**UNIT - 1**

**CHAPTER - 4**

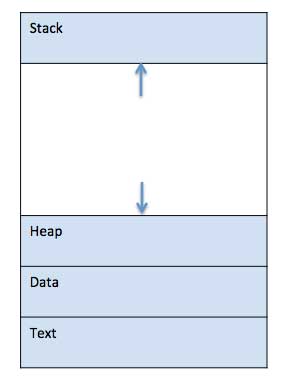
**Processes:** Process Concept, Process Scheduling, Operations on Processes, inter process Communication, IPC in shared-memory Systems, IPC in Message-passing Systems.

**Process Concepts**

1. **Process:**

* **A process is basically a program in execution.**
* The execution of a process must progress in a sequential fashion.
* A process is defined as an entity which represents the basic unit of work to be implemented in the system.
* To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.
* When a program is loaded into the memory and it becomes a process, it can be divided into four sections ─ stack, heap, text and data.
* **The following image shows a simplified layout of a process inside main memory:**

**Fig:** Process in memory.

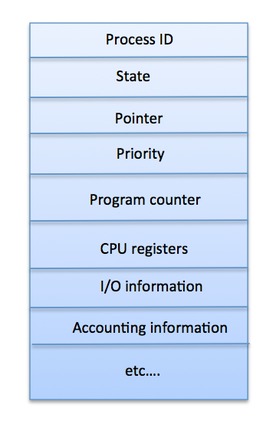


**Description:**

* **The text section** comprises the compiled program code, read in from non-volatile storage when the program is launched.
* **The data section** stores global and static variables, allocated and initialized prior to executing main.
* **The heap** is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
* **The Stack** contains the temporary data such as method/function parameters, return address and local variables.

1. **Process Control Block (PCB):**

* **A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID).**
* The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.
* The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB–



A PCB keeps all the information needed to keep track of a process as listed below in the table –

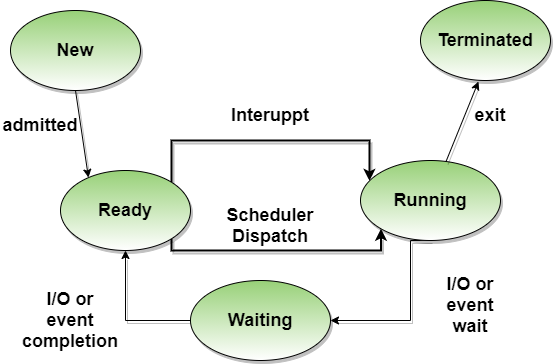
|  |  |
| --- | --- |
| **S.N.** | **Information & Description** |
| 1 | **Process State**  The current state of the process i.e., whether it is ready, running, waiting, or whatever. |
| 2 | **Process privileges**  This is required to allow/disallow access to system resources. |
| 3 | **Process ID**  Unique identification for each of the process in the operating system. |
| 4 | **Pointer**  A pointer to parent process. |
| 5 | **Program Counter**  Program Counter is a pointer to the address of the next instruction to be executed for this process. |
| 6 | **CPU registers**  Various CPU registers where process need to be stored for execution for running state. |
| 7 | **CPU Scheduling Information**  Process priority and other scheduling information which is required to schedule the process. |
| 8 | **Memory management information**  This includes the information of page table, memory limits, Segment table depending on memory used by the operating system. |
| 9 | **Accounting information**  This includes the amount of CPU used for process execution, time limits, execution ID etc. |
| 10 | **IO status information**  This includes a list of I/O devices allocated to the process. |

**Process Life Cycle Methods**

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized. In general, a process can have one of the following five states at a time.

Processes in the operating system can be in any of the following states:

* NEW- The process is being created.
* READY- The process is waiting to be assigned to a processor.
* 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).
* TERMINATED- The process has finished execution.



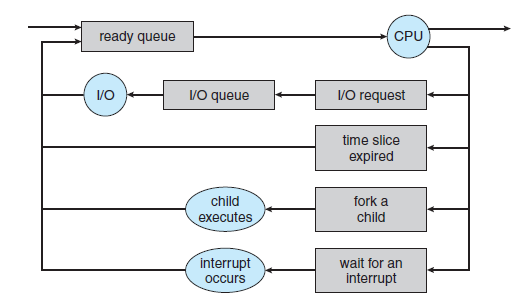
**Fig: Process state diagram**

**Process Scheduling**

* **The process scheduling** is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.
* **Process Scheduling** is an OS task that schedules processes of different states like ready, waiting, and running.
* **Process scheduling** allows OS to allocate a time interval of CPU execution for each process.
* Process scheduling is an essential part of Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

1. **Process Scheduling Queues:**

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.



**Fig: Queuing-diagram representation of process scheduling.**

* The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

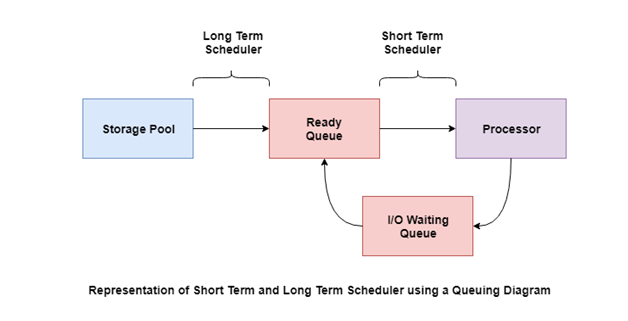
**Job queue**: This queue keeps all the processes in the system.

**Ready queue**: This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue.

**Device queues**: The processes which are blocked due to unavailability of an I/O device constitute this queue.

1. **Schedulers:**

* Schedulers are special system software which handles process scheduling in various ways.
* **Their main task is to select the jobs to be submitted into the system and to decide which process to run.**
* Schedulers are of three types:
* Long-Term Scheduler
* Short-Term Scheduler
* Medium-Term Scheduler



**Long Term Scheduler:It is also called a job scheduler.** A long-term scheduler determines which programs are admitted to the system for processing. It selects processes from the queue and loads them into memory for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound. It also controls the degree of multiprogramming. 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. On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

**Short Term Scheduler: Itis also called as CPU scheduler**. Its main objective is to increase system performance in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects a process among the processes that are ready to execute and allocates CPU to one of them. Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

**Medium Term Scheduler:Medium-term scheduling is a part of swapping**. **It is responsible for suspending and resuming the process.** It mainly does swap (moving processes from main memory to disk and vice versa). 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. It is helpful in maintaining a perfect balance between the I/O bound and the CPU bound. It reduces the degree of multiprogramming.

**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.

**(OR)**

* **A context switch** is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time. Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

****

**Operations on Processes**

**There are many operations that can be performed on processes. Some of these are process creation, process pre-emption, process blocking, and process termination.** These are given in detail as follows −

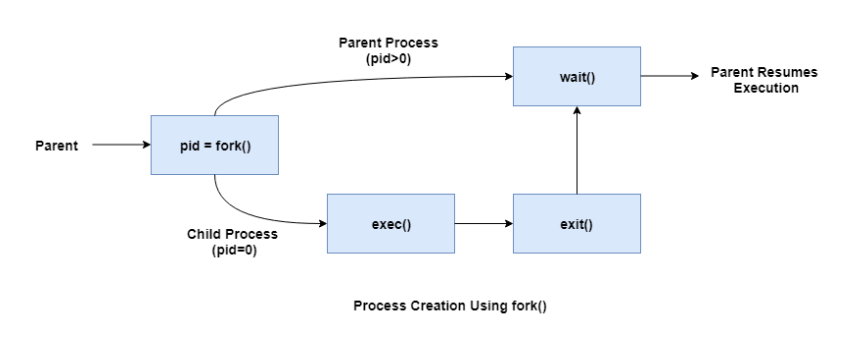
**Process Creation**

Processes need to be created in the system for different operations. This can be done by the following events −

* User request for process creation
* System initialization
* Execution of a process creation system call by a running process
* Batch job initialization

**A process may be created by another process using fork (). The creating process is called the parent process and the created process is the child process.** A child process can have only one parent but a parent process may have many children. Both the parent and child processes have the same memory image, open files, and environment strings. However, they have distinct address spaces.

A diagram that demonstrates process creation using fork() is as follows −

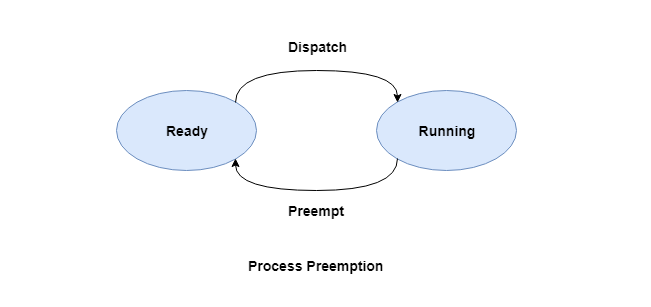


**Process Pre-emption**

**When a higher priority process becomes dispatchable, the kernel interrupts its computation and forces the context switch, pre-empting the currently running process. A process can be pre-empted at any time if the kernel finds that a higher-priority process is now dispatchable.**

An interrupt mechanism is used in pre-emption that suspends the process executing currently and the next process to execute is determined by the short-term scheduler. Pre-emption makes sure that all processes get some CPU time for execution.

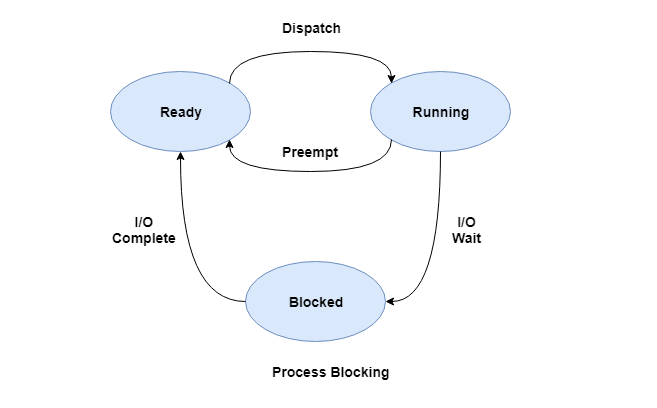
A diagram that demonstrates process pre-emption is as follows −



**Process Blocking**

**The process is blocked if it is waiting for some event to occur.** This event may be I/O as the I/O events are executed in the main memory and don't require the processor. After the event is complete, the process again goes to the ready state.

A diagram that demonstrates process blocking is as follows −



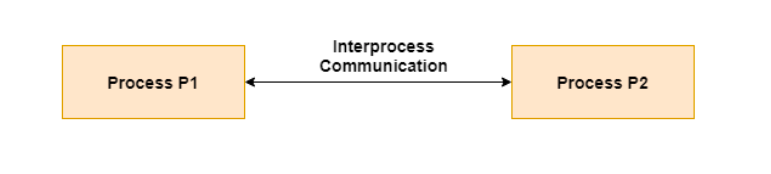
**Process Termination**

**After the process has completed the execution of its last instruction, it is terminated.** The resources held by a process are released after it is terminated.

A child process can be terminated by its parent process if its task is no longer relevant. The child process sends its status information to the parent process before it terminates. Also, when a parent process is terminated, its child processes are terminated as well as the child processes cannot run if the parent processes are terminated.

**Interprocess Communication**

* **Interprocess communication (IPC)** is the mechanism provided by the operating system that allows processes to communicate with each other. This communication could involve a process letting another process know that some event has occurred or the transferring of data from one process to another.
* A diagram that illustrates interprocess communication is as follows −



* Inter process communication (IPC) is a set of programming [interface](https://whatis.techtarget.com/definition/interface)s that allow a programmer to coordinate activities among different program [process](https://whatis.techtarget.com/definition/process)es that can run concurrently in an operating system. This allows a program to handle many user requests at the same time. Since even a single user request may result in multiple processes running in the operating system on the user's behalf, the processes need to communicate with each other. The IPC interfaces make this possible. Each IPC method has its own advantages and limitations so it is not unusual for a single program to use all of the IPC methods.
* Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
* **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 or Dependent** 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.

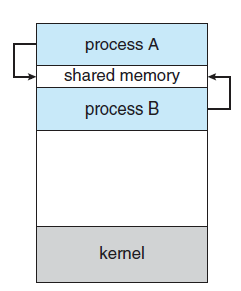
**Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing cores.

**Modularity**. We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.

**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 inter process communication (IPC) mechanism that will allow them to exchange data and information.**
* **There are two fundamental models of inter process communication: shared memory and message passing.**

1. **Shared memory:**

****

* **Inter process communication using shared memory requires communicating processes to establish a region of shared memory.**
* 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.
* Recall that, normally, the operating system tries to prevent one process from accessing another process’s memory. Shared memory requires that two or more processes agree to remove this restriction.
* 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.
* **Example :**To illustrate the concept of cooperating processes, let’s consider the producer–consumer problem, which is a common paradigm for cooperating processes.
* **Producer–consumer problem:** 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.
* **The following variables reside in a region of memory shared by the producer and consumer processes:**

#define BUFFER SIZE 10

Typedefstruct

{

. . .

}item;

item buffer[BUFFER SIZE];

int in = 0;

int out = 0;

* **The producer process using shared memory.**

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;

*}*

* **The consumer process using shared memory.**

item next consumed;

while (true)

{

while (in == out);

/\* do nothing \*/

next consumed = buffer[out];

out = (out + 1) % BUFFER SIZE;

/\* consume the item in next consumed \*/

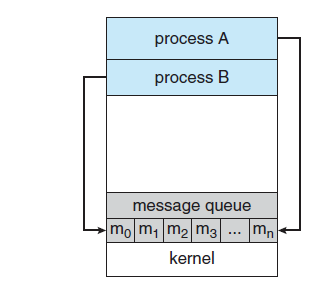
}

* **One solution to the producer–consumer problem uses shared memory.**
* To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
* This buffer will reside in a region of memory that is shared by the producer and consumer processes.
* 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.
* **Here 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.

1. **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)**

* Messages sent by a process can be either fixed or variable in size.
* If only fixed-sized messages can be sent, the system-level implementation is straightforward. 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.
* If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways. We are concerned here not with the link’s physical implementation (such as shared memory, hardware bus, or network. but rather with its logical implementation.
* **Here are several methods for logically implementing a link and the send()/receive() operations:**
* (a) Direct or indirect communication
* (b) Synchronous or asynchronous communication
* (c ) Automatic or explicit buffering
* We look at issues related to each of these features next.

1. **Direct or indirect communication**

* **Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.**
* **In direct communication or Symmetry scheme,** each process that wants to communicate must explicitly name the recipient or sender of the communication.
* **Here 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.

* **In Indirect communication or Asymmetry scheme**, indirect communication, the messages are sent to and received from mailboxes, or ports.
* **Here 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.

* In this scheme, a communication link has the following properties:
  + A communication 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.

**Example:**

* Now suppose that processes P1, P2, and P3 all share mailbox A. ProcessP1 sends a message to A, while both P2 and P3 execute a receive() from A. Which process will receive the message sent by P1?
* 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.

1. **Synchronization**

* Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive.
* **Message passing may be either blocking or non blocking—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.

• **Non blocking send**: The sending process sends the message and resumes operation.

• **Blocking receive**: The receiver blocks until a message is available.

• **Non blocking receive**: The receiver retrieves either a valid message or a null.

message next produced;

(c) **Buffering**

* Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue.
* **Basically, such queues can be implemented in three ways:**

**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 message scan 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.

**Examples of IPC**

* The following are the examples of IPC :
* POSIX Shared Memory
* Mach
* Windows
* Sockets
* Remote Procedure Calls
* Pipes