# REAL TIME OPERATING SYSTEM

# REAL TIME SYSTEM

**A system is said to be real time, if is required to complete its work and deliver its service on time. For example: Flight Control System.**

## Types of Real Time System

1. **Soft Real Time System:**

- ❖ **Tasks are performed as fast as possible**
- ❖ **Late Completion of Job is undesirable but not fatal.**
- ❖ **System performance degrades as more and more jobs are miss deadline.**
- ❖ **Most Operating system are soft real time system.**

**For example: Multimedia Transmission and reception, cellular Network, Websites, Computer Games, Online Database, ATM**

**2. Hard Real Time System:**
**There is no flexibility in time constraint.**
**It is purely deterministic.**

**For example: Air traffic control, Vehicle Subsystem control, Nuclear Power Plant Control, Missile launching system, Air bags control in car.**

## REAL TIME OPERATING SYSTEM

- ❖A Real Time operating system is defined as a data processing system in which the time interval required to process and response to input is so small, that it controls the environment. For example automatic car, Street light

- ❖The time taken from input to output task is the response time.

- ❖RTOS responds to input immediately(Real Time).

- ❖Task is completed within a specified time delay.

- ❖In real life situations like controlling traffic signals or nuclear reactor or an aircraft , the operating system has to respond quickly.

# DESKTOP VS. RTOS

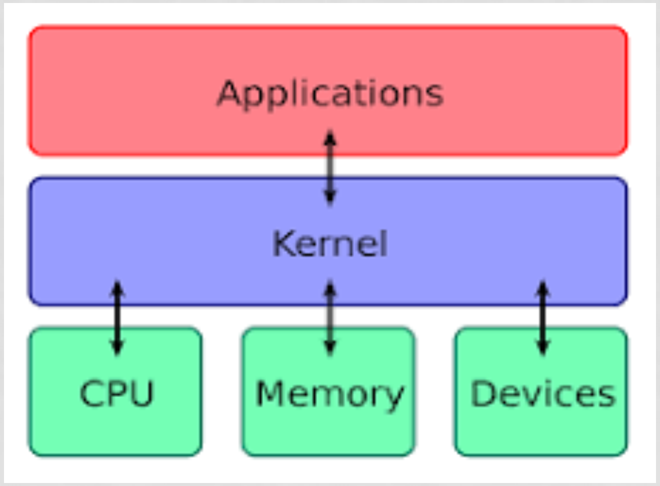
| Desktop OS | RTOS |
|---|---|
| During boot up,  OS gets the control first then the application software. | During boot up, Application software gets the control first then the OS. |
| Application and OS share the different address space. | Application and OS share the same address space. |
| OS protects itself carefully from the application. | OS does not protect itself carefully from the application. |
| General Purpose. | Dedicated to a single embedded application. |
| There may not be fixed time defined for any service in case of Desktop OS. | RTOS are deterministic in nature i.e., the time required to execute the services are fixed. |
| High priority process may be delayed to perform several low priority taskss. | High priority process execution will override the low priority ones. |
| Windows, Mac OS etc. | Vxworks, Vtrx, Nucleus, LynxOS etc… |

## CHARACTERISTICS OF RTOS

- **Reliability**
  - **Systems must be reliable.**
  - **Needs to operate without human intervention.**
- **Predictability**
  - **Should be deterministic(meeting time requirements should be predictable).**
- **Performance**
  - **Fast enough to fulfill the timing requirement.**
- **Compactness**
  - **Should be small and efficient to face memory constraints.**
- **Scalability**
  - **Should be capable of adding/deleting modular components.**

## KERNEL

- It is the central component of most computer operating system.
- The main aim of kernel is to manage communication between Software and Hardware.
- Software means the user level application.
- Hardware means the CPU, Hard Disk, memory etc.



## KERNEL FEATURES

- **Process Management:**

  The kernel supervise processes, does scheduling, creation, termination, and communication.

- **Interrupt handling**:

  Kernel manages system interrupts, prioritizing processes for foreground or background execution.

- **I/O Device Management**:

  Kernel operations through low-level Kernel routes I/O requests from various applications to the suitable I/O devices. The 'device manager' service handles all I/O device system calls known as device drivers.

- **File System Management:**

  The kernel manages file systems, encompassing tasks like reading, writing, and organizing files on storage devices. It offers a uniform interface for applications to interact with and control files and directories.

## KERNEL FEATURES

- **Memory Management**:

  The kernel supervises memory allocation, deallocation, and protection for processes, including managing virtual memory to provide each process with its own virtual address space.

- **System Calls**:

  The kernel offers system calls as interfaces for user-level applications to request services securely and under control, ensuring a safe interaction between applications and the kernel.

- **Error Handling**:

  The kernel manages hardware and software errors, preventing system crashes and data corruption through mechanisms like exception handling and error recovery routines.

- **Interprocess Communication (IPC):**

  The kernel enables process communication and data sharing using diverse IPC mechanisms like pipes, sockets, and shared memory.

## KERNEL FEATURES

- **Security:** The kernel ensures system security by controlling access, permissions, and isolation among processes and users to prevent unauthorized access and malicious activities.
- **Scheduling:** The kernel employs scheduling algorithms to allocate CPU access and duration for processes, striving to achieve balanced resource utilization.
- **Virtualization:** Modern kernels often support virtualization, enabling concurrent execution of multiple operating systems or virtual machines on the same physical hardware.
- **Power Management:** The kernel optimizes power consumption and extends battery life in portable devices by managing power-related features like sleep modes etc..

# REAL TIME KERNEL

Real-Time Kernels, also known as Real-Time Operating System (RTOS) kernels, are designed to meet stringent timing requirements and provide predictable behavior for time-critical applications. Here are the key features commonly found in real-time kernels:

1. **Deterministic Scheduling**: Real-time kernels use deterministic scheduling algorithms to ensure that tasks with higher priorities are executed before lower-priority tasks. This guarantees that critical tasks get timely access to the CPU, meeting their deadlines.
2. **Preemptive Multitasking**: Real-time kernels support preemptive multitasking, allowing higher-priority tasks to interrupt and preempt lower-priority tasks. This feature ensures that time-critical tasks can run without delay, even if lower-priority tasks are already executing.
3. **Task Prioritization**: Tasks in a real-time kernel are assigned priorities, and the scheduler uses these priorities to determine the order in which tasks are executed. Higher-priority tasks are given precedence over lower-priority tasks.
4. **Minimal Latency**: Real-time kernels aim to minimize interrupt latency and context-switching time. Low latency is crucial in meeting strict timing requirements and achieving predictable behavior.
5. **Fixed-Time Services**: Some real-time kernels provide fixed-time services for critical operations like context switches, memory allocation, and inter-process communication. These services ensure that these operations complete within predetermined time bounds.
6. **Timers and Clocks**: Real-time kernels often include precise timers and clocks that applications can use to schedule events accurately. These timers are essential for meeting specific timing requirements.

## REAL TIME KERNEL

6. **Reliable Task Synchronization**: Real-time kernels ensure effective task cooperation and communication with robust synchronization mechanisms (e.g., semaphores, mutexes, and message queues) to avoid race conditions and deadlocks.
7. **Interrupt Handling**: Real-time kernels efficiently handle hardware interrupts, promptly servicing critical events.
8. **Memory Management**: Real-time kernels offer memory management for tasks, ensuring fast and predictable memory allocation and deallocation.
9. **Small Footprint**: Real-time kernels are optimized for resource-constrained embedded systems with minimal memory and processing overhead.
10. **Hard and Soft Real-Time Support**: Real-time kernels can cater to both hard real-time and soft real-time needs, where hard real-time systems must meet strict timing deadlines, while soft real-time systems allow some flexibility for occasional deadline misses.
11. **Error Handling**: Real-time kernels incorporate fault handling mechanisms for graceful recovery or safe failure.

# TYPES OF REAL TIME KERNEL

1. **Monolithic Kernel:**

   In this design, the entire set of vital operating system functions and services are consolidated within a single, comprehensive executable file. The kernel directly interacts with the hardware, overseeing resource management such as memory, processes, device drivers, file systems, and networking protocols.

**Key characteristics of a monolithic kernel:**

**i. Single Address Space**:

All kernel functions and services run in the same address space, allowing for efficient communication and data sharing between different kernel components.

**ii. Efficient Performance**: Due to minimal inter-process communication overhead among kernel modules within the same address space, they exhibit enhanced performance.

**iii. Complexity and Stability**: Monolithic kernels may become intricate and challenging to maintain because of their extensive codebase. A bug in one section of the kernel could impact the overall system stability.

**iv. Limited Modularity**: Modifying features in a monolithic kernel often demands recompiling the whole kernel, reducing its flexibility and modularity compared to other kernel architectures.

**v. Device Drivers**: Device drivers in a monolithic kernel operate in kernel space, enabling direct hardware access for improved performance but introducing potential security vulnerabilities.

## TYPES OF REAL TIME KERNEL

**2. Micro Kernel:**

Microkernel design reduces kernel size and complexity by moving most operating system functions to user-space processes. Only essential services like inter-process communication and basic memory management remain in the kernel space. Other functions, such as device drivers, file systems, and networking protocols, are implemented as separate user-space processes known as "servers" or "services.“

**Key characteristics of a microkernel:**

**Modularity**: The microkernel design promotes modularity by keeping the kernel small and focused on essential tasks. Additional services run as separate user-space processes, which can be loaded or unloaded dynamically without the need to restart the entire system.

**Isolation**: Since most services run in user space, a bug or crash in one service does not affect other parts of the system. This isolation improves system stability and security.

**Portability**: Microkernel's are often designed to be more portable across different hardware architectures because they rely on a smaller, more standardized kernel interface.

**Performance Trade-offs**: While micro kernels aim to improve system stability and security, they can introduce additional overhead due to the need for inter-process communication between user-space services. This overhead can impact overall system performance compared to monolithic kernels.

## TYPES OF REAL TIME KERNEL

3. **Hybrid Kernel:**

   A hybrid kernel blends monolithic and microkernel aspects, aiming to balance performance and modularity.

   It keeps some OS services in kernel space for better performance and implements others as separate user-space processes for enhanced modularity and isolation.

Key characteristics of a hybrid kernel:

- **Modularity**: Similar to microkernels, hybrid kernels prioritize modularity by running specific OS services as separate user-space processes. This approach enhances system stability and simplifies the addition or removal of services.
- **Performance Optimization**: To optimize system performance, critical components like device drivers and low-level hardware access are typically retained in kernel space to reduce inter-process communication overhead.
- **Flexibility**: Hybrid kernels offer flexibility by allowing designers to choose which services operate in kernel space and which in user space, enabling customization for specific hardware and performance needs.
- **Isolation**: Through user space separation, a hybrid kernel achieves improved component isolation, minimizing the impact of failures in one part of the system on others.
- **Complexity**: Designing and maintaining hybrid kernels can be more intricate compared to monolithic or microkernels, as they require management of both kernel and user-space components.

## TYPES OF REAL TIME KERNEL

4. **Exo Kernel:**

- ❖ The exokernel takes kernel modularity to an extreme level.
- ❖ It directly exposes the lowest level of hardware abstractions to applications.
- ❖ It allows applications to have explicit control over hardware resource utilization, providing high flexibility and efficiency by minimizing the kernel's role and maximizing application control.

Key  characteristics of an exokernel:

- **Resource Management**: The main purpose of an exokernel is to efficiently manage hardware resources with a minimal interface to applications. Rather than offering high-level abstractions, the exokernel exposes raw hardware resources like CPU time, memory, and network access.
- **Application Flexibility**: Exokernel empowers applications with greater control over resource management, enabling customization of memory allocation, CPU scheduling, and network communication to meet specific requirements.
- **Modularity:** Exokernel design prioritizes modularity, segregating resource management functions (handled by the exokernel) from higher-level abstractions and services (implemented by library operating systems or "libOSes").
- **Library Operating Systems (libOSes)**: In an exokernel setup, conventional OS services such as file systems, process management, and network protocols are user-level libraries (libOSes) built on the exokernel. LibOSes provide higher-level abstractions to applications, while the exokernel concentrates on efficient resource allocation.
- **Performance:** Exokernels strive for high performance in resource-intensive applications by reducing the kernel's involvement and enabling direct hardware access for fine-grained resource control.

## TYPES OF REAL TIME KERNEL

5. **Nano Kernel:**

More minimalistic than a microkernel, providing only essential functionalities for hardware resource management and communication between software components.

Most OS services, including device drivers, file systems, and networking protocols, are implemented as user-space processes instead of residing in kernel space.

Key characteristics of a nanokernel:

**Minimalism**: The nanokernel strives for extreme minimalism, focusing solely on critical functions such as CPU scheduling, memory management, and inter-process communication.

**Modularity**: Like microkernels, the nanokernel fosters modularity by assigning most services to user-space processes. This enables seamless addition, removal, or replacement of services without impacting the kernel's core.

**Performance**: The nanokernel's small size and minimal functionality lead to lower overhead and potentially superior performance compared to feature-rich kernel architectures.

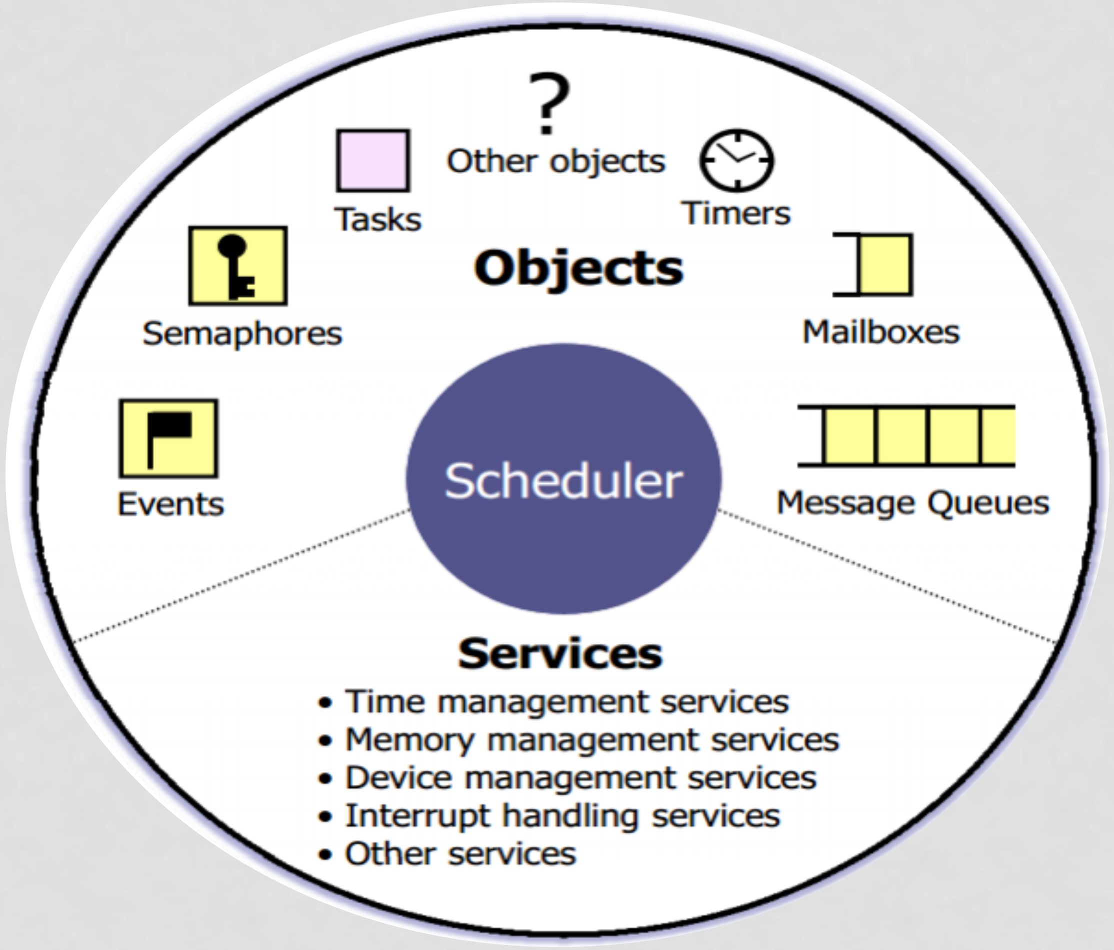
**Flexibility**: By delegating most services to user-space processes, the nanokernel gains flexibility and adaptability, allowing system customization to meet specific requirements.

**User-Space Services**: The majority of OS functionality, such as file systems, device drivers, and networking protocols, are implemented as user-space services or servers. They communicate with the nanokernel through well-defined interfaces.

# REAL TIME KERNEL

**Essential Components of Real Time Kernel:**

- Scheduler
- Objects
- Services


?
Other objects
Tasks
Timers
Semaphores
Objects
Mailboxes
Events
Scheduler
Message Queues
Services
• Time management services
• Memory management services
• Device management services
• Interrupt handling services
• Other services

## SCHEDULER

A component of an operating system or RTOS responsible for managing the execution of tasks or processes, deciding which task to run next and allocating system resources efficiently.

Types of Scheduler:

**Non-Premptive Scheduling**

- **FIFO**(First Come First Served) Scheduling: E.g Ticket Reservation System
- **LIFO**(Last Come and First Served) Scheduling
- **SJF**(Short Job First) Scheduling

  For instance, consider the following tasks and their respective burst times in a real-time system: **Task A: 2 ms Task B: 5 ms Task C: 3 ms Task D: 1 ms** In SJF scheduling, the task with the shortest burst time will be given the highest priority and executed first. So, in this example, the scheduling order will be: **Task D (1 ms) Task A (2 ms) Task C (3 ms) Task B (5 ms)**
- **Priority Based Scheduling**: Priority is set based on time req. to complete the complete the task.

# SCHEDULER

2. Preemptive Scheduling

- ❖ SJF(Shortest Remaining Time) Scheduling:

  When new process enters the ready queue and checks whether the execution time of new process is shorter than the remaining of the total estimated time for the currently executing process.

- ❖ Round Robin Scheduling:

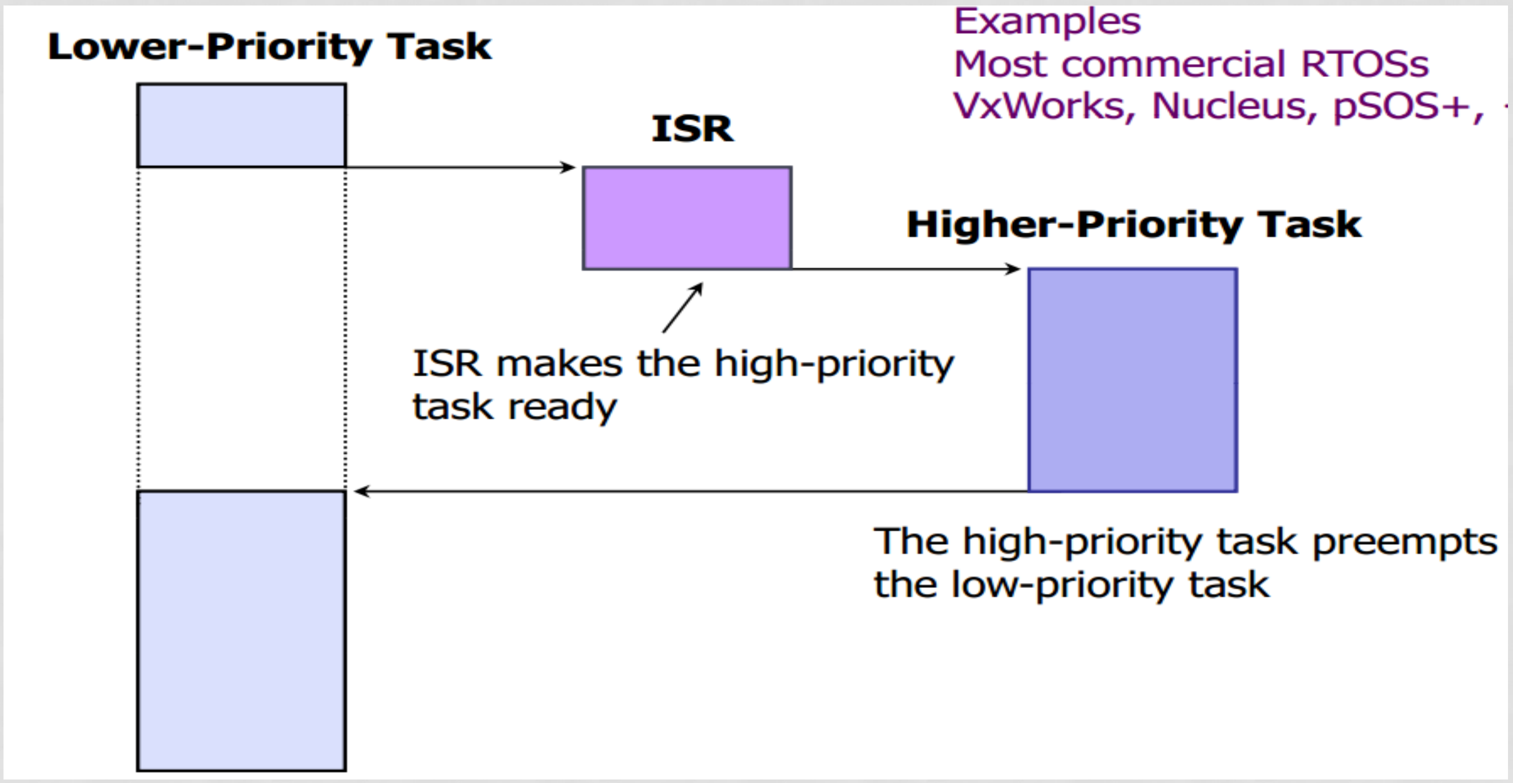
  Each Process in the ready queue is executed for a pre-defined time slot.

- ❖ Priority Based Scheduling:

  It is same as priority based scheduling of non-preemptive except for switching of execution between process.

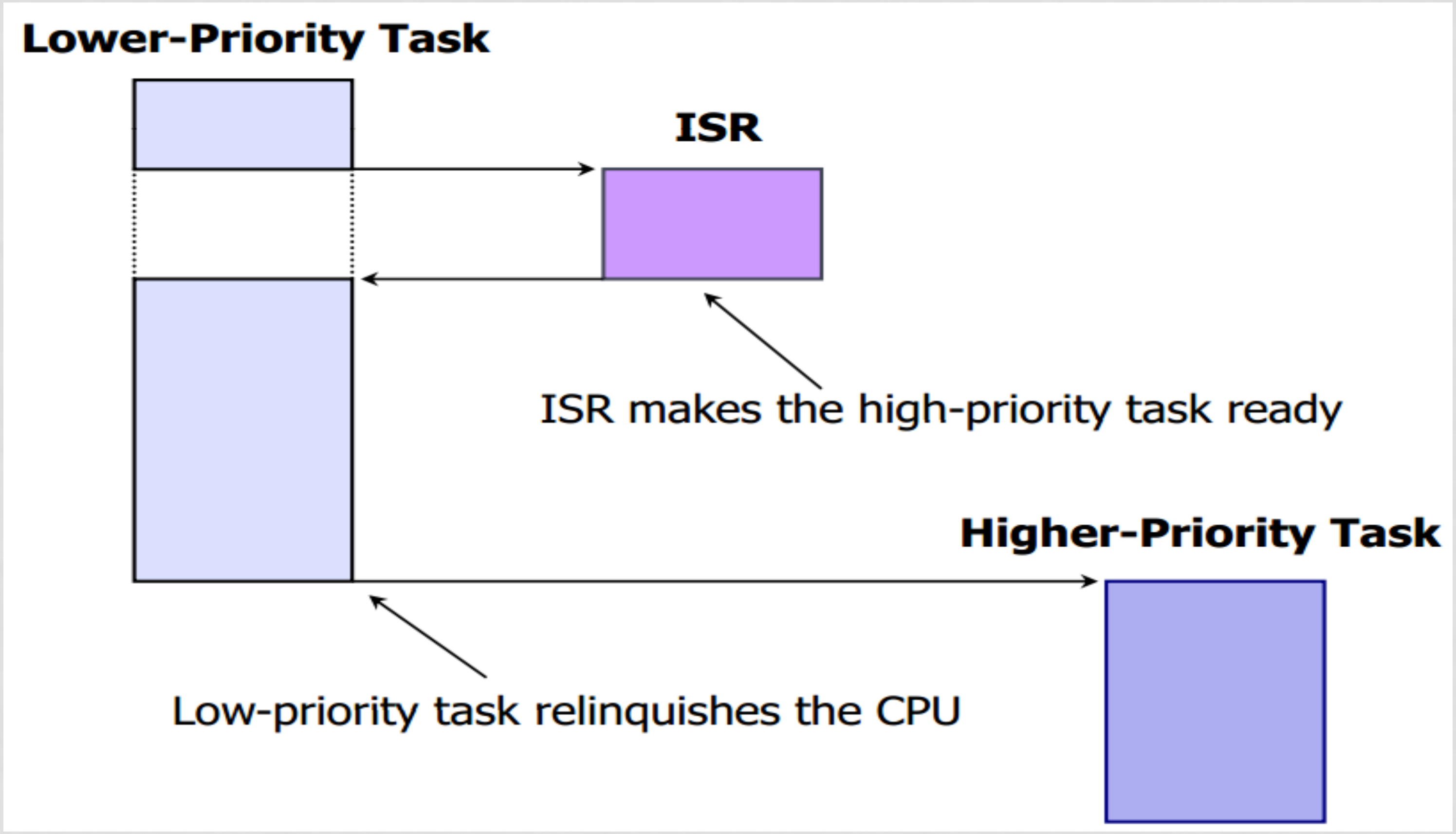
## SCHEDULING

**Preemptive Scheduling:**



# SCHEDULING

**Non preemptive Scheduling:**


Lower-Priority Task
ISR
ISR makes the high-priority task ready
Higher-Priority Task
Low-priority task relinquishes the CPU

## SCHEDULING

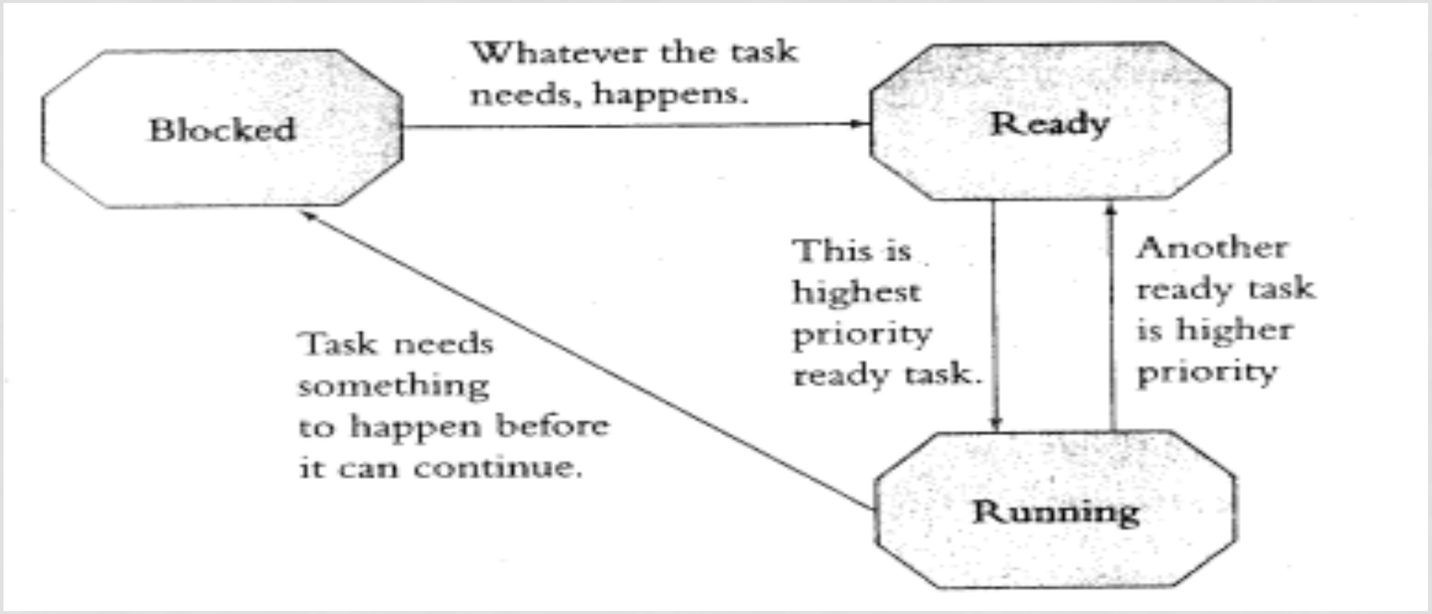
| Preemptive | Non-Preemptive |
|---|---|
| The resources are allocated to a process for a limited time. | Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state. |
| Process can be interrupted in between. | Process can not be interrupted till it terminates or switches to waiting state. |
| If a high priority process frequently arrives in the ready queue, low priority process may starve. | If a process with long burst time is running CPU, then another process with less CPU burst time may starve. |
| Preemptive scheduling has overheads of scheduling the processes. | Non-preemptive scheduling does not have overheads. |
| Preemptive scheduling is flexible. | Non-preemptive scheduling is rigid. |

## TASK AND ITS STATES

**Task** – a simple subroutine.

set of program instructions that are loaded in memory.

**Task states:** Running or Executing, Blocked(waited, dormant, delayed), Ready(pending, suspended).



Blocked
Whatever the task needs, happens.
Ready
This is highest priority ready task.
Another ready task is higher priority
Task needs something to happen before it can continue.
Running

## TASK AND ITS STATES

**Task** = Code + Data + State.

Task State is stored in a TCB(Task Control Block).

TCB:

| ID |
|---|
| Priority |
| Status |
| Registers |
| Saved PC |
| Saved SP |

## TASK AND ITS STATES

- **What happens if two tasks with the same priority are ready?**
- **If one task is running and another high priority task unblocks, does the task that is running get stopped and moved to the ready right away?**
- **What happens if all the tasks are blocked?**

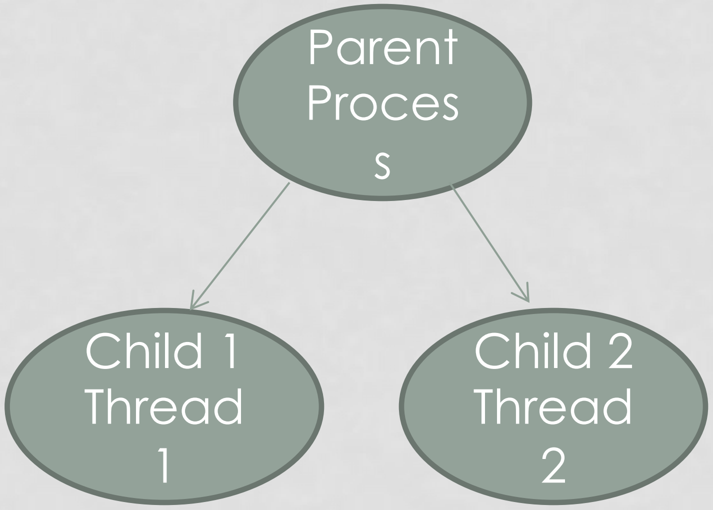
## PROCESS AND THREAD

Each application has multiple process.

Each process has its own address space, cpu quota, access to hardware resources and kernel services.

Process is allocated memory for instruction and data.(process needs memory for code and data.)

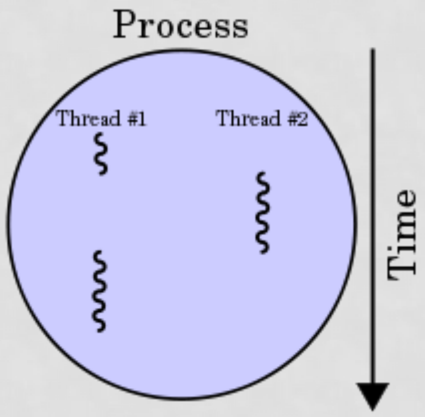
Example: Parent process is auto created during system start up. Thread is created dynamically.



# PROCESS AND THREADS

Process

| Process | Thread |
|---|---|
| Program in execution. | Function within executable. |
| Requires separate address space. | Shares same address space. |
| Inter Process communication is expensive | Inter thread communication is less expensive when compared to process. |
| Context switching from one process to other is costly. | Context switching between thread is less costly. |
| It has its own memory, program counter and stack. | It has shared memory. |

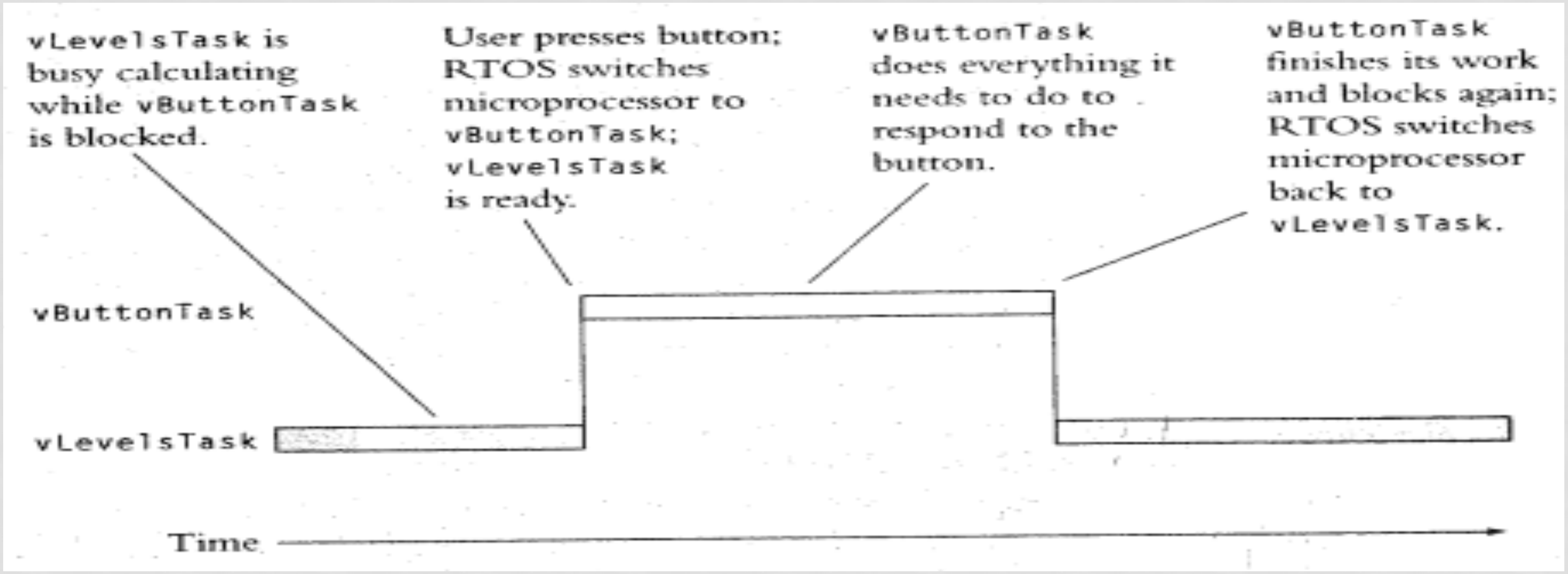


## TASK PRIORITY EXAMPLE

```
/*Button Task*/
void vButtonTask(void)          /*high priority*/
{
        while(True)
        {
                block until user pushes a button
                respond to the user
        }
}
/*Level Task*/
void vLevelTask(void)          /*low priority*/
{
        while(True)
        {
                Read levels of floats in the tank.
                Calculate average float level.
                Do some interminable calculations.
                figure out which tank to do next.
        }
}
```

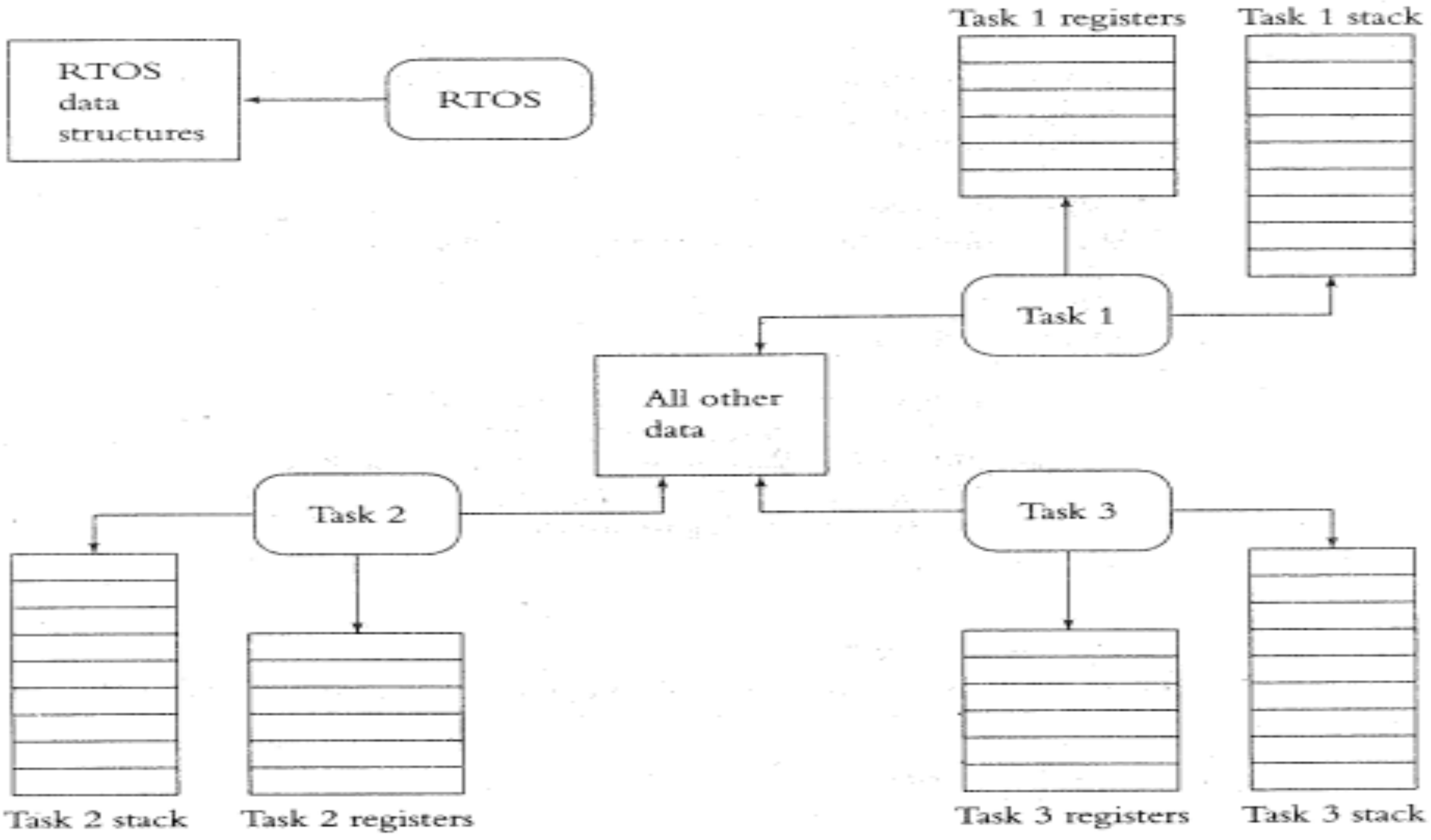
## TASK PRIORITY EXAMPLE

- RTOS will stop the low-priority **vLevelsTask** in its track and move it to ready state.


vLevelsTask is busy calculating while vButtonTask is blocked.
User presses button; RTOS switches microprocessor to vButtonTask; vLevelsTask is ready.
vButtonTask does everything it needs to do to respond to the button.
vButtonTask finishes its work and blocks again; RTOS switches microprocessor back to vLevelsTask.
vButtonTask
vLevelsTask
Time

- Runs the high priority **vButtonTask** task to let it respond to the user.
- **vButtonTask** task is finished responding, it blocks and the RTOS gives up back to the **vLevelsTask** task once again.

# TASK AND DATA



## SHARED DATA PROBLEM

Shared data problem" refers to the challenges and potential issues that arise when multiple threads or processes try to access and modify shared data concurrently.

The main issues related to the shared data problem are:

- **Data Race**

  A data race occurs when two or more threads access shared data simultaneously, at least one of them performs a write operation, and there is no proper synchronization mechanism in place.

- **Race Condition**

  A race condition occurs when the outcome of a program depends on the relative timing of events, such as the order of execution of threads, rather than the logical flow of the program.

- **Deadlock**

  A deadlock happens when two or more threads or processes are waiting indefinitely for each other to release a resource they need.

- **Data Inconsistency**

  If shared data is not synchronized correctly, it can lead to data inconsistency, where different threads or processes see different versions of the shared data, causing inconsistencies and unexpected results.

## SHARED DATA PROBLEM

```
void Task1(void)
        {
                vCountErrors(9);
        }
void Task2(void)
        {
                vCountErrors(11);
        }
Static int cErrors;
void vCountErrors(int cNewErrors)
{
                Mov R1, cErrors
                Add R1, cNewErrors
                Mov cErrors, R1
                return
}
```

## HOW TO DEAL WITH SHARED DATA PROBLEM: REENTRANCY

Functions that can be called by more than one task and will always work correctly even if the RTOS switches from one task to another in the middle of execution. (Or)
Function that can be interrupted at any time and resumed at a later time without loss of data. It either uses local variables or protects data when global variables are used. It can be used by more than one task without fear of data corruption.

3 rules to decide if a function is reentrant:

- It uses all shared variables in an atomic way.
- It does not call non-entrant functions.
- It does not use the hardware in a non-atomic way.

```
temp=foobar;                                  mov ax, [foobar]
temp+=1;                                       inc ax
foobar=temp;                                  mov [foobar], ax
```

**Atomic Version: foobar+=1**          **Atomic Version: inc[foobar]**