Chapter 3 : Processes

Process Concept

- ***Process,*** *which is a program in execution.*
- *The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs,it also needs to take care of various system tasks that are better left outside the kernel itself.*
- ***A batch job*** *is a computer program or set of programs processed in batch mode. This means that a sequence of commands to be executed by the operating system is listed in a file (often called a batch file, command file, or shell script) and submitted for execution as a single unit.*
- **A batch system executes jobs, whereas a time-shared system has user programs,or tasks.**
- And even if a user can execute only one program at a time, such as on an embedded device that does not support multitasking, the operating system may need to support its own internal programmed activities, such as memory management. In many respects, all these activities are similar, so we call all of them processes.
- Processes also includes the **current activity, as represented by the value of the program counter and the contents of the processor's registers. A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section, which contains global variables. Aprocess may also include a heap, which is memory that is dynamically allocated during process run time.**
- We emphasize that a program by itself is not a process. **A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an executable file). In contrast, a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources.** A program becomes a process when an executable file is loaded into memory. **Two common techniques for loading executable**
    1. **double-clicking an icon representing the executable file**
    2. **entering the name of the executable file on the command line (as in prog.exe or a.out).**

- Although two processes may be associated with the same program, they are nevertheless considered two separate execution sequences.

- **A process itself can be an execution environment for other code. The Java programming environment provides a good example.** In most circumstances, an executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions (via native machine instructions) on behalf of that code. For example, to run the compiled Java program Program.class, we would enter java Program The command java runs the JVM as an ordinary process, which in turns executes the Java program Program in the virtual machine. The concept is the same as simulation, except that the code, instead of being written for a different instruction set, is written in the Java language.

- As a process executes, it changes state. The state of a process is defined in part by the current activity of that process. **A process may be in one of the following states:**
    - **New.** The process is being created.
    - **Running.** Instructions are being executed.
    - **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
    - **Ready.** The process is waiting to be assigned to a processor.
    - **Terminated.** The process has finished execution.

- It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting, however.

PCB's

- **Each process is represented in the operating system by a process control block (PCB)—also called a task control block**

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

• **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
• **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system (Chapter 8).
• **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
• **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

- ***Since PCB contains the critical information for the process, it must be kept in an area of memory protected from normal user access. In some operating systems the PCB is placed in the beginning of the kernel stack of the process as it is a convenient protected location***

Threads

- **This single thread of control allows the process to perform only one task at a time.** The user cannot simultaneously type in characters and run the spell checker within the same process, for example.

- Most modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time. This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. **On a system that supports threads, the PCB is expanded to include information for each thread.**

PROCESS REPRESENTATION IN LINUX

- The process control block in the Linux operating system is represented by the C structure task struct, which is found in the **<linux/sched.h>** include file in the kernel source-code directory.

- **This structure contains all the necessary information for representing a process, including the state of the process, scheduling and memory-management information, list of open files, and pointers to the process's parent and a list of its children and siblings.** (A process's parent is the process that created it; its children are any processes that it creates. Its siblings are children with the same parent process.)

- **Within the Linux kernel, all active processes are represented using a doubly linked list of task struct.** The kernel maintains a pointer— current

Process Scheduling

- **The objective of multiprogramming is to have some process running at all times, to maximizeCPU utilization.**
- **The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.**
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. **For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.**
- **As processes enter the system, they are put into a job queue,** which consists of all processes in the system.
- The processes that are residing in **main memory and are ready and waiting to execute are kept on a list called the ready queue**. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.
- Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. T**he list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue**
- Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.
- A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. **Once the process is allocated the CPU and is executing, one of several events could occur:**

  • The process could issue an I/O request and then be placed in an I/O queue.
  • The process could create a new child process and wait for the child’s termination.
  • The process could be removed forcibly from the CPU,asaresult ofan interrupt, and be put back in the ready queue

- In the first two cases, the process eventually switches fromthe waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

- The selection process is carried out by the appropriate scheduler.

- Often, in a batch system,more processes are submitted than canbe executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The long-term scheduler,or job scheduler, selects processes from this pool and loads them into memory for execution.

Schedulers

- **The short-term scheduler,or CPU scheduler**, selects from among the processes that are ready to execute and allocates the CPU to one of them.

- The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then 10/(100 10) = 9 percent of the CPU is being used (wasted) simply for scheduling the work.

- **The long-term scheduler executes much less frequently**; minutes may sep- arate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in mem- ory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

- **An I/O-bound process** is one that spends more of its time doing I/O than it spends doing computations.

- **A CPU-bound process**, in contrast, generates I/O requests infrequently, using more of its time doing computations.

- **It is important that the long-term scheduler select a good process mix of I/O-bound and CPU-bound processes.** If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all

processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced

- **Time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.** The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If performance declines to unacceptable levels on a multiuser system, some users will simply quit.

- **The key idea behind a medium-term scheduler** is that sometimes it can be advantageous to remove a process from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. **The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.**

Context Switch

- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory-management information. Generically, we perform a state save of the current state of the CPU,be it in kernel or user mode, and then a state restore to resume operations

- **Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.**

- **Context-switch time is pure overhead, because the system does no useful work while switching.** Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). A typical speed is a few milliseconds.

- Some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch here simply requires changing the pointer to the current register set. Of course, if there are more

active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch

Multi-tasking in Mobile Systems

For iOS

- The foreground application is the application currently open and appearing on the display. The background application remains in memory, but does not occupy the display screen.)

- However, it is limited and only available for a limited number of application types, including applications
  - running a single, finite-length task (such as completing a download of content from a network);
  - receiving notifications of an event occurring (such as a new email message);
  - with long-running background tasks (such as an audio player.)

For Android

- **If an application requires processing while in the background, the application must use a service, a separate application component that runs on behalf of the background process.** Consider a streaming audio application: if the application moves to the background, the service continues to send audio files to the audio device driver on behalf of the background application. In fact, the service will continue to run even if the background application is suspended. Services do not have a user interface and have a small memory footprint, thus providing an efficient technique for multitasking in a mobile environment.

Operation on Processes

- **Identify processes according to a unique process identifier (or pid), which is typically an integer number.** The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel

- **The init process (which always has a pid of 1) serves as the root parent process for all user processes.** Once the system has booted, the initprocess can also create various user processes, such as a web or print server, an ssh server, and the like.

- **The kthreadd process is responsible for creating additional processes that perform tasks on behalf of the kernel**

- **The sshd process** is responsible for managing clients that connect to the system by using ssh

- **The login process** is responsible for managing clients that directly log onto the system. In this example, a client has logged on and is using the bash shell, which has been assigned pid 8416. Using the bash command-line interface, this user has created the process ps as well as the emacs editor.

Linux Command : **ps -el**

- A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes.

- For example, consider a process whose function is to display the contents of a file —say, image.jpg—on the screen of a terminal. When the process is created, it will get, as an input from its parent process, the name of the file image.jpg. Using that file name, it will open the file and write the contents out. It may also get the name of the output device. Alternatively, some operating systems pass resources to child processes. On such a system, the new process may get two open files, image.jpg and the terminal device, and may simply transfer the datum between the two.

- **When a process creates a new process, two possibilities for execution exist:**
    1. The parent continues to execute concurrently with its children.
    2. The parent waits until some or all of its children have terminated.

- **There are also two address-space possibilities for the new process:**
    - The child process is a duplicate of the parent process (it has the same program and data as the parent).
    - The child process has a new program loaded into it.

- **A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process.**

- Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

- **After a fork() system call,** one of the two processes typically **uses the exec() system call to replace the process's memory space with a new program. The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways.** The parent can then create more children; or, if it has nothing else to do while the child runs, it can issue a wait() system call to move itself off the ready queue until the termination of the child.

- **A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.** At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

- A process can cause the termination of another process via an appropriate system call . Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs. Note that a parent needs to know the identities of its children if it is to terminate them. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.

- **A parent may terminate the execution of one of its children for a variety of reasons, such as these:**
  - The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
  - The task assigned to the child is no longer required.
  - The parent is exiting, and the operating system does not allow a child to continue if its parent terminates

- **If a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system**

- A parent process may wait for the termination of a child process by using the wait() system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated:

pid t pid;
int status;
pid = wait(&status);

- **A process that has terminated, but whose parent has not yet called wait(),is known as a zombie process.** All processes transition to this state when they terminate, but generally they exist as zombies only briefly. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.
- **If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as orphans.** Linux and UNIX address this scenario by assigning the init process as the new parent to orphan processes

- **The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.**
  Forking Example :

```
        #include < sys/types.h >
#include < stdio.h >
#include < unistd.h >
int main()
{
        pid t pid;
        /* fork a child process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls","ls",NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait(NULL);
                printf("Child Complete");
        }
}
```

Inter-Proccess Communication

- A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes

executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

- There are several reasons for providing an environment that allows process cooperation:

  • **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
  • **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.
  • **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads, as we discussed in Chapter 2.
  • **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

- **Cooperating processes require an interprocess communication (IPC) mech- anism that will allow them to exchange data and information.** There are two fundamental models of interprocess communication: **shared memory and mes- sage passing.**

- **In the shared-memory model,** a region of memory that is shared by cooperating processes is established. Processes can then exchange informa- tion by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes

- **Message passing** is useful for exchanging smaller amounts of data, because no conflicts need be avoided.

- Message passing is also easier to implement in a distributed system than sharedmemory.

- Shared memory can be faster than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

- **Systems with several processing cores indicates that message passing provides better performance than shared**

memory on such systems. Shared memory suffers from cache coherency issues, which arise because shared data migrate among the several caches. As the number of processing cores on systems increases, it is possible that we will see message passing as the preferred mechanism for IPC.

- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space.
- They can then exchange information by reading and writing data in the shared areas
- **The form of the data and the location are determined by these processes and are not under the operating system's control.** The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- **A producer process produces information that is consumed by a consumer process.** For example, a compiler may produce assembly code that is consumed by an assembler. The assembler, in turn, may produce object modules that are consumed by the loader

Producer-Consumer Problem

- A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Producer :

```
item next produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER SIZE ) == out)
                /* do nothing */
buffer[in] = next produced;
in = (in + 1) % BUFFER SIZE ;
}
```

Consumer :

```
item next consumed;
while (true) {
        while (in == out)
                /* do nothing */
next consumed = buffer[out];
```

```
        out = (out + 1) % BUFFER SIZE ;
}
```

- Two types of buffers can be used. **The unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. **The bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-passing

- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, an Internet chat program could be designed so that chat participants communicate with one another by exchanging messages.

- A message-passing facility provides at least two operations: send(message) receive(message)

- If only fixed-sized messages can be sent, the system-level implementation is straight- forward. This restriction, however, makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler communication link must exist between them. This link can be implemented in a variety of ways.

- **Here are several methods for logically implementing a link and the send()/receive() operations:**
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering

- **Under direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send() and receive() primitives are defined as:
    - send(P, message)—Send a message to process P.
    - receive(Q, message)—Receive a message from process Q.

- **A communication link in this scheme has the following properties:**
    - A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other’s identity to communicate.
    - A link is associated with exactly two processes.

    - Between each pair of processes, there exists exactly one link.

- **This scheme exhibits symmetry in addressing;** that is, both the sender process and the receiver process must name the other to communicate.

- **A variant of this scheme employs asymmetry in addressing.** Here, only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send() and receive() primitives are defined as follows:
    - send(P, message)—Send a message to process P.
    - receive(id, message)—Receive a message from any process. **The variable id is set to the name of the process with which communication has taken place.**

- The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier. In general, any such hard-coding techniques, where identifiers must be explicitly stated, are less desirable than techniques involving indirection, as described next.

- **With indirect communication,** the messages are sent to and received from mailboxes,or ports. Amailbox can be viewed abstractly as an object into which messages canbe placed byprocesses and fromwhichmessages canbe removed. Each mailbox has a unique identification.

- For example, POSIX message queues use an integer value to identify a mailbox. A process can communicate with another process via a number of different mailboxes, but two processes can communicate only if they have a shared mailbox. The send() and receive() primitives are defined as follows:
    - send(A, message)—Send a message to mailbox A.
    - receive(A, message)—Receive a message from mailbox A.

- **A link is established between a pair of processes only if both members of the pair have a shared mailbox.**
    - A link may be associated with more than two processes.
    - Between each pair of communicating processes, a number of different links may exist, with each link corresponding to one mailbox.

- **Now suppose that processes P 1 , P 2 , and P 3 all share mailbox A. Process P 1 sends a message to A, while both P 2 and P 3 execute a receive() from A.Which process will receive the message sent by P 1 ? The answer depends on which of the following methods we choose:**

• Allow a link to be associated with two processes at most.
• 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 (that is, either P2 or P3, but not both, will receive the message). The system may define an algorithm for selecting which process will receive the message (for example, round robin, where processes take turns receiving messages). The system may identify the receiver to the sender or example, **round robin, where processes take turns receiving message**

- Since each mailbox has a unique owner, there can be no confusion about which process should receive a message sent to this mailbox. When a process that owns a mailbox terminates, the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

- **The operating system then must provide a mechanism that allows a process to do the following:**
  • Create a new mailbox.
  • Send and receive messages through the mailbox.
  • Delete a mailbox. The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox

- However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course, this provision could result in multiple receivers for each mailbox.

Synchronization

- **Message passing may be either blocking or nonblocking—** also **known as synchronous and asynchronous.** (Throughout this text, you will encounter the concepts of synchronous and asynchronous behavior in relation to various operating-system algorithms.)
  • Blocking send. The sending process is blocked until the message is received by the receiving process or by the mailbox.
  • Nonblocking send. The sending process sends the message and resumes operation.
  • Blocking receive. The receiver blocks until a message is available.
  • Nonblocking receive. The receiver retrieves either a valid message or a null.
  ◦
- Different combinations of send() and receive() are possible. When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver

- **The solution to the producer–consumer problem becomes trivial when we use blocking send() and receive() statements.** The producer merely invokes the blocking send() call and waits until the message is delivered to either the receiver or the mailbox. Likewise, when the consumer invokes receive(), it blocks until a message is available.

Buffering

- **Zero capacity.** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
- **Bounded capacity.** The queue has finite length n; thus, at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue (either the message is copied or a pointer to the message is kept), and the sender can continue execution without waiting. The link's capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
- **Unbounded capacity.** The queue's length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

An Example: POSIX Shared Memory

- create a shared-memory object using the shm open() system call, as follows:

  shm fd = shm open(name, O_CREAT | O_RDRW, 0666);
- The first parameter specifies the name of the shared-memory object. Processes that wish to access this shared memory must refer to the object by this name.
- The subsequent parameters specify that the shared-memory object is to be created if it does not yet exist (O CREAT) and that the object is open for reading and writing (O RDRW).
- The last parameter establishes the directory permissions of the shared-memory object. A successful call to shm open() returns an integer file descriptor for the shared-memory object.
- Once the object is established, the ftruncate() function is used to configure the size of the object in bytes. The call

  ftruncate(shm fd, 4096);

  sets the size of the object to 4,096 bytes.
- Finally, the mmap() function establishes a memory-mapped file containing the shared-memory object. It also returns a pointer to the memory-mapped file that is used for accessing the shared-memory object.
- **The flag MAP_SHARED** specifies that changes to the shared-memory object will be visible to all processes sharing the object.
- The consumer also invokes the shm unlink() function, which removes the shared-memory segment after the consumer has accessed it.

Producer :

```
#include< stdio.h >
#include< stlib.h >
#include< string.h >
#include< fcntl.h >
#include< sys/shm.h >
#include< sys/stat.h >
int main()
{
        /* the size (in bytes) of shared memory object */
        const int SIZE 4096;
        /* name of the shared memory object */
        const char *name = "OS";
        /* strings written to shared memory */
        const char *message 0 = "Hello";
        const char *message 1 = "World!";
        /* shared memory file descriptor */
        int shm fd;
        /* pointer to shared memory obect */
        void *ptr;
        /* create the shared memory object */
        shm fd = shm open(name, O CREAT | O RDRW, 0666);
        /* configure the size of the shared memory object */
        ftruncate(shm fd, SIZE);
        /* memory map the shared memory object */
        ptr = mmap(0, SIZE, PROT WRITE, MAP SHARED, shm fd, 0);
        /* write to the shared memory object */
        sprintf(ptr,"%s",message 0);
        ptr += strlen(message 0);
        sprintf(ptr,"%s",message 1);
        ptr += strlen(message 1);
        return 0;
}
```

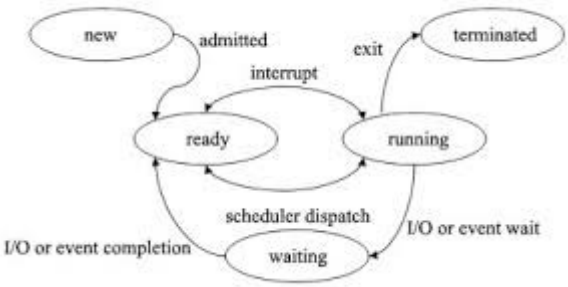
Consumer :

```
#include< stdio.h >
#include< stlib.h >
#include< fcntl.h >
#include< sys/shm.h >
#include< sys/stat.h >
int main()
{
        /* the size (in bytes) of shared memory object */
        const int SIZE 4096;
        /* name of the shared memory object */
        const char *name = "OS";
```

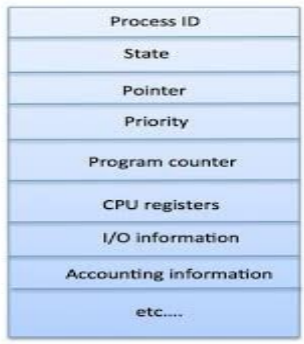
```
        /* shared memory file descriptor */
        int shm fd;
        /* pointer to shared memory obect */
        void *ptr;
        /* open the shared memory object */
        shm fd = shm open(name, O RDONLY, 0666);
        /* memory map the shared memory object */
        ptr = mmap(0, SIZE, PROT READ, MAP SHARED, shm fd, 0);
        /* read from the shared memory object */
        printf("%s",(char *)ptr);
        /* remove the shared memory object */
        shm unlink(name);
        return 0;
}
```

Diagram :

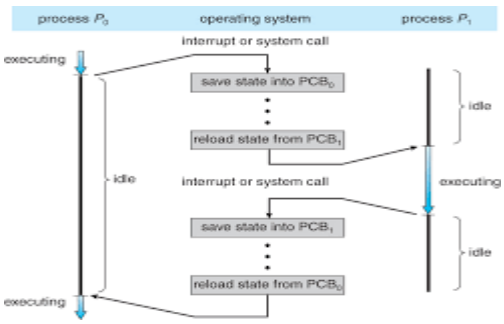
1. Process State Diagram



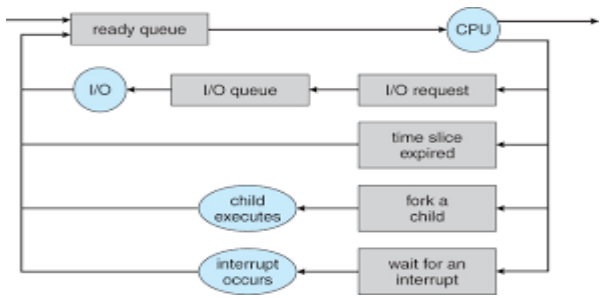
2. PCB's



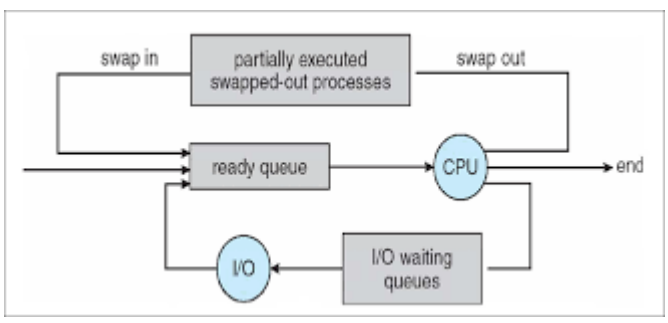
3. Context Switch



4. Queuing Diagram



5. Medium Scheduler



6. Message and Shared