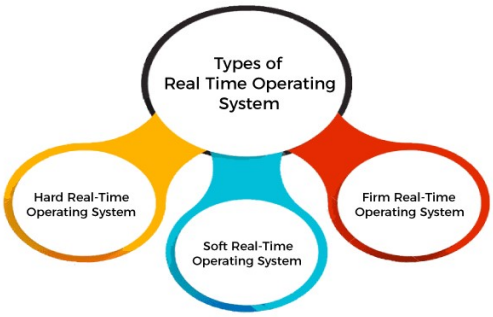
- In this type of operating system, the failure of any node in a system affects the whole system.
- Security and performance are important issues. So trained network administrators are required for network administration.

Real Time Operating System

In Real-Time Systems, each job carries a certain deadline within which the job is supposed to be completed, otherwise, the huge loss will be there, or even if the result is produced, it will be completely useless.



The Application of a Real-Time system exists in the case of military applications, if you want to drop a missile, then the missile is supposed to be dropped with a certain precision.



Advantages of Real-time operating system:

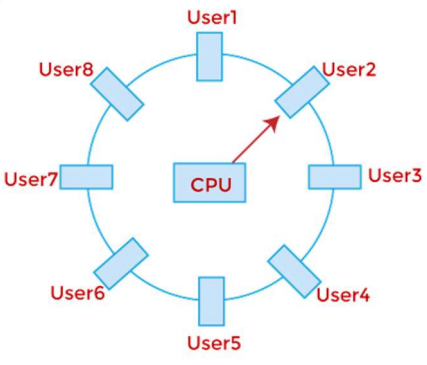
- Easy to layout, develop and execute real-time applications under the real-time operating system.
- In a Real-time operating system, the maximum utilization of devices and systems.

Disadvantages of Real-time operating system:

- Real-time operating systems are very costly to develop.
- Real-time operating systems are very complex and can consume critical CPU cycles.

Time-Sharing Operating System

In the Time Sharing operating system, computer resources are allocated in a time-dependent fashion to several programs simultaneously. Thus it helps to provide a large number of user's direct access to the main computer. It is a logical extension of multiprogramming. In time-sharing, the CPU is switched among multiple programs given by different users on a scheduled basis.



Timesharing in case of 8 users

A time-sharing operating system allows many users to be served simultaneously, so sophisticated CPU scheduling schemes and Input/output management are required.
Time-sharing operating systems are very difficult and expensive to build.

Advantages of Time Sharing Operating System

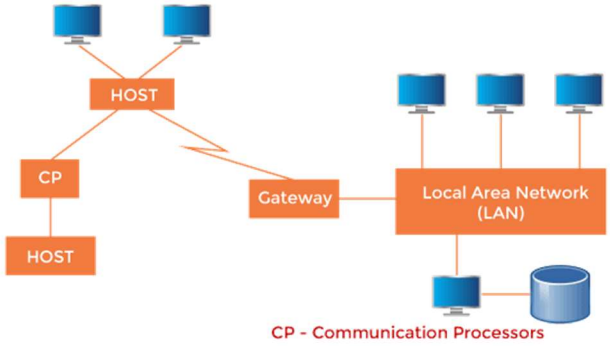
- The time-sharing operating system provides effective utilization and sharing of resources.
- This system reduces CPU idle and response time.

Disadvantages of Time Sharing Operating System

- Data transmission rates are very high in comparison to other methods.
- Security and integrity of user programs loaded in memory and data need to be maintained as many users access the system at the same time.

Distributed Operating System

The Distributed Operating system is not installed on a single machine, it is divided into parts, and these parts are loaded on different machines. A part of the distributed Operating system is installed on each machine to make their communication possible. Distributed Operating systems are much more complex, large, and sophisticated than Network operating systems because they also have to take care of varying networking protocols.



A Typical View of a Distributed System

Advantages of Distributed Operating System

- The distributed operating system provides sharing of resources.
- This type of system is fault-tolerant.

Disadvantages of Distributed Operating System

- Protocol overhead can dominate computation cost.

Operating System Services

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)
  - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
- Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- I/O operations - A running program may require I/O, which may involve a file or an I/O device.
- File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- User interface - Almost all operating systems have a user interface (UI)
  - Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
- Program execution - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- I/O operations - A running program may require I/O, which may involve a file or an I/O device.
- File-system manipulation - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

Another set of operating system services are used for ensuring the efficient operation of the system itself via resource sharing

- **Resource allocation -** When multiple users or multiple jobs are running concurrently, resources must be allocated to each of them
  - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code.
- **Accounting -** To keep track of which users use how much and what kinds of computer resources
- **Protection and security -** The owners of information stored in a multi-user or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts

User Operating System Interface

The Command Line Interface

- There are two fundamental approaches for users to interface with the operating system: command line interface and graphical user interface
- The command line interface, or command interpreter, allows users to directly enter commands to be performed by the operating system
  - Part of the interface is implemented in the form of a single shell program; the other part is implemented as system programs
  - There are various kinds of shell programs (C shell, Bourne shell, korn shell, etc.)
  - The interface fetches a command from the user and executes it

```
$> pwd
/home/jjt107/http/cs410
$> date
Tue Aug 15 12:49:41 CDT 2006
$>
```

The Graphical User Interface (GUI)

- A graphical user interface (GUI) provides a mouse-based interface consisting of windows and menus
- Depending on the position of the mouse cursor, clicking on a mouse button can invoke a program, select a file or directory, or pull down a menu that contains commands
- Graphical user interfaces first appeared because of the research that took place in the early 1970s at the Xerox PARC research facility
- Many systems now include both CLI and GUI interfaces
  - Microsoft Windows is GUI with CLI "command" shell
  - Apple Mac OS X has "Aqua" GUI interface with UNIX kernel underneath and shells available
  - Solaris is CLI with optional GUI interfaces (X Window System, Java Desktop, KDE)

System Calls

System Calls in Operating System (OS)

A system call is a way for a user program to interface with the operating system. The program requests several services, and the OS responds by invoking a series of system calls to satisfy the request. A system call can be written in assembly language or a high-level language like **C** or **Pascal**. System calls are predefined functions that the operating system may directly invoke if a high-level language is used.

What is a System Call?

A system call is a method for a computer program to request a service from the kernel of the operating system on which it is running. A system call is a method of interacting with the operating system via programs. A system call is a request from computer software to an operating system's kernel.

The **Application Program Interface (API)** connects the operating system's functions to user programs. It acts as a link between the operating system and a process, allowing user-level programs

to request operating system services. The kernel system can only be accessed using system calls. System calls are required for any programs that use resources.

How are system calls made?

When a computer software needs to access the operating system's kernel, it makes a system call. The system call uses an API to expose the operating system's services to user programs. It is the only method to access the kernel system. All programs or processes that require resources for execution must use system calls, as they serve as an interface between the operating system and user programs.

Below are some examples of how a system call varies from a user function.

1. A system call function may create and use kernel processes to execute the asynchronous processing.
2. A system call has greater authority than a standard subroutine. A system call with kernel-mode privilege executes in the kernel protection domain.
3. System calls are not permitted to use shared libraries or any symbols that are not present in the kernel protection domain.
4. The code and data for system calls are stored in global kernel memory.

Why do you need system calls in Operating System?

There are various situations where you must require system calls in the operating system. Following of the situations are as follows:

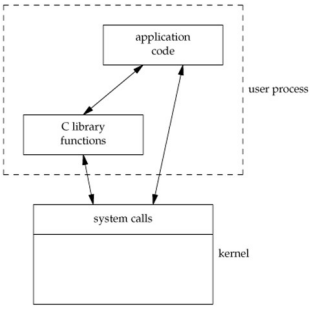
1. It is must require when a file system wants to create or delete a file.
2. Network connections require the system calls to sending and receiving data packets.
3. If you want to read or write a file, you need to system calls.
4. If you want to access hardware devices, including a printer, scanner, you need a system call.
5. System calls are used to create and manage new processes.

How System Calls Work

The Applications run in an area of memory known as user space. A system call connects to the operating system's kernel, which executes in kernel space. When an application creates a system call, it must first obtain permission from the kernel. It achieves this using an interrupt request, which pauses the current process and transfers control to the kernel.

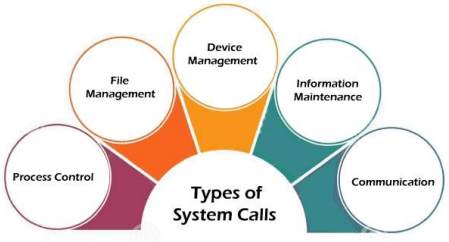
If the request is permitted, the kernel performs the requested action, like creating or deleting a file. As input, the application receives the kernel's output. The application resumes the procedure after the input is received. When the operation is finished, the kernel returns the results to the application and then moves data from kernel space to user space in memory.

A simple system call may take few nanoseconds to provide the result, like retrieving the system date and time. A more complicated system call, such as connecting to a network device, may take a few seconds. Most operating systems launch a distinct kernel thread for each system call to avoid bottlenecks. Modern operating systems are multi-threaded, which means they can handle various system calls at the same time.



Types of System Calls

There are commonly five types of system calls. These are as follows:



1. **Process Control**
2. **File Management**
3. **Device Management**
4. **Information Maintenance**
5. **Communication**

Process Control

Process control is the system call that is used to direct the processes. Some process control examples include creating, load, abort, end, execute, process, terminate the process, etc.

File Management

File management is a system call that is used to handle the files. Some file management examples include creating files, delete files, open, close, read, write, etc.

Device Management

Device management is a system call that is used to deal with devices. Some examples of device management include read, device, write, get device attributes, release device, etc.

Information Maintenance

Information maintenance is a system call that is used to maintain information. There are some examples of information maintenance, including getting system data, set time or date, get time or date, set system data, etc.

Communication

Communication is a system call that is used for communication. There are some examples of communication, including create, delete communication connections, send, receive messages, etc.

Examples of Windows and Unix system calls

There are various examples of Windows and Unix system calls. These are as listed below in the table:

Process	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	Fork() Exit() Wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	Open() Read() Write() Close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	Ioctl() Read() Write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	Getpid() Alarm() Sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	Pipe() Shmget() Mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorgroup()	Chmod() Umask() Chown()

open()

The **open()** system call allows you to access a file on a file system. It allocates resources to the file and provides a handle that the process may refer to. Many processes can open a file at once or by a single process only. It's all based on the file system and structure.

read()

It is used to obtain data from a file on the file system. It accepts three arguments in general:

- A file descriptor.
- A buffer to store read data.
- The number of bytes to read from the file.

The file descriptor of the file to be read could be used to identify it and open it using **open()** before reading.

wait()

In some systems, a process may have to wait for another process to complete its execution before proceeding. When a parent process makes a child process, the parent process execution is suspended until the child process is finished. The **wait()** system call is used to suspend the parent process. Once the child process has completed its execution, control is returned to the parent process.

write()

It is used to write data from a user buffer to a device like a file. This system call is one way for a program to generate data. It takes three arguments in general:

- A file descriptor.
- A pointer to the buffer in which data is saved.
- The number of bytes to be written from the buffer.

fork()

Processes generate clones of themselves using the **fork()** system call. It is one of the most common ways to create processes in operating systems. When a parent process spawns a child process, execution of the parent process is interrupted until the child process completes. Once the child process has completed its execution, control is returned to the parent process.

close()

It is used to end file system access. When this system call is invoked, it signifies that the program no longer requires the file, and the buffers are flushed, the file information is altered, and the file resources are de-allocated as a result.

exec()

When an executable file replaces an earlier executable file in an already executing process, this system function is invoked. As a new process is not built, the old process identification stays, but the new process replaces data, stack, data, head, etc.

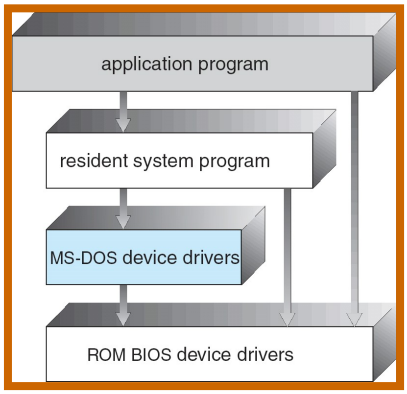
exit()

The **exit()** is a system call that is used to end program execution. This call indicates that the thread execution is complete, which is especially useful in multi-threaded environments. The operating system reclaims resources spent by the process following the use of the **exit()** system function.

Operating System Structure

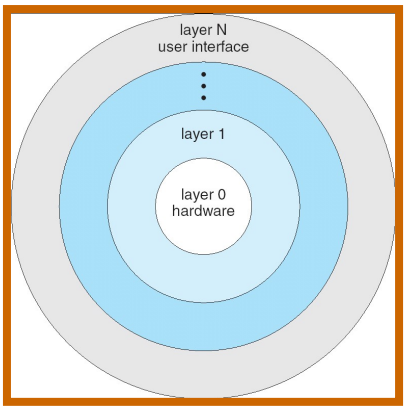
Simple Structure

- MS-DOS – written to provide the most functionality in the least space
  - Not divided into modules
  - Although MS-DOS has some structure, its interfaces and levels of functionality are not well separated



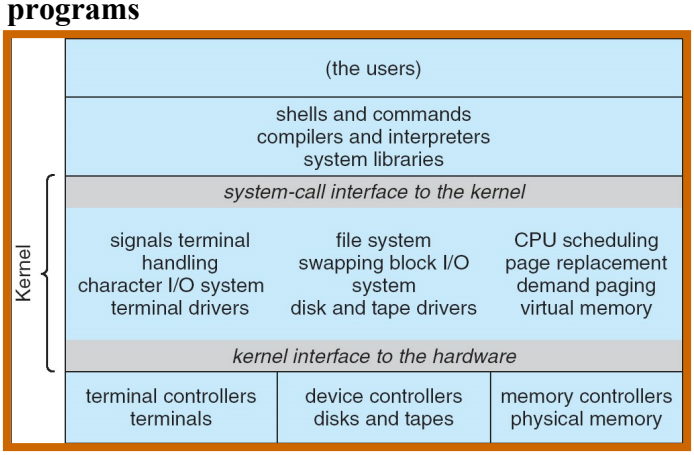
Layered Approach

- In a layered approach, the operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



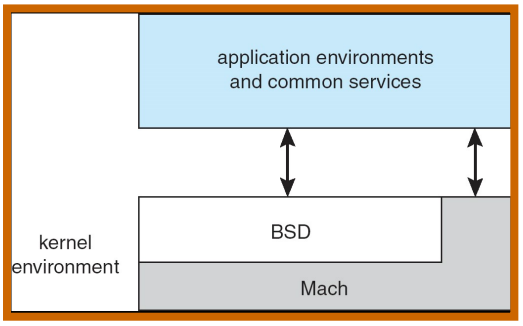
UNIX

- Limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
  - **The kernel**
    - Consists of everything below the system-call interface and above the physical hardware
    - Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level
  - **Systems programs**



Microkernel System Structure

- Moves as much from the kernel into "*user*" space
- Communication takes place between user modules using message passing
- Benefits:
  - Easier to extend a microkernel
  - Easier to port the operating system to new architectures
  - More reliable (less code is running in kernel mode)
  - More secure
- Detriments:
  - Performance overhead of user space to kernel space communication



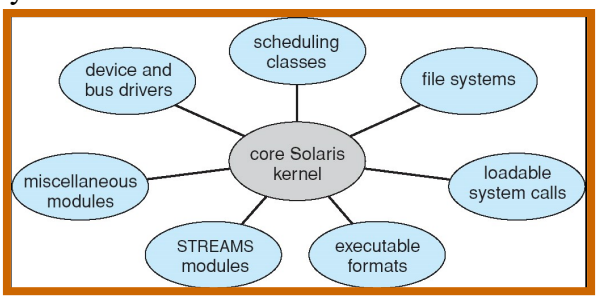
Mac OS X Structure

Modules

- Most modern operating systems implement kernel modules

- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel

- Overall, similar to layers but more flexible

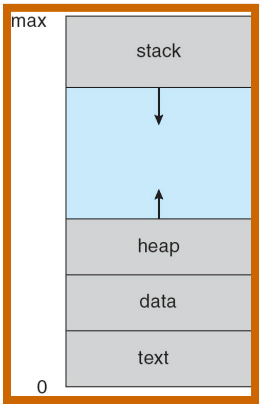


Solaris Modular Approach

Process

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter and process registers
  - stack and heap
  - text section
  - data section



Process in Memory

Attributes of a process

The Attributes of the process are used by the Operating System to create the process control block (PCB) for each of them. This is also called context of the process. Attributes which are stored in the PCB are described below.

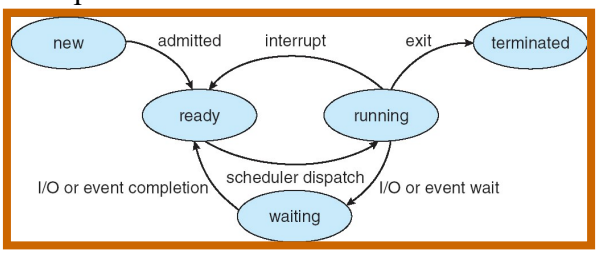
1. Process ID: When a process is created, a unique id is assigned to the process which is used for unique identification of the process in the system.
2. Program counter: A program counter stores the address of the last instruction of the process on which the process was suspended. The CPU uses this address when the execution of this process is resumed.
3. Process State: The Process, from its creation to the completion, goes through various states which are new, ready, running and waiting. We will discuss about them later in detail.
4. Priority: Every process has its own priority. The process with the highest priority among the processes gets the CPU first. This is also stored on the process control block.

Process Attributes

5. General Purpose Registers: Every process has its own set of registers which are used to hold the data which is generated during the execution of the process.
6. List of open files: During the Execution, Every process uses some files which need to be present in the main memory. OS also maintains a list of open files in the PCB.
7. List of open devices: OS also maintain the list of all open devices which are used during the execution of the process.

Process State

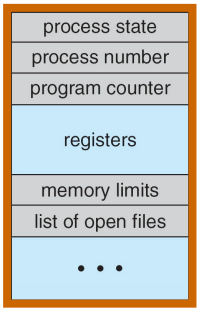
- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal)
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution

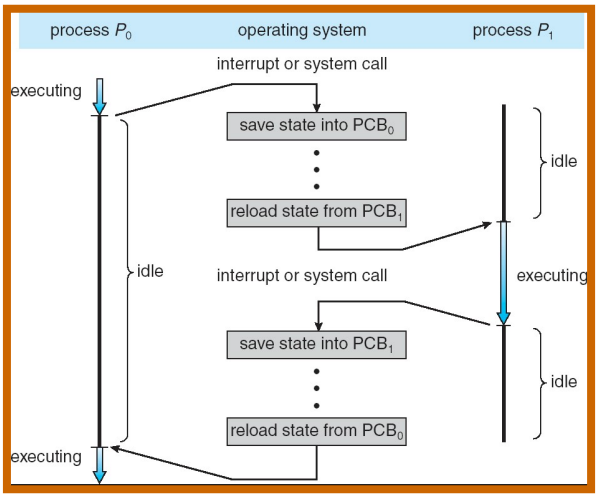


Process Control Block (PCB)

Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information





CPU Switch From Process to Process

Context Switch

- When CPU switches to another process (because of an interrupt), the system must save the state of the old process and load the saved state for the new process
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

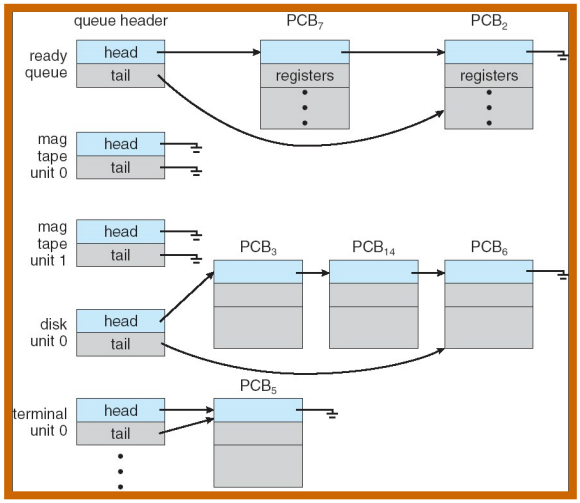
Process Scheduling

Process Scheduling Queues

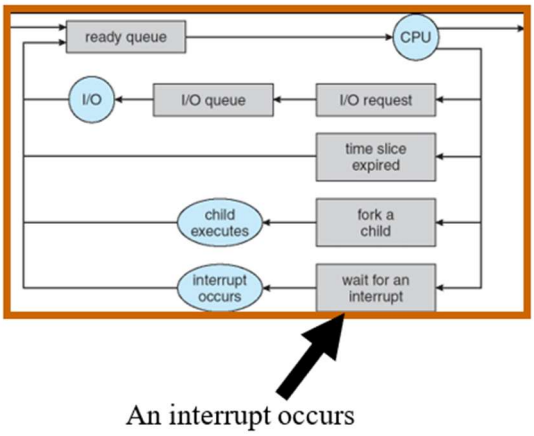
- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute

- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue and Various I/O Device Queues



Representation of Process Scheduling



Schedulers

- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU (covered in Chapter 5)
- Short-term scheduler is invoked very frequently (milliseconds) Þ (must be fast)
- Long-term scheduler is invoked very infrequently (seconds, minutes) Þ (may be slow)
- The long-term scheduler controls the *degree of multiprogramming*
- Processes can be described as either:
  - I/O-bound process – spends more time doing I/O than computations, many short CPU bursts
  - CPU-bound process – spends more time doing computations; few very long CPU bursts
- Some systems have no long-term scheduler
  - Every new process is loaded into memory
  - System stability effected by physical limitation and self-adjusting nature of the human user

Operations on Processes

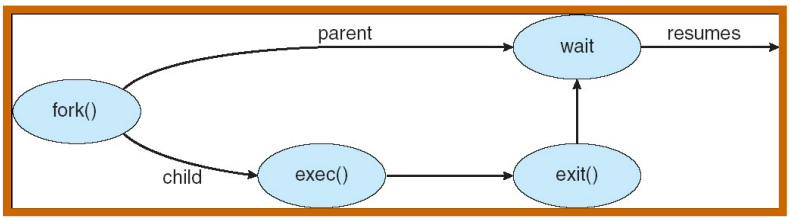
Process Creation

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources

- Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until some or all of its children have terminated
- Address space options
  - Child process is a duplicate of the parent process (same program and data)
  - Child process has a new program loaded into it
- UNIX example
  - **fork()** system call creates a new process
  - **exec()** system call used after a **fork()** to replace the memory space of the process with a new program

C Program Forking Separate Process

```
int main(void)
{
pid_t  processID;
processID = fork(); // Create a process
if (processID < 0)
   { // Error occurred
   fprintf(stderr, "Fork failed");
   exit(-1);
   } // End if
else if (processID == 0)
   { // Inside the child process (no execlp() this time)
   printf ("(Child) I am doing something");
   } // End else if
else
   { // Inside the parent process
   printf("(Parent) I am waiting for PID#%d to finish\n", processID);
   wait(NULL);
   printf ("\n(Parent) I have finished waiting; the child is done");
   exit(0);
   } // End else
return 0;
} // End main
```



Process Termination

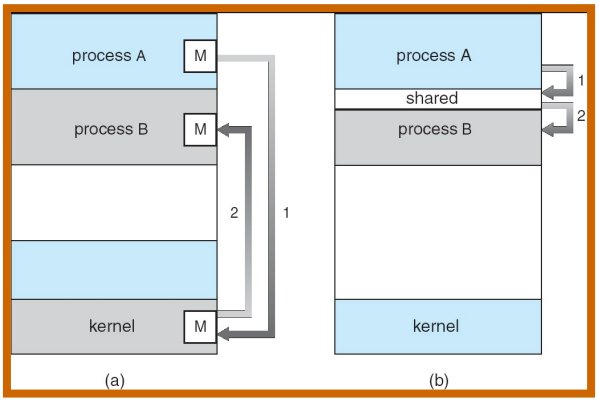
- Process executes last statement and asks the operating system to terminate it (via **exit**)
  - Exit (or return) status value from child is received by the parent (via **wait()**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**kill() function**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
      - All children terminated - *cascading termination*

Interprocess Communication

- **An independent** process is one that cannot affect or be affected by the execution of another process
- **A cooperating** process can affect or be affected by the execution of another process in the system
- Advantages of process cooperation
  - Information sharing (of the same piece of data)

- Computation speed-up (break a task into smaller subtasks)
- Modularity (dividing up the system functions)
- Convenience (to do multiple tasks simultaneously)

- Two fundamental models of interprocess communication
  - Shared memory (a region of memory is shared)
  - Message passing (exchange of messages between processes)



Communications Models

Shared Memory Systems

- Shared memory requires communicating processes to establish a region of shared memory
- Information is exchanged by reading and writing data in the shared memory
- A common paradigm for cooperating processes is the producer-consumer problem
- A *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Message-Passing Systems

- Mechanism to allow processes to communicate and to synchronize their actions
- No address space needs to be shared; this is particularly useful in a distributed processing environment (e.g., a chat program)
- Message-passing facility provides two operations:
  - **send**(*message*) – message size can be fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Logical implementation of communication link
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically between every pair of processes that want to communicate
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional
- Disadvantages
  - Limited modularity of the resulting process definitions
  - Hard-coding of identifiers are less desirable than indirection techniques
- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P

- **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically between every pair of processes that want to communicate
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional
- Disadvantages
  - Limited modularity of the resulting process definitions
  - Hard-coding of identifiers are less desirable than indirection techniques
- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
    - send(A, message) – send message to mailbox A
    - receive(A, message) – receive message from mailbox A
- Properties of communication link
  - Link is established between a pair of processes only if both have a shared mailbox
  - A link may be associated with more than two processes
  - Between each pair of processes, there may be many different links, with each link corresponding to one mailbox
- For a shared mailbox, messages are received based on the following methods:
  - Allow a link to be associated with at most two processes
  - Allow at most one process at a time to execute a receive() operation
  - Allow the system to select arbitrarily which process will receive the message (e.g., a round robin approach)
- Mechanisms provided by the operating system
  - Create a new mailbox
  - Send and receive messages through the mailbox
  - Destroy a mailbox

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null
- When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver

Buffering

- Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue
- These queues can be implemented in three ways:
  1. **Zero capacity** – the queue has a maximum length of zero
  - Sender must block until the recipient receives the message
  2. **Bounded capacity** – the queue has a finite length of *n*
  - Sender must wait if queue is full
  3. **Unbounded capacity** – the queue length is unlimited

  Sender never blocks

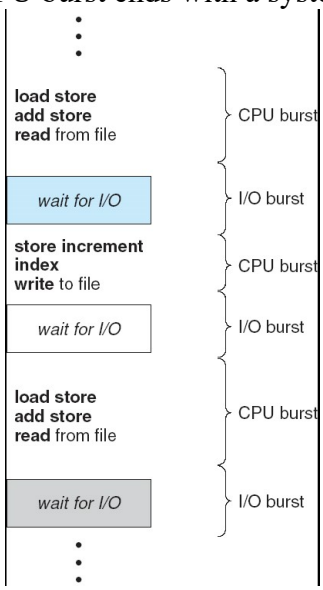
CPU Scheduling

Basic Concepts

- Maximum CPU utilization is obtained with multiprogramming
  - Several processes are kept in memory at one time

- Every time a running process has to wait, another process can take over use of the CPU

- Scheduling of the CPU is fundamental to operating system design
- Process execution consists of a *cycle* of a **CPU time burst** and an **I/O time burst** (i.e. wait) as shown on the next slide
  - Processes alternate between these two states (i.e., CPU burst and I/O burst)
  - Eventually, the final CPU burst ends with a system request to terminate execution



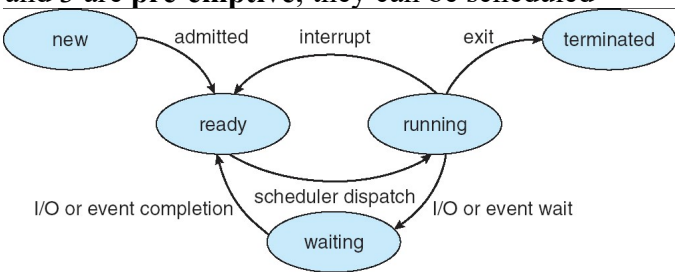
Alternating Sequence of CPU And I/O Bursts

CPU Scheduler

- The CPU scheduler selects from among the processes in memory that are ready to execute and allocates the CPU to one of them
- CPU scheduling is affected by the following set of circumstances:

1. (N) A process switches from **running** to **waiting** state
2. (P) A process switches from **running** to **ready** state
3. (P) A process switches from **waiting** to **ready** state
4. (N) A processes switches from **running** to **terminated** state

- Circumstances 1 and 4 are **non-preemptive**; they offer no schedule choice
- Circumstances 2 and 3 are **pre-emptive**; they can be scheduled



Dispatcher

- The dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- The dispatcher needs to run as fast as possible, since it is invoked during process context switch
- The time it takes for the dispatcher to stop one process and start another process is called **dispatch latency**

Scheduling Criteria

- Different CPU scheduling algorithms have different properties
- The choice of a particular algorithm may favor one class of processes over another

- In choosing which algorithm to use, the properties of the various algorithms should be considered
- Criteria for comparing CPU scheduling algorithms may include the following
  - **CPU utilization** – percent of time that the CPU is busy executing a process
  - **Throughput** – number of processes that are completed per time unit
  - **Response time** – amount of time it takes from when a request was submitted until the first response occurs (but not the time it takes to output the entire response)
  - **Waiting time** – the amount of time before a process starts after first entering the ready queue (or the sum of the amount of time a process has spent waiting in the ready queue)
  - **Turnaround time** – amount of time to execute a particular process from the time of submission through the time of completion

Optimization Criteria

- It is desirable to
  - Maximize CPU utilization
  - Maximize throughput
  - Minimize turnaround time
  - Minimize start time
  - Minimize waiting time
  - Minimize response time
- In most cases, we strive to optimize the average measure of each metric
- In other cases, it is more important to optimize the minimum or maximum values rather than the average

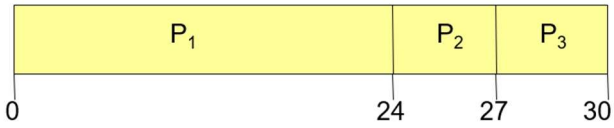
Single Processor Scheduling Algorithms

- First Come, First Served (FCFS)
- Shortest Job First (SJF)
- Priority
- Round Robin (RR)

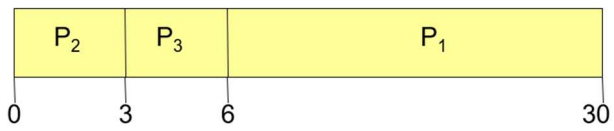
First Come, First Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- With FCFS, the process that requests the CPU first is allocated the CPU first
- Case #1: Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:



  Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
- Average waiting time: $(0 + 24 + 27)/3 = 17$
- Average turn-around time: $(24 + 27 + 30)/3 = 27$
- Case #2: Suppose that the processes arrive in the order: P_2 , P_3 , P_1
- The Gantt chart for the schedule is:



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: $(6 + 0 + 3)/3 = 3$ (Much better than Case #1)
- Average turn-around time: $(3 + 6 + 30)/3 = 13$
- Case #1 is an example of the **convoy effect;** all the other processes wait for one long-running process to finish using the CPU

- This problem results in lower CPU and device utilization; Case #2 shows that higher utilization might be possible if the short processes were allowed to run first

- The FCFS scheduling algorithm is **non-preemptive**
  - Once the CPU has been allocated to a process, that process keeps the CPU until it releases it either by terminating or by requesting I/O
  - It is a troublesome algorithm for time-sharing systems

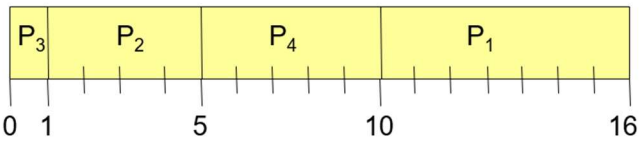
Shortest Job First (SJF) Scheduling

- The SJF algorithm associates with each process the length of its next CPU burst
- When the CPU becomes available, it is assigned to the process that has the smallest next CPU burst (in the case of matching bursts, FCFS is used)
- Two schemes:
  - **Nonpreemptive** – once the CPU is given to the process, it cannot be preempted until it completes its CPU burst
  - **Preemptive** – if a new process arrives with a CPU burst length less than the remaining time of the current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF)

Example #1: Non-Preemptive SJF (simultaneous arrival)

Process	Arrival Time	Burst Time
P_1	0.0	6
P_2	0.0	4
P_3	0.0	1
P_4	0.0	5

- SJF (non-preemptive, simultaneous arrival)

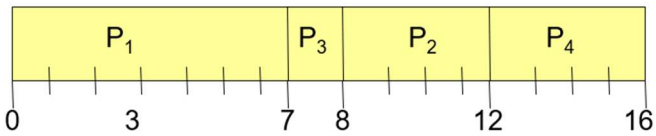


- Average waiting time = (0 + 1 + 5 + 10)/4 = 4
- Average turn-around time = (1 + 5 + 10 + 16)/4 = 8

Example #2: Non-Preemptive SJF (varied arrival times)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non-preemptive, varied arrival times)



- Average waiting time

$$= ((0 – 0) + (8 – 2) + (7 – 4) + (12 – 5))/4$$
$$= (0 + 6 + 3 + 7)/4 = 4$$

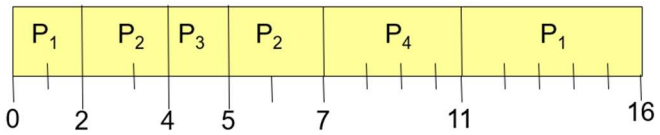
- Average turn-around time:

$$= ((7 – 0) + (12 – 2) + (8 - 4) + (16 – 5))/4$$
$$= (7 + 10 + 4 + 11)/4 = 8$$

Example #3: Preemptive SJF (Shortest-remaining-time-first)

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (preemptive, varied arrival times)



- Average waiting time
  = ([(0 – 0) + (11 - 2)] + [(2 – 2) + (5 – 4)] + (4 - 4) + (7 – 5))/4
  = 9 + 1 + 0 + 2)/4
  = 3
- Average turn-around time = (16 + 7 + 5 + 11)/4 = 9.75

Priority Scheduling

- The SJF algorithm is a special case of the general priority scheduling algorithm
- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
- Priority scheduling can be either preemptive or non-preemptive
  - A **preemptive** approach will preempt the CPU if the priority of the newly-arrived process is higher than the priority of the currently running process
  - A **non-preemptive** approach will simply put the new process (with the highest priority) at the head of the ready queue
- SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time
- The main problem with priority scheduling is **starvation**, that is, low priority processes may never execute
- A solution is **aging**; as time progresses, the priority of a process in the ready queue is increased

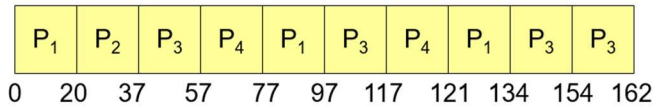
Round Robin (RR) Scheduling

- In the round robin algorithm, each process gets a small unit of CPU time (a *time quantum*), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are *n* processes in the ready queue and the time quantum is *q*, then each process gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No process waits more than (*n*-1)*q* time units.
- Performance of the round robin algorithm
  - *q* large Þ FCFS
  - *q* small Þ *q* must be greater than the context switch time; otherwise, the overhead is too high
- One rule of thumb is that 80% of the CPU bursts should be shorter than the time quantum

Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

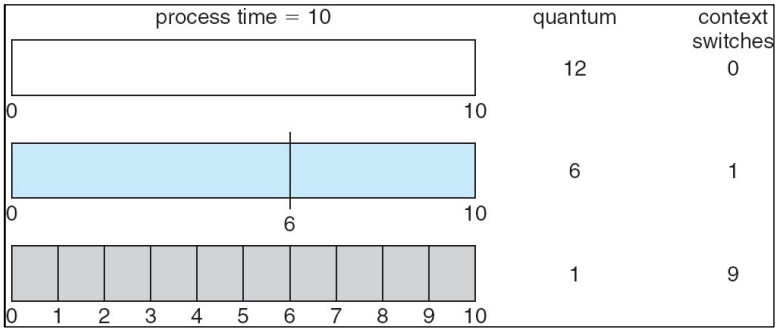
- The Gantt chart is:



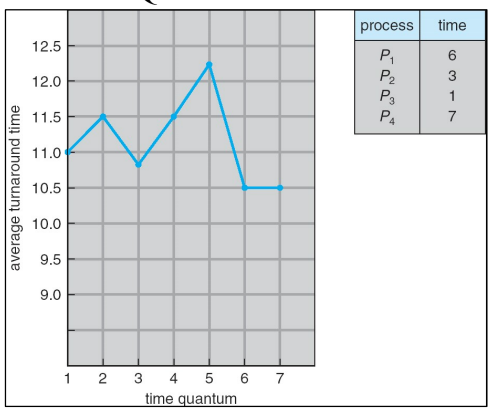
- Typically, higher average turnaround than SJF, but better *response time*
- Average waiting time
  = ([(0 – 0) + (77 - 20) + (121 – 97)] + (20 – 0) + [(37 – 0) + (97 - 57) + (134 – 117)] + [(57 – 0) + (117 – 77)]) / 4
  = (0 + 57 + 24) + 20 + (37 + 40 + 17) + (57 + 40)) / 4
  = (81 + 20 + 94 + 97)/4
  = 292 / 4 = 73

- Average turn-around time = 134 + 37 + 162 + 121) / 4 = 113.5

Time Quantum and Context Switches

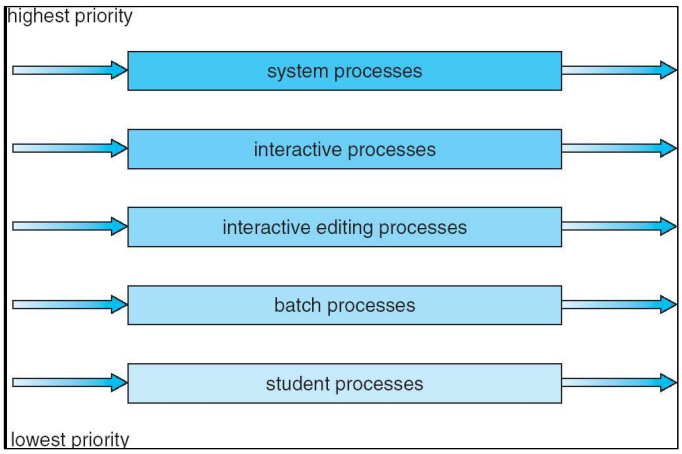


Turnaround Time Varies with The Time Quantum



Multi-level Queue Scheduling

- Multi-level queue scheduling is used when processes can be classified into groups
- For example, **foreground** (interactive) processes and **background** (batch) processes
  - The two types of processes have different response-time requirements and so may have different scheduling needs
  - Also, foreground processes may have priority (externally defined) over background processes
- A multi-level queue scheduling algorithm partitions the ready queue into several separate queues
- The processes are permanently assigned to one queue, generally based on some property of the process such as memory size, process priority, or process type
- Each queue has its own scheduling algorithm
  - The foreground queue might be scheduled using an RR algorithm
  - The background queue might be scheduled using an FCFS algorithm
- In addition, there needs to be scheduling among the queues, which is commonly implemented as fixed-priority pre-emptive scheduling
  - The foreground queue may have absolute priority over the background queue
- One example of a multi-level queue are the five queues shown below
- Each queue has absolute priority over lower priority queues
- For example, no process in the batch queue can run unless the queues above it are empty
- However, this can result in starvation for the processes in the lower priority queues



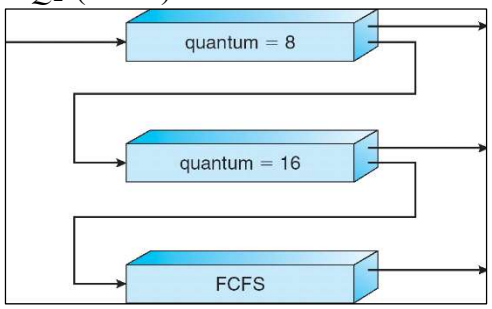
- Another possibility is to time slice among the queues
- Each queue gets a certain portion of the CPU time, which it can then schedule among its various processes
  - The foreground queue can be given 80% of the CPU time for RR scheduling
  - The background queue can be given 20% of the CPU time for FCFS scheduling

Multi-level Feedback Queue Scheduling

- In multi-level feedback queue scheduling, a process can move between the various queues; aging can be implemented this way
- A multilevel-feedback-queue scheduler is defined by the following parameters:
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to promote a process
  - Method used to determine when to demote a process
  - Method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Scheduling
  - A new job enters queue Q_0 *(RR)* and is placed at the end. When it gains the CPU, the job receives 8 milliseconds. If it does not finish in 8 milliseconds, the job is moved to the end of queue Q_1.
  - A Q_1 (RR) job receives 16 milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 (FCFS).



Multiple-Processor Scheduling

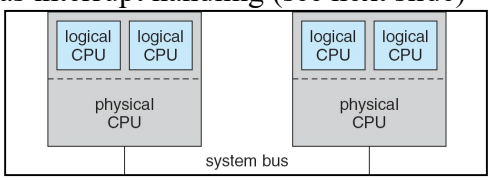
- If multiple CPUs are available, load sharing among them becomes possible; the scheduling problem becomes more complex
- We concentrate in this discussion on systems in which the processors are identical (homogeneous) in terms of their functionality
  - We can use any available processor to run any process in the queue
- Two approaches: **Asymmetric** processing and **symmetric** processing (see next slide)
- *Asymmetric multiprocessing (ASMP)*
  - One processor handles all scheduling decisions, I/O processing, and other system activities
  - The other processors execute only user code

- Because only one processor accesses the system data structures, the need for data sharing is reduced

- Symmetric multiprocessing (SMP)
  - Each processor schedules itself
  - All processes may be in a common ready queue or each processor may have its own ready queue
  - Either way, each processor examines the ready queue and selects a process to execute
  - Efficient use of the CPUs requires load balancing to keep the workload evenly distributed
    - In a **Push** migration approach, a specific task regularly checks the processor loads and redistributes the waiting processes as needed
    - In a **Pull** migration approach, an idle processor pulls a waiting job from the queue of a busy processor
  - Virtually all modern operating systems support SMP, including Windows XP, Solaris, Linux, and Mac OS X

Symmetric Multithreading

- Symmetric multiprocessing systems allow several threads to run concurrently by providing multiple physical processors
- An alternative approach is to provide multiple **logical** rather than **physical** processors
- Such a strategy is known as symmetric multithreading (SMT)
  - This is also known as hyperthreading technology
- The idea behind SMT is to create multiple logical processors on the same physical processor
  - This presents a view of several logical processors to the operating system, even on a system with a single physical processor
  - Each logical processor has its own **architecture state**, which includes general-purpose and machine-state registers
  - Each logical processor is responsible for its own interrupt handling
  - However, each logical processor shares the resources of its physical processor, such as cache memory and buses
- SMT is a feature provided in the hardware, not the software
  - The hardware must provide the representation of the architecture state for each logical processor, as well as interrupt handling (see next slide)



A typical SMT architecture