A LECTURE NOTE ON

CSC 205

OPERATING SYSTEM I

BY

HENRY ONYEOMA MAFUA
BRIGHT EZIEKEL CHIBUZOR

SECTION ONE

1.0 INTRODUCTION TO OPERATING SYSTEMS

1.1 DEFINITIONS OF OPERATING SYSTEMS

An **operating system** (commonly abbreviated *OS* and *O/S*) is the infrastructure software component of a computer system; it is responsible for the management and coordination of activities and the sharing of the limited resources of the computer. An operating system is the set of programs that controls a computer. The operating system acts as a host for applications that are run on the machine. As a host, one of the purposes of an operating system is to handle the details of the operation of the hardware. This relieves application programs from having to manage these details and makes it easier to write applications. Operating Systems can be viewed from two points of views: **Resource manager** and **Extended machines**. From Resource manager point of view, Operating Systems manage the different parts of the system efficiently and from extended machines point of view, Operating Systems provide a virtual machine to users that is more convenient to use.

1.2 HISTORICAL DEVELOPMENT OF OPERATING SYSTEMS

Historically operating systems have been tightly related to the computer architecture, it is good idea to study the history of operating systems from the architecture of the computers on which they run. Operating systems have evolved through a number of distinct phases or generations which corresponds roughly to the decades.

The 1940's - First Generation

The earliest electronic digital computers had no operating systems. Machines of the time were so primitive that programs were often entered one bit at a time on rows of mechanical switches (plug boards). Programming languages were unknown (not even assembly languages). Operating systems were unheard of.

The 1950's - Second Generation

By the early 1950's, the routine had improved somewhat with the introduction of punch cards. The General Motors Research Laboratories implemented the first operating systems in early 1950's for their IBM 701. The system of the 50's generally ran one job at a time. These were called single-stream batch processing systems because programs and data were submitted in groups or batches.

The 1960's - Third Generation

The systems of the 1960's were also batch processing systems, but they were able to take better advantage of the computer's resources by running several jobs at once. So operating systems designers developed the concept of multiprogramming in which several jobs are in main memory at once; a processor is switched from job to job as needed to keep several jobs advancing while keeping the peripheral devices in use.

For example, on the system with no multiprogramming, when the current job paused to wait for other I/O operation to complete, the CPU simply sat idle until the I/O finished. The solution for this problem that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU.

Another major feature in third-generation operating system was the technique called spooling (simultaneous peripheral operations on line). In spooling, a high-speed device like a disk interposed between a running program and a low-speed device involved with the program in input/output. Instead of writing directly to a printer, for example, outputs are written to the disk. Programs can run to completion faster, and other programs can be initiated sooner when the printer becomes available, the outputs may be printed.

Note that spooling technique is much like thread being spun to a spool so that it may be later be unwound as needed.

Another feature present in this generation was time-sharing technique, a variant of multiprogramming technique, in which each user has an on-line (i.e., directly connected)

terminal. Because the user is present and interacting with the computer, the computer system must respond quickly to user requests, otherwise user productivity could suffer. Timesharing systems were developed to multiprogram large number of simultaneous interactive users.

Fourth Generation

With the development of LSI (Large Scale Integration) circuits, chips, operating system entered in the personal computer and the workstation age. Microprocessor technology evolved to the point that it became possible to build desktop computers as powerful as the mainframes of the 1970s. Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the large personal computers using the Motorola 6899 CPU family.

1.3 OBJECTIVES OF OPERATING SYSTEMS

Modern Operating systems generally have following three major goals. Operating systems generally accomplish these goals by running processes in low privilege and providing service calls that invoke the operating system kernel in high-privilege state.

To hide details of hardware by creating abstraction

An abstraction occurs when software hides lower level details and provides a set of higher-level functions. An operating system transforms the physical world of devices, instructions, memory, and time into virtual world that is the result of abstractions built by the operating system. There are several reasons for abstraction.

First, the code needed to control peripheral devices is not standardized. Operating systems provide subroutines called device drivers that perform operations on behalf of programs for example, input/output operations.

Second, the operating system introduces new functions as it abstracts the hardware. For instance, operating system introduces the file abstraction so that programs do not have to deal with disks. *Third*, the operating system transforms the computer hardware into multiple virtual computers, each belonging to a different program. Each program that is running is called a process. Each

process views the hardware through the lens of abstraction.

Fourth, the operating system can enforce security through abstraction.

To allocate resources to processes (Manage resources)

An operating system controls how **processes** (the active agents) may access **resources** (passive entities).

Provide a pleasant and effective user interface

The user interacts with the operating systems through the user interface and usually interested in the “look and feel” of the operating system. The most important components of the user interface are the command interpreter, the file system, on-line help, and application integration. The recent trend has been toward increasingly integrated graphical user interfaces that encompass the activities of multiple processes on networks of computers.

1.4 HOW AN OPERATING SYSTEM WORKS

When the power of computer is turned on, the first program that runs is usually a set of instructions kept in the computer's read- only memory (ROM). This code examines the system hardware to make sure everything is functioning properly. This **power-on self test** (POST) checks the CPU, memory and basic input-output system (BIOS) for errors and stores the result in a special memory location. Once the POST has successfully completed, the software loaded in ROM (sometimes called the BIOS or **firmware**) will begin to activate the computer's disk drives. In most modern computers, when the computer activates the hard disk drive, it finds the first piece of the operating system: the **bootstrap loader**.

The bootstrap loader is a small program that has a single function: It loads the operating system into memory and allows it to begin operation. In the most basic form, the bootstrap loader sets up the small driver programs that interface with and control the various hardware subsystems of the computer. It sets up the divisions of memory that hold the operating system, user information and applications. It establishes the data structures that will hold the myriad signals, flags and semaphores that are used to communicate within and between the subsystems and applications of the computer. Then it turns control of the computer over to the operating system.

1.5 TYPES OF OPERATING SYSTEMS

Within the broad family of operating systems, there are generally four types, categorized based on the types of computers they control and the sort of applications they support. The categories are:

Real-time operating system (RTOS) - Real-time operating systems are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-interface capability, and no end-user utilities, since the system will be a "sealed box" when delivered for use. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time, every time it occurs. In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not move at all because the system is busy.

Single-user, single task operating system - As the name implies, this operating system is designed to manage the computer so that one user can effectively do one thing at a time. The Palm OS for Palm handheld computers is a good example of a modern single-user, single-task operating system.

Single-user, multi-tasking operating system - This is the type of operating system most people use on their desktop and laptop computers today. Microsoft's Windows and Apple's MacOS platforms are both examples of operating systems that will let a single user have several programs in operation at the same time. For example, it's entirely possible for a Windows user to be writing a note in a word processor while downloading a file from the Internet while printing the text of an e-mail message.

Multi-user operating system - A multi-user operating system allows many different users to take advantage of the computer's resources simultaneously. The operating system must make sure that the requirements of the various users are balanced, and that each of the programs they are using has sufficient and separate resources so that a problem with one user doesn't affect the entire community of users. Unix, VMS and mainframe operating systems, such as *MVS*, are examples of multi-user operating systems.

1.6 FUNCTIONS OF OPERATING SYSTEMS

The major functions of operating systems are;

Processor management: An operating system deals with the assignment of processor to different tasks being performed by the computer system.

Memory management: An operating system deals with the allocation of main memory and other storage areas to the system programs as well as user programs and data.

Input/output management: An operating system deals with the co-ordination and assignment of the different output and input device while one or more programs are being executed.

File management: An operating system deals with the storage of file of various storage devices to another. It also allows all files to be easily changed and modified through the use of text editors or some other files manipulation routines.

Other functions of the operating systems are;

Establishment and enforcement of a priority system: The operating system determines and maintains the order in which jobs are to be executed in the computer system.

The automatic transition from job to job as directed by special control statements. Interpretation of commands and instructions.

Coordination and assignment of compilers, assemblers, utility programs, and other software to the various user of the computer system.

Facilitates easy communication between the computer system and the computer operator (human). It also establishes data security and integrity.

OPERATING SYSTEMS CONCERN

An operating system is a computer program. Operating systems are written by human programmers who make mistakes. Therefore, there can be errors in the code even though there may be some testing before the product is released. Some companies have better software

quality control and testing than others so one may notice varying levels of quality from operating system to operating system. Errors in operating systems cause three main types of problems:

System crashes and instabilities - These can happen due to a software bug typically in the operating system, although computer programs being run on the operating system can make the system more unstable or may even crash the system by themselves. This varies depending on the type of operating system. A system crash is the act of a system freezing and becoming unresponsive which would cause the user to need to reboot.

Security flaws - Some software errors leave a door open for the system to be broken into by unauthorized intruders. As these flaws are discovered, unauthorized intruders may try to use these to gain illegal access to your system. Patching these flaws often will help keep your computer system secure. How this is done will be explained later.

Sometimes errors in the operating system will cause the computer not to work correctly with some peripheral devices such as printers.

SECTION TWO

2.0 PROCESSES AND THREADS

2.1 PROCESSES

There is no universally agreed upon definition, however, some of the definitions of a process include;

A program in Execution.

An asynchronous activity.

The 'animated sprit' of a procedure in execution. The entity to which processors are assigned.

The 'dispatchable' unit.

The definition "*Program in Execution"* seems to be most frequently used.

A process is the unit of work in a system. In Process model, all software on the computer is organized into a number of sequential processes. A process includes PC, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, the CPU switches back and forth among processes. (The rapid switching back and forth is called multiprogramming).

2.1.1 PROCESS STATES

A process goes through a series of discrete process states.

New State The process being created.

Terminated State The process has finished execution.

Blocked (waiting) State When a process blocks, it does so because logically it cannot continue, typically because it is waiting for input that is not yet available. Formally, a process is said to be blocked if it is waiting for some event to happen (such as an I/O completion) before it can proceed. In this state a process is unable to run until some external event happens.

Running State A process is said to be running if it currently has the CPU, that is, actually using the CPU at that particular instant.

Ready State A process is said to be ready if it use a CPU if one were available. It is runable but temporarily stopped to let another process run.

Logically, the 'Running' and 'Ready' states are similar. In both cases the process is willing to run, only in the case of 'Ready' state, there is temporarily no CPU available for it. The 'Blocked' state is different from the 'Running' and 'Ready' states in that the process cannot run, even if the CPU is available.

2.1.2 PROCESS STATE TRANSITION

The following are six (6) possible transitions among above mentioned five (5) states

Transition 1 occurs when process discovers that it cannot continue. If running process initiates an I/O operation before its allotted time expires, the running process voluntarily relinquishes the CPU.

This state transition is:

Block (process-name): Running → Block.

Transition 2 occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.

This state transition is:

Time-Run-Out (process-name): Running → Ready.

Transition 3 occurs when all other processes have had their share and it is time for the first process to run again

This state transition is:

Dispatch (process-name): Ready → Running.

Transition 4 occurs when the external event for which a process was waiting (such as arrival of input) happens.

This state transition is:

Wakeup (process-name): Blocked → Ready.

Transition 5 occurs when the process is created.

This state transition is:

Admitted (process-name): New → Ready.

Transition 6 occurs when the process has finished execution.

This state transition is:

Exit (process-name): Running → Terminated.

2.1.3 PROCESSING MODES

There are various processing modes, two of which are majorly connected with the study of operating systems, they are;

Batch Processing: The earliest computers were extremely expensive devices, and very slow. Machines were typically dedicated to a particular set of tasks and operated by control panel, the operator manually entering small programs via switches in order to load and run other programs. These programs might take hours, even weeks, to run. As computers grew in speed, run times dropped, and suddenly the time taken to start up the next program became a concern. The batch processing methodologies evolved to decrease these dead times, cuing up programs so as soon as one completed the next would start.

To support a batch processing operation, a number of card punch or paper tape writers would be used by programmers, who would use these inexpensive machines to write their programs "offline". When they completed typing them, they were submitted to the operations team, who would schedule them for running. Important programs would be run quickly, less important ones were unpredictable. When the program was finally run, the output, generally printed, would be

returned to the programmer. The complete process might take days, during which the programmer might never see the computer.

The alternative, allowing the user to operate the computer directly, was generally far too expensive to consider. This was because the user had long delays where they were simply sitting there entering code. This limited developments in direct interactivity to organizations that could afford to waste computing cycles, large universities for the most part. Programmers at the universities decried the inhumanist behaviors that batch processing imposed, to the point that Stanford students made a short film humorously critiquing it. They experimented with new ways to directly interact with the computer, a field today known as human machine interaction.

Benefits of Batch Processing

It allows sharing of computer resources among many users and programs,
It shifts the time of job processing to when the computing resources are less busy,
It avoids idling the computing resources with minute-by-minute human interaction and supervision,
By keeping high overall rate of utilization, it better amortizes the cost of a computer, especially an expensive one.

Time Sharing: Time-sharing developed out of the realization that while any single user was inefficient, a large group of users together were not. This was due to the pattern of interaction; in most cases users entered bursts of information followed by long pause, but a group of users working at the same time would mean that the pauses of one user would be used up by the activity of the others. Given an optimal group size, the overall process could be very efficient. Similarly, small slices of time spent waiting for disk, tape, or network input could be granted to other users.

Implementing a system able to take advantage of this would be difficult. Batch processing was really a methodological development on top of the earliest systems; computers still ran single programs for single users at any time, all that batch processing changed was the time delay between one program and the next. Developing a system that supported multiple users at the

same time was a completely different concept, the "state" of each user and their programs would

have to be kept in the machine, and then switch between them quickly. This would take up computer cycles, and on the slow machines of the era this was a concern. However, as computers rapidly improved in speed, and especially size of core memory to keep the state, the overhead of time-sharing continually reduced in overall terms.

2.2.0 THREADS

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*. In a process, threads allow multiple executions of streams. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or Terminated).

The following are some reasons why threads are used in the design of operating systems.

1. A process with multiple threads makes a great server for example printer server.
2. Because threads can share common data, they do not need to use interprocess communication.
3. Because of the very nature, threads can take advantage of multiprocessors.

2.2.1 PROCESSES VS. THREADS

In many respect threads operate in the same way as that of processes. Some of the similarities and differences are:

Similarities

Like processes threads share CPU and only one thread active (running) at a time.

Like processes, threads within a process executes sequentially.

Like processes, thread can create children.

And like process, if one thread is blocked, another thread can run.

Differences

Unlike processes, threads are not independent of one another.

Unlike processes, all threads can access every address in the task.

Unlike processes, threads are designed to assist one other. Processes might or might not assist one another because processes may originate from different users.

SECTION THREE

3.0 PROCESS SCHEDULING AND SCHEDULING ALGORITHM

3.1 PROCESS SCHEDULING

The assignment of physical processors to processes allows processors to accomplish work. The problem of determining when processors should be assigned and to which processes is called processor scheduling or CPU scheduling. When more than one process is runable, the operating system must decide which one first. The part of the operating system concerned with this decision is called the scheduler, and algorithm it uses is called the scheduling algorithm.

3.1.1 GOALS OF SCHEDULING

Many objectives must be considered in the design of a scheduling discipline. In particular, a scheduler should consider fairness, efficiency, response time, turnaround time, throughput, etc., Some of these goals depends on the system one is using for example batch system, interactive system or real-time system, etc. but there are also some goals that are desirable in all systems.

General Goals

Fairness: Fairness is important under all circumstances. A scheduler makes sure that each process gets its fair share of the CPU and no process can suffer indefinite postponement. Note that giving equivalent or equal time is not fair. Think of *safety control* and *payroll* at a nuclear plant.

Policy Enforcement: The scheduler has to make sure that system's policy is enforced. For example, if the local policy is safety then the *safety control processes* must be able to run whenever they want to, even if it means delay in *payroll processes*.

Efficiency: Scheduler should keep the system (or in particular CPU) busy cent percent of the time when possible. If the CPU and all the Input/output devices can be kept running all the time, more work gets done per second than if some components are idle.

Response Time: A scheduler should minimize the response time for interactive user.

Turnaround: A scheduler should minimize the time batch users must wait for an output.

Throughput: A scheduler should maximize the number of jobs processed per unit time.

3.1.2 Preemptive Vs Nonpreemptive Scheduling

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Nonpreemptive Scheduling

A scheduling discipline is nonpreemptive if, once a process has been given the CPU, the CPU cannot be taken away from that process.

Following are some characteristics of nonpreemptive scheduling

1. In nonpreemptive system, short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.
2. In nonpreemptive system, response times are more predictable because incoming high priority jobs can not displace waiting jobs.
3. In nonpreemptive scheduling, a schedular executes jobs in the following two situations.
    a. When a process switches from running state to the waiting state.
    b. When a process terminates.

Preemptive Scheduling

A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.

The strategy of allowing processes that are logically runable to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

3.2 SCHEDULING ALGORITHMS

CPU Scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU.

Following are some scheduling algorithms we will study

FCFS Scheduling.

SRT Scheduling.

SJF Scheduling.

Round Robin Scheduling.

Multilevel Feedback Queue Scheduling.

3.2.1 First-Come-First-Served (FCFS) SCHEDULING

Other names of this algorithm are:

First-In-First-Out (FIFO)

Run-to-Completion

Run-Until-Done

Perhaps, First-Come-First-Served algorithm is the simplest scheduling algorithm is the simplest scheduling algorithm. Processes are dispatched according to their arrival time on the ready queue. Being a nonpreemptive discipline, once a process has a CPU, it runs to completion. The FCFS scheduling is fair in the formal sense or human sense of fairness but it is unfair in the sense that long jobs make short jobs wait and unimportant jobs make important jobs wait.

FCFS is more predictable than most of other schemes since it offers time. FCFS scheme is not useful in scheduling interactive users because it cannot guarantee good response time. The code for FCFS scheduling is simple to write and understand. One of the major drawback of this scheme is that the average time is often quite long.

The First-Come-First-Served algorithm is rarely used as a master scheme in modern operating systems but it is often embedded within other schemes.

3.2.2 SHORTEST-REMAINING-TIME (SRT) SCHEDULING

The SRT is the preemtive counterpart of SJF and useful in time-sharing environment.

In SRT scheduling, the process with the smallest estimated run-time to completion is run next, including new arrivals.

In SJF scheme, once a job begin executing, it run to completion.

In SJF scheme, a running process may be preempted by a new arrival process with shortest estimated run-time.

The algorithm SRT has higher overhead than its counterpart SJF.

The SRT must keep track of the elapsed time of the running process and must handle occasional preemptions.

In this scheme, arrival of small processes will run almost immediately. However, longer jobs have even longer mean waiting time.

3.2.3 SHORTEST-JOB-FIRST (SJF) SCHEDULING

Another name of this algorithm is Shortest-Process-Next (SPN).

Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run-time-to-completion is run next. In other words, when CPU is available, it is assigned to the process that has smallest next CPU burst.

The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. Since the SJF scheduling algorithm gives the minimum average time for a given set of processes, it is probably optimal.

The SJF algorithm favors short jobs (or processors) at the expense of longer ones.

The obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run, and this information is not usually available.

The best SJF algorithm can do is to rely on user estimates of run times.

Like FCFS, SJF is non preemptive therefore, it is not useful in timesharing environment in which reasonable response time must be guaranteed.

3.2.4 ROUND ROBIN SCHEDULING

One of the oldest, simplest, fairest and most widely used algorithm is round robin (RR).

In the round robin scheduling, processes are dispatched in a FIFO manner but are given a limited amount of CPU time called a time-slice or a quantum.

If a process does not complete before its CPU-time expires, the CPU is preempted and given to the next process waiting in a queue. The preempted process is then placed at the back of the ready list.

Round Robin Scheduling is preemptive (at the end of time-slice) therefore it is effective in time-sharing environments in which the system needs to guarantee reasonable response times for interactive users.

The only interesting issue with round robin scheme is the length of the quantum. Setting the quantum too short causes too many context switches and lower the CPU efficiency. On the other hand, setting the quantum too long may cause poor response time and approximates FCFS.

In any event, the average waiting time under round robin scheduling is often quite long.

3.2.5 Multilevel Feedback Queue Scheduling

Multilevel feedback queue-scheduling algorithm allows a process to move between queues. It uses many ready queues and associates a different priority with each queue.

The Algorithm chooses to process with highest priority from the occupied queue and run that process either preemptively or nonpreemptively. If the process uses too much CPU time it will moved to a lower-priority queue. Similarly, a process that wait too long in the lower-priority

queue may be moved to a higher-priority queue may be moved to a highest-priority queue. Note that this form of aging prevents starvation.

Process Management

An operating system executes a variety of programs: o Batch system – jobs

o Time-shared systems – user programs or tasks

Textbook uses the terms *job* and *process* almost interchangeably

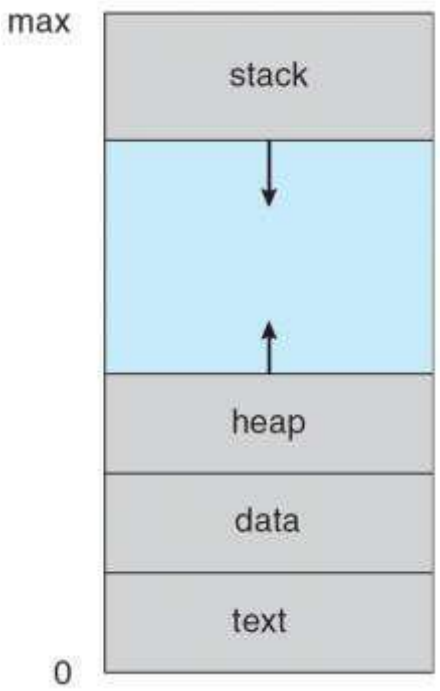
Process – a program in execution; process execution must progress in sequential fashion

A process includes:

program counter

stack

data section



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

ready: The process is waiting to be assigned to a processor

terminated: The process has finished execution

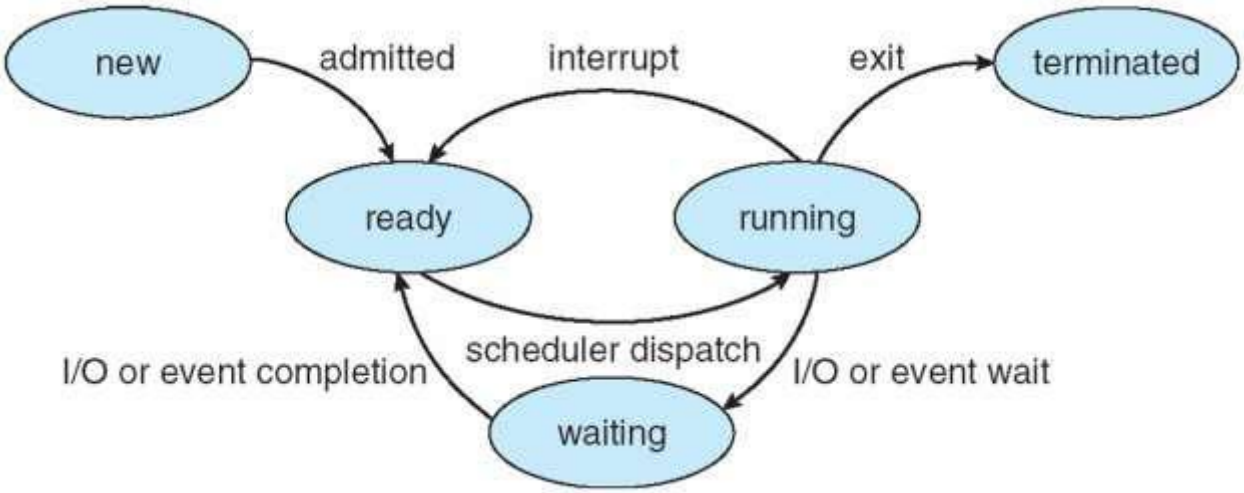


Fig: Process Transition Diagram

PCB: Process Control Block

Information associated with each process

Process state

Program counter

CPU registers

CPU scheduling information

Memory-management information

Accounting information

I/O status information

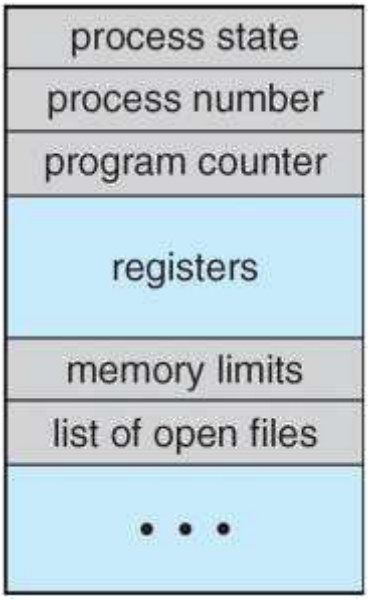


Fig: PCB

Context Switching

When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch

Context of a process represented in the PCB

Context-switch time is overhead; the system does no useful work while switching

Time dependent on hardware support

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

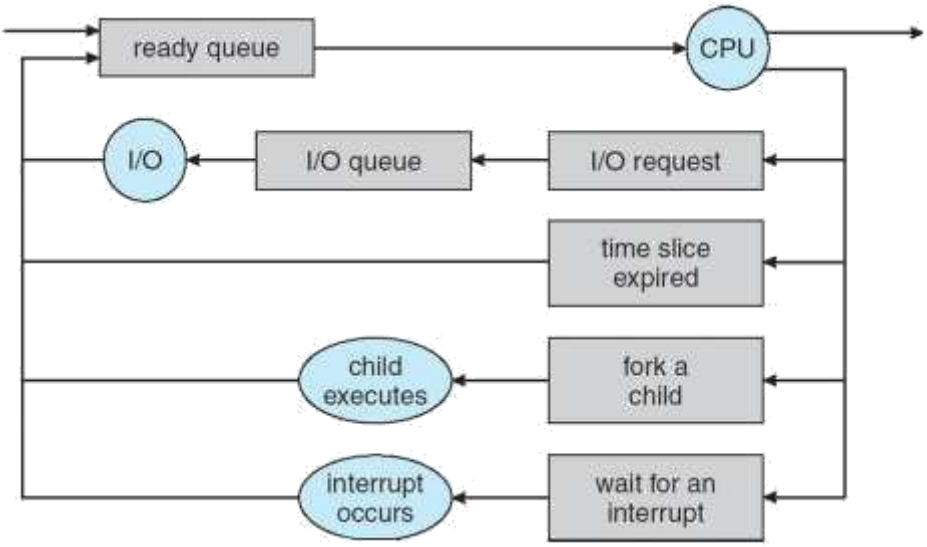
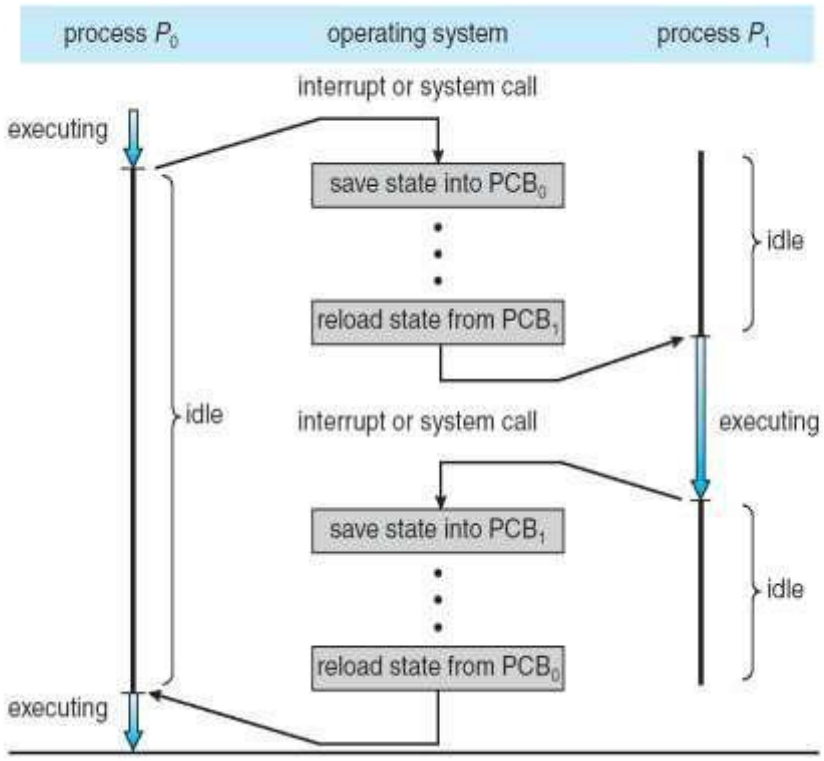


Fig: Process Scheduling



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

Short-term scheduler is invoked very frequently (milliseconds) □ (must be fast)

Long-term scheduler is invoked very infrequently (seconds, minutes) □ (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

Process Creation

Parent process create children processes, which, in turn create other processes, forming a tree of processes

Generally, process identified and managed via a process identifier (pid)

Resource sharing

Parent and children share all resources

Children share subset of parent's resources

Parent and child share no resources

Execution

Parent and children execute concurrently

Parent waits until children terminate

Address space

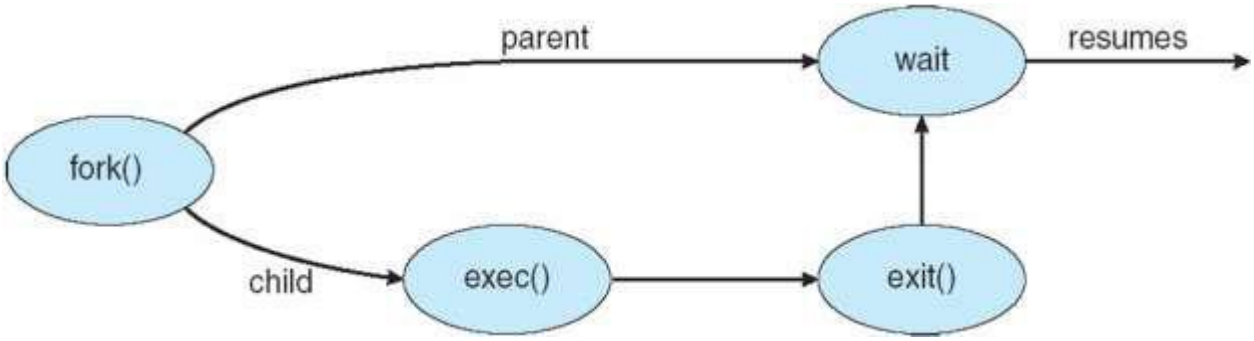
Child duplicate of parent

Child has a program loaded into it

UNIX examples

fork system call creates new process

exec system call used after a fork to replace the process' memory space with a new program



Process Termination

Process executes last statement and asks the operating system to delete it (exit) o Output data from child to parent (via wait)

o Process' resources are deallocated by operating system

Parent may terminate execution of children processes (abort)

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

Inter Process Communication

Processes within a system may be independent or cooperating

Cooperating process can affect or be affected by other processes, including sharing data

Reasons for cooperating processes:

Information sharing

Computation speedup

Modularity

Convenience

Cooperating processes need interprocess communication (IPC)

Two models of IPC

Shared memory

Message passing

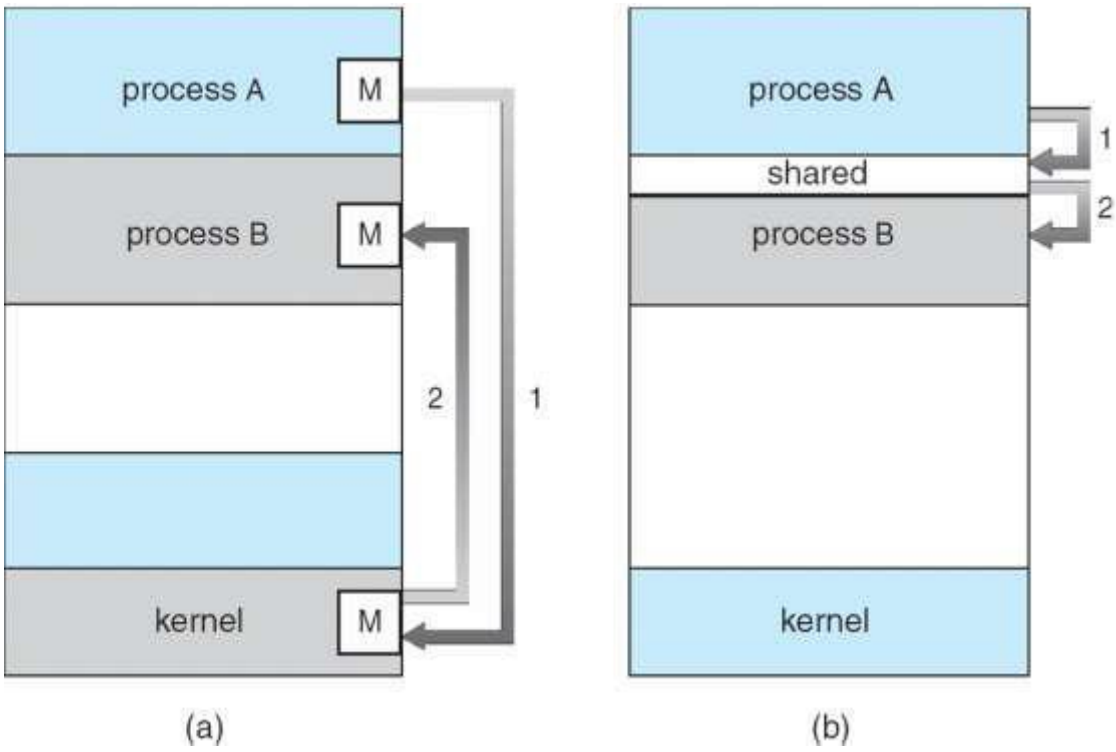


Fig:a- Message Passing, b- Shared Memory

Cooperating Process

Independent process cannot affect or be affected by the execution of another process

Cooperating process can affect or be affected by the execution of another process

Advantages of process cooperation

Information sharing

Computation speed-up

Modularity

Convenience

Producer Consumer Problem

Paradigm for cooperating processes, *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

```
while (true) {

/* Produce an item */

while (((in = (in + 1) % BUFFER SIZE count) == out)

        /* do nothing -- no free buffers */ buffer[in] = item;

in = (in + 1) % BUFFER SIZE;

}
```

Fig: Producer Process

```
while (true) {

while (in == out)

        // do nothing -- nothing to consume

        remove an item from the buffer

item = buffer[out];
```

Fig: Consumer Process

out = (out + 1) % BUFFER SIZE;

return item;

IPC-Message Passing

Mechanism for processes to communicate and to synchronize their actions

Message system – processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

send(*message*) – message size 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

Implementation of communication link

physical (e.g., shared memory, hardware bus)

logical (e.g., logical properties)

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

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

Indirect Communication

Messages are directed and received from mailboxes (also referred to as ports) o Each mailbox has a unique id

o Processes can communicate only if they share a mailbox

Properties of communication link

Link established only if processes share a common mailbox

A link may be associated with many processes

Each pair of processes may share several communication links

Link may be unidirectional or bi-directional

Operations

- create a new mailbox
- send and receive messages through mailbox
- destroy a mailbox

Primitives are defined as:
send(*A, message*) – send a message to mailbox A

receive(*A, message*) – receive a message from mailbox A

Allow a link to be associated with at most two processes

Allow only one process at a time to execute a receive operation

Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronisation

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

Buffering

Queue of messages attached to the link; implemented in one of three ways

Zero capacity – 0 messages

Sender must wait for receiver (rendezvous)

Bounded capacity – finite length of *n* messages Sender must wait if link full

Unbounded capacity – infinite length Sender never waits

SECTION FOUR

4.0 INTERRUPTS, VIRTUAL MEMORY AND VIRTUAL MACHINE

4.1 INTERRUPTS

An **interrupt** is an asynchronous signal from hardware indicating the need for attention or a synchronous event in software indicating the need for a change in execution. A *hardware interrupt* causes the processor to save its state of execution via a context switch, and begin execution of an interrupt handler. *Software interrupts* are usually implemented as instructions in the instruction set, which cause a context switch to an interrupt handler similar to a hardware interrupt. Interrupts are a commonly used technique for computer multitasking, especially in real-time computing. Such a system is said to be interrupt-driven. An act of *interrupting* is referred to as an interrupt request ("IRQ"). Interrupts were introduced as a way to avoid wasting the processor's valuable time in polling loops, waiting for external events.

An interrupt that leaves the machine in a well-defined state is called a **precise interrupt**. Such an interrupt has four properties:

The Program Counter (PC) is saved in a known place.

All instructions before the one pointed to by the PC have fully executed.

No instruction beyond the one pointed to by the PC has been executed (That is no prohibition on instruction beyond that in PC, it is just that any changes they make to registers or memory must be undone before the interrupt happens).

The execution state of the instruction pointed to by the PC is known.

An interrupt that does not meet these requirements is called an **imprecise interrupt**.

The phenomenon where the overall system performance is severely hindered by excessive amounts of processing time spent handling interrupts is called an interrupt storm.

4.1.1 INTERRUPT HANDLERS

An **interrupt handler**, also known as an **interrupt service routine** (**ISR**), is a callback subroutine in an operating system or device driver whose execution is triggered by the reception of an interrupt. Interrupt handlers have a multitude of functions, which vary based on the reason the interrupt was generated and the speed at which the Interrupt Handler completes its task.

In modern operating systems, interrupt handlers are divided into two parts: the **First-Level Interrupt Handler** (**FLIH**) and the **Second-Level Interrupt Handlers** (**SLIH**). FLIHs are also known as *hard interrupt handlers*, *fast interrupt handlers* and *top-half of interrupt*, and SLIHs are also known as *interrupt threads*, *slow interrupt handlers* and *bottom-half of interrupt*.

A FLIH implements at minimum platform-specific interrupt handling similarly to *interrupt routines*. In response to an interrupt, there is a context switch, and the code for the interrupt is loaded and executed. The job of a FLIH is to quickly service the interrupt, or to record platform-specific critical information which is only available at the time of the interrupt, and schedule the execution of a SLIH for further long-lived interrupt handling. A SLIH completes long interrupt processing tasks similarly to a process.

4.2 VIRTUAL MEMORY AND VIRTUAL MACHINE

Virtual memory is a computer system technique which gives an application program the impression that it has contiguous working memory (an address space), while in fact it may be physically fragmented and may even overflow on to disk storage. Systems that use this technique make programming of large applications easier and use real physical memory (e.g. RAM) more efficiently than those without virtual memory.

Virtual machine is a software implementation of a machine (computer) that executes programs like a real machine. An example of a virtual machine scenario is when a program written in Java receives services from the Java Runtime Environment (JRE) software by issuing commands from which the expected result is returned by the Java software. By providing these services to the program, the Java software is acting as a "virtual machine", taking the place of the operating system or hardware for which the program would ordinarily be tailored.

Virtual machines are separated into two major categories, based on their use and degree of correspondence to any real machine. A system virtual machine provides a complete system platform which supports the execution of a complete operating system (OS). In contrast, a process virtual machine is designed to run a single program, which means that it supports a single process. An essential characteristic of a virtual machine is that the software running inside is limited to the resources and abstractions provided by the virtual machine -- it cannot break out of its virtual world.