**–Operating System Notes:**

An operating system acts as an – - **Resource allocator**

* **Control program**

First operating system –

* ATLAS (Manchester Univ., late 1950s – early 1960s)
* evolved from control programs

Types of operating systems –

* Single process operating system [MS DOS, 1981]
* Batch-processing operating system [ATLAS, Manchester Univ., late 1950s – early 1960s]
* Multiprogramming operating system [THE, Dijkstra, early 1960s]
* Multitasking operating system [CTSS, MIT, early 1960s]

**Multiprogramming** increases CPU utilization by keeping multiple jobs (code and data) in the memory so that the CPU always has one to execute.

|  |
| --- |
| Job 1 |
| Job 2 |
| Job 3 |
| Operating system |

**Multitasking** is a logical extension of multiprogramming.

CPU executes multiple tasks by switching among them.

The switching is very fast.

Requires an interactive (hands-on) computer where the user can directly interact with the computer. Response time should be minimal.

A **kernel** is that part of the operating system which interacts directly with the hardware and performs the most crucial tasks.

A **microkernel** is much smaller in size than a conventional kernel and supports only the core operating system functionalities.

A **shell**, also known as a command interpreter, is that part of the operating system that receives commands from the users and gets them executed.

A **system call** is a mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform.

User programs typically do not have permission to perform operations like accessing I/O devices and communicating other programs.

A user program invokes system calls when it requires such services.

System calls provide an interface between a program and the operating system.

System calls are of different types.

E.g. – fork, exec, getpid, getppid, wait, exit.

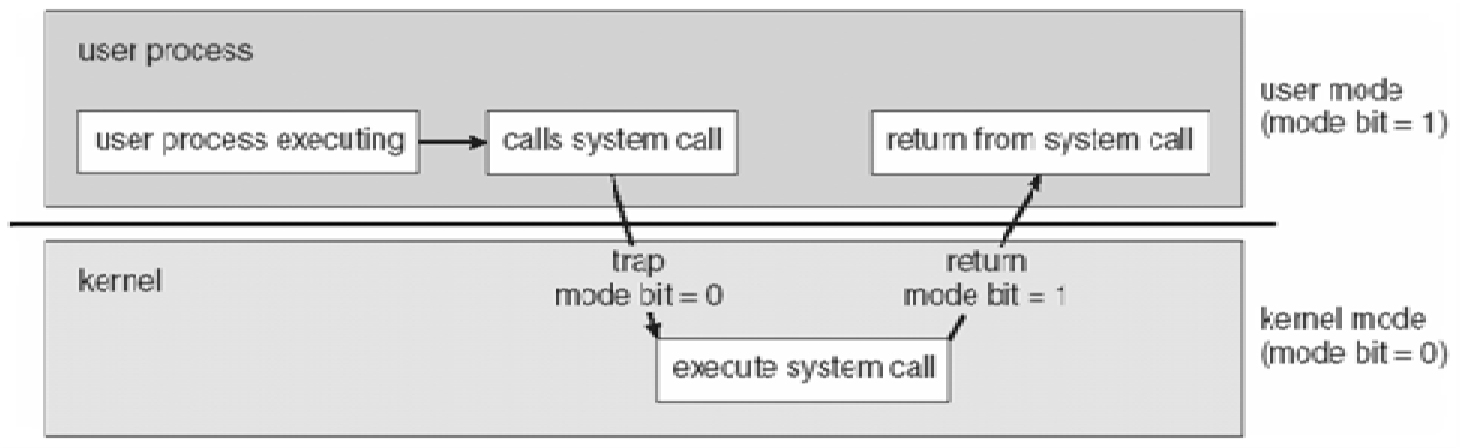
**Dual-mode operation** – - **User mode**

* + **Kernel mode** / supervisor mode / system mode / privileged mode

Mode bit– 0: kernel, 1: user

Request using a system call

*An Introduction to Operating Systems* 3



Duties of an operating system –

* + **Process management**
  + creating and deleting user and system processes
  + suspending and resuming processes
  + interprocess communication
  + process synchronization
  + deadlock handling
  + **Memory management**
  + Keeping track of which part of memory is being used by which job
  + Allocating and deallocating memory space
  + **Storage management**
  + file system management
  + creating, deleting and manipulating files and directories
  + mass storage management
  + free space management
  + storage allocation
  + disk scheduling
  + **Caching**
  + **Input-output management**

Operating system services –

* + Helpful to the user
  + user interface (CUI/shell and GUI)
  + program execution
  + I/O operation
  + file system manipulation
  + communication
  + error detection
  + Helpful to the system
  + resource allocation
  + accounting
  + protection and security

Operating system structures –

* + Monolithic [MS DOS, Unix, Linux]
  + Layered [THE]
  + Microkernel [Mach, MINIX]

A **real-time operating system** (**RTOS**) has well-defined and fixed time constraints which have to be met or the system will fail.

An RTOS is used when rigid time constraints have been placed on the operation of processes or flow of data.

An RTOS is often used in the control device in a dedicated application.

Hard- and soft- RTOS.

Applications: embedded systems, robotics, scientific utilities, etc.

Operating systems for smart phones –

* CPUs of smart phones are made to be much slower to conserve energy

**Booting** is the process of starting the computer and loading the kernel.

When a computer is turned on, the power-on self-tests (POST) are performed.

Then the bootstrap loader, which resides in the ROM, is executed.

The bootstrap loader loads the kernel or a more sophisticated loader.

**UNIT 2**

**PROCESS AND PROCESS SCHEDULING**

A **process** is a program in execution.

A process is a unit of work in a computer system.

The terms process and job are used interchangeably.

A process comprises of –

* text section containing the program code
* current activity represented by the values of the program counter and other registers - program stack
* data section containing global variables
* heap

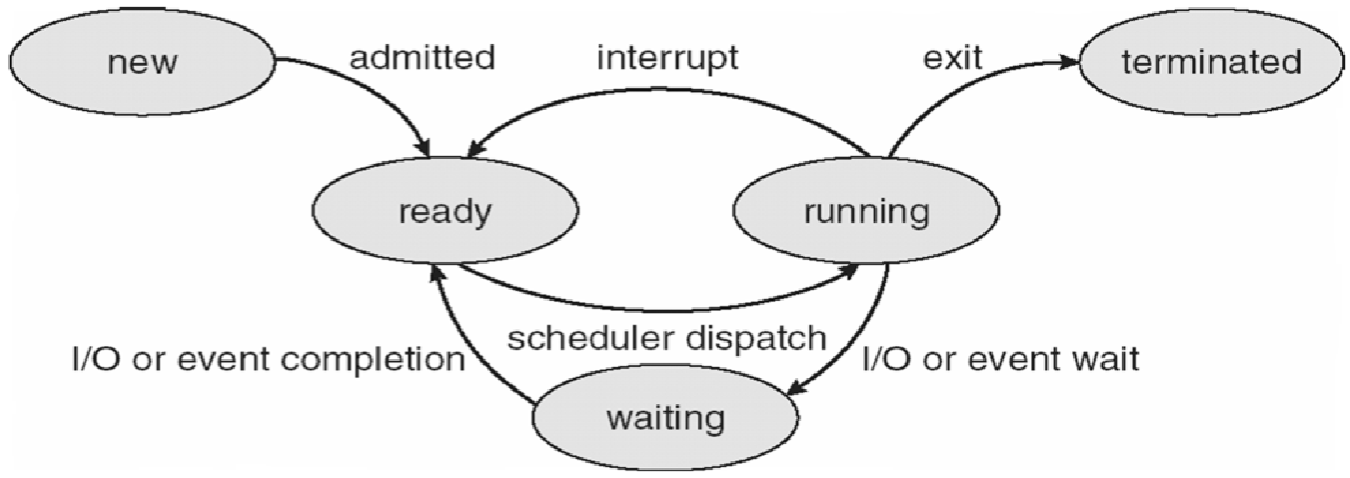
|  |
| --- |
| Stack |
| ↓    ↑ |
| Heap |
| Data |
| Text |

max

0

A program is a passive entity while a process is an active entity.

**Process state** is defined by the current activity of the process. As a process executes, its state changes.



Only one process can be in the running state at any instant.

Many processes can be ready or waiting.

Each process is internally represented by the operating system by a **process control block** (**PCB**) also called task control block.

PCB contains all information associated with the process –

* Process state
* Values of program counter and other registers
* CPU scheduling information - priority, pointer to scheduling queue, etc.
* Accounting information - process id, CPU- and real- time used, time limits, etc.

I/O status information - list of i/o devices allocated, list of open files, etc.

**Process scheduling** is selecting one process for execution out of all the ready processes.

The objective of multiprogramming is to have some process running at all times so as to maximize CPU utilization.

The objective of multitasking is to switch the CPU among the processes so frequently that the user can interact with each process while it is running.

To meet these objectives the process scheduler selects one of the available processes for execution.

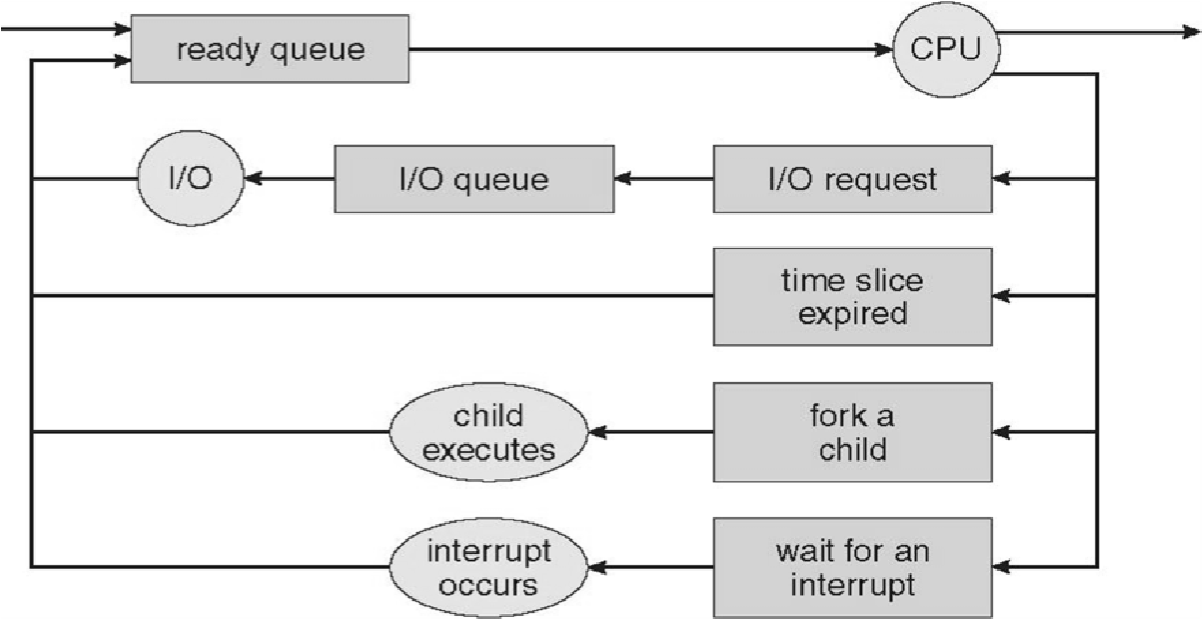
**Scheduling queues** are used to perform process scheduling.

As a process enters the system, it is put in a **job queue** that contains all the processes in the system. The processes that are residing in the memory and are ready for execution are kept in the **ready queue**.

The ready queue is implemented as a linked list of PCBs with a header containing pointers to the first and the last PCBs.

The list of the processes waiting for a particular i/o device is called a **device queue**. Each device has its own device queue.

A queuing diagram shows how the processes migrate among the various scheduling queues.



There are three types of **schedulers**.

A process migrates among the various scheduling queues throughout its lifetime.

The operating system has to select the processes from the queues according to some criteria. The selection is done by the appropriate scheduler.

A **long-term scheduler** (**job scheduler**) selects processes from those submitted by the user and loads them into the memory.

The long-term scheduler controls the degree of multiprogramming which is represented by the number of processes in the memory.

It is invoked less frequently.

A **short-term scheduler** (**CPU scheduler**) selects one of the processes in the memory and allocates the CPU to it.

The short-term scheduler is invoked frequently and should be very fast.

The long-term scheduler should select a proper mix of CPU-bound processes and i/o-bound processes.

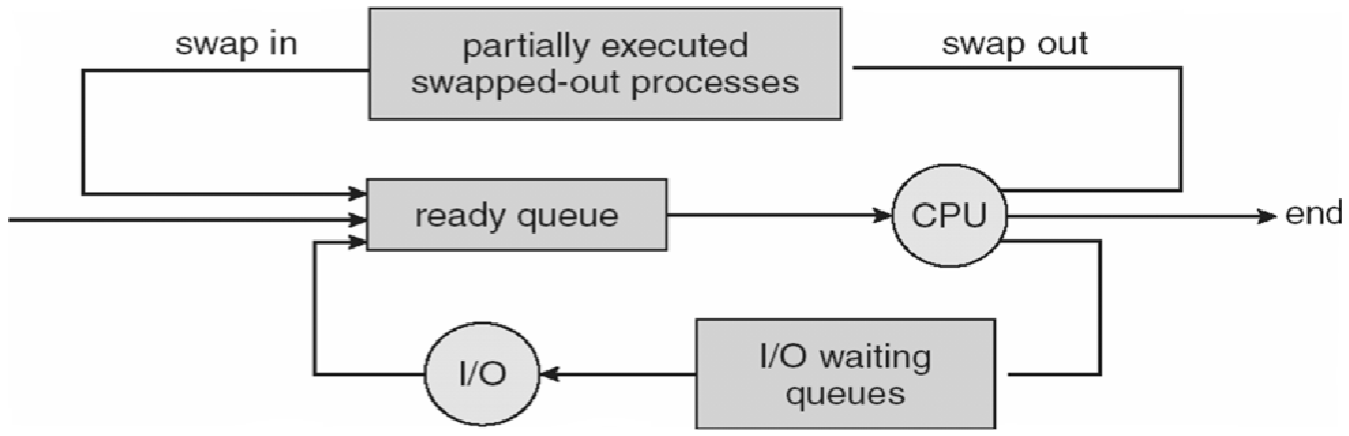
A CPU-bound process spends most of its time doing computations. An i/o-bound process spends most of its time doing i/o.

Some multitasking operating systems, like Unix and Windows, do not use long-term schedulers. All new processes are put in the memory for the perusal of the short-term scheduler.

A **medium-term scheduler** removes processes from the memory and from the competition for the CPU, thus reducing the degree of multiprogramming.

The processes can be later reintroduced in the memory and their executions can be resumed. This scheme is called **swapping**.

Swapping may be used to improve process mix and to free up some memory in uncontrollable circumstances.



**Context switching** is done to switch between processes.

Switching the CPU to another process requires saving the state of the current process and reloading the state of another process.

States are saved into and reloaded from PCBs.

**Process creation** is one process creating another process.

The processes are called parent process and child process, respectively.

Each process has a unique id.

A process may obtain resources either from its parent or from the operating system directly.

A parent process may continue executing with its children processes or may wait for them to complete.

A process may be a duplicate of its parent process (same code and data) or may have a new program loaded into it.

**Process termination** marks the deletion of the PCB of the process.

A parent process may terminate a child process –

* if it has exceeded its resource usage
* if its result is no more needed
* if the parent process is terminating and the operating system does not allow an **orphan process** (this may lead to cascading process terminations)

Typically, the kernel is the first process to be created, is the ancestor of all other processes and is at the root of the process tree.

A **zombie process** is a process that has terminated but its PCB still exists because its parent has not yet accepted its return value.

**Interprocess communication** –

* **Reasons –**
* information sharing
* computational speedup
* modularity
* convenience
* **Models –**
* shared memory
* message passing [send(P,message) and receive(id,message)]

A **thread** is the smallest sequence of instructions that can be managed independently by a scheduler.

A thread is a component of a process.

Multiple threads can exist within the same process, executing concurrently and share resources such as memory.

The threads of a process share its instructions (executable code) and its context (the values of its variables at any given moment).

Difference between process and thread –

* processes are typically independent while threads exist as parts of a process
* processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources
* processes have separate address spaces, whereas threads share their address space
* processes interact only through system-provided inter-process communication mechanisms
* context switching between threads in the same process is typically faster than context switching between processes

Advantages of multi-threaded programming – - responsiveness

* faster execution
* better resource utilization
* easy communication
* parallelization

The execution of a process consists of alternate **CPU bursts** and **i/o bursts**, starting and ending with CPU bursts.

The CPU scheduler is invoked when a process –

* switches from running state to waiting state (condition 1) -
* switches from running state to ready state (condition 2)
* switches from waiting state to ready state (condition 3)
* terminates (condition 4)

In **non-preemptive scheduling** or cooperating scheduling, a process keeps the CPU until it terminates or switches to the waiting state.

Some machines support non-preemptive scheduling only. E.g. – Window 3.1x.

In **preemptive scheduling**, a process can be forced to leave the CPU and switch to the ready queue. E.g. – Unix, Linux, Windows 95 and higher.

CPU scheduling is optional for conditions 2 and 3, but necessary in the other two conditions.

MS DOS does not support multiprogramming, hence no CPU scheduling.

A **dispatcher** is the module of the operating system that gives control of the CPU to the process selected by the CPU scheduler.

Steps –

* switching context
* switching to user mode
* jumping to the proper location in the user program

**Dispatch latency** is the time taken to stop a process and start another. Dispatch latency is a pure overhead.

Scheduling criteria –

|  |  |  |
| --- | --- | --- |
| - | ↑ **CPU utilization** | - percentage |
| - | ↑ **Throughput** | - number of processes completed per unit time |
| - | ↓ **Turnaround time** | - time from submission to completion (time spent in different queues + time spent in CPU + time spent in different i/o devices) |
| - | ↓ **Waiting time** | - time spent in ready queue (only) |
| - | ↓ **Response time** | - time from submission to first response |

Variance in response time should be minimal.

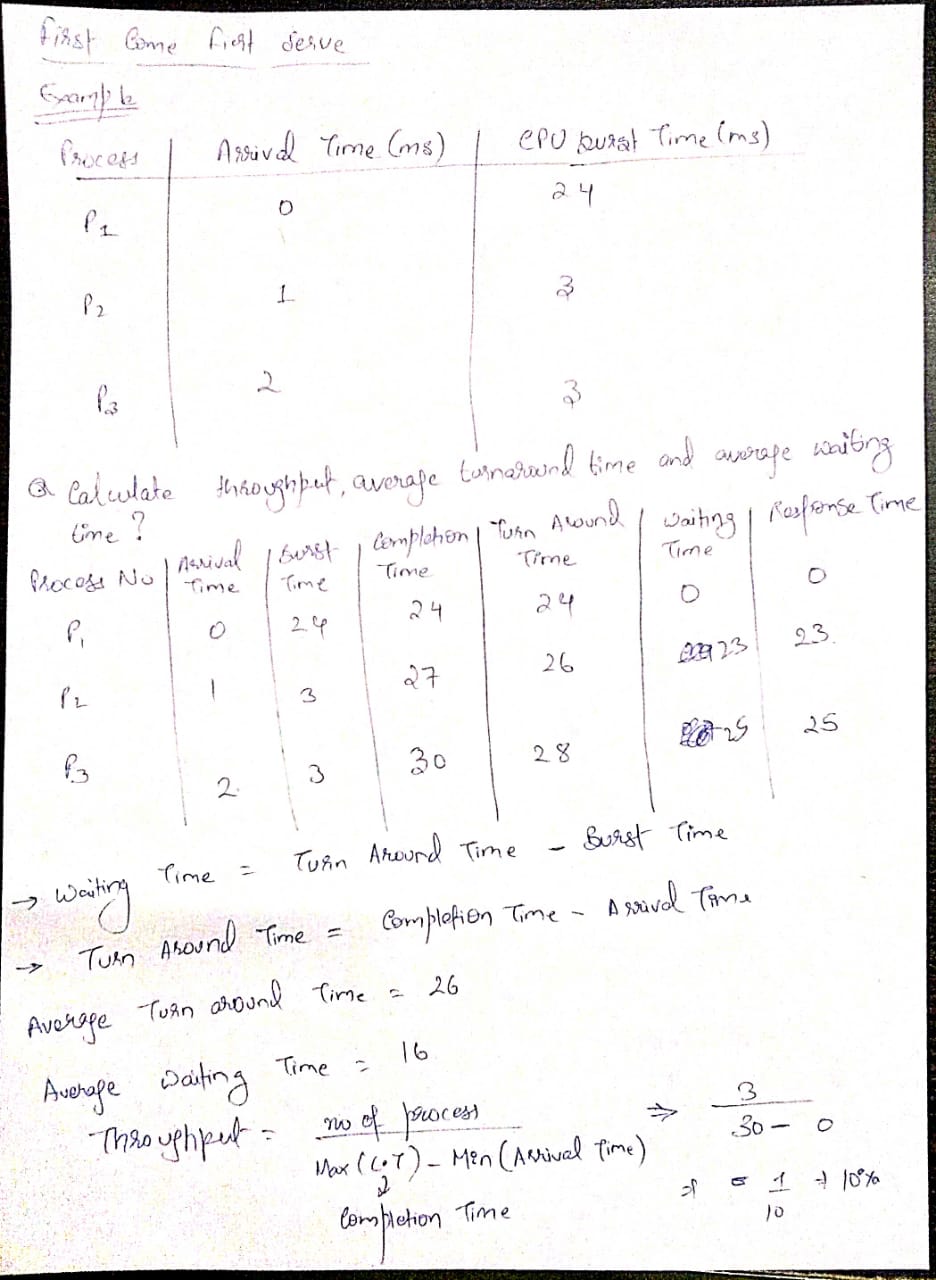
**First-come first-served (FCFS) scheduling** – - non-preemptive

* high average waiting time
* **convoy effect** - several small processes may need to wait if a large process is given the CPU

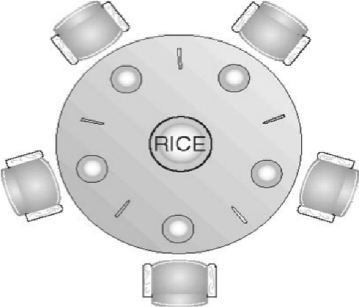
***Exercise 1.***

|  |  |  |
| --- | --- | --- |
| *Process* | *Arrival time (ms)* | *CPU burst time (ms)* |
| *P1* | *0* | *24* |
| *P2* | *1* | *3* |
| *P3* | *2* | *3* |

*Calculate throughput, average turnaround time and average waiting time.*



**Dining philosophers problem**



18

Semaphores – chopsticks[5]; all initialized to 1 Pi

do { wait (chopstick[i]); wait (chopStick[(i+1)%5]);

// eat

signal (chopstick[i]); signal (chopstick[(i+1)%5]);

// think

} while (true);

If all philosophers get hungry simultaneously then there will be a deadlock.

A **monitor** is a high-level process synchronization construct.

Only one process can be active within the monitor at a time.

A monitor type presents a set of programmer defined operations that are provided mutual exclusion within the monitor.

A monitor can have variables of the condition type that can be accessed by wait() and signal() operations only.

**Memory Management**

