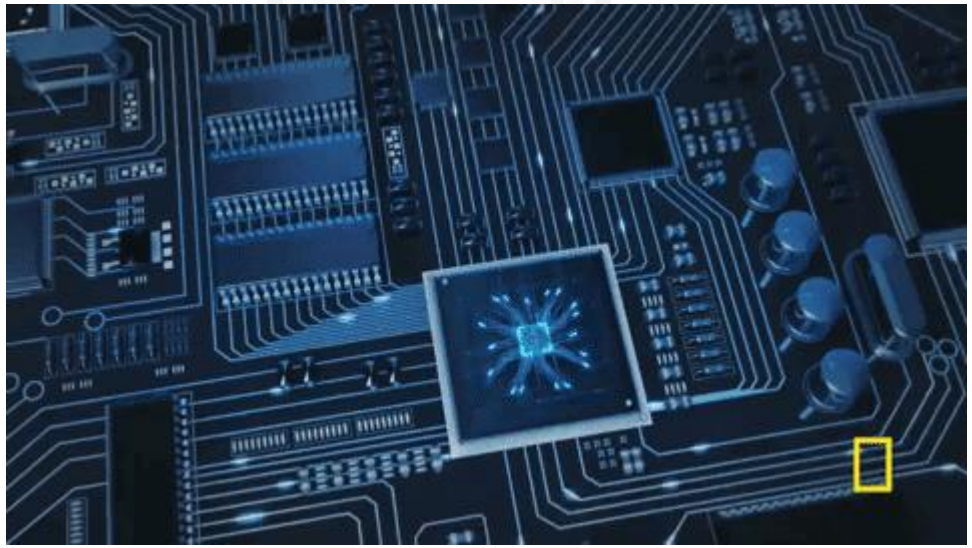
ATME College of Engineering

Department of EEE
Emitting Elite Energy

BEE613B
Embedded Systems Design
Module-5: Real-time Operating System(RTOS) based Embedded System Design



Presented by,

Mr.Shreeshayana R

Assistant Professor

Electrical and Electronics Engineering

ATME College of Engineering, Mysuru

ATME College of Engineering

Department of EEE
Emitting Elite Energy

CONTENTS

Real-time Operating System(RTOS) based Embedded System Design:
Operating System basics,
Types of Operating Systems,
Tasks, Process and Threads,
Multiprocessing and Multitasking,
Task Scheduling (Chapter 10 – Text 1: 10.1 to 10.5)

Real-Time Operating System (RTOS) based Embedded System Design

1. Operating System (OS) Basics:

Real-Time Operating System (RTOS) based Embedded System Design

1. Operating System (OS) Basics:

An **Operating System (OS)** is system software that manages hardware and software resources and provides services for computer programs.

Key Functions:

- **Process Management**: Creation, scheduling, and termination of processes.
- **Memory Management**: Efficient allocation/deallocation of memory.
- **I/O Management**: Handling input/output operations.
- **File System Management**: Organizing and accessing files.
- **Security & Protection**: Ensuring data and resource access control.

Embedded Systems Overview:

An **embedded system** is a computer system with a dedicated function within a larger system.

Characteristics:

- Real-time constraints
- Low power and resource usage
- High reliability and stability

What is a Real-Time Operating System (RTOS)?

An **RTOS** is a specialized OS designed to meet **real-time constraints**, where task completion must occur within strict time limits.

Types of RTOS:

- **Hard RTOS**: Missing a deadline is catastrophic (e.g., pacemakers).
- **Soft RTOS**: Occasional deadline misses are tolerable (e.g., audio streaming).

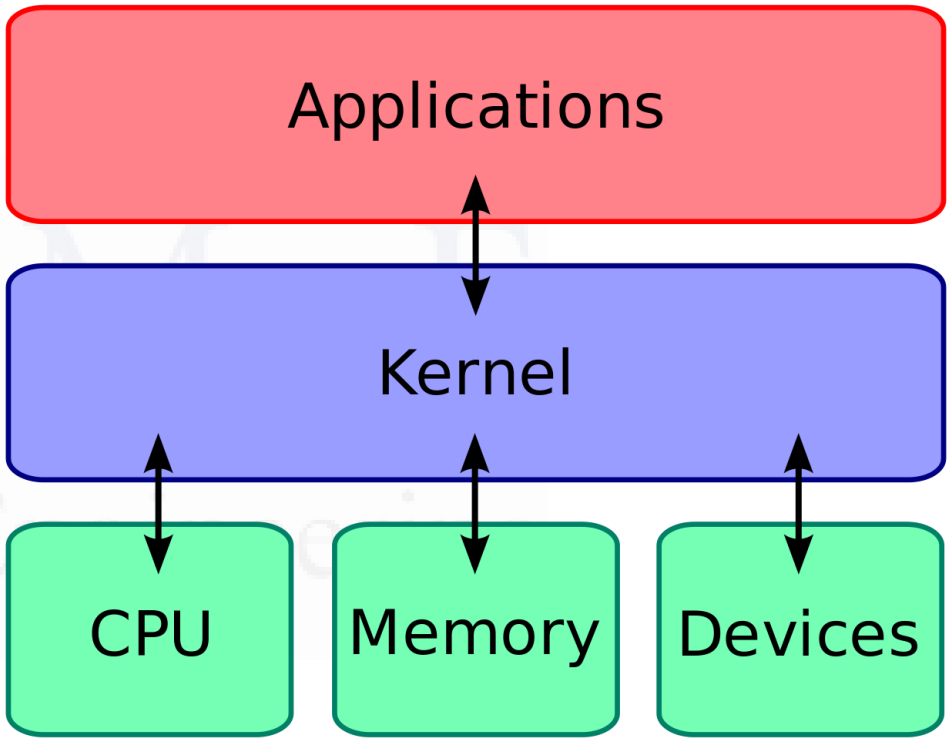
Differences between General-Purpose OS and RTOS:

Feature	General-Purpose OS	RTOS
Scheduling	Fairness-oriented	Deadline-oriented
Response Time	Unpredictable	Predictable
Priority Inversion	Less control	Mechanisms to prevent
Determinism	Low	High

Kernel

The kernel is the **core part of the OS, managing**:

- **Process Management**: Handles execution, scheduling, and communication between processes.
- **Primary Memory Management**: Manages RAM usage dynamically using the Memory Management Unit (MMU).
- **File System Management**: Manages file creation, deletion, storage, and directories.
- **I/O Device Management**: Uses device drivers to handle input/output device requests.
- **Secondary Storage Management**: Manages disk space, scheduling, and free space.
- **Protection System**: Implements user access control and security policies.
- **Interrupt Handler**: Manages external and internal system interrupts.

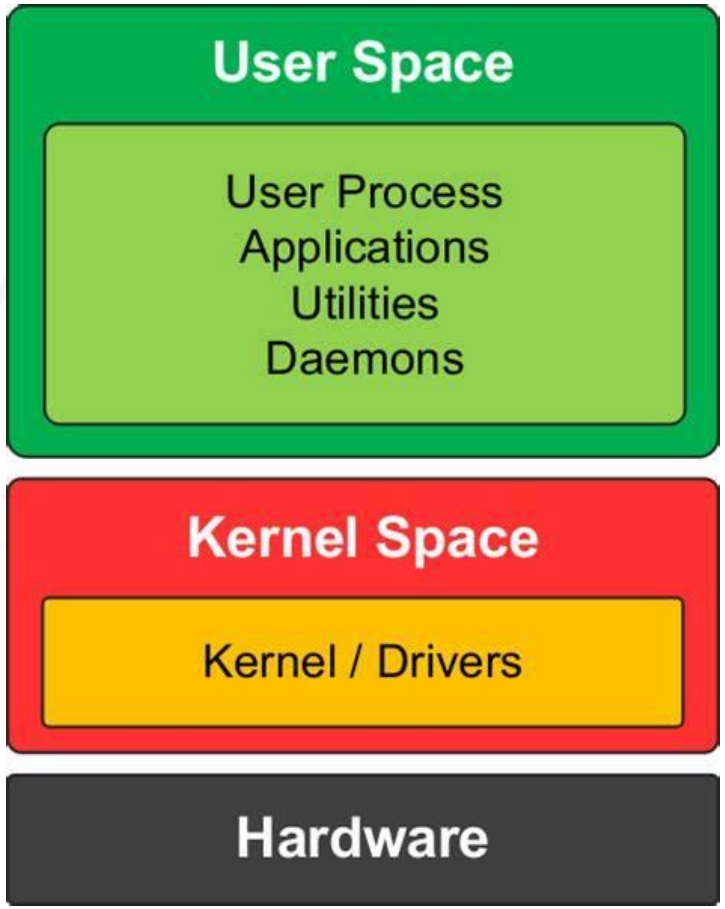


Kernel and User Space

- **Kernel Space**: Area in memory reserved for kernel code; protected from user access.

- **User Space**: Memory where user applications run; separate from kernel for security and stability.

- **Swapping**: Transfers parts of programs between main and secondary memory as needed.



What is Kernel in Operating System?

Kernel Types

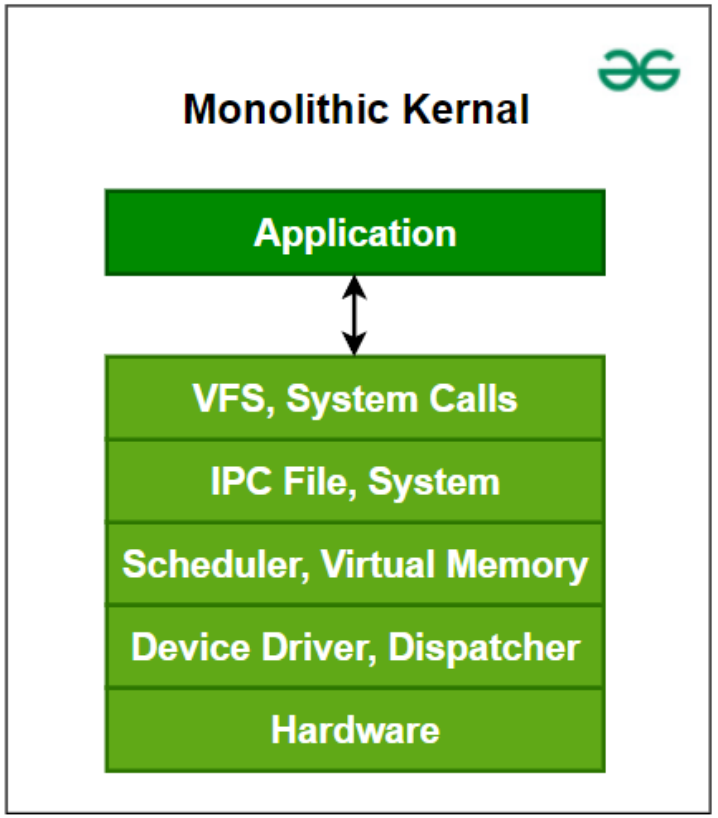
•**Monolithic Kernel**: All services run in the same space; efficient but less stable (e.g., Linux, MS-DOS).

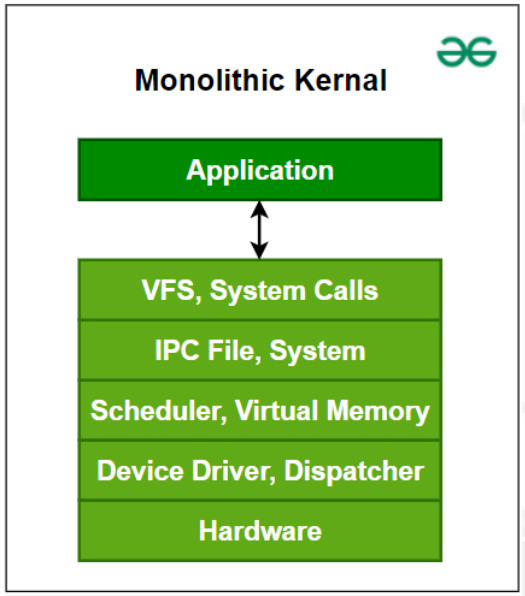
Monolithic Kernel

All core services (like memory management, file system, device drivers) run in kernel space.

•**Examples:**

- **Linux** (e.g., Ubuntu, CentOS, Fedora)
- **MS-DOS**
- **Unix** (traditional)
- **Solaris (earlier versions)**





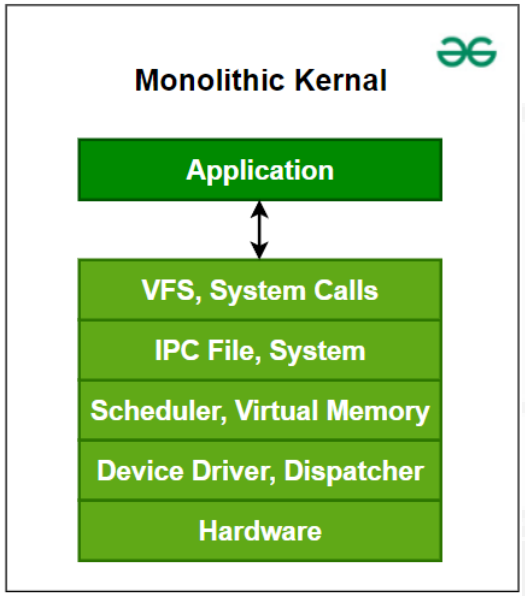
Advantages of Monolithic Kernel

•One of the major advantages of having a monolithic kernel is that it provides **CPU scheduling, memory management, file management, and other operating system functions through system calls.**

•The other one is that it is a single large process running entirely in a single address space.

•It is a single static binary file. **Examples** of some Monolithic Kernel-based OSs are Unix, Linux, Open VMS, XTS-400, z/TPF.

•No need for complex inter-process communication (IPC), which speeds up system call execution.

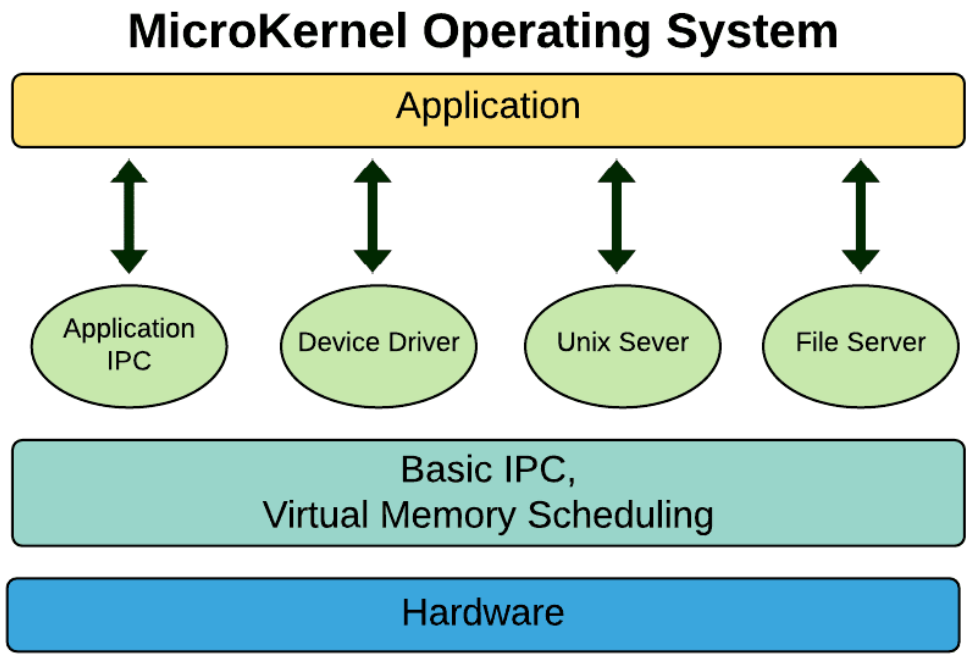


Disadvantages of Monolithic Kernel

- **Stability Issues**: One of the major disadvantages of a monolithic kernel is that if anyone service fails it leads to an entire system failure.
- **Lack of Modularity**: If the user has to add any new service. The user needs to modify the entire operating system.
- **Security Risks**: A bug or vulnerability in any service can affect the entire system since all services run in kernel mode.
- **Large Size**: The kernel can become very large and complex as more services are added.

Kernel Types

•**Microkernel**: Only essential services run in the kernel; others run as user-space servers, offering better **robustness** and **configurability** (e.g., Minix 3, QNX).



3. Hybrid Kernel

Combines features of both monolithic and microkernel—runs some services in kernel space for performance but maintains modularity.

•**Examples:**

- **Windows NT Kernel** (used in Windows XP, 10, 11)
- **macOS (XNU Kernel)** – hybrid of Mach and BSD
- **ReactOS**

4. Exokernel

Gives minimal abstraction; applications manage hardware resources directly.

•**Examples:**

- **MIT Exokernel** (experimental/research OS)
- **Nemesis OS** (research project)

5. Nanokernel / Picokernel (very minimal, niche)

•**Examples:**

- **L4 Microkernel family**
- **CodeZero** (ARM nanokernel)

Aspect	Microkernel	Monolithic Kernel
Size	Small	Large
Execution	Slower (message passing)	Faster (direct calls)
Extensibility	Easy to extend	Harder to extend
Security	More secure (less in kernel space)	Less secure (more code in kernel space)
Crash Impact	Limited to service	Can crash the whole system
Code Complexity	More code required	Less code required
Examples	L4Linux, macOS	Windows, Linux, BSD
Portability	High (services in user space)	Lower (hardware-dependent)
Communication	Message passing	Direct function calls
Performance	Lower	Higher

Real-Time Operating System (RTOS) based Embedded System Design

2. Types of Operating Systems

GPOS

RTOS

General Purpose Operating System (GPOS)

1.Definition:

General Purpose Operating Systems (GPOS) are **operating systems designed for general computing environments**.

2.Kernel Characteristics:

- The kernel is **generalized**.
- It includes a **wide range of services** to support execution of **various generic applications**.

3.Behavior:

- GPOS are often **non-deterministic** in nature.
- They may introduce **random delays** during execution.
- This can result in **unpredictable responsiveness** of applications.

4.Deployment:

- Used in systems **where deterministic behaviour is not crucial**.
- Commonly deployed in **personal computers, desktops**, and similar platforms.

5.Examples:

- **Windows XP**
- **MS-DOS**

Real-Time Operating System (RTOS)

1. Definition of Real-Time

There is no universal definition of "real-time" in the context of Operating Systems. However, in general:

- **Real-time** implies **deterministic timing behavior**.
- RTOS ensures that OS services execute in **predictable and bounded time**, regardless of system load

2. Key Features of RTOS:

- Deterministic and predictable performance.
- Time-critical resource allocation.
- Implements **policies** and **rules** for consistent task execution.

Examples: Windows CE, QNX, VxWorks, MicroC/OS-II.

Real-Time Kernel

The **kernel** in an RTOS is minimal and optimized for:

1.Task/Process Management
2.Task Scheduling
3.Synchronization
4.Error/Exception Handling
5.Memory Management
6.Interrupt Handling
7.Time Management

1. Task/Process Management

Handles:

- Creating and deleting tasks
- Allocating memory and resources
- Maintaining **Task Control Block (TCB)**

TCB Includes:

- Task ID
- Task State (e.g., Ready, Running)
- Task Type (Hard, Soft, Background)
- Priority
- Context Pointer
- Memory Pointers
- Resource Pointers (e.g., semaphores)
- Pointers to other TCBs

2. Task/Process Scheduling

Managed by a **Scheduler**.

- Uses algorithms to assign CPU time to tasks.
- Ensures **efficient, optimal, and deterministic** task execution.
- Types of scheduling discussed in later chapters (e.g., Round Robin, Rate Monotonic).

3. Synchronization

•Manages access to **shared resources**.
•Enables **inter-task communication**.
•Uses **semaphores, mutexes, and message queues**.

4. Error/Exception Handling

Handles:
•Kernel-level exceptions: e.g., deadlocks
•Task-level exceptions: e.g., timeouts
Mechanisms:
•GetLastError() (Windows CE API)
•**Watchdog Timers**: prevent hang-ups from unresponsive tasks by enforcing timeout policies.

6. Memory Management

•**Block-based allocation** for deterministic timing.
•Uses **fixed-size memory blocks** stored in a **Free Buffer Queue**.
•Avoids:
- Memory fragmentation
- Garbage collection overhead

Trade-offs:
•Suboptimal memory use
•Limited block size options
Some RTOSs support **Virtual Memory** when secondary storage is available.

6. Interrupt Handling

•Interrupts enable **real-time responsiveness**.

•Two types:

- **Synchronous** (e.g., divide-by-zero): Occur during task execution.
- **Asynchronous** (e.g., I/O interrupt): Occur anytime.

Key Concepts:

•**Interrupt Service Routine (ISR)**

•**Context switching** (especially for asynchronous interrupts)

•**Interrupt priority and nesting** (higher-priority interrupt can preempt lower-priority ISR)

7. Time Management

Uses **high-resolution RTC** to generate regular **Timer Ticks**.

Timer Tick is:

•Used as kernel's time base (e.g., 1 μs or 1 ms interval).

•Drives the **System Time Register**.

Functions of Timer Interrupt Handler:

•Save current task context

•Update system time

•Manage software timers

•Trigger periodic tasks

•Invoke scheduler

•Remove terminated tasks

•Load next task context

📊 Calculation Example from the Time Management Section

Let's assume:

- The **System Time Register** is 32 bits wide (i.e., it can count up to 2^{32}).
- The **Timer tick** (interrupt) occurs every 1 microsecond or 1 millisecond.

Case 1: Timer Tick Interval = 1 Microsecond (µs)

$$\frac{2^{32} \times 10^{-6}}{24 \times 60 \times 60} \approx 49700 \text{ days} \approx 1.19 \text{ hours}$$

〓 The system register will overflow in ~1.19 hours.

Case 2: Timer Tick Interval = 1 Millisecond (ms)

$$\frac{2^{32} \times 10^{-3}}{24 \times 60 \times 60} \approx 497 \text{ days} \approx 50 \text{ days}$$

〓 The system register will overflow in ~50 days.

Optional Kernel Services

Some RTOSs allow **custom kernel builds** where the developer selects only necessary services (e.g., Windows CE).

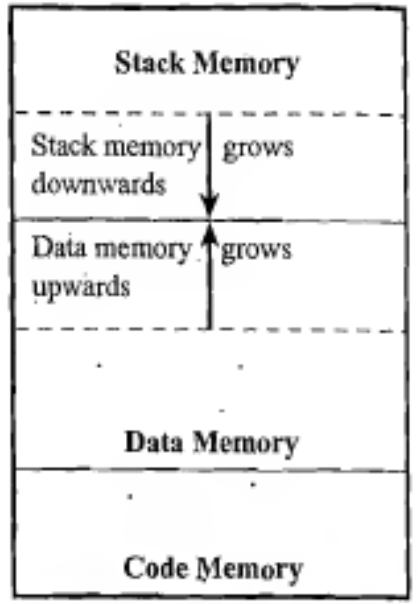
Additional features may include:

- File systems
- Networking
- I/O management

3. Tasks, Process and Threads,

Tasks, Processes, and Threads in Operating Systems:

- **Task/Process/Job**: All refer to the same—**a program in execution**.
- A **process** requires CPU, memory, I/O, etc., and is **sequential** in executio
- A **process** has three memory segments:
  - **Stack**: Holds local variables.
  - **Data**: Holds global data.
  - **Code**: Holds program instructions.



Memory Organization of Process

Process Life Cycle

- **States**:
  - **Created**: Process is created.
  - **Ready**: Waiting for CPU.
  - **Running**: Currently executing.
  - **Blocked/Wait**: Waiting for a resource or event.
  - **Completed**: Finished execution.
- **State Transitions** occur due to events like I/O wait, CPU scheduling, etc.

Example OS variations:

- **VxWorks**: States include READY, PEND, DELAY, SUSPEND.
- **MicroC/OS-II**: States include DORMANT, READY, RUNNING, WAITING, INTERRUPTED.

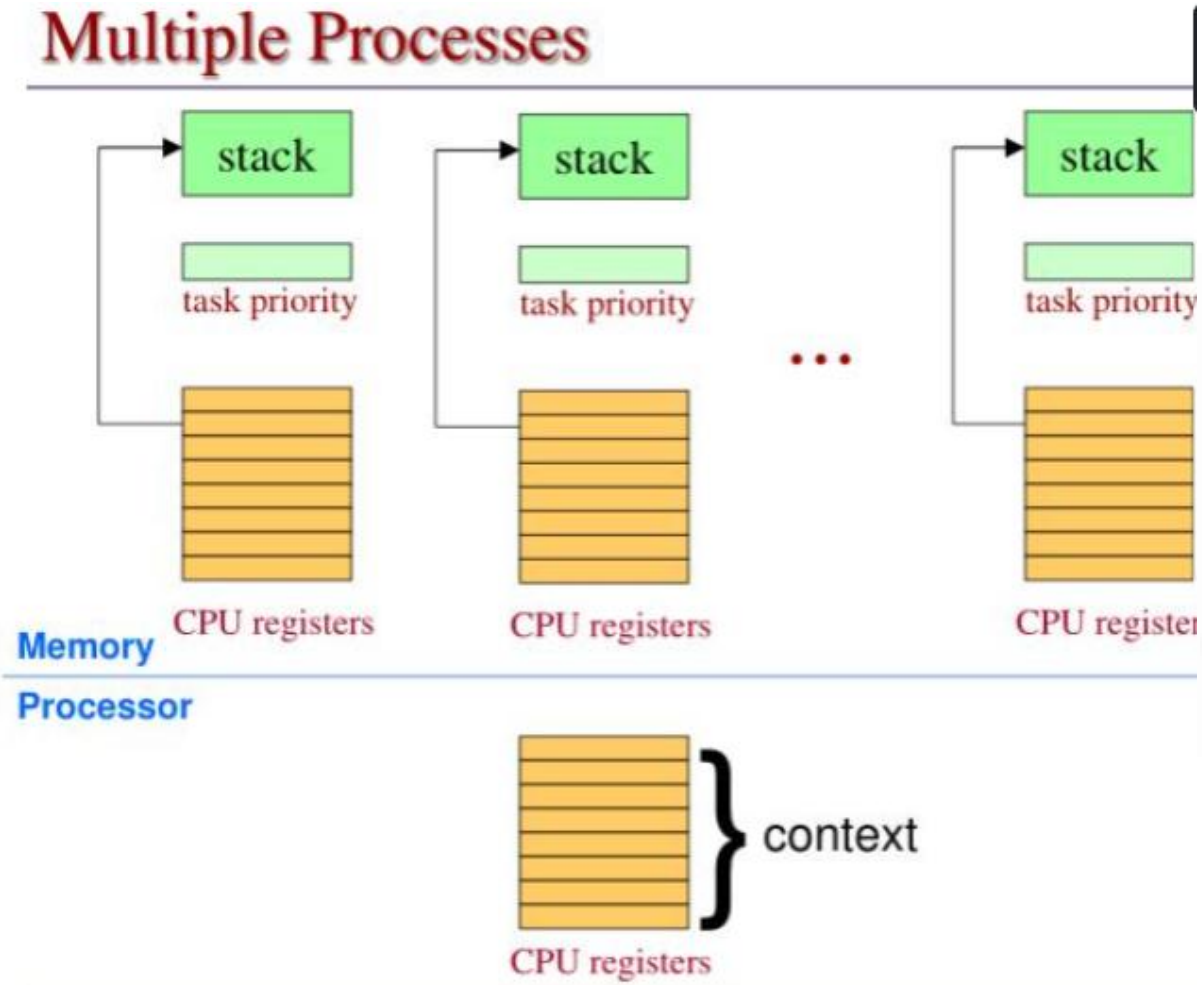
Process Management:

•Involves creating, loading, assigning memory, setting up the **Process Control Block (PCB)**, and terminating the process.

Process : A process which inherits all the properties of a CPU is called Virtual Processor

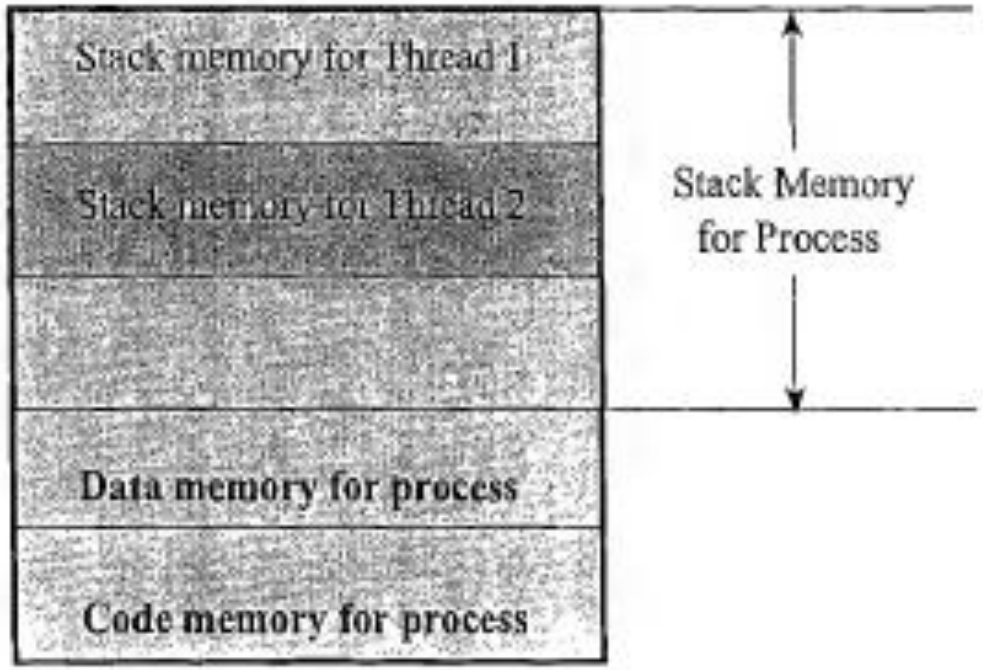
Structure

- **Stack (Stack pointer)**
- Used for function calls, local variables, and control flow.
- Stack pointer points to the top of the stack.
- **Working registers**
- Temporary storage used during execution (like general-purpose registers).
- **Status registers**
- Hold flags that reflect the outcome of operations (e.g., zero flag, carry flag).
- **Program Counter (PC)**
- Holds the address of the next instruction to be executed.


Multiple Processes
stack
task priority
CPU registers
stack
task priority
CPU registers
stack
task priority
CPU registe
Memory
Processor
context
CPU registers

Threads and Multithreading:

- A **thread** is a lightweight process and the smallest unit of execution.
- **Multithreading** splits a process into sub-tasks (threads) to improve:
  - CPU utilization
  - Execution speed
  - Memory efficiency
- Threads share **code**, **data**, and **heap**, but each has its own **stack**, **program counter**, and **registers**.



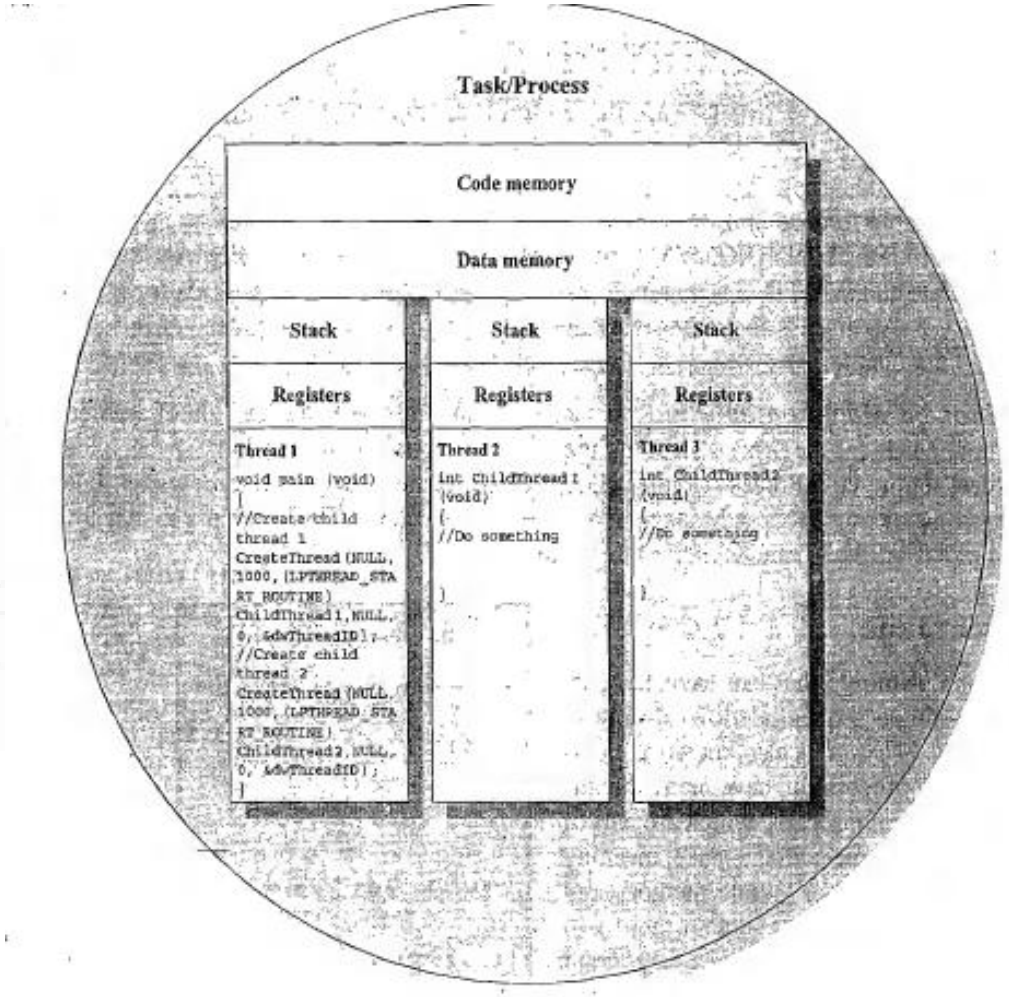
Memory organisation of a Process and its associated Threads

Portable Operating System Interface.

The POSIX.4 standard deals with the Real-Time extensions and POSIX.4a standard deals with thread extensions. The POSIX standard library for thread creation and management is 'Pthreads'. 'Pthreads' library defines the set of **POSIX thread creation and management functions in 'C' language**

Thread Standards

- **POSIX Threads (Pthreads)**:
  - **pthread_create**: Creates a new thread.
  - **pthread_join**: Waits for a thread to finish.
  - Returns **0 on success**.



Process with multi-threads

POSIX multithreaded program in C

Part	**Description**
#include <pthread.h>	Includes the POSIX thread (pthreads) library.
pthread_t tcb;	Declares a thread control block tcb for the new thread.
pthread_create(...)	Creates a new thread that runs new_thread().
new_thread(void *args)	Function executed by the new thread, prints a message 5 times.
pthread_join(tcb, NULL)	Waits for the new thread to finish before main() terminates.
Delay loops	These are busy-wait loops to simulate a pause. You can replace them with sleep(1); for actual delays.
printf(...)	Prints the message indicating which thread is executing.

POSIX multithreaded program in C **using pthread_create() and pthread_join()**

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

// Function executed by the new thread
void *new_thread(void *thread_args) {
    int i, j;
    for (j = 0; j < 5; j++) {
        printf("Hello I'm in new thread\n");

        // Busy-wait delay (can be replaced by sleep(1) if needed)
        for (i = 0; i < 1000000; i++);
    }
    return NULL;
}
```

```
int main(void) {
    int i, j;
    pthread_t tcb;

    // Create a new thread to run the new_thread function
    if (pthread_create(&tcb, NULL, new_thread, NULL)) {
        printf("Error in creating new thread\n");
        return -1;
    }

    // Main thread also prints its message 5 times
    for (j = 0; j < 5; j++) {
        printf("Hello I'm in main thread\n");

        // Busy-wait delay
        for (i = 0; i < 1000000; i++);
    }

    // Wait for the new thread to finish
    if (pthread_join(tcb, NULL)) {
        printf("Error in joining thread\n");
        return -1;
    }

    return 0;
}
```

Win32 multithreaded application using the Windows API to create a counter in both the main thread and a child thread.

The program includes **a 500 ms delay between successive print statements.**

- **Solution:**
- **Parallel Execution**: Both main and child threads run *simultaneously*.
- **Synchronization**: WaitForSingleObject() ensures the main thread waits for the child to finish.
- **Error Handling**: If thread creation fails, it prints the error number using GetLastError().

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

// Function executed by the new thread
void *new_thread(void *thread_args) {
    int i, j;
    for (j = 0; j < 5; j++) {
        printf("Hello I'm in new thread\n");

        // Busy-wait delay (can be replaced by sleep(1) if needed)
        for (i = 0; i < 1000000; i++);
    }
    return NULL;
}
```

```
int main(void) {
    int i, j;
    pthread_t tcb;

    // Create a new thread to run the new_thread function
    if (pthread_create(&tcb, NULL, new_thread, NULL)) {
        printf("Error in creating new thread\n");
        return -1;
    }
```

```
    // Main thread also prints its message 5 times
    for (j = 0; j < 5; j++) {
        printf("Hello I'm in main thread\n");

        // Busy-wait delay
        for (i = 0; i < 1000000; i++);
    }

    // Wait for the new thread to finish
    if (pthread_join(tcb, NULL)) {
        printf("Error in joining thread\n");
        return -1;
    }

    return 0;
}
```

Part	Description
#include <windows.h>	Windows-specific functions (like CreateThread, Sleep, etc.).
CreateThread(...)	Creates a new thread and runs ChildThread() in parallel.
ChildThread(...)	The function executed by the new thread, prints numbers 0–10 with 500ms delay.
Sleep(500)	Causes the current thread to pause for 500 milliseconds.
WaitForSingleObject(hThread, INFINITE)	Waits for the child thread to complete before continuing.
CloseHandle(hThread)	Closes the handle to the child thread to free resources.
printf(...)	Prints the output from each thread.

Java Threads

Java supports **multithreaded programming**, which allows concurrent execution of two or more parts of a program for maximum utilization of CPU. In Java, multithreading is achieved using the Thread class, which is part of the java.lang package.

You must import this package to create and use threads.

There are two ways to create a thread in Java:

- By extending the Thread class
- By implementing the Runnable interface

1. Creating a Thread by Extending Thread Class

This method involves creating a new class that extends Thread and overrides its **run() method.**

Sample Code:

```
public class MyThread extends Thread
 {
    public void run() {
        System.out.println("Hello from MyThread!");
    }

    public static void main(String[] args)
{
        new MyThread().start();
}
}
```

Understanding Java Thread (Extending Thread Class) in 4 Steps:

1.Create a Thread Class

➤ Define a new class that extends the Thread class.

➤ Override the run() method to define the task to be performed by the thread.

2.Start the Thread

➤ In the main() method, create an object of your new thread class.

➤ Call the start() method (not run() directly) to begin execution.

3.Thread Enters Ready State

➤ The start() method places the thread in the **Ready queue**, waiting for CPU allocation by the **scheduler**.

4.Thread Execution by Scheduler

➤ The Java scheduler picks threads from the ready queue based on priority and availability.

➤ The thread executes the code inside the run() method.

Output

Hello from MyThread!

The run() method gets executed in a separate thread when start() is called.

Thread Pre-emption

•Temporarily stops a running thread to switch CPU to another.
•Controlled by the **Operating System (OS)**.
•Enables **CPU time sharing** via **Thread Context Switching**.

Types of Threads

1.User-Level Threads

1. Managed by the application (not the OS).
2. OS sees them as **one single thread**.
3. **Non-preemptive** at OS level.
4. Very **fast** context switching (no system calls).

2.Kernel-Level Threads

1. Managed by the **Operating System**.
2. OS treats each thread individually.
3. **Preemptive** — OS can interrupt and switch between them.
4. Context switching is **slower** due to kernel overhead.

Thread Binding Models

1.Many-to-One

1. Many user threads mapped to **one** kernel thread.
2. No parallelism; threads share same system thread.
3. Ex: Solaris Green Threads, GNU Portable Threads.

2.One-to-One

1. Each user thread maps to **one** kernel thread.
2. True concurrency, better CPU utilization.
3. Ex: Linux, Windows XP/NT/2000.

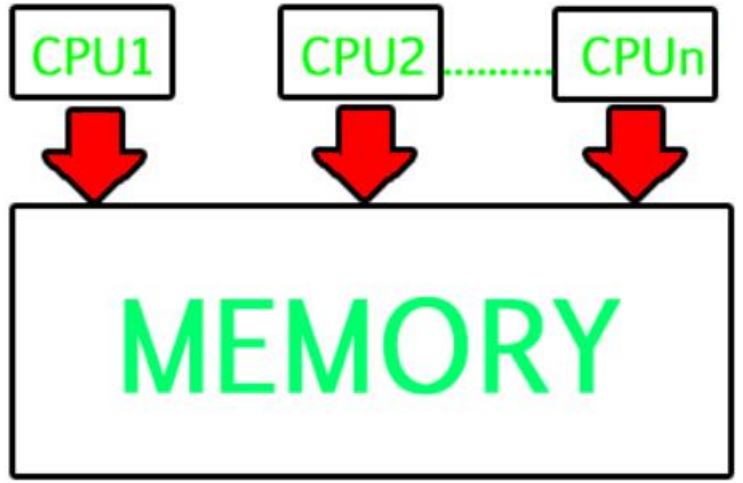
3.Many-to-Many

1. Many user threads mapped to many kernel threads.
2. Combines flexibility with concurrency.
3. Ex: Windows NT/2000 with ThreadFibre.

Aspect	Thread	Process
Definition	A thread is a single unit of execution and is part of a process.	A process is a program in execution and contains one or more threads.
Memory	Thread does not have its own data memory and heap; it shares memory with other threads of the same process.	Process has its own data memory, heap, and stack.
Independence	Thread cannot live independently; it lives within the process.	A process contains at least one thread and is independent.
Context Switching	Threads of the same process share code and memory, making context switching fast and efficient.	Processes do not share memory and require more overhead for context switching.
Overhead	Threads use fewer resources (lightweight).	Processes use more resources (heavyweight).

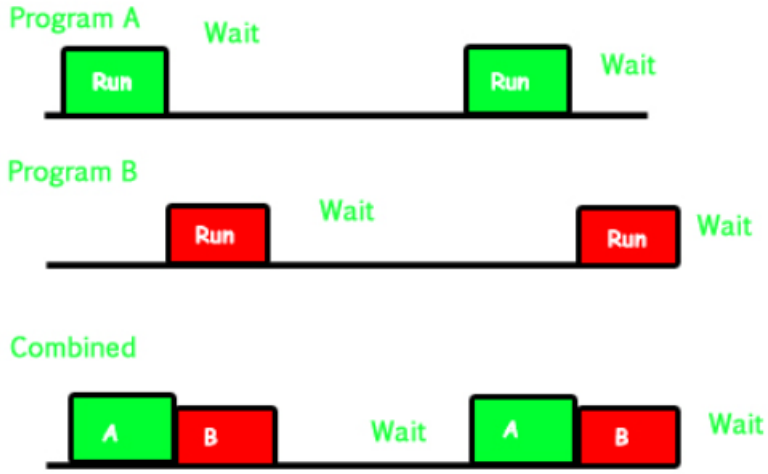
Multiprocessing

- The ability of a system to **execute multiple processes simultaneously** using **multiple CPUs**.
- Used in **multiprocessor systems**.



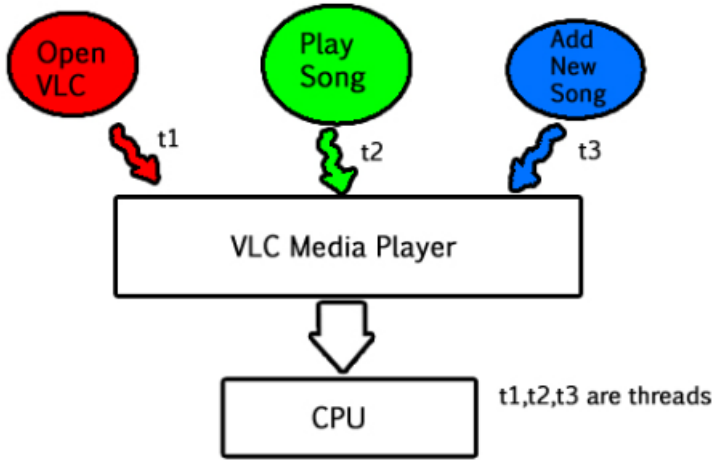
Multiprogramming

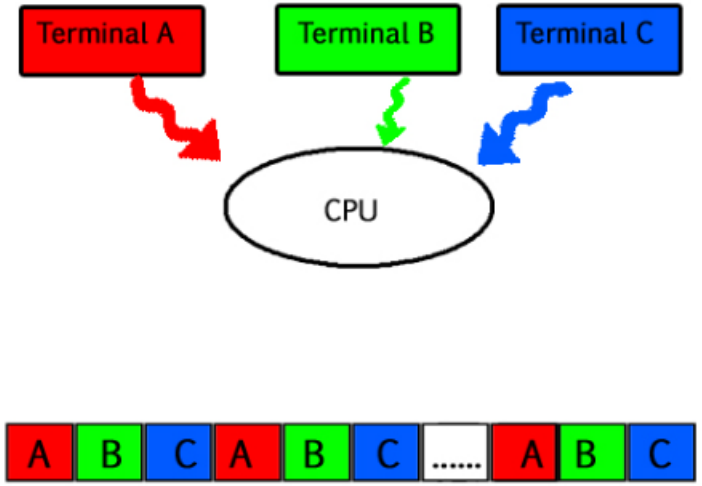
- Multiple programs are **kept in memory** so the CPU can switch between them.
- **Only one program executes at a time** in a uniprocessor system, but switching increases efficiency.



Multitasking

- OS handles **multiple tasks** by quickly switching between them.
- Creates an **illusion of parallel execution** even on a single CPU.
- Involves:
  - **Context Switching**: Switching between processes.
  - **Context Saving**: Saving the state of the current process.
  - **Context Retrieval**: Loading the state of the next process.





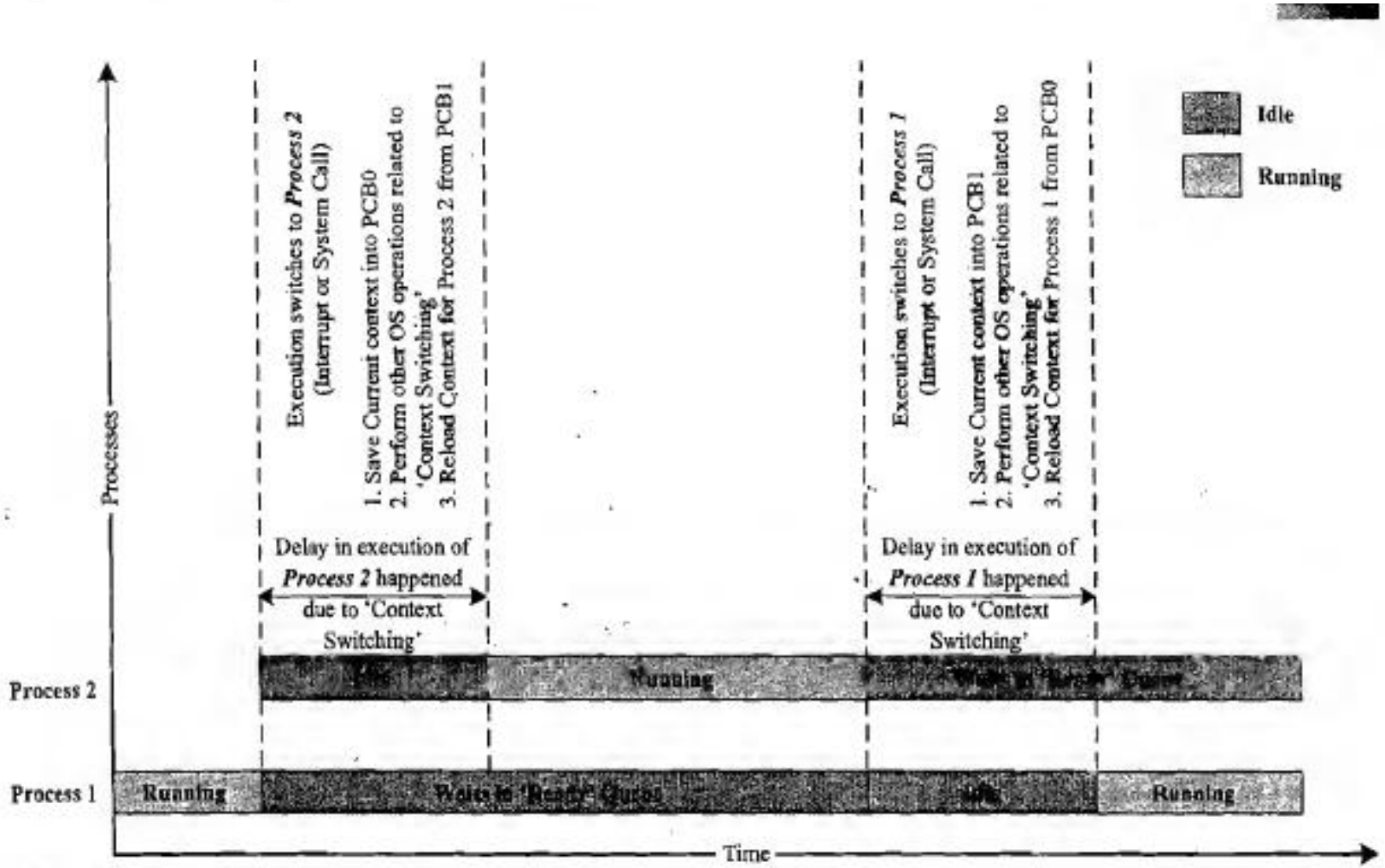


Fig. 10.11 Context switching

Types of Multitasking

Type	Description
Co-operative	Task releases CPU **voluntarily**; one task may dominate the CPU.
Preemptive	OS **forcibly switches** tasks based on **time or priority**.
Non-preemptive	Task runs until it finishes or waits for I/O; **no forced switching**.

Analogy:

Like **juggling**, multitasking gives the illusion of handling multiple balls (tasks) at once, though only one is in hand (CPU) at a time.

Task Scheduling in Multitasking Systems

- **Multitasking** requires a mechanism to decide which task gets CPU time — this is called **task/process scheduling**.
- The **scheduler** (a kernel service) implements **scheduling policies** using algorithms to manage task execution.

When Scheduling Decisions Occur:

1. **Running → Ready**: Preempted by another task.
2. **Running → Blocked/Wait**: Waiting for I/O or resource.
3. **Blocked/Wait → Ready**: May preempt if priority-based.
4. **Process Completion**: CPU assigned to another task.

Scheduling Criteria:

- **CPU Utilization**: Maximize usage.
- **Throughput**: Number of tasks completed per time unit.
- **Turnaround Time**: Total time to execute a task.
- **Waiting Time**: Time in Ready queue.
- **Response Time**: Time from submission to first response.

Process Queues:

- **Job Queue**: All system processes.
- **Ready Queue**: Tasks ready for CPU.
- **Device Queue**: Tasks waiting for I/O.

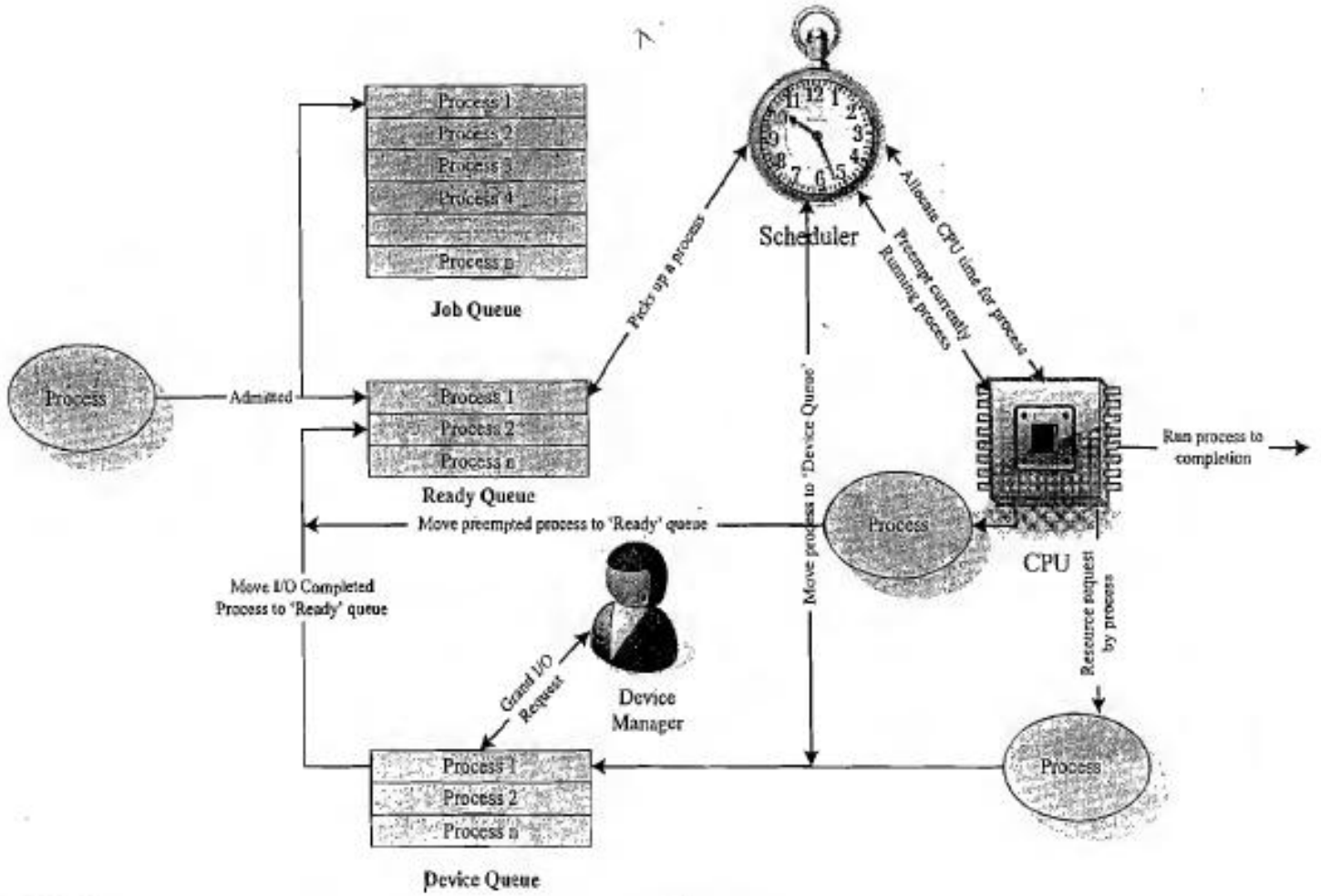


Fig. 10.12 **Illustration of process transition through various queues**

Non-Preemptive Scheduling:

•CPU is released only when a task finishes or waits.

•Types:

- **FCFS (FIFO)**: First entered, first executed.
- **LCFS (LIFO)**: Last entered, first executed.
- **Priority-based**: High-priority tasks served first.
- **SJF (Shortest Job First)**: Task with shortest execution time first.

Preemptive Scheduling:

•CPU can be forcibly taken from a running task.

•Types:

- **Preemptive SJF / SRT (Shortest Remaining Time)**: Compares new task's time with remaining time of current task.
- **Priority-based**: High-priority tasks preempt current ones.

Key Differences:

- **Non-preemptive**: Task runs until done or waiting.
- **Preemptive**: Scheduler can interrupt and switch tasks.

Example

Three processes with process IDs **P1**, **P2**, and **P3** have estimated completion times of 10 ms, 5 ms, and 7 ms respectively. All three processes enter the ready queue at the same time.

1.Calculate the **Waiting Time (WT)** and **Turn Around Time (TAT)** for each process and the **average waiting time** and **average turnaround time** assuming the processes enter the ready queue in the order **P1, P2, P3**.

2.Repeat the calculation if the processes enter the ready queue in the order **P2, P1, P3**.

Assume there is no I/O waiting and the CPU is available when the first process arrives.

Solution

Process	Execution Time (ms)
P1	10
P2	5
P3	7

Case 1: Order of arrival = P1, P2, P3

Waiting Time (WT):

- WT(P1) = 0 ms (P1 starts immediately)
- WT(P2) = Execution time of P1 = 10 ms
- WT(P3) = Execution time of P1 + P2 = 10 + 5 = 15 ms

Turn Around Time (TAT) = WT + Execution Time

- TAT(P1) = 0 + 10 = 10 ms
- TAT(P2) = 10 + 5 = 15 ms
- TAT(P3) = 15 + 7 = 22 ms

Average Waiting Time:

$$\frac{0+10+15}{3} = \frac{25}{3} = 8.33 \text{ ms}$$

Average Turn Around Time:

$$\frac{10+15+22}{3} = \frac{47}{3} = 15.66 \text{ ms}$$

Case 2: Order of arrival = P2, P1, P3

Waiting Time (WT):

- WT(P2) = 0 ms (P2 starts immediately)
- WT(P1) = Execution time of P2 = 5 ms
- WT(P3) = Execution time of P2 + P1 = 5 + 10 = 15 ms

Turn Around Time (TAT):

- TAT(P2) = 0 + 5 = 5 ms
- TAT(P1) = 5 + 10 = 15 ms
- TAT(P3) = 15 + 7 = 22 ms

Average Waiting Time:

$$\frac{0+5+15}{3} = \frac{20}{3} = 6.66 \text{ ms}$$

Average Turn Around Time:

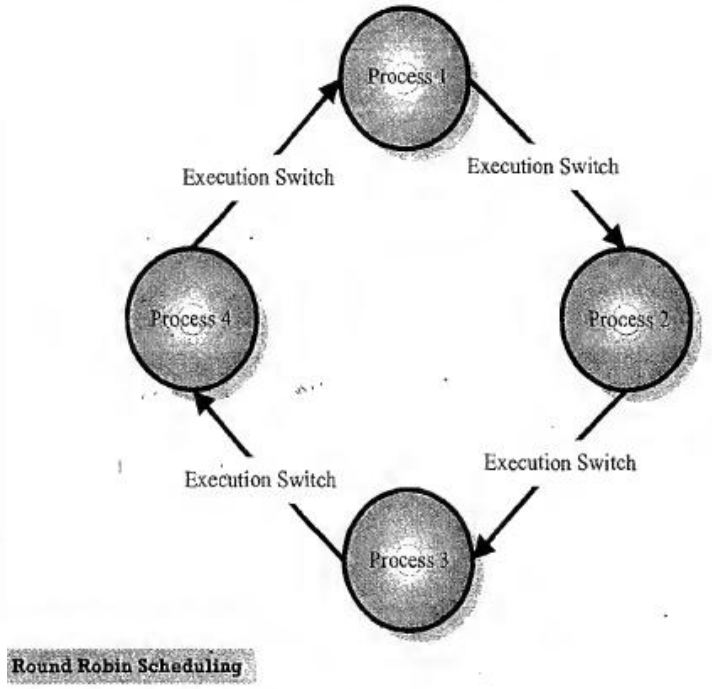
$$\frac{5+15+22}{3} = \frac{42}{3} = 14 \text{ ms}$$

Summary Table:

Arrival Order	Average Waiting Time (ms)	Average Turnaround Time (ms)
P1, P2, P3	8.33	15.66
P2, P1, P3	6.66	14

Round Robin (RR) Scheduling

- Round Robin scheduling gives each process in the Ready queue an equal, fixed time slice for execution.
- The scheduler runs the first process for the time slice; if it finishes early, it moves to the next process.
- If not, the running process is preempted and placed at the queue's end, and the next process runs. This cycle repeats in a circular manner.
- It's similar to First-Come-First-Served (FCFS) but includes time-based preemption to share CPU fairly. The time slice is set by the OS timer and may be configurable.
- **Round Robin** ensures fair and efficient CPU sharing, commonly used in time-sharing systems.



Round Robin Scheduling

Priority-Based Scheduling

The **Priority-Based Preemptive Scheduling Algorithm** functions similarly to the **Non-Preemptive Priority-Based Scheduling Algorithm**, with the key difference being in how tasks are switched during execution.

In **Preemptive Scheduling**, **any high-priority process** entering the **Ready Queue** is **immediately scheduled** for execution, **interrupting** the currently running lower-priority process.

In contrast, in **Non-Preemptive Scheduling**, a high-priority process entering the Ready Queue will only be scheduled **after the currently executing process finishes** its execution or **voluntarily releases** the CPU.

The method used to assign and represent priorities in **Preemptive Scheduling** is the same as that adopted in **Non-Preemptive Multitasking**

ATME College of Engineering

Department of EEE
Emitting Elite Energy

Thank You