Contents

Operating System as a Process Manager: 1

Mutual Exclusion 5

Job Scheduler 7

Process Scheduler 8

Operating System as a Process Manager:

We are all familiar with the modern operating system running many tasks all at once or multitasking. We can think of each process as a bundle of elements kept by the kernel to keep track of all these running tasks.

An OS *is* just a program: It has a **main()** function, which gets called only once (during boot) and like any program, it consumes resources (such as memory), and has the responsibility of executing various functions (like generating an exception), etc.

But it different to standard programs:

- It is "entered" from different locations in response to external events
- It does not have a single thread of control, it can be invoked simultaneously by two different events (e.g. system call & an interrupt)
- It is not supposed to terminate
- It can execute any instruction in the machine

After basic processes have started, the OS runs user programs, if available, otherwise enters the ***idle loop***

In the idle loop:

- OS executes an infinite loop (UNIX)
- OS performs some system management & profiling
- OS halts the processor and enter in low-power mode (notebooks)

OS wakes up on:

- Interrupts from hardware devices
- Exceptions from user programs
- System calls from user programs

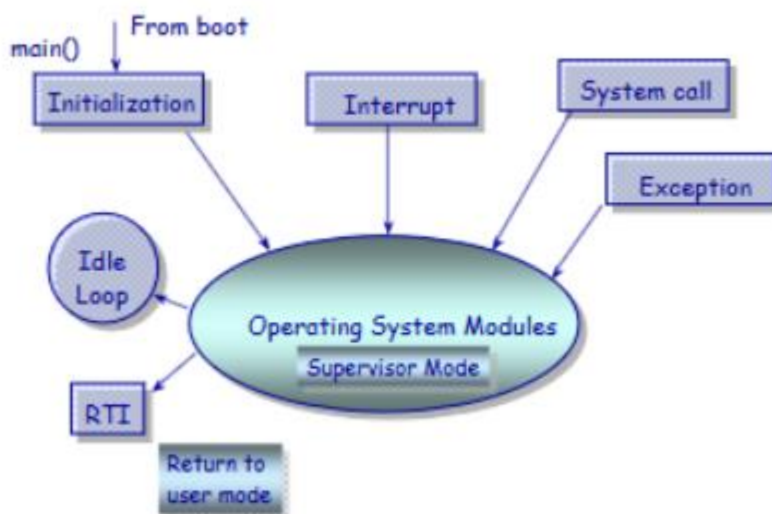

Figure 1 - Control flow of an OS

The fundamental task of any operating system (OS) is process management.

Process management describes how the OS manage the multiple processes running at a particular instance of time. For example Real-time systems require specially designed OS's. Real-time and embedded systems require an OS of minimal size and minimal resources utilisation. Wireless networks, which combine the compactness of embedded systems with issues characteristic of networked systems, have also motivated innovations in OS design.

The process manager implements the process abstraction by creating a model for the way the process uses CPU and any system resources.

Much of the complexity of the operating system stems from the need for multiple processes to share the hardware at the same time. As a consequence of this goal, major activities of an operating system in regard to process management include:

- Creation and deletion of user and system processes.
- Suspension and resumption of processes.
- Allocating resources to processes
- A mechanism for inter process communication to enable processes to share and exchange information
- Protecting the resources of each process from other processes and
- A mechanism for process synchronisation (among multiple processes).
- Interleave the execution of multiple processes to maximise processor utilisation while providing reasonable response time
- A mechanism for deadlock handling.

In addition, the process manager implements part of the operating systems

- protection and
- security.

To meet these requirements the OS must maintain a data structure for each process that describes the state and resource ownership of that process and that enables the OS to exert process control.

The OS must have control of the processor (as well as other resources) because one of its many tasks is ***scheduling the processes*** that use the CPU.

The OS relinquishes control of the CPU to various application programs during the course of their execution. The OS is dependent on the processor to regain control when the application either no longer requires the CPU or gives up the CPU as it waits for other resources.

<<recall user mode and kernel mode>>

Process management in operating systems can be classified broadly into three categories:

1. ***Multiprogramming*** involves multiple processes on a system with a single processor (the processor is used by multiple programs/processes).
2. ***Multiprocessing*** involves multiple processes on a system with multiple processors. On a multiprocessor, process execution may be interleaved and also multiple processes can be executed simultaneously.
3. ***Distributed*** processing involves multiple processes on multiple systems. Multiple CPUs must be kept busy and if scheduling is not done properly, the inherent advantages of the multiprocessor parallelism are not fully realised, and therefore performance will suffer.

The process manager needs to arrange the execution of the programs/processes (scheduling) to promote the most efficient use of resources (hardware and software) and to avoid competition and deadlock.

Both interleaved and simultaneous execution are types of concurrency and lead to a host of difficult problems, both from the application programmer and the OS.

Concurrent processing is thus central to operating systems and their design. Concurrency is the interleaving of processes in time, to give the appearance of simultaneous execution.

Difficulties of concurrency include:

- sharing global resources safely is difficult;
- optimal allocation of resources is difficult;
- locating programming errors can be difficult, because the contexts in which errors occur cannot always be reproduced easily.

For example, when one process is inside a critical section of code, other processes must be prevented from entering that section of code. This requirement is known as *mutual exclusion*.

Mutual Exclusion

Mutual exclusion is in many ways the fundamental issue in concurrency. It is the requirement that when a process *P* is accessing a shared resource *R*, no other process should be able to access *R* until *P* has finished with *R*. Examples of such resources include files, I/O devices such as printers, and shared data structures

Process management includes the introduction of the concept of a thread.

Recall:

i. A ***program*** is a non-active, passive collection of instructions stored on disk
ii. A program becomes a ***job*** from the moment it is selected for execution until it has finished running and becomes a program again.
iii. A ***process*** is a program in execution. It is a program that has started but has not finished. A process has one or more threads, along with their execution state. A process is the actual execution of program instructions
iv. ***Thread***: Executes a series of instructions in order (only one thing happens at a time). A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. A Thread is a portion of a process that can be run independently. Modern systems allow a single process to have multiple threads of execution, which execute concurrently

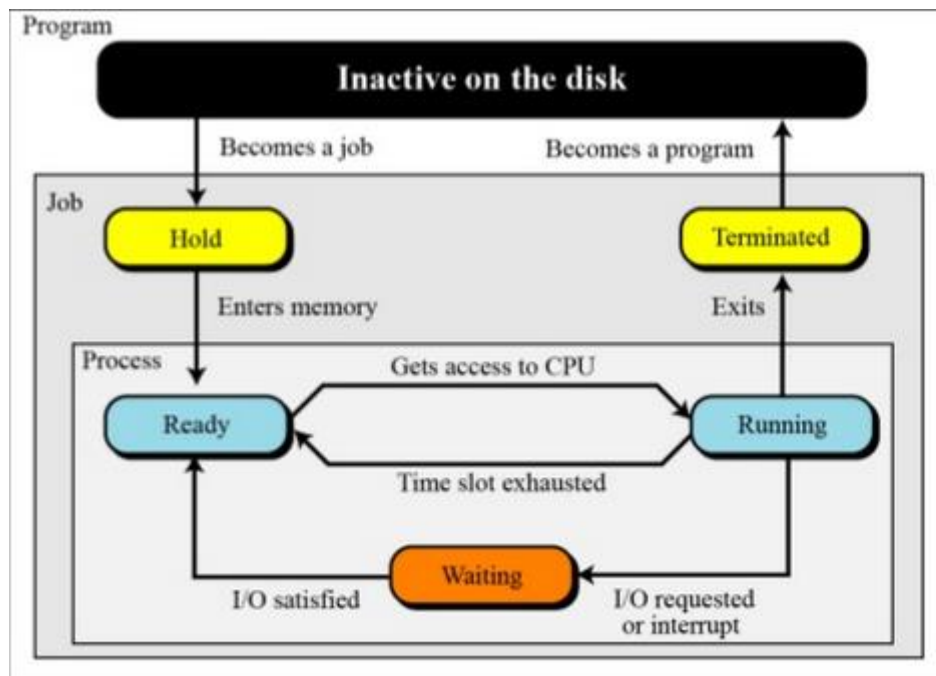

Figure 2: Program, Job and Process State Diagram

Almost all computers today can execute multiple threads simultaneously:

- Each processor chip typically contains multiple *cores*
- Each core contains a complete CPU capable of executing threads

Many modern processors support *hyperthreading*: each physical core behaves as if it is actually two cores, so it can run two threads simultaneously (e.g. execute one thread while the other is waiting on a cache miss).

> For example, a server might contain 2 Intel processor chips, each with 12 cores, where each core supports 2-way hyperthreading. Overall, this server can run 48 threads simultaneously.

The OS may have more threads than cores

At any given time, most threads do not need to execute (they are waiting for something).

A process in execution needs resources like processing resource, memory and IO resources. Current machines allow several processes to share resources although in reality one processor is shared amongst many processes: the owner of every process gets an illusion that the server (read processor) is available to their process without any interruption

To move a job or process from one state to another, the process manager uses two schedulers:

1. the **job scheduler** and

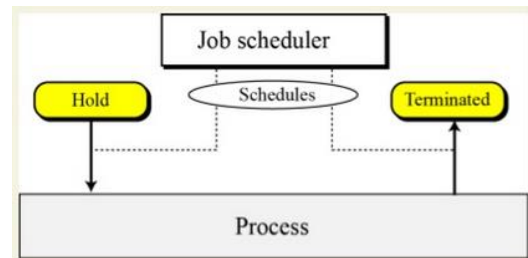

Figure 3: Job Scheduler

2. the **process scheduler**

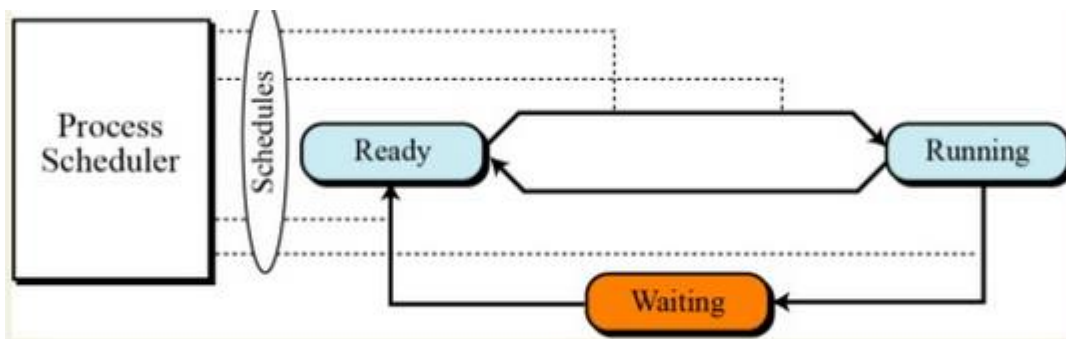

Figure 4: Process Scheduler

Job Scheduler

When several jobs are ready are ready to be brought into memory, and there is not enough room for all of them, then the decision made by the system to choose among them can be termed as job scheduling. This can also be termed as a ***long term scheduler***.

> It is typical of a batch system or a very heavily loaded system. It runs infrequently, and can afford to take the time to implement intelligent and advanced scheduling algorithms.

Process Scheduler

The objective of multiprogramming is to have some process running at all times to maximise CPU utilization. This could be termed as ***short term scheduler*** or CPU scheduler. It must very quickly swap one process out of the CPU and swap in another one.

Our state diagram shows *one job* or *one process* moving from one state to another. In reality, there are many jobs and many processes ***competing*** with each other for computer resources. To handle multiple processes and jobs, the process manager uses queues (waiting lists).

A ***job control block*** or ***process control block*** is associated with each job or process. This is a block of memory that stores information about that job or process.