KARNAVATI
UNIVERSITY

Operating Systems
Course Code: **71203002004**
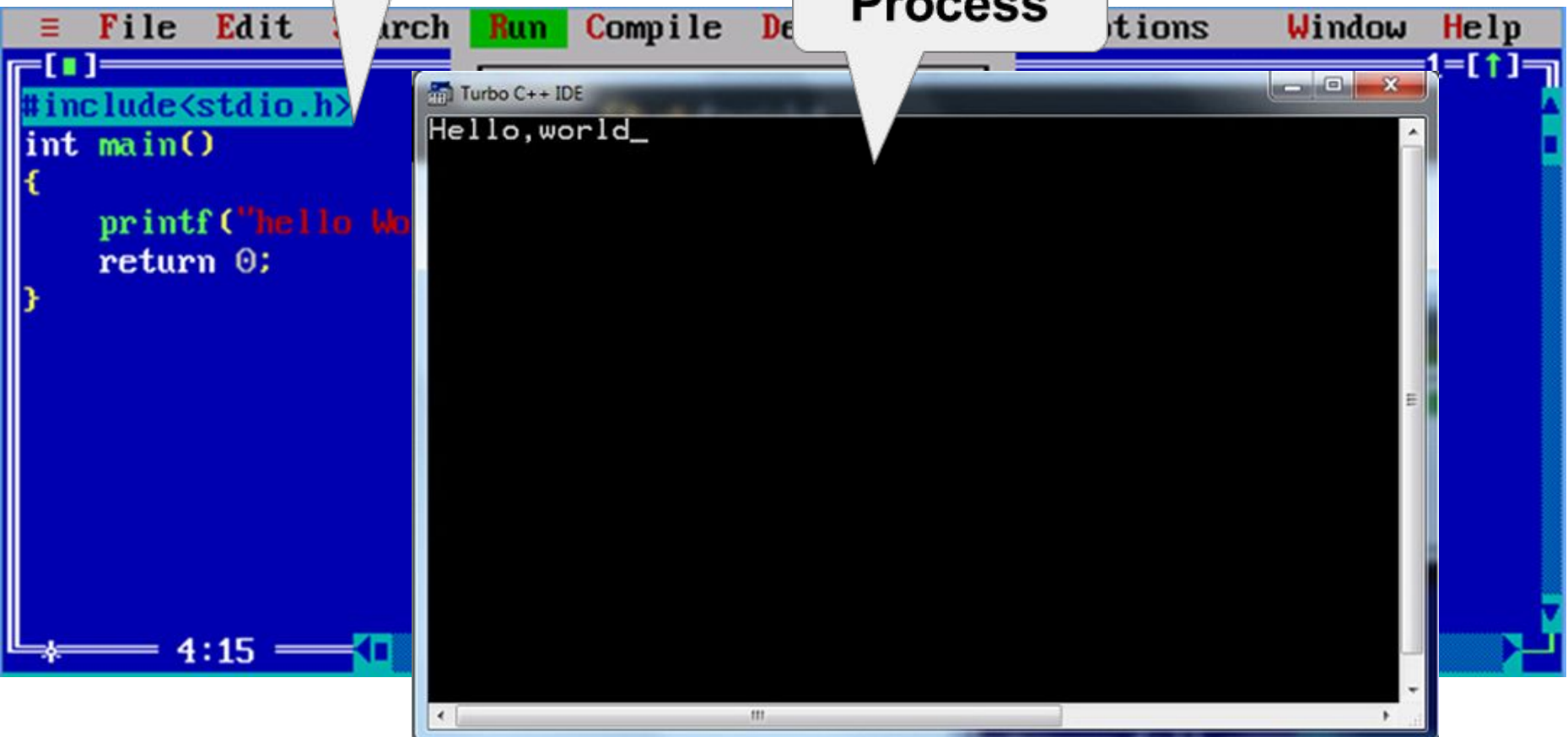*Process*

by -
Minal Rajwar

What is a Process?

- A **process** is a program in execution. A program (like a compiled C/C++ binary) is a **passive** entity stored on disk.
- When run, it becomes an **active** process with its own memory and system resources.
- Running the same program multiple times creates multiple **processes**, each functioning independently.
- Two essential elements of a process are program code:
  - Program code
  - Set of Data associated with code

Attributes of a Process

A process has several key attributes stored in a **Process Control Block (PCB)**, which helps the operating system manage it. Important attributes include:

- **Process ID (PID):** Unique ID for each process.
- **Process State:** Current status (e.g., running, waiting).
- **CPU Scheduling Info:** Priority and scheduling data.
- **I/O Info:** Devices the process is using.
- **File Descriptors:** Open files and network connections.
- **Accounting Info:** Resource usage (CPU time, etc.).
- **Memory Info:** Memory layout (stack, heap, code, data).

States of a Process

A process moves through various states during its lifecycle:

1. **New:** Process is being created.
2. **Ready:** Waiting to be assigned to the CPU.
3. **Running:** Currently being executed.
4. **Waiting (Blocked):** Waiting for I/O.
5. **Terminated:** Execution is complete.
6. **Suspended Ready:** Ready but temporarily moved out of main memory.
7. **Suspended Blocked:** Blocked and also suspended from memory.
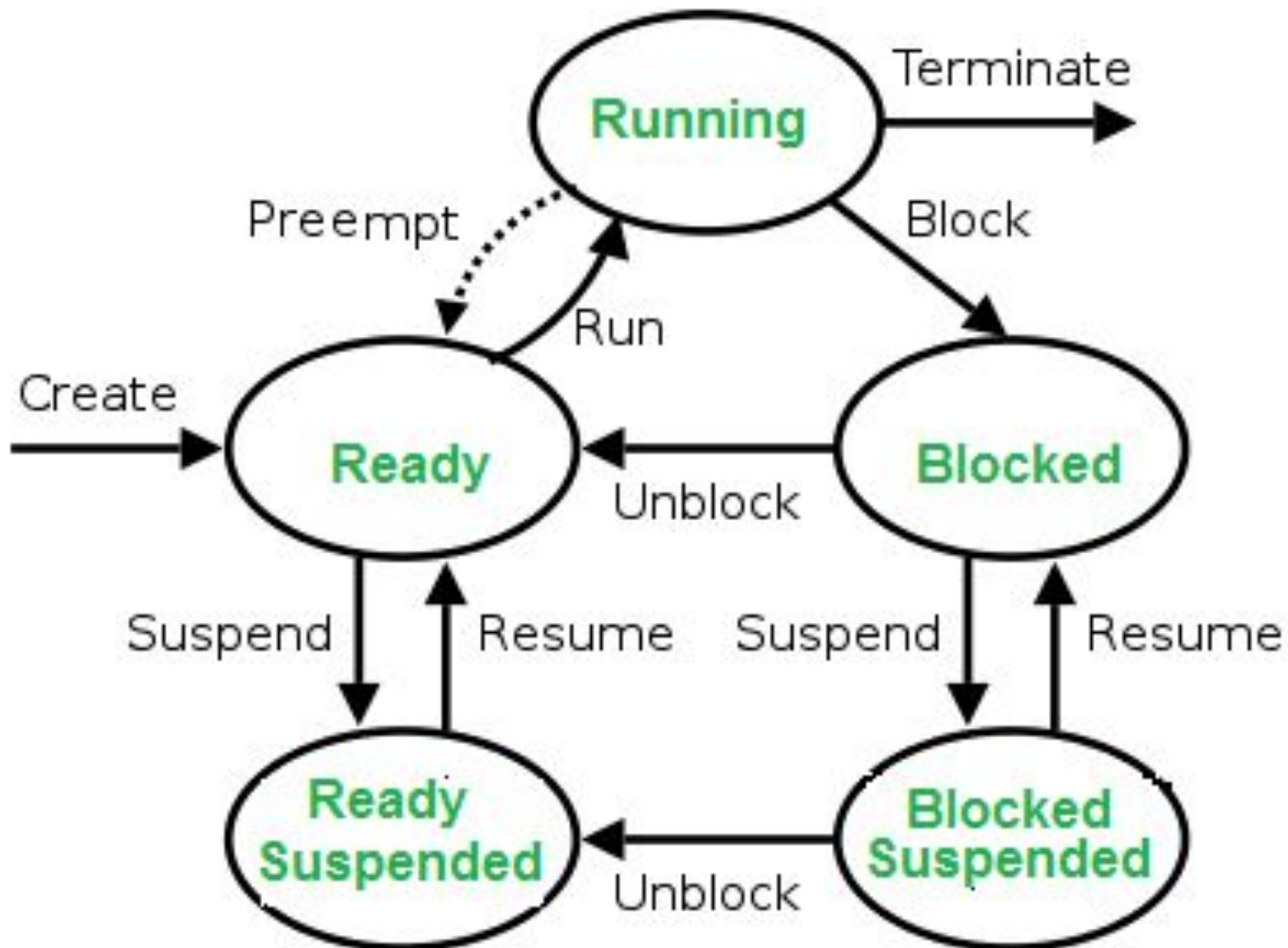
What is Process Management?

- **Process Management** is a core function of the **Operating System (OS)** that handles the creation, scheduling, execution, and termination of processes.

- In systems that support **multitasking or multiprocessing**, it ensures smooth, efficient, and fair execution of all running processes.

Key Process Operations

Process Creation

- A new process is created (using fork() or similar).
- The OS assigns it a **PID** and creates a **PCB** (Process Control Block).
- Example: Opening Chrome creates a new process.

Scheduling

- The OS decides **which process gets the CPU** next.
- Uses algorithms like **Round Robin**, **FCFS**, or **Priority Scheduling**.
- Example: While listening to music and browsing, the OS switches between both processes.

Execution

- CPU starts running the process.
- It may move to **waiting** if it needs I/O, or be **preempted** by a higher-priority process.
- Example: Downloading a file pauses briefly when the system is busy.

Killing (Termination)

- Process finishes or is forcefully stopped (e.g., due to an error).
- OS removes it from memory and releases resources.
- Example: Closing a game or killing it via Task Manager.

Process Management Tasks

Task	Description	Example
Process Creation & Termination	Starts or ends processes and manages their resources.	A program starts with a PID, ends by freeing RAM.
CPU Scheduling	Decides which process runs on CPU.	Multitasking between YouTube and Excel.
Deadlock Handling	Prevents situations where processes wait on each other forever.	Two apps waiting on the same locked file.
Inter-Process Communication (IPC)	Allows processes to talk to each other.	Chat apps or client-server programs sharing data.
Process Synchronization	Ensures correct access to shared data in multi-process apps.	Bank transactions updating the same account balance.

Process Creation

A process can be created in several ways during the operation of an OS. Common reasons include:

1. System Initialization

When the OS starts, it creates essential processes like system daemons (background services).

Example: On Linux, the init (or systemd) process is the first process created. It starts other system services like networking, logging, etc.

2. User Request

A user runs a program manually.
Example: Double-clicking on Chrome.exe on Windows creates a new **Chrome** process.

3. Batch Job Initialization

In server environments, jobs are submitted in batches (e.g., in data centers or banks).
**Example:** Running a scheduled script every night for database backup spawns a new process.

4. Process Calls fork()

One process creates another using a system call like fork() (common in UNIX/Linux).

- **Parent process** calls fork(), and a **child process** is created.
- The child gets a **copy** of the parent's memory but has a separate address space.

*Parent and child may perform different tasks after creation. Web servers like **Apache** use this model—one main process forks multiple worker processes to handle requests.*

Process Creation Using fork()
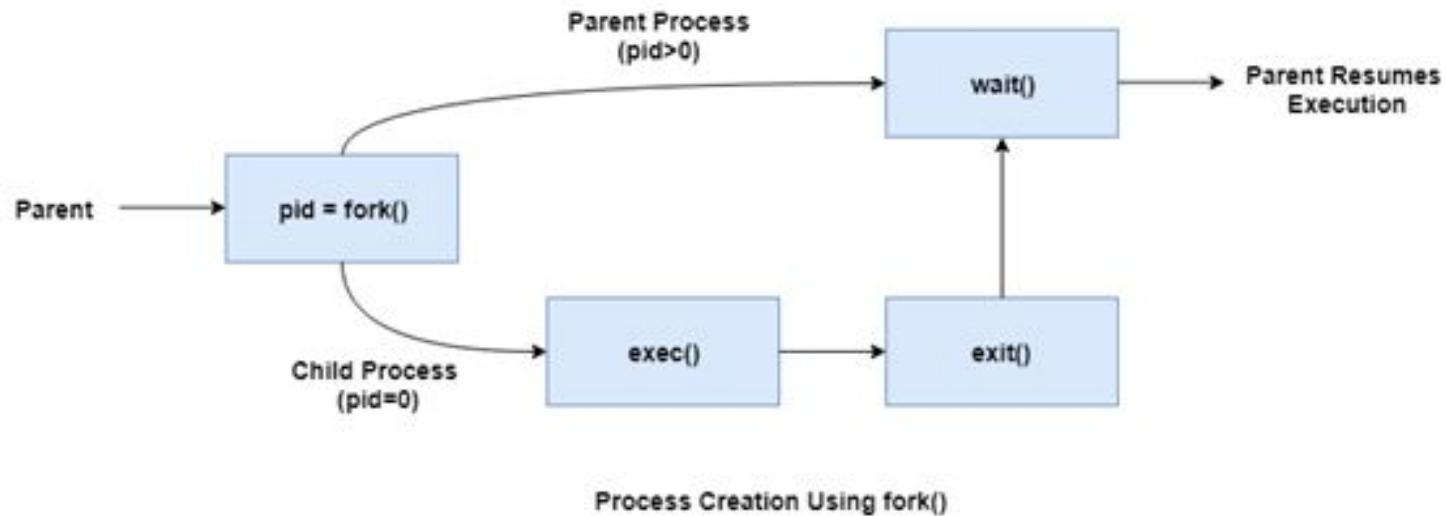
```
1   #include <stdio.h>
2   #include <unistd.h>
3
4   int main() {
5       for (int i = 1; i <= 3; i++) {
6           pid_t pid = fork();
7
8           if (pid == 0) {
9               // Child process
10              printf("I am Child %d. My PID is %d. My Parent's PID is %d\n", i, getpid(), getppid());
11              return 0; // Exit child so it doesn't create more children
12          }
13      }
14
15      // Only parent reaches here
16      sleep(1); // Wait to ensure all child messages print first
17      printf("I am the Parent. My PID is %d\n", getpid());
18
19      return 0;
20  }
```

```
I am Child 1. My PID is 1330. My Parent's PID is 1326
I am Child 3. My PID is 1332. My Parent's PID is 1326
I am Child 2. My PID is 1331. My Parent's PID is 1326
I am the Parent. My PID is 1326
```

Process Termination

Processes terminate for several reasons:

1. Normal Exit

After completing its task, a process ends by calling exit().
Example: A calculator program ends after you close it.

2. **Illegal Operation**

If a process tries to do something not allowed, the OS may terminate it.
Example: Writing to a read-only file or accessing memory outside allowed space causes **segmentation fault**.

3. Parent Requests Termination

A parent can terminate its child process using system calls
(like kill() in Linux).
 Example: A shell script that starts a process and kills it after
a timeout.

```
#!/bin/bash
my_process &
sleep 5
kill $!
```

4. I/O Failure

If a process depends on a device (e.g., printer, disk) that fails, the OS may stop it.
 Example: A process waiting for a disconnected USB drive might crash.

5. Memory Scarcity

If a process tries to use more memory than available, the OS might terminate it.
 Example: Opening a huge image in a low-RAM system can crash the image viewer.

6. Parent Dies

In many systems, if the parent process dies, its children are also terminated (or adopted by another process like init in Linux).

What is an Interrupt?

An **interrupt** is a signal sent by hardware or software to the processor, asking it to pause its current task and handle a more urgent task. This allows the processor to respond quickly to important events.

When an interrupt occurs:

1. The processor finishes the current instruction.
2. It saves the address of the next instruction.
3. It jumps to a special function called the **Interrupt Service Routine (ISR)** to handle the event.
4. After handling the interrupt, it resumes the paused task.

This whole process takes a small delay, called **interrupt latency**, because the processor must save its current state and notify the device that its request is being handled.

Types of Interrupt?
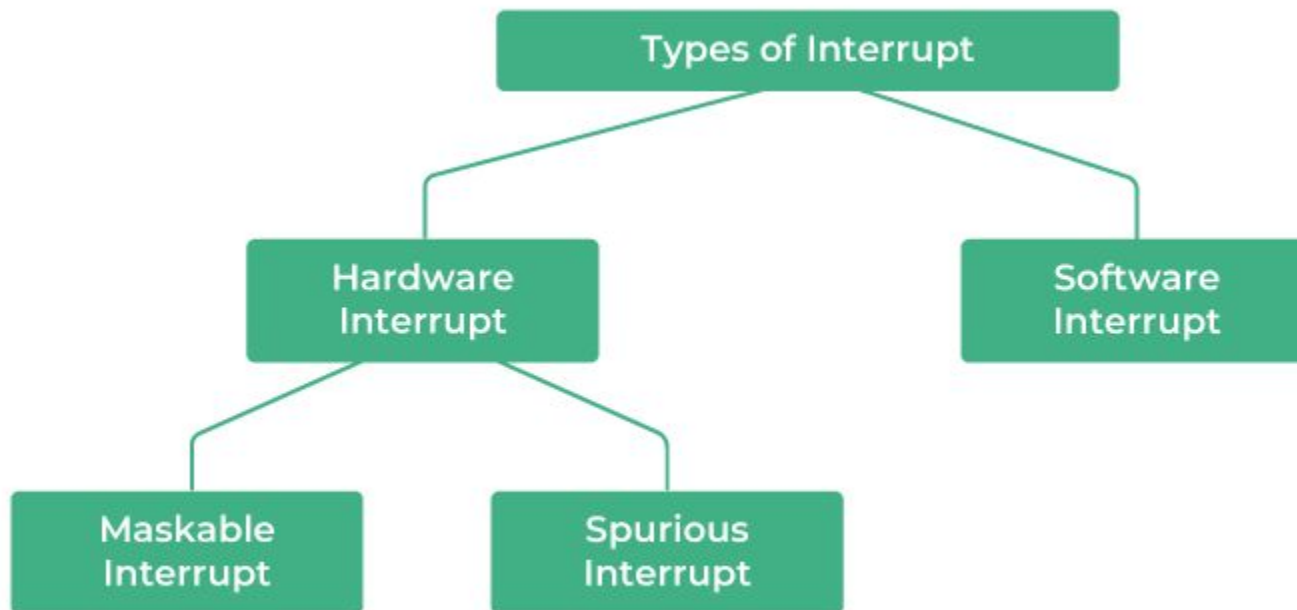
Interrupts can be triggered by **software** or **hardware**:

1. Software Interrupts

● Caused by programs, not hardware.
● Used for system calls or error handling (e.g., division by zero).
● Triggered using special instructions that tell the processor to run a specific routine.

2. Hardware Interrupts

● Triggered by devices (like a keyboard or mouse) connected to the **Interrupt Request Line (INTR)**.
● Devices signal interrupts by sending a signal through this line.

Types of hardware interrupts:

● **Maskable Interrupt:** Can be turned on or off using special settings.
● **Spurious Interrupt:** Happens without a clear cause—may be due to faulty hardware or noise.

Interrupt Handling Mechanism

1. An interrupt is raised (from a device or system).
2. The processor saves the current state (registers and program counter).
3. It identifies the correct ISR from the **interrupt vector table**.
4. The processor runs the ISR (located in the operating system's kernel).
5. The ISR does its job (e.g., processing input).
6. The processor restores the saved state.
7. The original task continues.
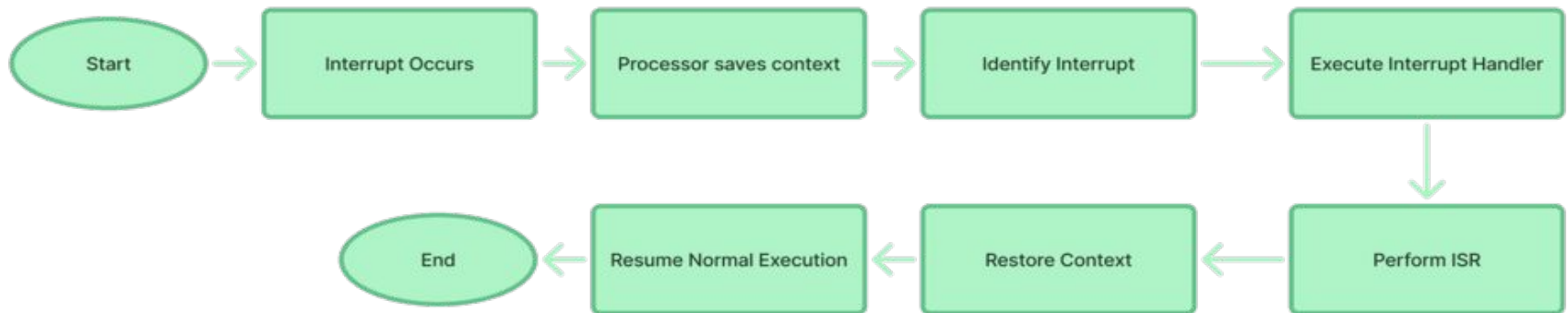
IPC - Inter Process Communication

IPC is a way for different processes (programs running on a computer) to talk to each other, share data, and work together without causing conflicts.

Types of Processes

1. **Independent Process**
   - Doesn't share data or resources with others.
   - No need for communication.
2. **Co-operating Process**
   - Shares data or resources with other processes.
   - Needs communication using IPC.

Why is IPC Needed?

- To **share information** between processes
- To **coordinate tasks**
- To **avoid conflicts** when accessing the same data

Two Main Methods of IPC

1. **Shared Memory**
   - Processes share a part of memory.
   - One process writes data; another reads it.
2. **Message Passing**
   - Processes send and receive messages.
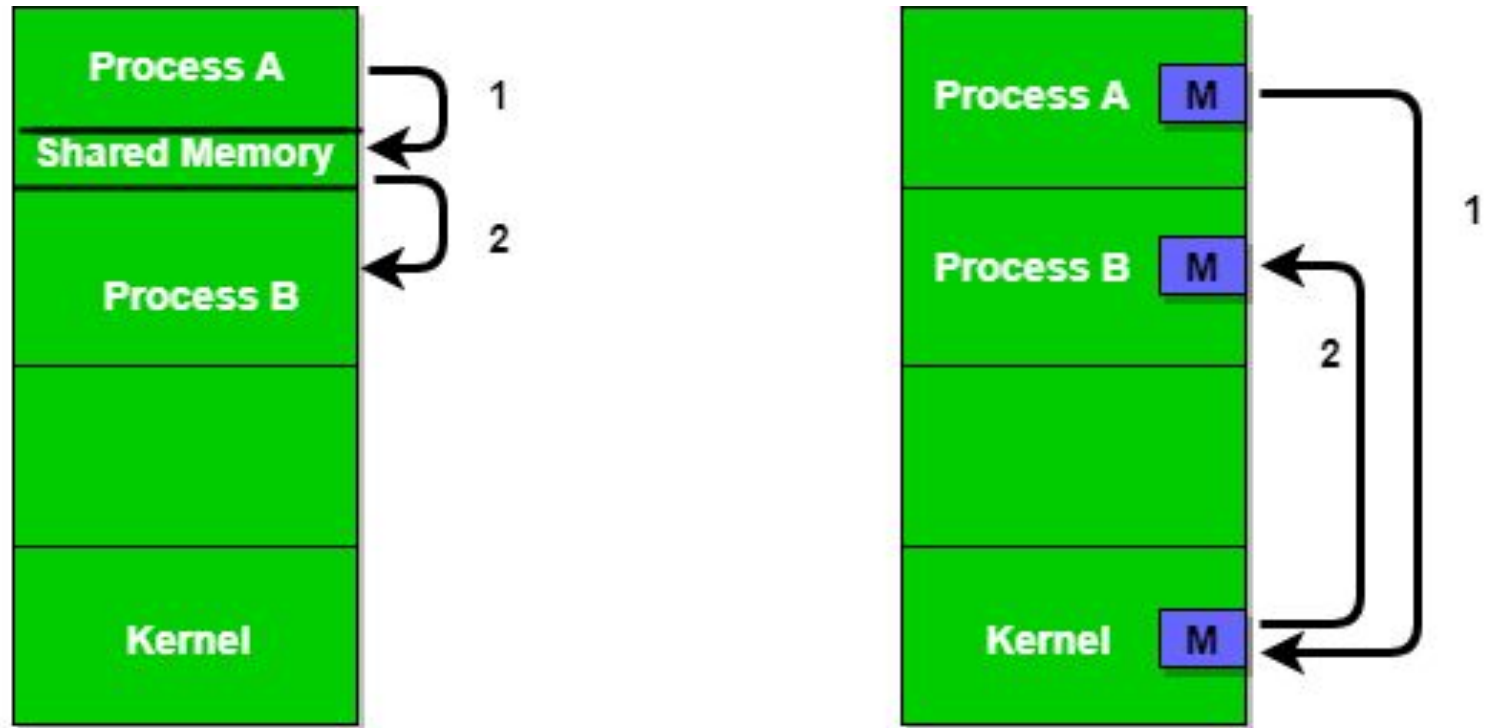   - No need to share memory.

Figure 1 - Shared Memory and Message Passing

DISCUSSION & REVISION

1. What does OS assign to each new process?
2. Which component decides which process runs next?
3. What happens to a process when it finishes execution?
4. What do processes use to communicate with each other?
5. What prevents processes from waiting forever in a loop?
6. What signal tells the processor to stop and handle an urgent task?
7. Which type of process shares data with other processes?
8. What IPC method involves sending and receiving data without shared memory?