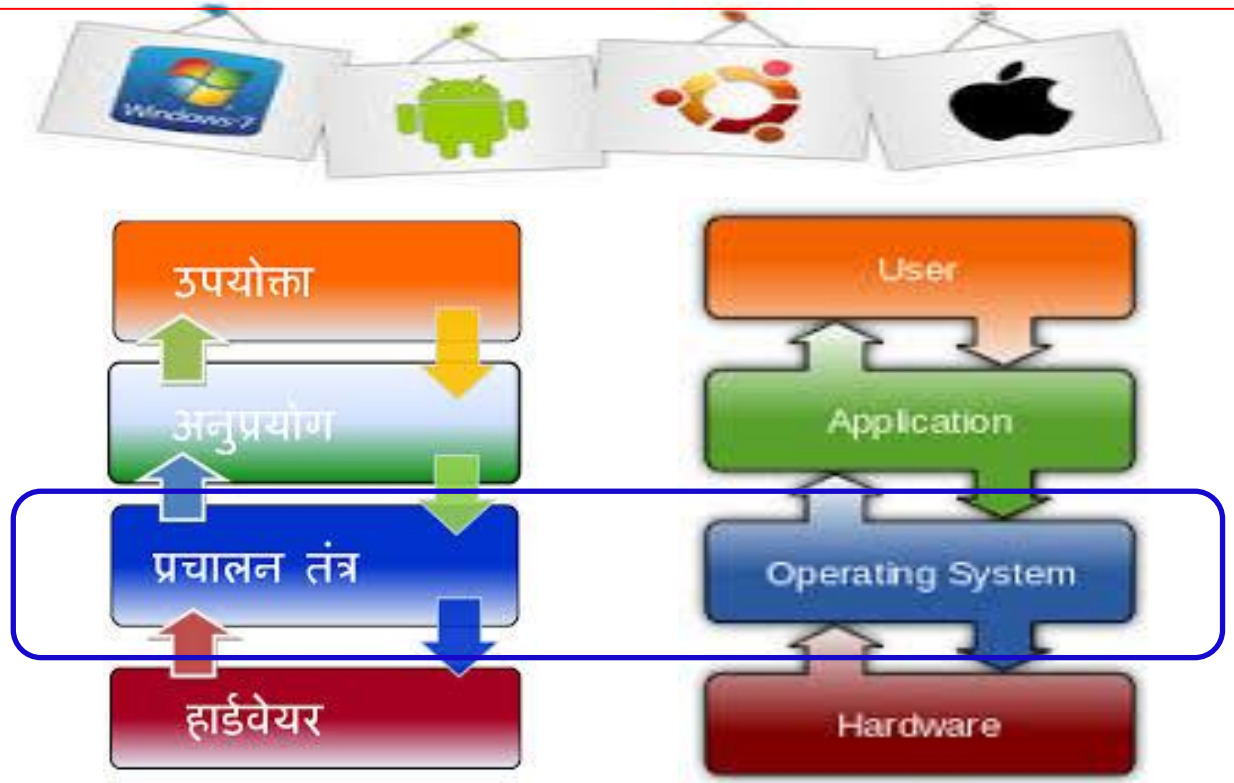
Real Time Operating System



Outline

- System component
- Why Operating System require ???
- What is Operating System ???
- The Kernel
  - Kernel Space and User Space
  - Monolithic kernel v/s Microkernel
- Types of Operating Systems (GPOS & RTOS)
- Why RTOS ???
- The real-time kernel
- Task Scheduling Algorithms

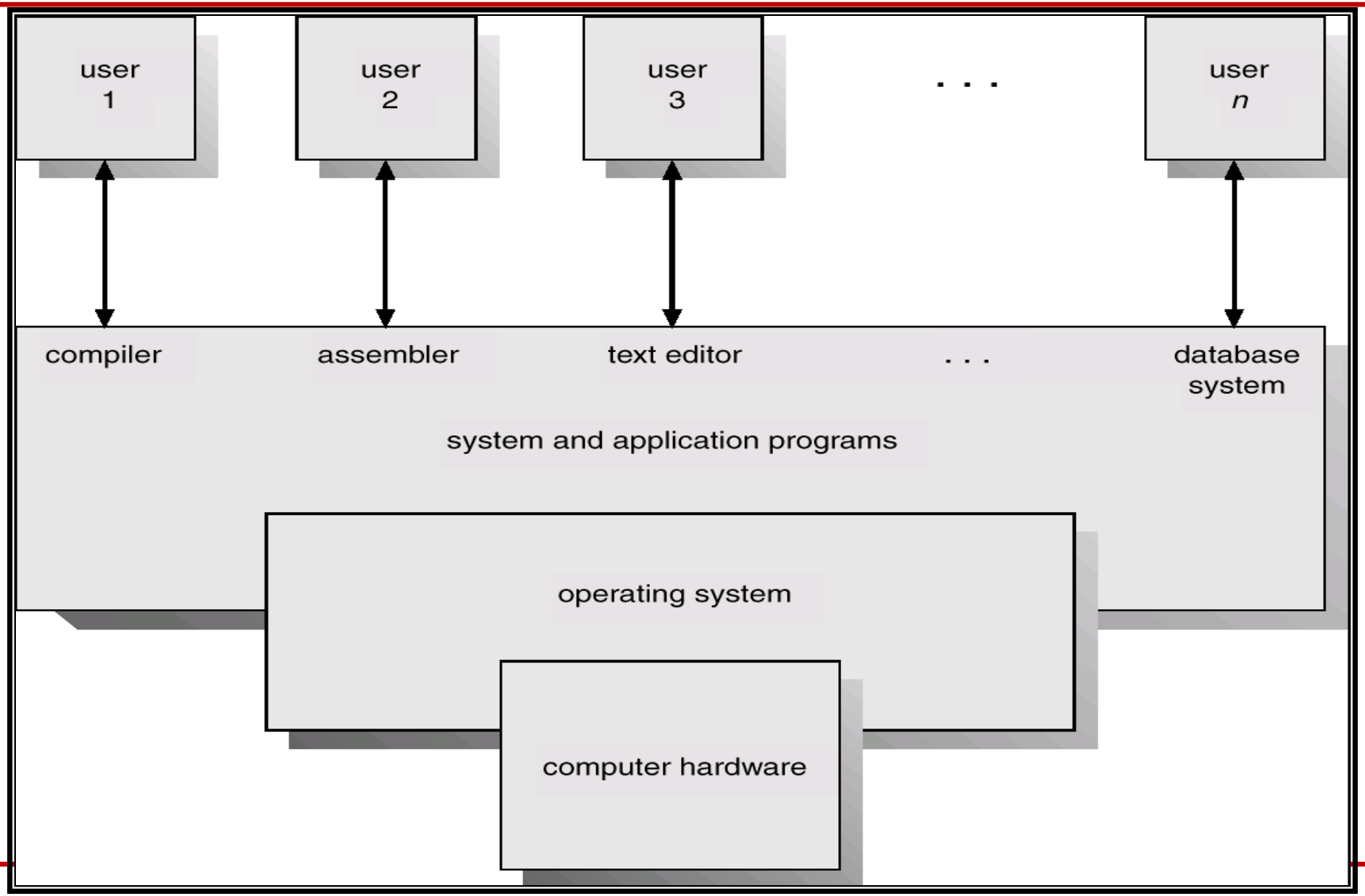
Basic Hardware Elements

- **Processor**
  - controls operation of the system and data processing function
- **Memory**
  - Stores data and programs
- **I/O modules**
  - move data between computer and external environment
- **System Bus**
  - communication among processors, memory, and I/O module

Computer System Components

1. **Hardware**
   - computing resources (CPU, memory, I/O devices).
2. **Operating system**
   - controls and coordinates the use of the hardware among the various user application programs
3. **Applications programs**
   - Solves user's problems (.exe)
4. **System Program**
   - manages Computer operations (Compiler, Editor, interpreter)
5. **Users**
   - people, machines, other computers

Abstract View of System Components



Super Loop

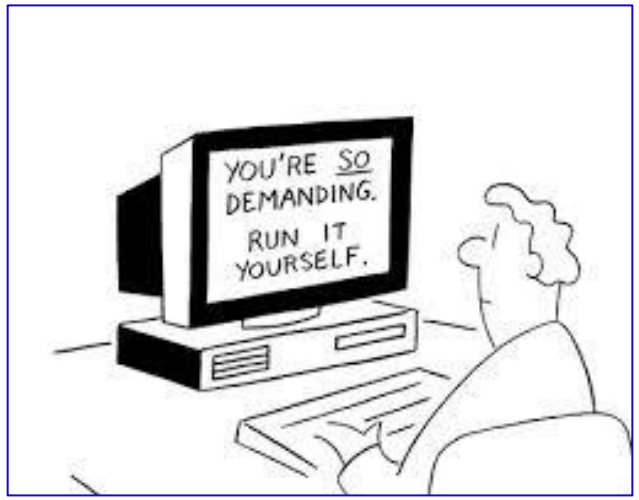
- In embedded systems, especially those with real-time constraints, efficient code execution is critical. One common architectural pattern used to achieve this efficiency is called the "super loop" or "main loop" architecture.
- The super loop architecture is a simple and straightforward design where the entire program's functionality is contained within a single loop, often referred to as the "super loop" or "main loop." This loop continuously iterates, executing tasks sequentially, and handles all system activities.
- It is essential to ensure that the loop's execution time doesn't exceed the system's real-time requirements and that all critical tasks are handled promptly within each iteration of the loop.

Example of Super Loop

```
int main() {
    initialize_system(); // Initialize hardware and peripherals

    while (1) { // Super loop
        // Task 1: Read sensor data
        read_sensors();

        // Task 2: Process sensor data
        process_data();

        // Task 3: Control actuators
        control_actuators();

        // Task 4: Handle communication
        handle_communication();

        // Task 5: Handle other system activities
        handle_other_activities();

        // Add any necessary delays or timing control to meet system requirements
        // delay();
    }

    return 0;
}
```

Why OS is required for Embedded System?

- Super-loop
- Draw back of super-loop
  - *Executes task sequentially in order, at regular interval, and executes in non-real time.*
  - *No of task increases ⇔ time interval at which the task get the service is also increases*
  - *Latency depends on the external events also*
  - *Priority of task execution is also fixed*
  - *Solution to this time critical responses for the task/events ??????*

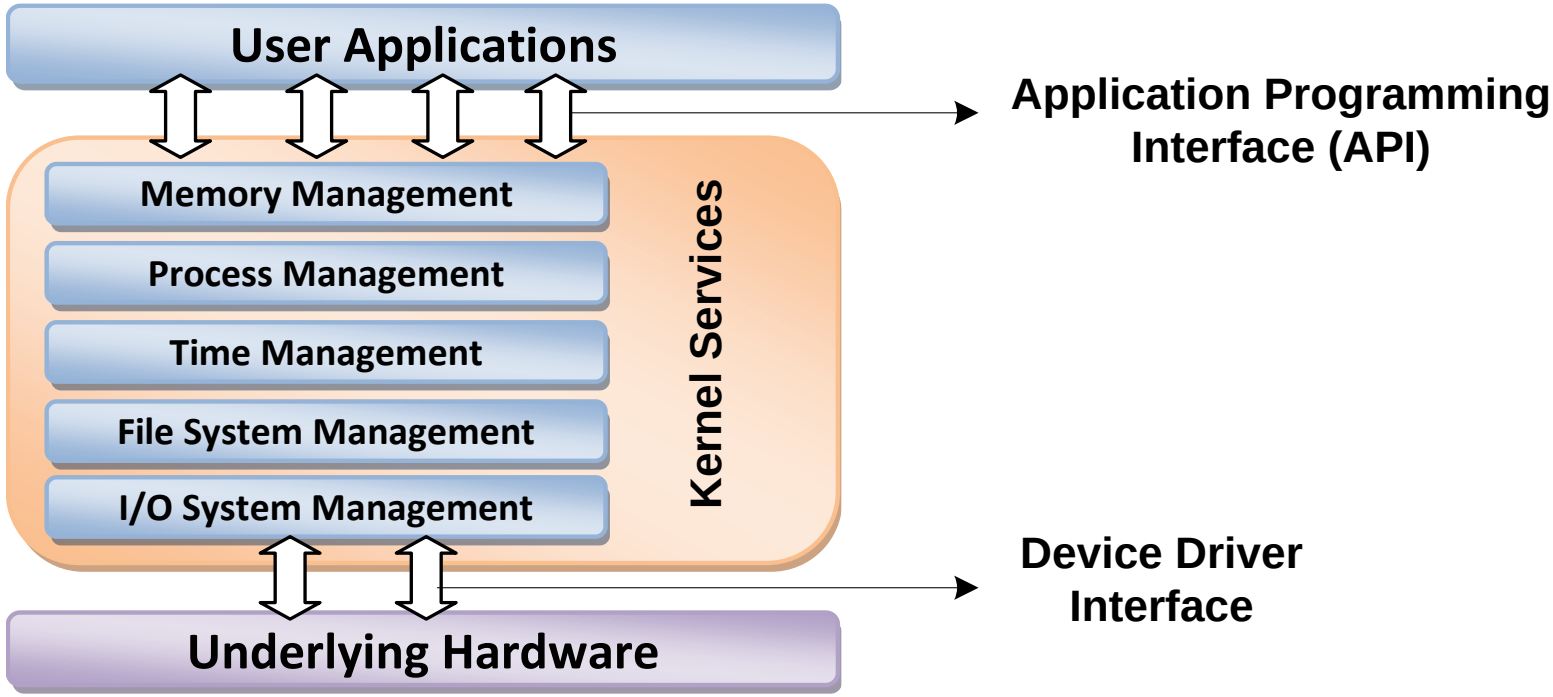


Why OS is required for Embedded System?

- Assign Priority to task and execute the high Priority Task
- Dynamically change the Priority of task if needed.
- Schedule the execution of task based on priorities.
- Switch the execution of task when a task is waiting for an external event or a system including I/O device operation.

What is Operating System

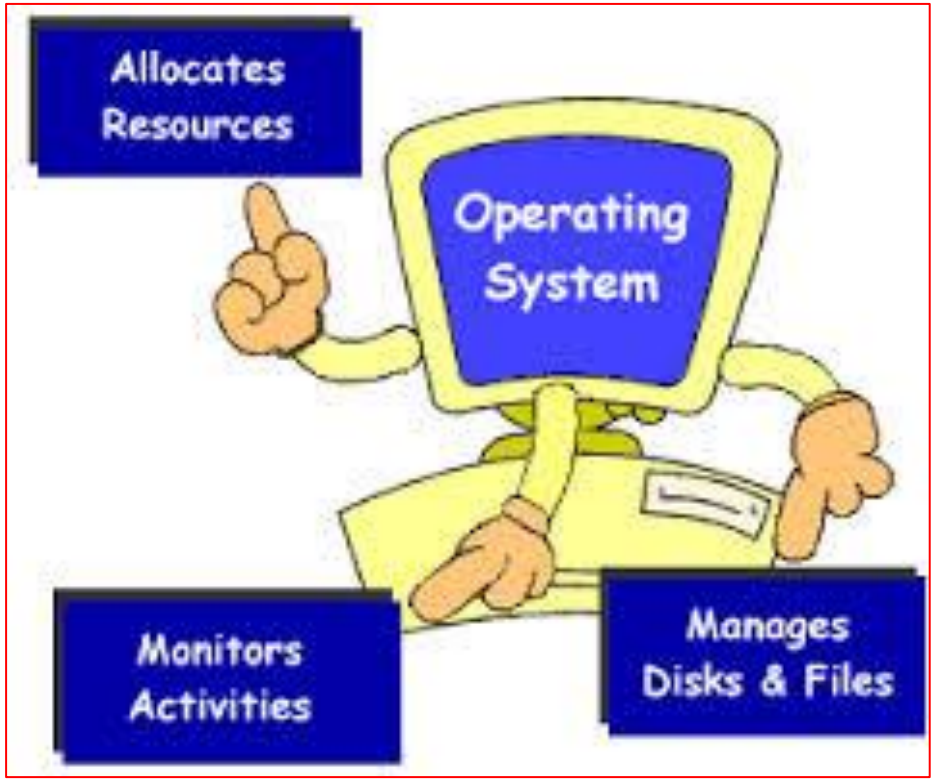
- *The Operating System acts as a bridge between the user applications/tasks and the underlying system resources through a set of system functionalities and services*



The Operating System Architecture

What is Operating System

- The low-level software that supports a computer's basic functions, such as scheduling tasks and controlling peripherals.

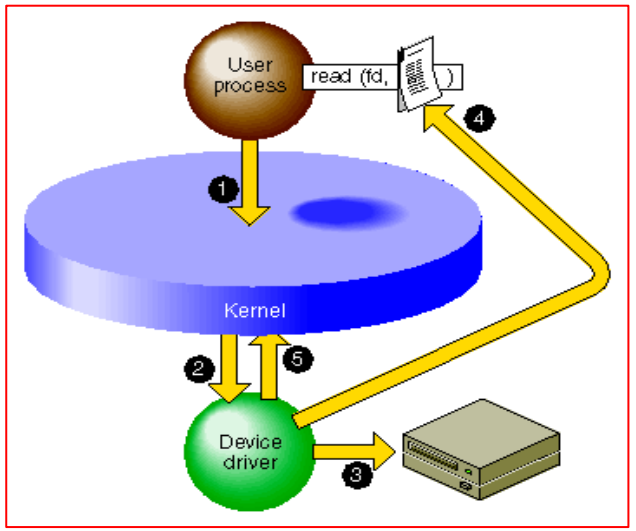


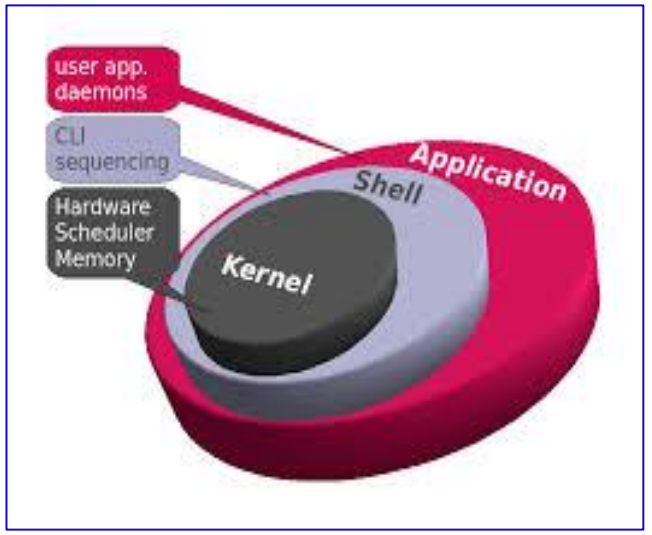
Operating System Basics

- OS manages the system resources and makes them available to the user applications/tasks on a need basis

- The **primary functions** of an Operating system is
  - Make the system convenient to use
  - Organize and manage the system resources efficiently and correctly

The Kernel

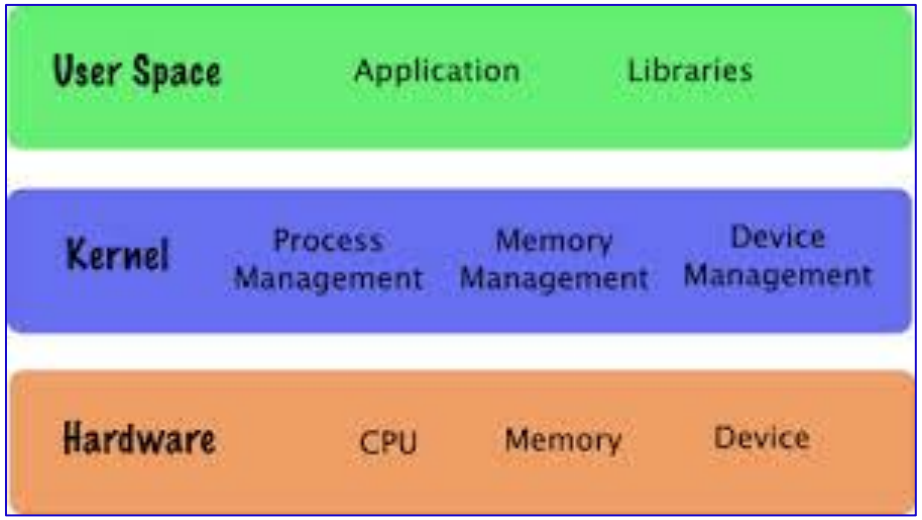




- The kernel is the core of the operating system
- It is responsible for managing the system resources and the communication among the hardware and other system services
- Kernel acts as the abstraction layer between system resources and user applications

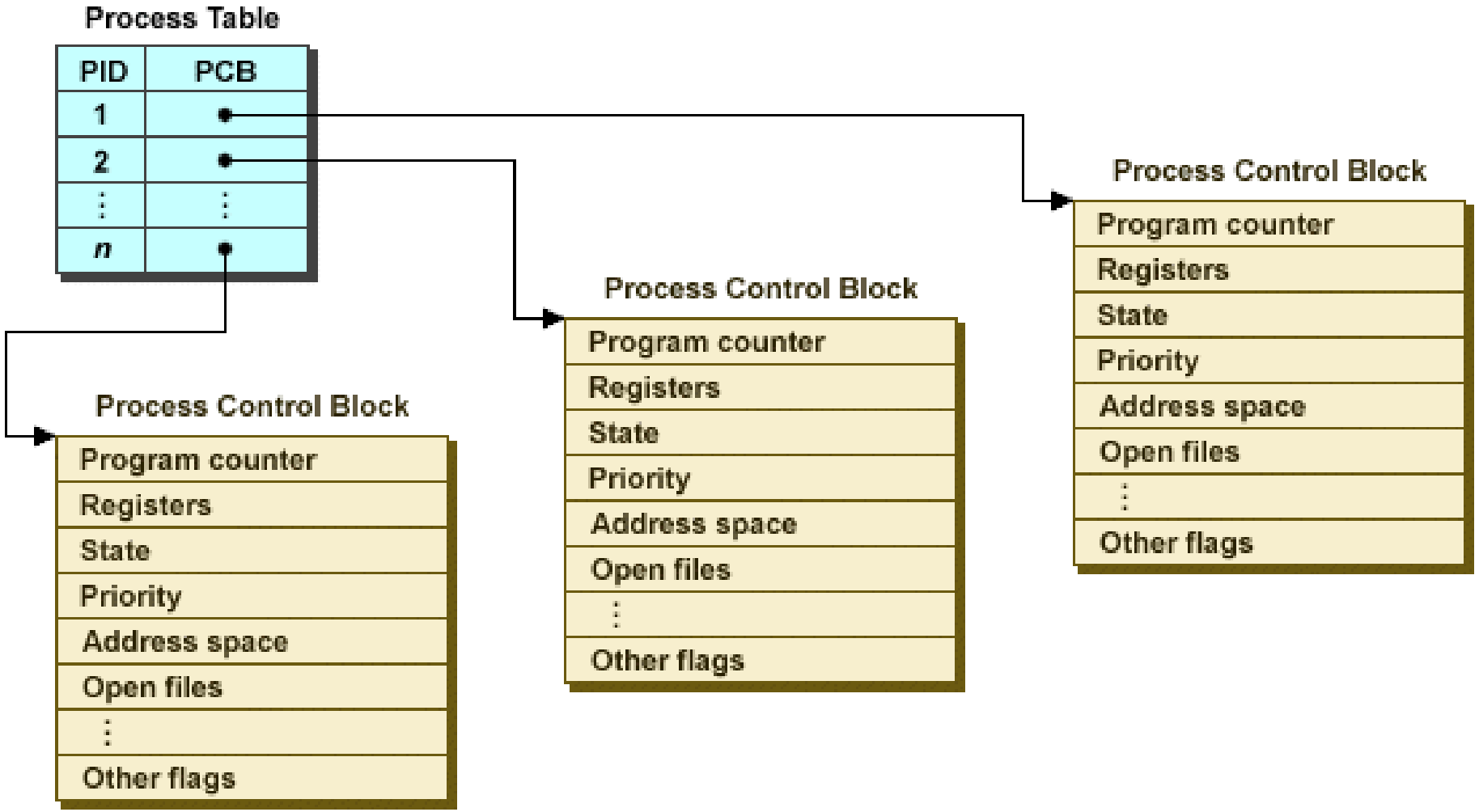
The Kernel

- Kernel contains a set of system libraries and services. For a general purpose OS, the kernel contains different services like
  - Process Management
  - Primary Memory Management
  - File System management
  - I/O System (Device) Management
  - Secondary Storage Management
  - Protection
  - Time management
  - Interrupt Handling
  - Others . . . .



The Kernel- Process Management

- Deals with ***managing*** Process/tasks.
- Setting up ***memory space*** for process
- Loading the process code into memory space allocating system resources
- ***Scheduling*** and ***managing*** the execution of process
- Setting and managing the ***Process Control Box (PCB)***
- Inter-process communication, synchronization, process termination/deletion


Process Table
PID
PCB
1
2
n
Process Control Block
Program counter
Registers
State
Priority
Address space
Open files
Other flags
Process Control Block
Program counter
Registers
State
Priority
Address space
Open files
Other flags
Process Control Block
Program counter
Registers
State
Priority
Address space
Open files
Other flags

The Kernel- Primary Management Unit

- Refers to Volatile Memory (RAM) where process are loaded and variables, shared data associated with process are stored.

- The Memory Management Unit (MMU) is responsible for
  - Keeping track of which part of memory area is used by which process
  - Allocating the memory space on need basis

The Kernel- File System Management

- File is collection of related information, could be any text, image, word, audio/video etc…
- Each are different in kind of information they hold and the way information is stored.
- The file management service in kernel is responsible for
  - The creation, deletion and alteration of files.
  - Creation, deletion and alteration of directories.
  - Saving file in secondary storage Memory (e.g. HDD)
  - Providing automatic allocation of file space based on amount of free space available.
  - Providing flexible naming convention.
- ***File management is an OS dependent. MS DOS file system can not work by file system support provided by UNIX Kernel.***

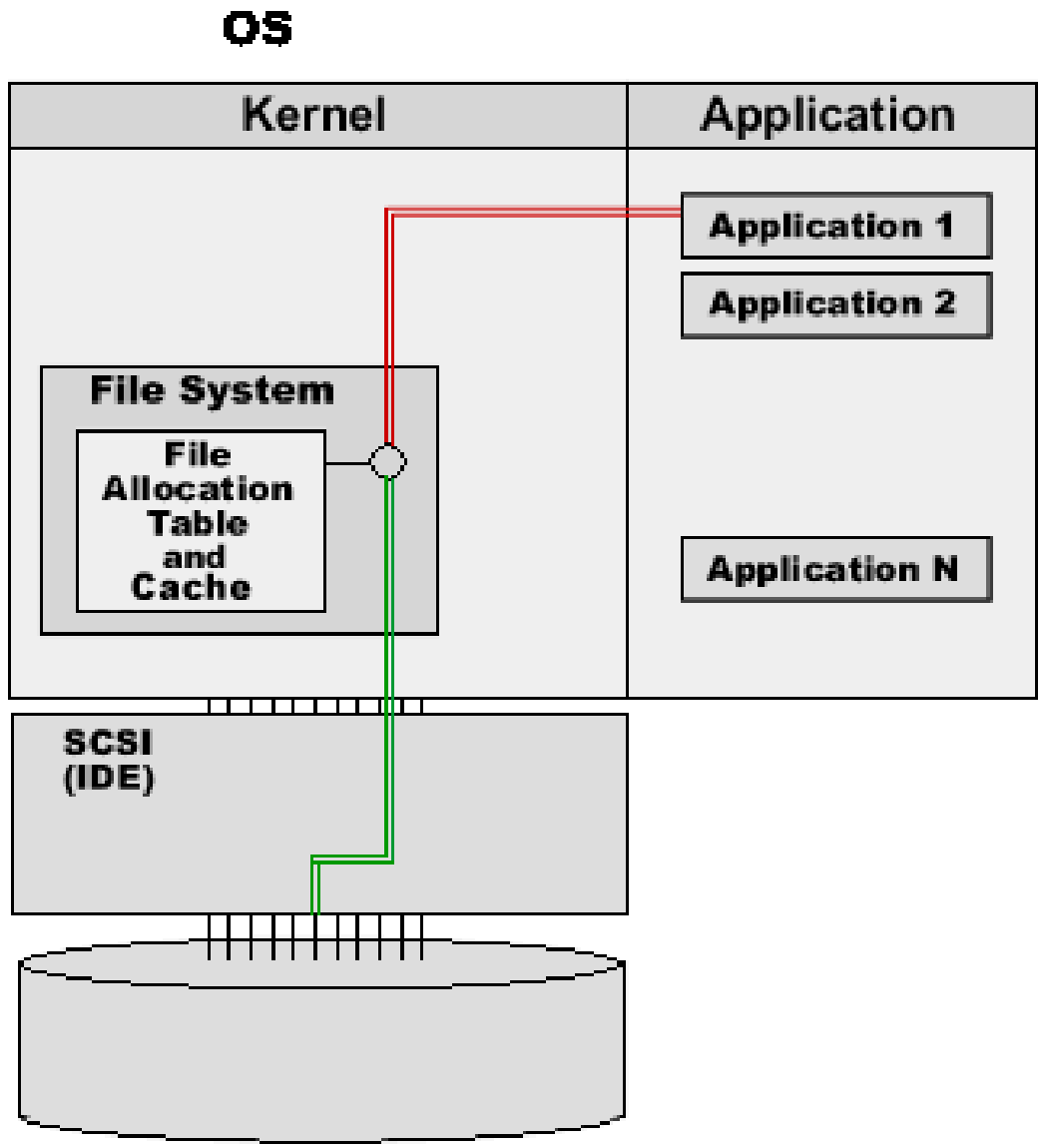
The Kernel- I/O System (Device) management

- Kernel is responsible for routing I/O request coming from different user applications to appropriate I/O devices.
- Direct accessing of I/O devices in well structured OS are not allowed.
- Access must be provided through set of Application Programming Interfaces (APIs) exposed by kernel.
- Kernel maintains list of devices in advance or may be upgraded when device is installed. e.g plug n play USB.
- The service Device Manager is responsible
- The kernel talks to I/O devices to set of the driver through a set of low level system call, which are implemented in service called device driver.
- **The Device Manager is responsible for**
  - **Loading and unloading of device Driver**
  - **Exchanging information and system specific control signal to and from the devices.**

The Kernel- Secondary Storage Management

- Deals with managing the secondary storage devices.
- Main duty of the Kernel are
  - Device storage allocation
  - Disk Scheduling (Time Interval at Disk is activated to take back up data)
  - Free Disk space Management
- *e.g. The Service storage in Computer management*

The Kernel- Secondary Storage Management



The Kernel- Protection System

- Most of the Modern OS support multiple user login
- Different user can be assigned different level of access permission
- Protection deals with implementing security policy to restrict the access to both User and system resources.
- Use of some application may not be granted.

The Kernel- Interrupt Handler

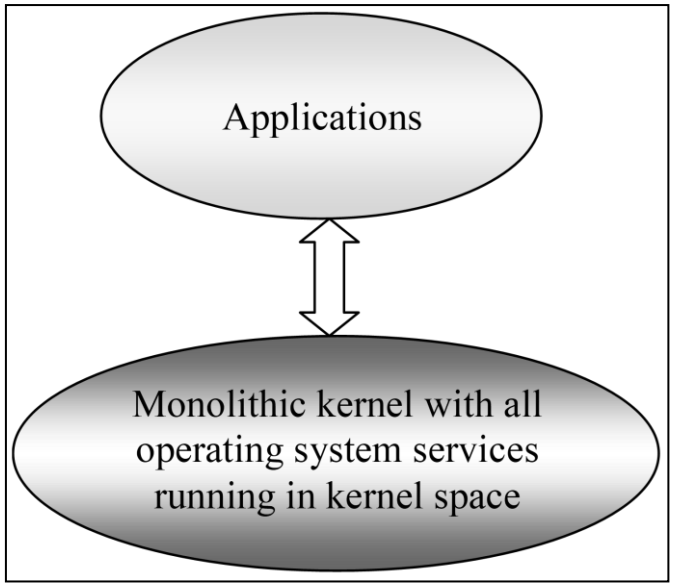
- Kernel provides handler mechanism for all external/internal interrupt generated by system.
- Other services offered by Kernel of OS.
  - Network Communication
  - Network management
  - User Interface
  - Graphics
  - Timer Service (Delays/Timeout)
  - Error Handler
  - Database Management
  - All these are handled using several APIs.

Kernel Space and User Space

- The program code corresponding to the kernel applications/services are kept in a contiguous area (OS dependent) of primary (working) memory and is protected from the un-authorized access by user programs/applications
- The memory space at which the kernel code is located is known as *'Kernel Space'*
- All user applications are loaded to a specific area of primary memory and this memory area is referred as *'User Space'*
- The partitioning of memory into kernel and user space is purely Operating System dependent
- An operating system with virtual memory support, loads the user applications into its corresponding virtual memory space with demand paging technique
- Most of the operating systems keep the kernel application code in main memory and it is not swapped out into the secondary memory

Monolithic Kernel

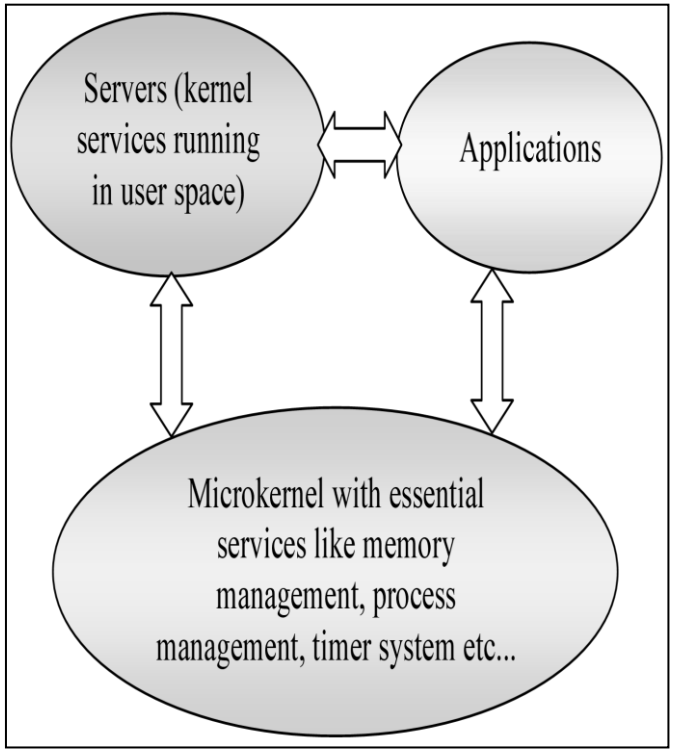
- All kernel services run in the kernel space
- All kernel modules run within the same memory space under a single kernel thread
- The tight internal integration of kernel modules in monolithic kernel architecture allows the effective utilization of the low-level features of the underlying system
- The major drawback of monolithic kernel is that any error or failure in any one of the kernel modules leads to the crashing of the entire kernel application



Microkernel

- The microkernel design incorporates only the essential set of Operating System services into the kernel
- Rest of the Operating System services are implemented in programs known as '*Servers*' which runs in user space
- The kernel design is highly modular

- Memory management, process management, timer systems and interrupt handlers are examples of essential services, which forms the part of the microkernel
- **MACH, QNX, Minix3** kernels are examples for microkernel



Offers Advantage
- Robustness
- Configurability.

GPOS Vs RTOS

Types of Operating Systems

Depending on:

- Type of kernel
- kernel services
- Purpose and type of computing systems where the OS is deployed
- Responsiveness to applications
- Operating Systems are classified into
  - GPOS
  - RTOS

General Purpose Operating System (GPOS)

- Operating Systems, which are deployed in general computing systems
- The kernel is more generalized and contains all the required services to execute generic applications
- Need not be deterministic in execution behavior
- May inject random delays into application software and thus cause slow responsiveness of an application at unexpected times
- Usually deployed in computing systems where deterministic behavior is not an important criterion
- Personal Computer/Desktop system is a typical example for a system where GPOSs are deployed.
- Windows XP/MS-DOS, Linux etc are examples of General Purpose Operating System

When RTOS is required

- RTOS requested when system has following constraints
  - Timing constraint – processes have deadline
  - Precedence constraint – processes runs according to priority (high/low)
  - Resource constraint
    - private resource – dedicated for process
    - shared resource – used by all
    - exclusive resource – shared but not simultaneous access

Real Time Purpose Operating System (RTOS)

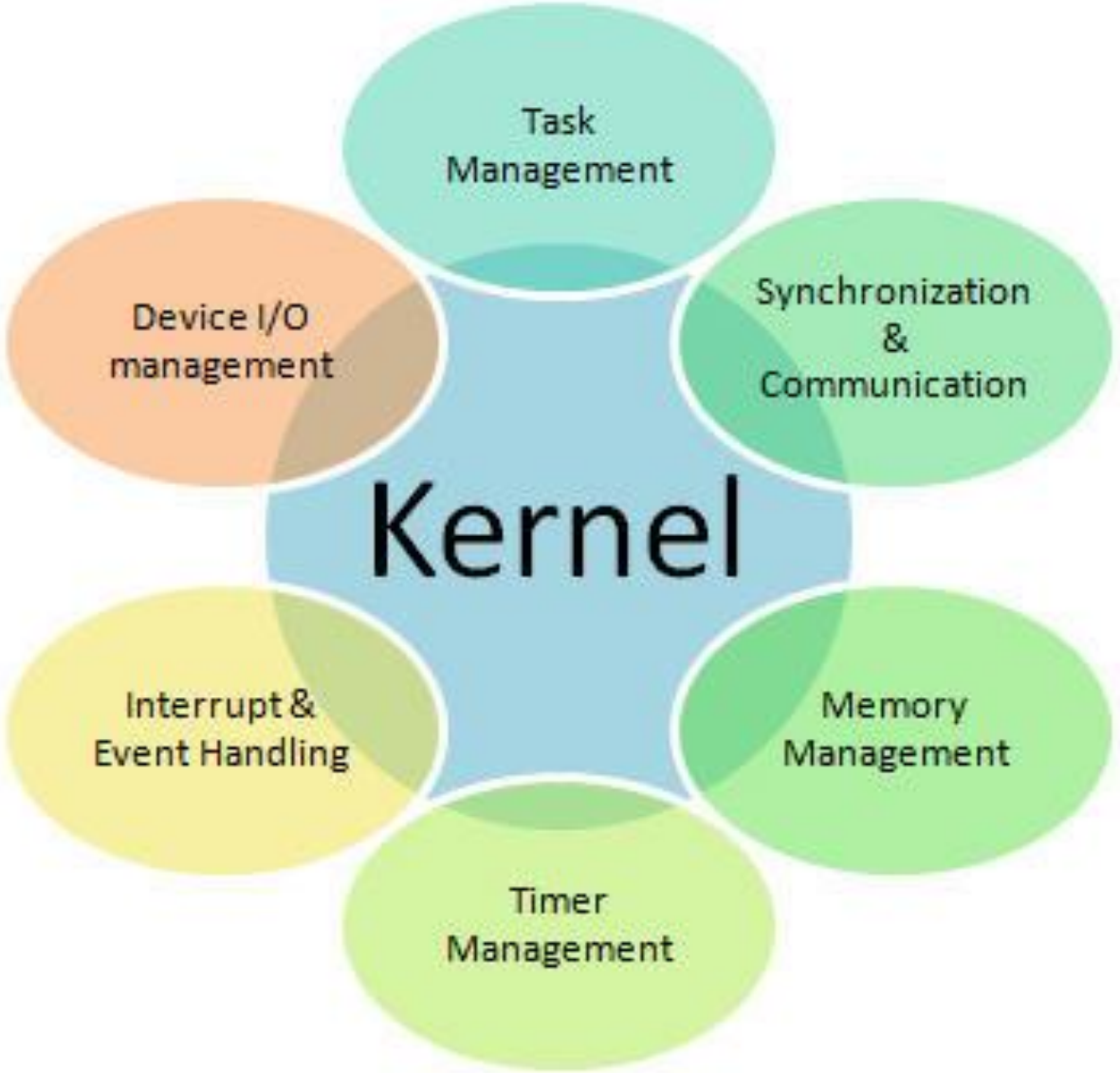
- Operating Systems, which are deployed in embedded systems demanding real-time response
- Deterministic in execution behavior. Consumes only known amount of time for kernel applications
- Implements scheduling policies for executing the highest priority task/application always
- Implements policies and rules concerning time-critical allocation of a system's resources
    - Provide necessary system calls to achieve real-time deadlines
    - Worst case execution time of each system call can be calculated
- Decides which Application should run in which order and how much time needed.
- Predictable performance is the Hallmark.
- It achieves using proper policies and rules.
- Windows CE, QNX, VxWorks, MicroC/OS-II etc are examples of Real Time Operating Systems (RTOS)

The Real Time Kernel

The Real Time Kernel

- The kernel of a Real Time Operating System is referred as Real Time kernel.
- In complement to the conventional OS kernel, the Real Time kernel is highly specialized and it contains only the minimal set of services required for running the user applications/tasks.
- **The basic functions of a Real Time kernel:**
  1. **Task/Process management**
  2. **Task/Process scheduling**
  3. **Task/Process synchronization**
  4. **Error/Exception handling**
  5. **Memory Management**
  6. **Interrupt handling**
  7. **Time management**

Functions of Real Time Kernel



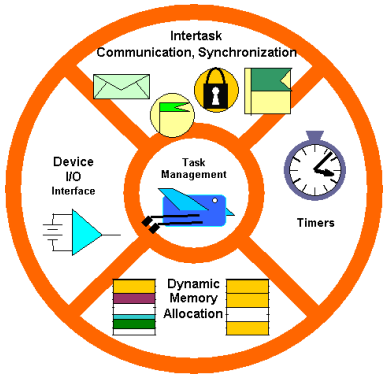
1. Task/Process Management

Deals with setting up

- The memory space for the tasks
- Loading the task's code into the memory space
- Allocating system resources
- Setting up a Task Control Block (TCB)

  for the task and task/process termination/deletion.

A Task Control Block (TCB) is used for holding the information corresponding to a task. TCB usually contains the following set of information



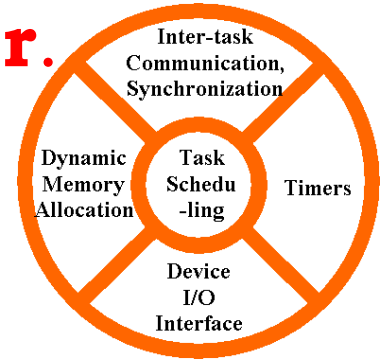
Task Control Block (TCB)

- *Task ID:* Task Identification Number
- *Task State:* The current state of the task. (E.g. State= 'Ready' for a task which is ready to execute)
- *Task Type*: Task type. Indicates what is the type for this task. The task can be a hard real time or soft real time or background task.
- *Task Priority*: Task priority (E.g. Task priority =1 for task with priority = 1)
- *Task Context Pointer*: Context pointer. Pointer for context saving
- *Task Memory Pointers*: Pointers to the code memory, data memory and stack memory for the task
- *Task System Resource Pointers*: Pointers to system resources (semaphores, mutex etc) used by the task
- *Task Pointers:* Pointers to other TCBs (TCBs for preceding, next and waiting tasks)
- *Other Parameters* Other relevant task parameters

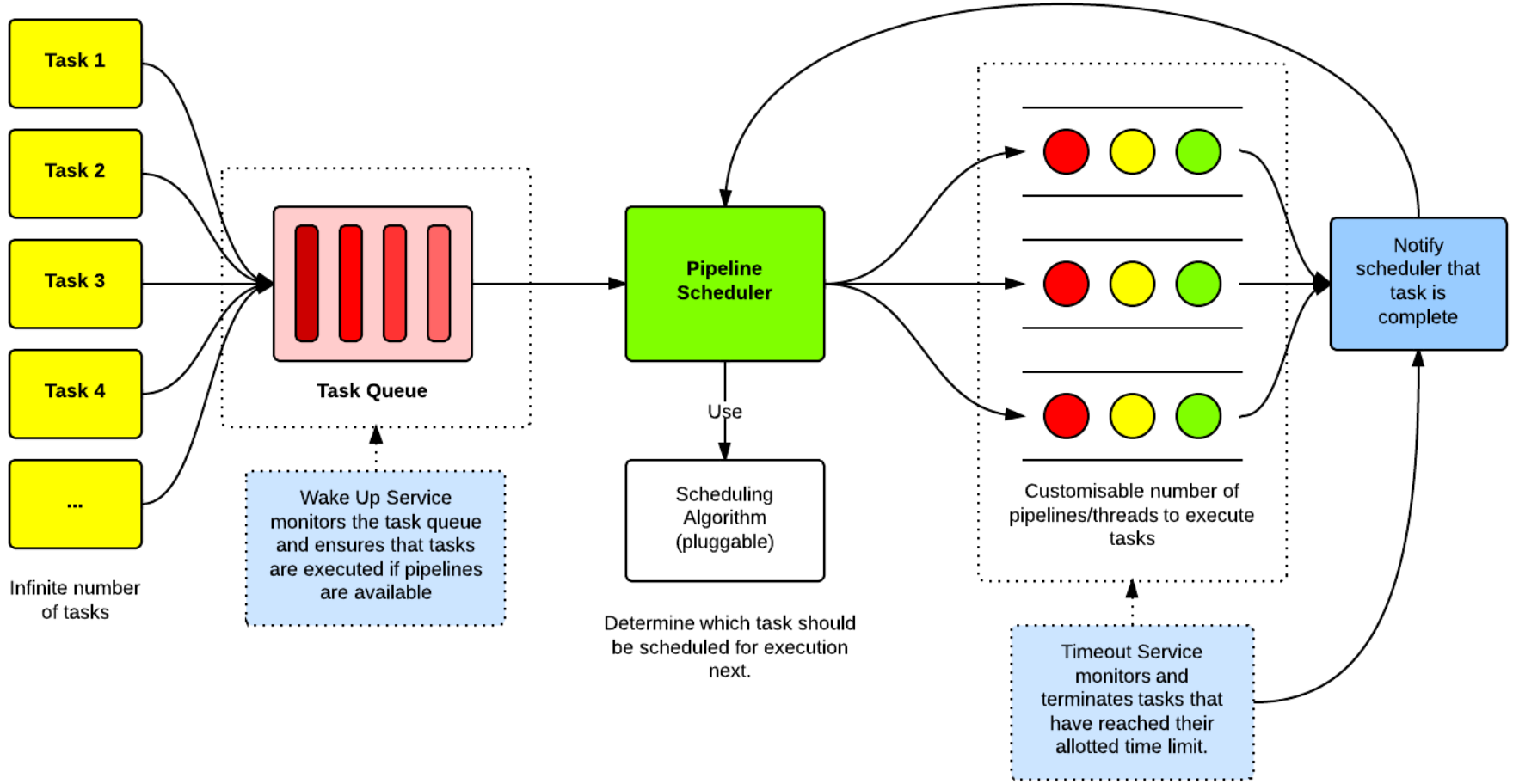
The parameters and implementation of the TCB is kernel dependent. The TCB parameters vary across different kernels, based on the task management implementation

2. Task/Process Scheduling

- Deals with sharing the CPU among various tasks/processes.
- A kernel application called *'Scheduler'* handles the task scheduling.
- Scheduler is nothing but an algorithm implementation, which **performs the efficient and optimal scheduling** of tasks to provide a **deterministic behavior**.



2. Task/Process Scheduling

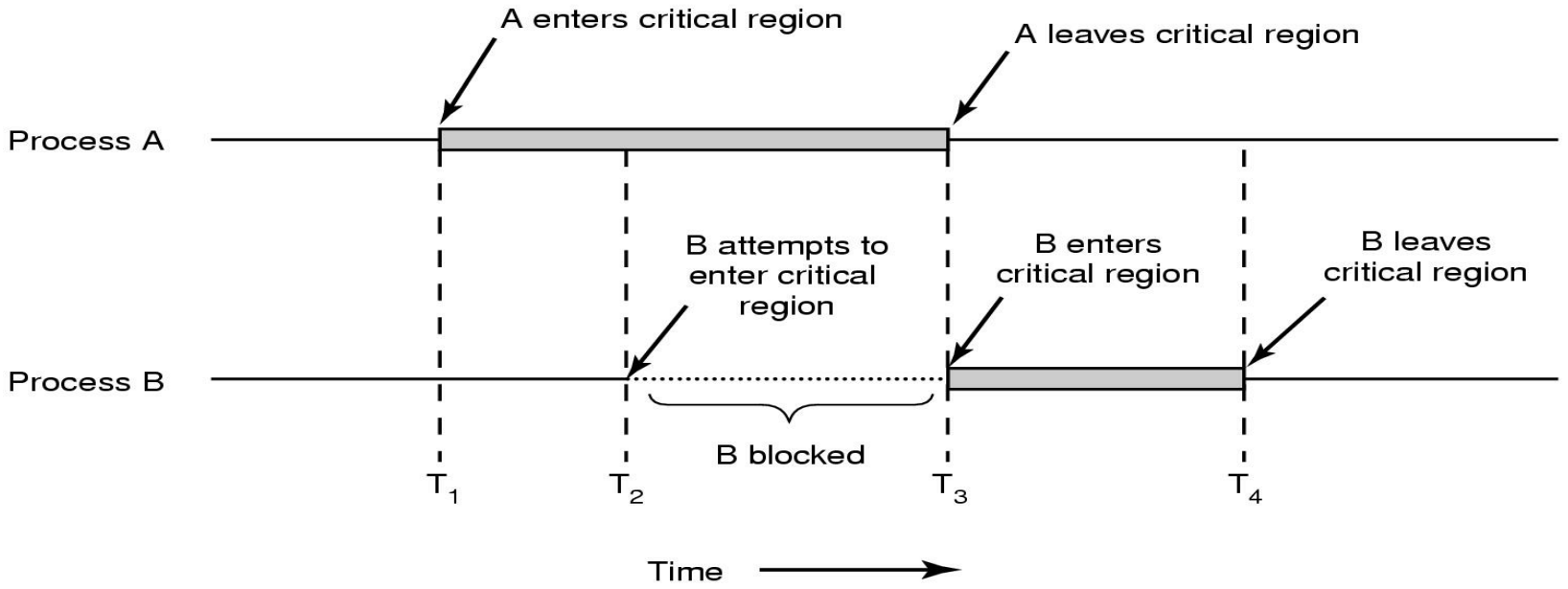


3. Task/Process Synchronization

- Deals with
  - synchronizing the concurrent access of a resource, which is shared across multiple tasks
  - the communication between various tasks.

The Critical-Section Problem

- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.



Solution to Critical-Section Problem

- Semaphores
  - Process executing critical region takes semaphores
  - On exit from the region release semaphore
  - Semaphore value 0 and 1
- Semaphore is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait and signal.
- These operations were originally termed P (for wait, take) and V (for signal, release)

Semaphore

- Semaphores are typically used to control access to shared resources such as critical sections, where only one process or thread can access the resource at a time.
- There are two main types of semaphores:

1. Binary Semaphores: Also known as mutexes (mutual exclusion), these can only take on the values 0 and 1. They are often used to provide exclusive access to a resource.

2. Counting Semaphores: These can take on any non-negative integer value and are used to control access to a resource with multiple instances available, such as a fixed-size buffer.

- Semaphores can be thought of as integer variables that are accessed through two atomic operations: "wait" (or "P" operation) and "signal" (or "V" operation).

4. Error/Exception handling

- Deals with registering and handling the errors occurred/exceptions raised during the execution of tasks.
- Examples:
  - Insufficient memory, timeouts, deadlocks, deadline missing, bus error, divide by zero, unknown instruction execution etc, are examples of errors/exceptions.
- Errors/Exceptions can happen at the kernel level services or at task level.
- Deadlock is an example for kernel level exception, whereas timeout is an example for a task level exception.
- The OS kernel gives the information about the error in the form of a system call (API).
- Certain Task may involve the waiting from external events from device, may enter in infinite loop and behave HANG UP. Proper timeout mechanism should be employed e.g WATCHDOG Timer.

5. Memory Management

- The memory management function of an RTOS kernel is slightly different compared to the General Purpose Operating Systems.
- In general, the memory allocation time increases depending on the size of the block of memory needs to be allocated and the state of the allocated memory block (initialized memory block consumes more allocation time than un-initialized memory block).
- Since predictable timing and deterministic behavior are the primary focus for an RTOS, RTOS achieves this by compromising the **effectiveness of memory allocation**
- RTOS generally uses '*block*' based memory allocation technique, instead of the usual dynamic memory allocation techniques used by the GPOS.

5. Memory Management (Conti . .)

- RTOS kernel uses blocks of fixed size of dynamic memory and the block is allocated for a task on a need basis. The blocks are stored in a '*Free buffer Queue*'.
- Most of the RTOS kernels allow tasks to access any of the memory blocks without any memory protection to achieve predictable timing and avoid the timing overheads.
- RTOS kernels assume that the whole design is proven correct and protection is unnecessary. Some commercial RTOS kernels allow memory protection as optional and the kernel enters a *fail-safe* mode when an illegal memory access occurs.

5. Memory Management (Conti . .)

- A few RTOS kernels implement *Virtual Memory* concept for memory allocation if the system supports secondary memory storage (like HDD and FLASH memory).

- In the '*block*' based memory allocation, a block of fixed memory is always allocated for tasks on need basis and it is taken as a unit. Hence, there will not be any memory fragmentation issues.

- The memory allocation can be implemented as constant functions and thereby it consumes fixed amount of time for memory allocation. This leaves the deterministic behavior of the RTOS kernel untouched

6. Interrupt Handling

- Interrupts provide Real Time Behavior to systems.
- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
- Interrupts can be either *Synchronous* or *Asynchronous*.
- Interrupts which occurs in sync with the currently executing task is known as *Synchronous* interrupts. Usually the software interrupts fall under the Synchronous Interrupt category. Divide by zero, memory segmentation error etc are examples of Synchronous interrupts.
- For synchronous interrupts, the interrupt handler runs in the same context of the interrupting task.
- Asynchronous interrupts are interrupts, which occurs at any point of execution of any task, and are not in sync with the currently executing task.

6. Interrupt Handling (Conti . .)

- The interrupts generated by external devices (by asserting the Interrupt line of the processor/controller to which the interrupt line of the device is connected) connected to the processor/controller, timer overflow interrupts, serial data reception/ transmission interrupts etc are examples for asynchronous interrupts.

- For asynchronous interrupts, the interrupt handler is usually written as separate task (Depends on OS Kernel implementation) and it runs in a different context. Hence, a context switch happens while handling the asynchronous interrupts.

- Priority levels can be assigned to the interrupts and each interrupts can be enabled or disabled individually.

- Most of the RTOS kernel implements '*Nested Interrupts*' architecture. Interrupt nesting allows the pre-emption (interruption) of an Interrupt Service Routine (ISR), servicing an interrupt, by a higher priority interrupt.

7. Time Management

- Interrupts inform the processor that an external device or an associated task requires immediate attention of the CPU.
- Accurate time management is essential for providing precise time reference for all applications.
- The time reference to kernel is provided by a high-resolution Real Time Clock (RTC) hardware chip (hardware timer).
- The hardware timer is programmed to interrupt the processor/controller at a fixed rate. This timer interrupt is referred as '*Timer tick*'.
- The '*Timer tick*' is taken as the timing reference by the kernel. The '*Timer tick*' interval may vary depending on the hardware timer. Usually the '*Timer tick*' varies in the microseconds range.
- The time parameters for tasks are expressed as the multiples of the '*Timer tick*'.

7. Time Management (Conti . .)

The '*Timer tick*' interrupt is handled by the 'Timer Interrupt' handler of kernel. The '*Timer tick*' interrupt can be utilized for implementing the following actions.

- Save the current context (Context of the currently executing task)
- Increment the System time register by one. Generate timing error and reset the System time register if the timer tick count is greater than the maximum range available for System time register
- Update the timers implemented in kernel (Increment or decrement the timer registers for each timer depending on the count direction setting for each register. Increment registers with count direction setting = '*count up*' and decrement registers with count direction setting = '*count down*')
- Activate the periodic tasks, which are in the idle state
- Invoke the scheduler and schedule the tasks again based on the scheduling algorithm
- Delete all the terminated tasks and their associated data structures (TCBs)
- Load the context for the first task in the ready queue. Due to the re-scheduling, the ready task might be changed to a new one from the task, which was pre-empted by the 'Timer Interrupt' task

Other Functions

- Apart from these basic function, Some RTOS provide some other function e.g. File Management and Network function.
- Some RTOS provide selecting the required kernel function at the time of building kernel. The User can pick required functions apart from the set of available function and compile to generate the kernel binary.
- e.g. In Windows CE, the User can select the required components for the kernel