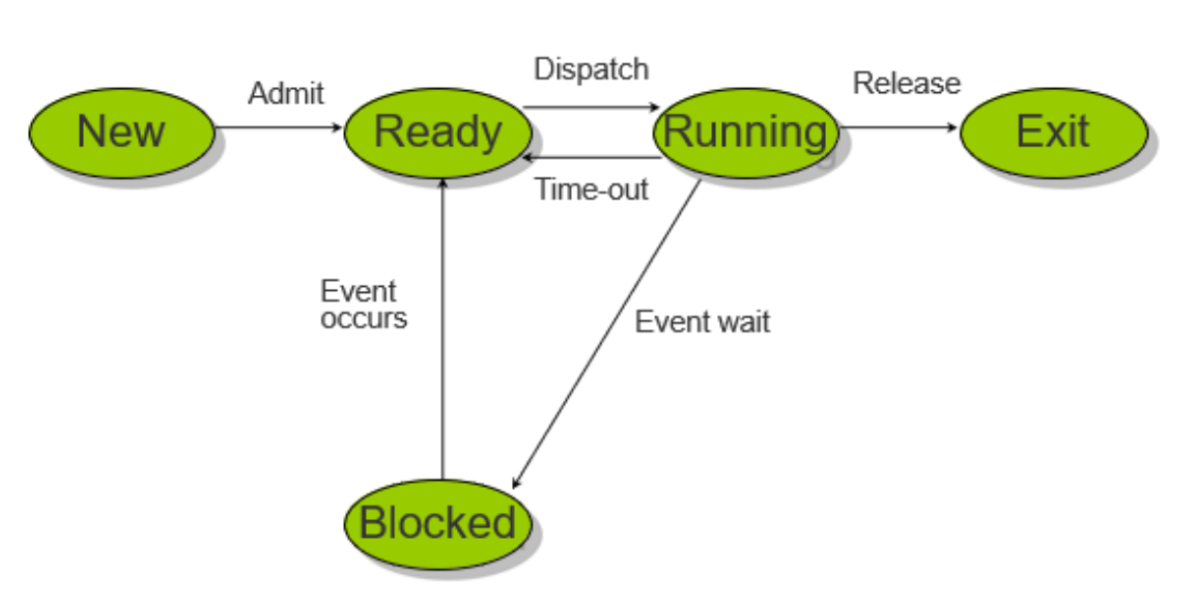
**Process Concept**

* **Definition:** A process is a program in execution. It is the basic unit of work in an operating system.

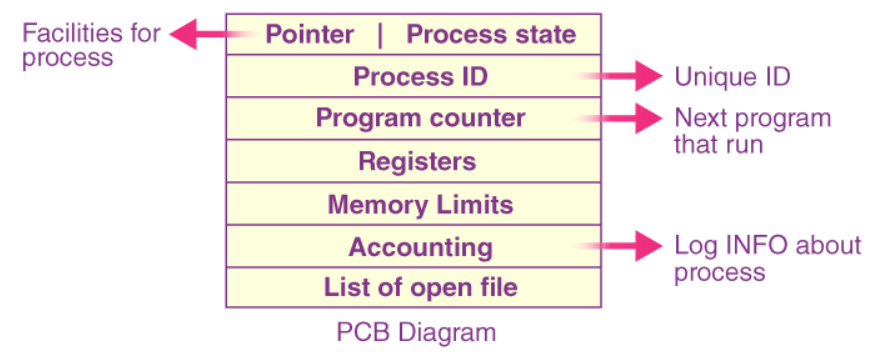
A process includes:

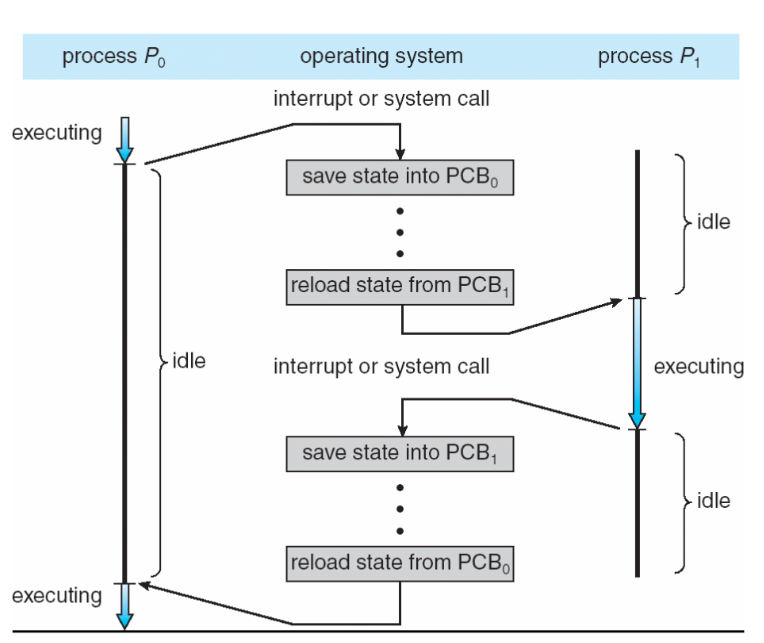
* Program code **(text section**).
* Current activity (program counter, **CPU registers**).
* **Stack** (temporary data like function parameters, return addresses, local variables).
* **Data** section (global variables).
* **Heap** (dynamically allocated memory during runtime).

**State Process Model**



* **New:** A process has been created but has not yet been admitted to the pool of executable processes.
* **Ready:** Processes that are prepared to run if given an opportunity. That is, they are not waiting on anything except the CPU availability.
* **Running:** The process that is currently being executed. (Assume single processor for simplicity.)
* **Blocked:** A process that cannot execute until a specified event such as an IO completion occurs.
* **Exit:** A process that has been released by OS either after normal termination or after abnormal termination (error).
* **Process Control Block (PCB)**

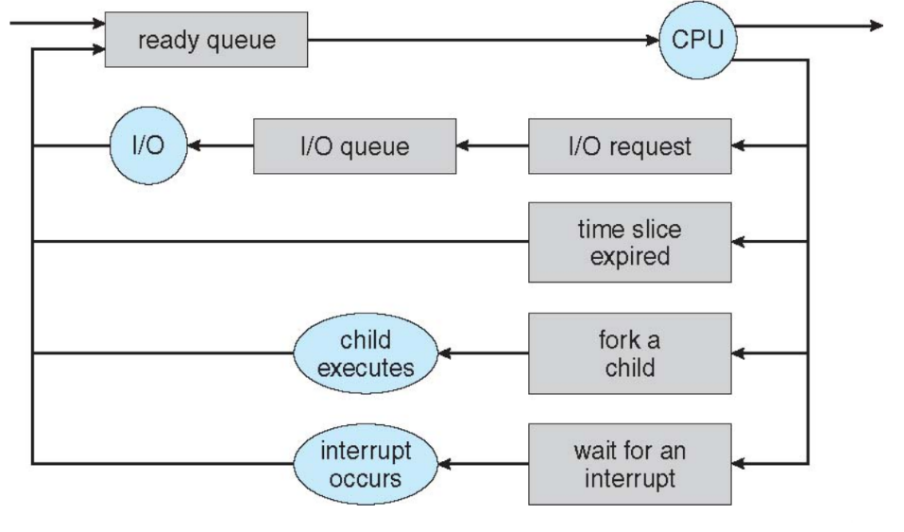


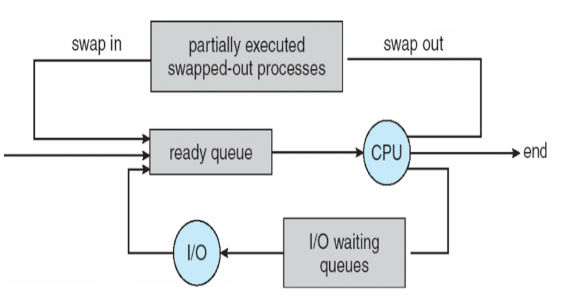
**Context Switching Between Process to Process**

* **Context Switching:**
  + When the CPU switches from one process to another, the state of the current process is saved in its PCB, and the state of the new process is loaded.
  + This is called a context switch.
* **Overhead:**
  + Context switching takes time and is considered overhead since no useful work is done during the switch.

**When does context switching happen?**

* **Multitasking:** Switching between processes in a time-shared system.
* **Interrupt Handling:** Switching from user mode to kernel mode to handle system calls or interrupts.
* **Process Priority Changes:** Switching to a higher-priority process.
* **I/O Blocking:** When a process is waiting for I/O, the CPU switches to another ready process.

**Process Scheduling**

* **Purpose**:
  + The **process scheduler** selects which process should execute next on the CPU.
* **Scheduling Queues**:
  + **Job queue**: All processes in the system.
  + **Ready queue**: Processes in memory ready to execute.
  + **Device queues**: Processes waiting for I/O devices.
* **Schedulers**:
  + **Short-term scheduler (CPU scheduler)**: Selects the next process to execute on the CPU (invoked frequently).
  + **Long-term scheduler (job scheduler)**: Selects which processes should be brought into the ready queue (invoked infrequently).
  + **Medium-term scheduler**: Handles **swapping** (moving processes between memory and disk).

**Operations on Processes**

* The OS provides mechanisms for:
  + **Process creation**.
  + **Process termination**.
  + **Inter Process communication (IPC)**.

**Process Creation**

Fork(); duplicates the code written below it and then runs it. The duplicated code is the child process.

The **total process** would be 2n and **total child** process would be 2n-1. So, the program will run **4 times**.

**A screen shot of a computer code

AI-generated content may be incorrect.Examples:** Total process are 4 out of which 3 are child processes. So, printf would run 4 times.

A screenshot of a computer

AI-generated content may be incorrect.

**Zombie Process**

When a child process terminates, it **does not immediately disappear**. Instead, it enters the **zombie state** until the parent process collects its exit status using wait().

If the parent process does not call wait(), the child process remains in the **process table** as a zombie.

This happens because the kernel **keeps the process metadata (PID, exit status, etc.)** to allow the parent process to retrieve it.

If the parent process **itself terminates**, the zombie process is adopted by **init (PID 1)**, which automatically cleans up the zombie.

**Orphan Process:**  
An **orphan process** is a process whose **parent process has terminated** (either normally or abnormally) without waiting for the child process to complete. In most operating systems, when a parent process terminates, its child processes are either terminated as well or become orphaned. Orphan processes are typically **adopted by a special system process** (often the **init process**, with PID 1 in Unix-like systems), which becomes their new parent.

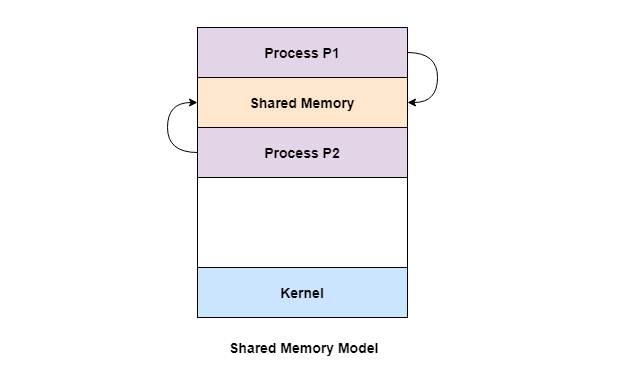
**Key Characteristics of Orphan Processes**

1. **Parent Termination**:
   * The parent process terminates before the child process completes its execution.
   * The child process is left without a parent.
2. **Adoption by Init**:
   * In Unix-like systems, orphan processes are adopted by the **init process** (or its modern equivalents like **systemd**).
   * The init process periodically calls **wait()** to collect the exit status of orphaned processes, preventing them from becoming zombies.
3. **No Direct Relationship**:
   * Once orphaned, the child process no longer has a direct relationship with its original parent.
4. **Resource Usage**:
   * Orphan processes continue to consume system resources (e.g., memory, CPU) until they terminate.



**Parent Terminates Without wait()**:

* If a parent process terminates without calling **wait()** or **waitpid()**, its child processes become orphans.

**Interprocess Communication (IPC):**

Process can be of two types: *1 Independent 2 Cooperating*

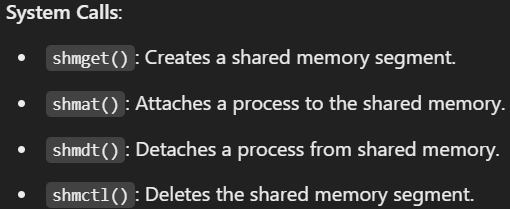
**- IPC** is used for **cooperating** processes

**- Reasons for cooperating processes:**

* Information sharing
* Computation speedup
* Modularity
* Convenience

**Two Models Of IPC:**

1. **Shared Memory:**

* **Mechanism**: Processes communicate by reading and writing to a shared memory area.
* **Synchronization**: Requires mechanisms to ensure that processes do not interfere with each other when accessing shared memory. (Use **Semaphore** for synchronization)
* 

**Buffer in Shared Memory:**

1. **Bounded Buffer:** Assume that there is fixed size of buffer.
2. **Unbounded Buffer:** Places no limited size of the buffer.
3. **Message Passing:**

Mechanism for processes to communicate and to synchronize their actions

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

**Operations**:

* **send(message)**: Sends a message to another process.
* **receive(message)**: Receives a message from another process

**Implementation Issues**:

* How are communication links established?
* Can a link be associated with more than two processes?
* What is the capacity of a link?
* Is the link unidirectional or bi-directional?

**Implementation of Message Passing:**

**1 Physical 2 Logical**

- Shared Memory - Direct or Indirect

- Hardware Bus - Synchronous or Asynchronous

- Network - Automatic or Explicit buffering

**🡪Direct Communication (Synchronous):**

* **Explicit Naming**: Processes explicitly name each other when communicating.

**Properties**:

* + Links are established automatically between communicating processes.
  + Each link is associated with exactly one pair of processes.
  + The link may be unidirectional, but usually is bi-directional

**Send (P, message)** 🡪 Send a message to process P

**Receive (Q, message)** 🡪 Received a message from process Q

**🡪 Indirect Communication (Asynchronous):**

* **Mailboxes**: Messages are sent to and received from mailboxes (or ports).
* **Properties**:
  + A link is established if processes share a common mailbox.
  + A link may be associated with multiple processes.
  + Each pair of processes can have several communication links.

**Send (A, message)** 🡪 Send a message to mailbox A

**Receive (A, message)** 🡪 Received a message mailbox A

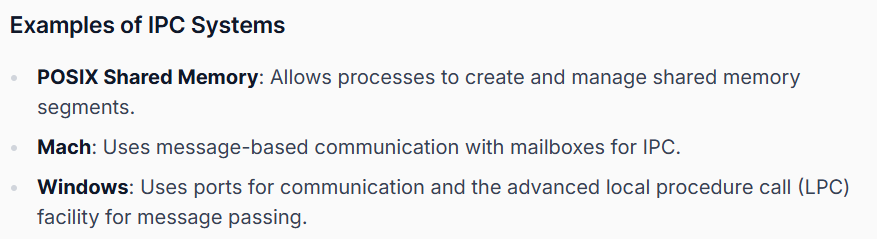
**Synchronization in Message Passing:**

* **Blocking (Synchronous):**
  + **Blocking Send:** The sender is blocked until the message is received.
  + **Blocking Receive:** The receiver is blocked until a message is available.
* **Non-Blocking (Asynchronous):**
  + **Non-Blocking Send**: The sender sends the message and continues execution.
  + **Non-Blocking Receive:** The receiver receives a valid message or a null message if none is available.

**\*\* Rendezvous:** If both sender and receiver are blocked \*\*

**Buffering in Message Passing:**

* **Zero Capacity**: No messages are queued; the sender must wait for the receiver.
* **Bounded Capacity**: A finite number of messages can be queued; the sender must wait if the buffer is full.
* **Unbounded Capacity**: An infinite number of messages can be queued; the sender never waits.

****

* **Producer-Consumer Problem and its Solution Using Shared Memory & Message Passing**

**What is the Producer-Consumer Problem?**

The Producer-Consumer Problem is a classic synchronization problem where:

* A Producer generates data and adds it to a shared buffer.
* A Consumer retrieves data from the buffer and processes it.
* The buffer has a limited size, so the producer must wait if it’s full, and the consumer must wait if it’s empty.

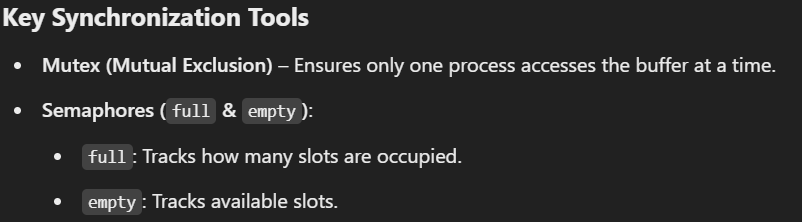
**We can solve this problem using two IPC mechanisms:**

1. Shared Memory (fast but requires synchronization).
2. Message Passing (structured but slower).

**Solution 1: Using Shared Memory**

**Concept**

* Producer writes data into shared memory.
* Consumer reads data from shared memory.
* Synchronization is achieved using semaphores

****

* **With Bounded Buffer:**

**A white background with black text

AI-generated content may be incorrect.** **Producer A white background with black text

AI-generated content may be incorrect.** **Consumer** A computer code with text

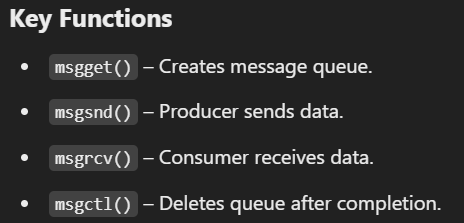
AI-generated content may be incorrect.



**Solution 2: Using Message Passing**

**Concept**

* Producer sends messages using a message queue.
* Consumer receives messages from the queue.
* Kernel manages synchronization.

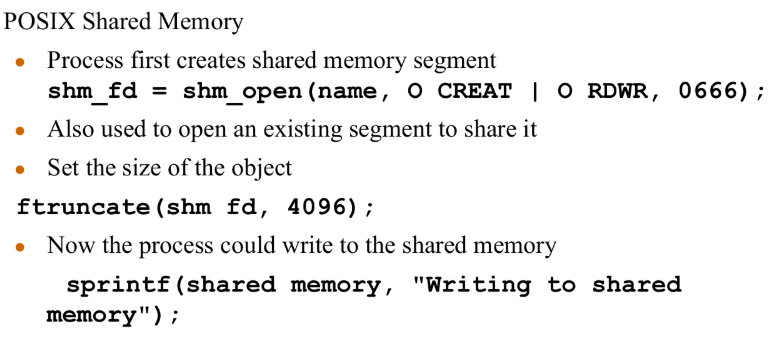
****

**A white screen with black text

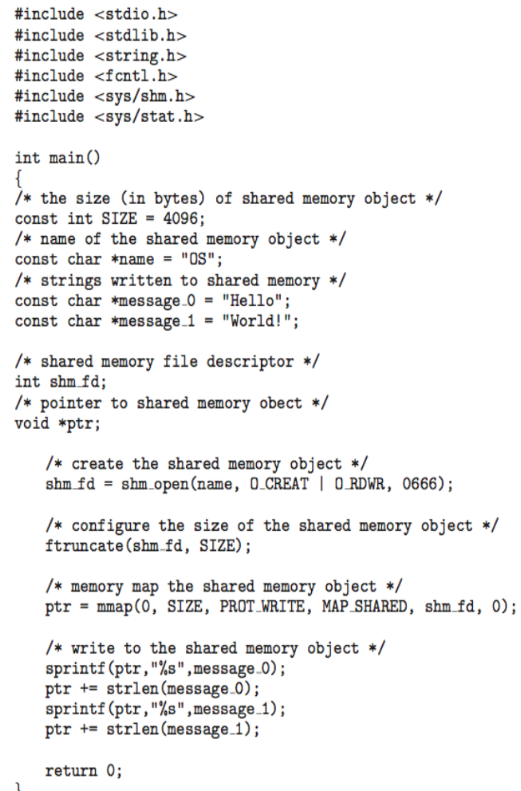
AI-generated content may be incorrect.**



**IPC System – POSIX**

****

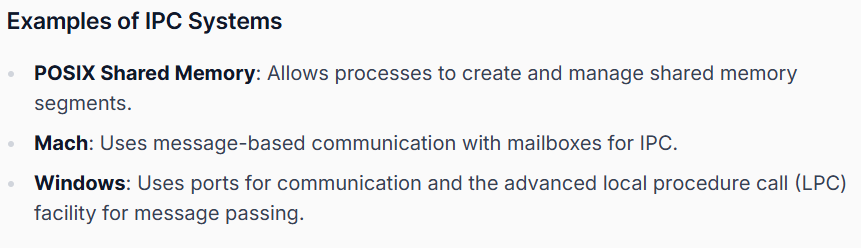
**IPC Producer:**

****

**IPC Consumer:**

**A white screen shot of a computer program

AI-generated content may be incorrect.**

****