

## Module 2

### 1.24 Process Concept

- A process is the unit-of-work.
- A system consists of a collection of processes:
  - 1) **OS process** can execute system-code and
  - 2) **User process** can execute user-code.

#### 1.24.1 The Process

- A process is a program in execution.
- It also includes (Figure 1.23):
  - 1) **Program Counter** to indicate the current activity.
  - 2) **Registers Content** of the processor.
  - 3) **Process Stack** contains temporary data.
  - 4) **Data Section** contains global variables.
  - 5) **Heap** is memory that is dynamically allocated during process run time.
- A program by itself is not a process.
  - 1) A process is an active-entity.
  - 2) A program is a passive-entity such as an executable-file stored on disk.
- A program becomes a process when an executable-file is loaded into memory.
- If you run many copies of a program, each is a separate process.

The text-sections are equivalent, but the data-sections vary.

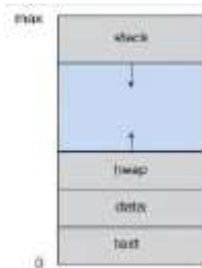


Figure 1.23 Process in memory

#### 1.24.2 Process State

- As a process executes, it changes state.
- Each process may be in one of the following states (Figure 1.24):
  - 1) **New**: The process is being created.
  - 2) **Running**: Instructions are being executed.
  - 3) **Waiting**: The process is waiting for some event to occur (such as I/O completions).
  - 4) **Ready**: The process is waiting to be assigned to a processor.
  - 5) **Terminated**: The process has finished execution.
- Only one process can be running on any processor at any instant.



Figure 1.24 Diagram of process state

## 1.25 Process Scheduling

- Objective of multiprogramming:

To have some process running at all times to maximize CPU utilization.

- Objective of time-sharing:

To switch the CPU between processes so frequently that users can interact with each program while it is running.

- To meet above 2 objectives: **Process scheduler** is used to select an available process for program-execution on the CPU.

### 1.25.1 Scheduling Queues

- Three types of scheduling-queues:

- 1) **Job Queue**

- This consists of all processes in the system.

- As processes enter the system, they are put into a job-queue.

- 2) **Ready Queue**

- This consists of the processes that are

- residing in main-memory and

- ready & waiting to execute (Figure 1.26).

- 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 has a pointer to the next PCB in the ready-queue.

- 3) **Device Queue**

- This consists of the processes that are waiting for an I/O device.

- Each device has its own device-queue.

- When the process is executing, one of following events could occur (Figure 1.27):

- 1) The process could issue an I/O request and then be placed in an I/O queue.

- 2) The process could create a new subprocess and wait for the subprocess's termination.

- 3) The process could be interrupted and put back in the ready-queue.

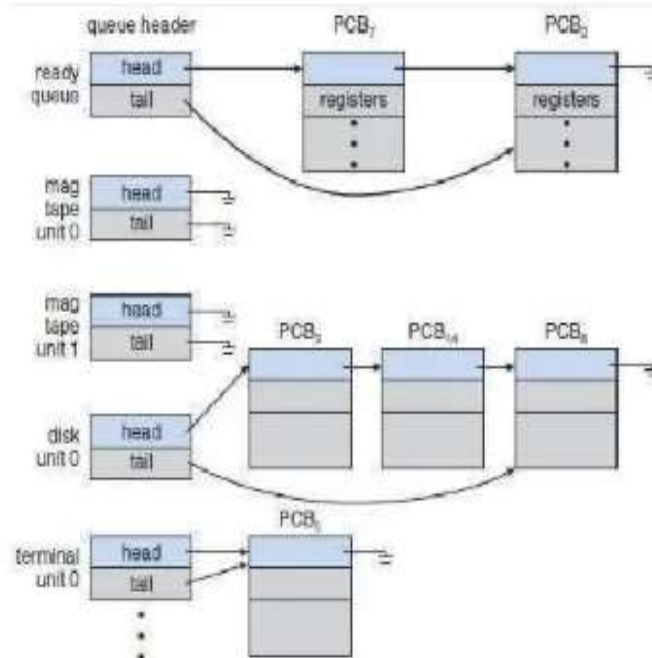


Figure 1.26 The ready-queue and various I/O device-queues

## 1.26 Operations on Processes

- 1) Process Creation and
- 2) Process Termination

### 1.26.1 Process Creation

- A process may create a new process via a **create-process** system-call.
- The creating process is called a parent-process.
  - The new process created by the parent is called the child-process (Sub-process).
- OS identifies processes by pid (process identifier), which is typically an integer-number.
- A process needs following resources to accomplish the task:
  - CPU time
  - memory and
  - I/O devices.
- Child-process may
  - get resources directly from the OS or
  - get resources of parent-process. This prevents any process from overloading the system.
- Two options exist when a process creates a new process:
  - 1) The parent & the children execute concurrently.
  - 2) The parent waits until all the children have terminated.
- Two options exist in terms of the address-space of the new process:
  - 1) The child-process is a duplicate of the parent-process (it has the same program and data as the parent).
  - 2) The child-process has a new program loaded into it.

#### Process creation in UNIX

- In UNIX, each process is identified by its process identifier (pid), which is a unique integer.
- A new process is created by the **fork()** system-call (Figure 1.29 & 1.30).
- The new process consists of a copy of the address-space of the original process.
- Both the parent and the child continue execution with one difference:
  - 1) The return value for the fork() is **zero** for the new (child) process.
  - 2) The return value for the fork() is **nonzero** pid of the child for the parent-process.
- Typically, the **exec()** system-call is used after a fork() system-call by one of the two processes to replace the process's memory-space with a new program.
- The parent can issue **wait()** system-call to move itself off the ready-queue.

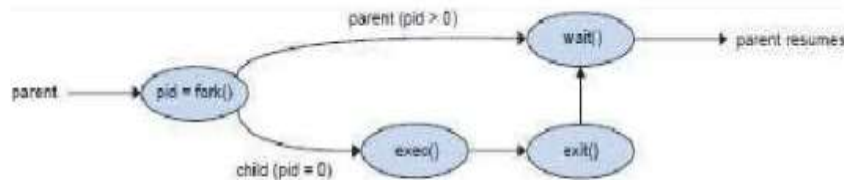


Figure 1.30 Process creation using the fork() system-call

### 1.26.2 Process Termination

- A process terminates when it executes the last statement (in the program).
- Then, the OS deletes the process by using **exit()** system-call.
- Then, the OS de-allocates all the resources of the process. The resources include
  - memory
  - open files and
  - I/O buffers.
- Process termination can occur in following cases:
  - A process can cause the termination of another process via **TerminateProcess()** system-call.
  - Users could arbitrarily **kill** the processes.
- A parent terminates the execution of children for following reasons:
  - 1) The child has exceeded its usage of some resources.
  - 2) The task assigned to the child is no longer required.
  - 3) The parent is exiting, and the OS does not allow a child to continue.
- In some systems, if a process terminates, then all its children must also be terminated. This phenomenon is referred to as **cascading termination**.

### 1.27 Inter Process Communication (IPC)

- Processes executing concurrently in the OS may be
  - 1) Independent processes or
  - 2) Co-operating processes.
- 1) A process is **independent** if
  - i) The process cannot affect or be affected by the other processes.
  - ii) The process does not share data with other processes.
- 2) A process is **co-operating** if
  - i) The process can affect or be affected by the other processes.
  - ii) The process shares data with other processes.
- Advantages of process co-operation:
  - 1) **Information Sharing**
    - Since many users may be interested in same piece of information (ex: shared file).
  - 2) **Computation Speedup**
    - We must break the task into subtasks.
    - Each subtask should be executed in parallel with the other subtasks.
    - The speed can be improved only if computer has multiple processing elements such as
      - CPUs or
      - I/O channels.
  - 3) **Modularity**
    - Divide the system-functions into separate processes or threads.
  - 4) **Convenience**
    - An individual user may work on many tasks at the same time.
    - For ex, a user may be editing, printing, and compiling in parallel.
- Two basic models of IPC (Figure 1.31):
  - 1) Shared-memory and
  - 2) Message passing.

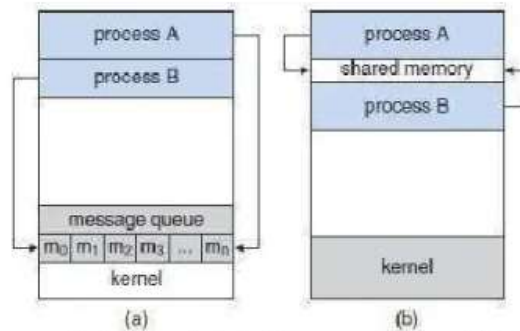


Figure 1.31 Communications models. (a) Message passing. (b) Shared-memory

#### 1.27.1 Shared-Memory Systems

- Communicating-processes must establish a region of shared-memory.
- A shared-memory resides in address-space of the process creating the shared-memory.
  - Other processes must attach their address-space to the shared-memory.
- The processes can then exchange information by reading and writing data in the shared-memory.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- For ex, **Producer-Consumer Problem**:
  - Producer-process produces information that is consumed by a consumer-process
- Two types of buffers can be used:
  - 1) **Unbounded-Buffer** places no practical limit on the size of the buffer.
  - 2) **Bounded-Buffer** assumes that there is a fixed buffer-size.
- Advantages:
  - 1) Allows maximum speed and convenience of communication.
  - 2) Faster.

## How to write answer

1. Definition
2. Introduction
3. Diagram
4. Explanation of Diagram
5. Explanation of Concept
6. Example
7. Conclusion

**Q: Explain Inter process Communication (IPC).**

**Ans: 1. Definition**

**Interprocess Communication (IPC)** is a mechanism that allows **processes to exchange data and coordinate their actions** while executing in an operating system.

It enables one process to communicate with another — either to share information or synchronize their execution.

### **2. Introduction**

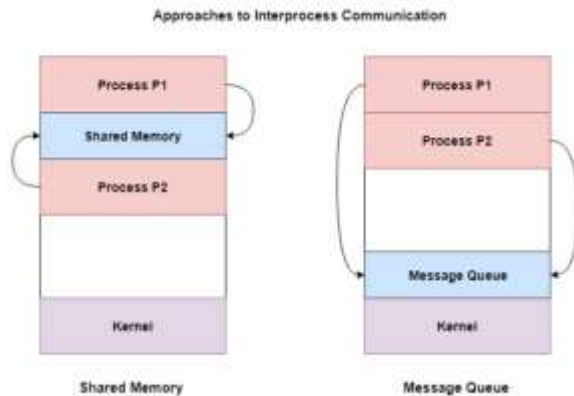
In a multitasking operating system, multiple processes run simultaneously. Each process operates in its own **address space** and cannot directly access the memory of another process.

To perform cooperative tasks, processes must communicate — this communication is achieved through **Interprocess Communication (IPC)**.

The operating system provides various IPC mechanisms to ensure **data sharing, synchronization, and coordination** among processes.



### 3. Diagram



### 4. Explanation of Diagram

- The diagram shows **two processes (A and B)** communicating with each other.
- They can communicate in two main ways:
  1. **Message Passing:** Exchange of messages using system calls like `send()` and `receive()`.
  2. **Shared Memory:** Both processes access a **common memory area** to read/write data.
- The operating system manages this communication safely and efficiently.

### 5. Explanation of Concept

#### □ Models of IPC

#### 1. Message Passing:

- Used when processes do **not share memory**.
- The OS provides system calls for sending and receiving messages.
- Example: Client-server communication over a network.

#### 2. Shared Memory:

- A section of memory is shared among processes.
- Faster than message passing but requires **synchronization** (to avoid conflicts).
- Example: Producer-consumer problem.

## □ Need for IPC

1. **Information Sharing** – Processes exchange data (e.g., shared files, buffers).
2. **Computation Speedup** – Divide large tasks among several processes.
3. **Modularity** – Design applications as independent, communicating modules.
4. **Convenience** – Background processes can interact with user applications.

## □ Importance of IPC

1. Ensures **coordination** between processes.
2. Enables **data exchange** between user programs and system services.
3. Improves **efficiency** and **system performance**.
4. Essential for **distributed systems** and **network-based applications**.

## Example

When you download a file, one process handles **network data** (producer), while another writes it to **disk storage** (consumer). Both communicate through IPC.

## 6. Conclusion

Interprocess Communication (IPC) is an essential function of the operating system that allows processes to **cooperate, share data, and synchronize** their execution. It enhances system performance, supports multitasking, and ensures smooth interaction between processes — making it a core concept of modern operating systems.