BECE403E-EMBEDDED SYSTEM DESIGN

MODULE-7

EMBEDDED REAL-TIME OPERATING SYSTEMS

MODULE-7

Embedded Real-Time Operating Systems

Introduction to basic concepts of RTOS - Task, process & threads, Multiprocessing and Multitasking, Preemptive and non-preemptive scheduling, Schedulability Analysis, Inter process Communication, Performance Metrics of RTOS

RTOS - INTRODUCTION

RTOS - INTRODUCTION

NEED FOR RTOS

Why not GPOS or EOS?

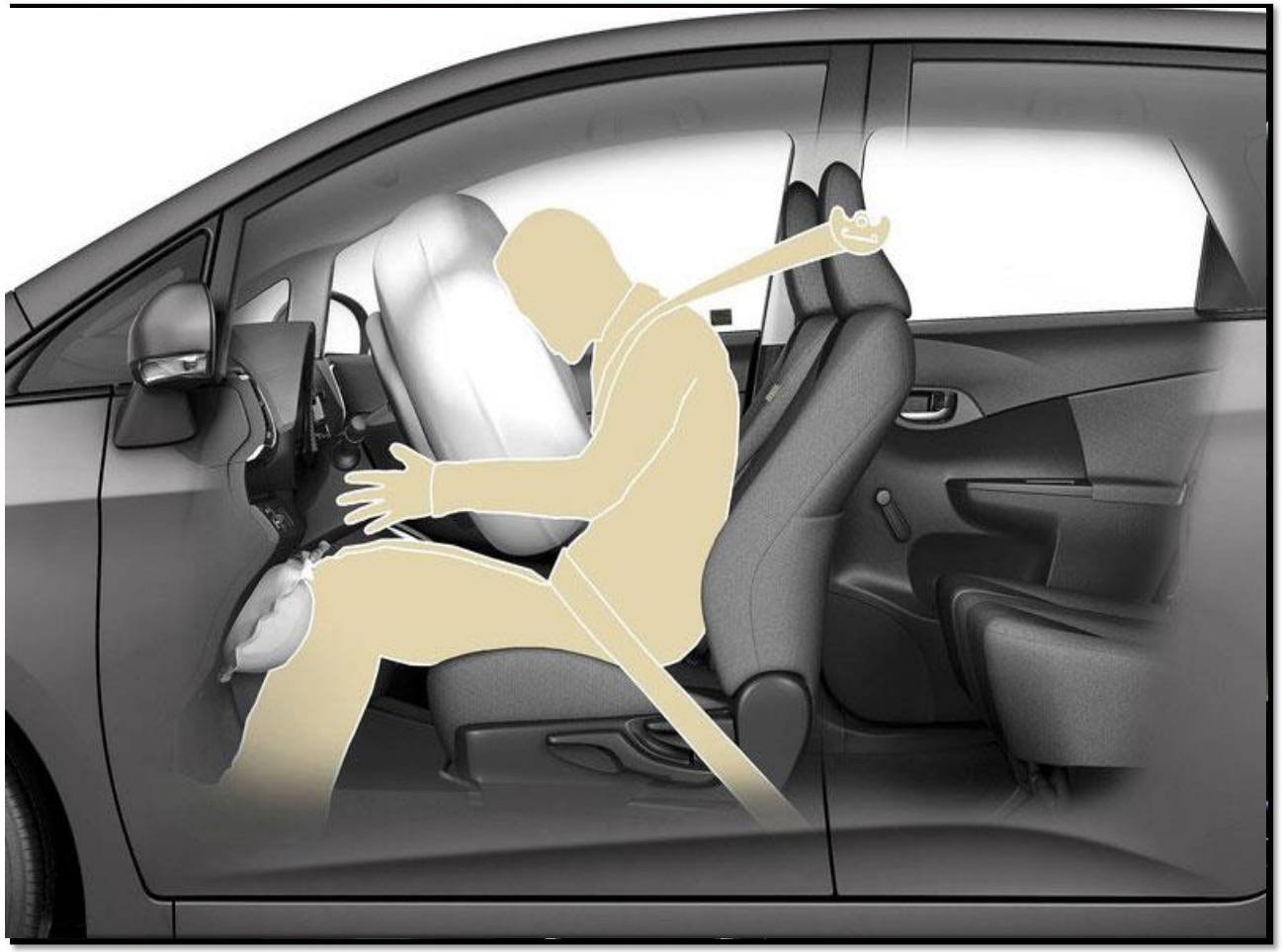
RTOS - INTRODUCTION

NEED FOR RTOS

When a fighter jet control system runs on GPOS encountered a hill, and you attempted to avoid…

RTOS - INTRODUCTION

NEED FOR RTOS



While you drive your car at high speed, suddenly collided with another vehicle if airbag activation system runs on EOS opens only after 10 sec….

RTOS - INTRODUCTION

WHAT IS RTOS?

- "Real time in operating systems is the ability of the OS to provide a required level of service in a bounded response time." - POSIX Standard 1003.1
- "A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application requests." - Wikipedia
- "RTOS: Any OS where interrupts are guaranteed to be handled within a certain specified time, there by making it suitable for control hardware in embedded system & time critical applications." - Dictionary.com
- A real-time operating system (RTOS) is an operating system that guarantees a certain capability within a specified time constraint to serve real time embedded applications.

RTOS - INTRODUCTION

RTOS - FEATURES

- **Multitasking and Pre-emptibility:** An RTOS must be multi-tasked and pre-emptible to support multiple tasks in real-time applications.
- **Task Priority:** In RTOS, pre-emption capability is achieved by assigning individual task with the appropriate priority level.
- **Inter Task Communication:** For multiple tasks to communicate in a timely manner and to ensure data integrity among each other, reliable and sufficient inter-task communication and synchronization mechanisms are required.
- **Priority Inheritance:** To allow applications with stringent priority requirements to be implemented, RTOS must have a sufficient number of priority levels when using priority scheduling.

RTOS - INTRODUCTION

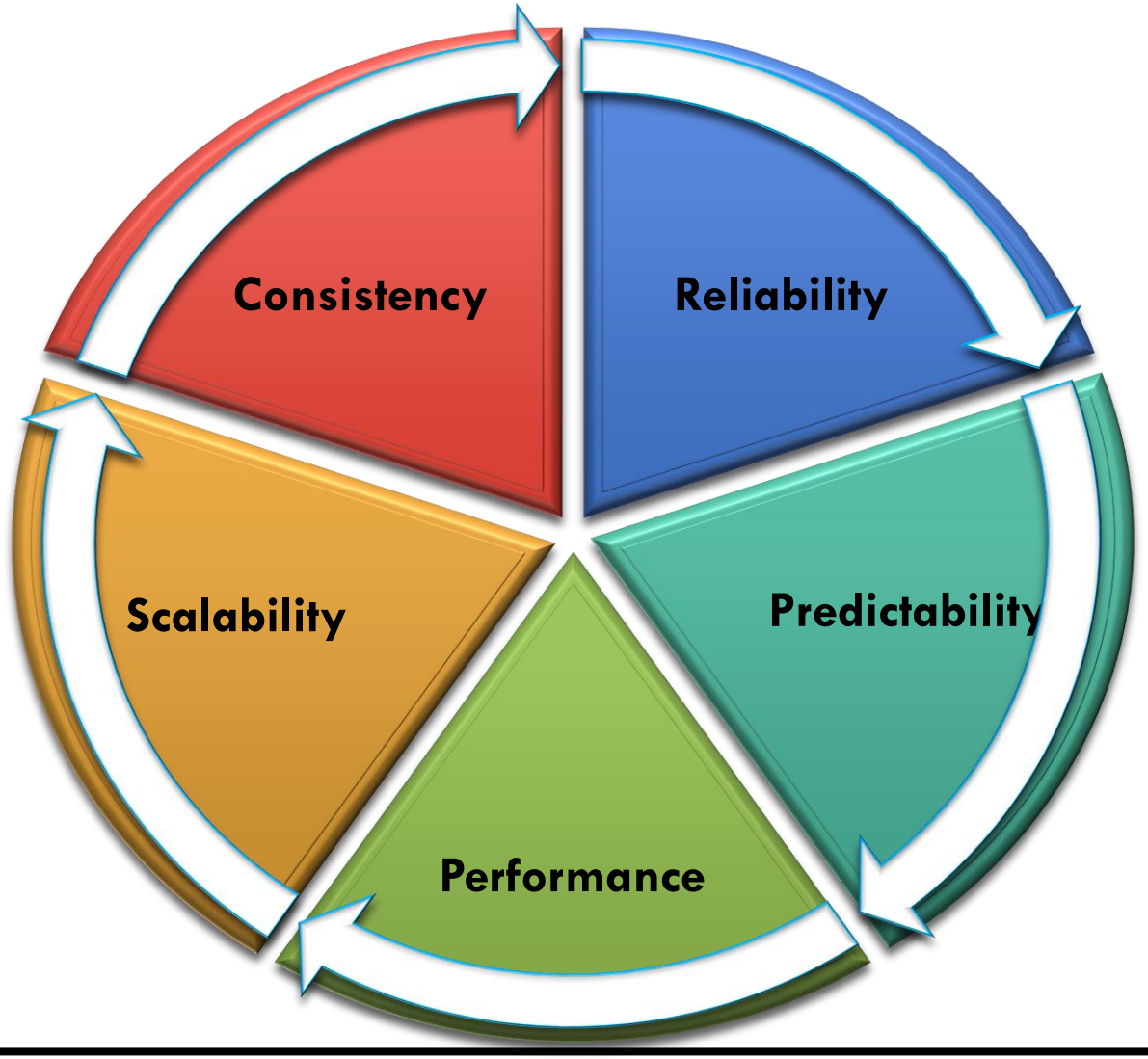
RTOS - FEATURES

- **Control of Memory Management:** To ensure predictable response to an interrupt, an RTOS should provide way for task to lock its code and data into real memory.

- **Predefined Short Latencies:**
  - **Task switching latency:** The time needed to save the context of a currently executing task and switching to another task is desirable to be short.
  - **Interrupt latency:** The time elapsed between execution of the last instruction of the interrupted task and the first instruction in the interrupt handler.
  - **Interrupt dispatch latency:** The time from the last instruction in the interrupt handler to the next task scheduled to run.

RTOS - INTRODUCTION

RTOS - CHARACTERISTICS



RTOS - INTRODUCTION

RTOS - TYPES

- HARD RTOS strictly adhere to the deadline associated with the tasks and the degree of tolerance for missed deadlines is negligible.
- A missed deadline can result in catastrophic failure of the system
- Examples:
  - Missile Navigation Systems
  - Vehicle Air Bags Control
  - Nuclear Power Plant Control

RTOS - INTRODUCTION

RTOS - TYPES

- FIRM RTOS tolerates a low occurrence of missing a deadline.
- Missing a deadline may result in an unacceptable reduction in quality of a product not lead to failure of the complete system.
- Examples:
  - Robot in car assembly section
  - Food processing control system
  - Weather monitoring system

RTOS - INTRODUCTION

RTOS - TYPES

- SOFT RTOS allows for frequently missed deadlines, and as long as tasks are timely executed their results continue to have value.
- Even the soft real time systems cannot miss the deadline for every task or process according to the priority it should meet the deadline. (Best effort system)
- Examples:
  - Multimedia transmission & reception
  - Digital cameras & mobile phones
  - Computer games

RTOS - INTRODUCTION

RTOS - APPLICATIONS

RTOS - INTRODUCTION

RTOS - APPLICATIONS

Mars Curiosity Rover, Uses ***VxWorks*** (For Split-second decision taking) and ***µc/os-II*** (For Sample Analysis)

RTOS - INTRODUCTION

RTOS - APPLICATIONS

F-35 Fighter aircraft uses a ***Integrity*** DO-178B, a POSIX based RTOS developed by Green Hills Software

RTOS - INTRODUCTION

RTOS - APPLICATIONS

International medical technology group Elekta is basing its new generations of equipment on the ***LynxOS-SE*** (RTOS)

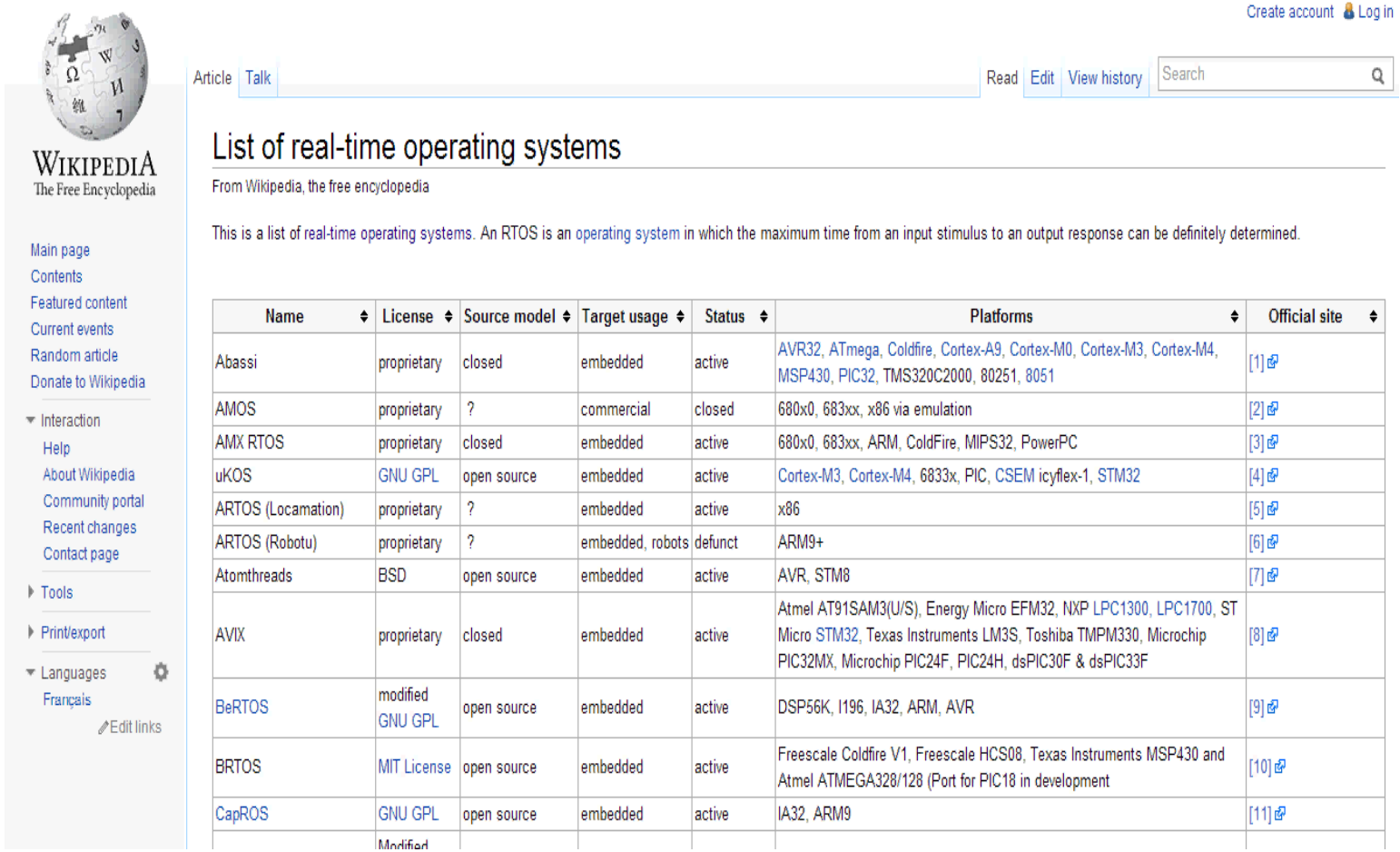
RTOS - INTRODUCTION

RTOS - APPLICATIONS

ThreadX RTOS is used by HP (All Printers) & Honeywell (Advanced security systems) in consumer electronics devices

RTOS - INTRODUCTION

RTOS IN EMBEDDED MARKET



Create account Log in

Article Talk | Read Edit View history | Search

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

Print/export

Languages
Français
Edit links

List of real-time operating systems

From Wikipedia, the free encyclopedia

This is a list of real-time operating systems. An RTOS is an operating system in which the maximum time from an input stimulus to an output response can be definitely determined.

Name	License	Source model	Target usage	Status	Platforms	Official site
Abassi	proprietary	closed	embedded	active	AVR32, ATmega, Coldfire, Cortex-A9, Cortex-M0, Cortex-M3, Cortex-M4, MSP430, PIC32, TMS320C2000, 80251, 8051	[1]
AMOS	proprietary	?	commercial	closed	680x0, 683xx, x86 via emulation	[2]
AMX RTOS	proprietary	closed	embedded	active	680x0, 683xx, ARM, ColdFire, MIPS32, PowerPC	[3]
uKOS	GNU GPL	open source	embedded	active	Cortex-M3, Cortex-M4, 6833x, PIC, CSEM icyflex-1, STM32	[4]
ARTOS (Locamation)	proprietary	?	embedded	active	x86	[5]
ARTOS (Robotu)	proprietary	?	embedded, robots	defunct	ARM9+	[6]
Atomthreads	BSD	open source	embedded	active	AVR, STM8	[7]
AVIX	proprietary	closed	embedded	active	Atmel AT91SAM3(U/S), Energy Micro EFM32, NXP LPC1300, LPC1700, ST Micro STM32, Texas Instruments LM3S, Toshiba TMPM330, Microchip PIC32MX, Microchip PIC24F, PIC24H, dsPIC30F & dsPIC33F	[8]
BeRTOS	modified GNU GPL	open source	embedded	active	DSP56K, I196, IA32, ARM, AVR	[9]
BRTOS	MIT License	open source	embedded	active	Freescale Coldfire V1, Freescale HCS08, Texas Instruments MSP430 and Atmel ATMEGA328/128 (Port for PIC18 in development	[10]
CapROS	GNU GPL	open source	embedded	active	IA32, ARM9	[11]
	Modified					

There are more than 100 RTOS currently available in embedded market

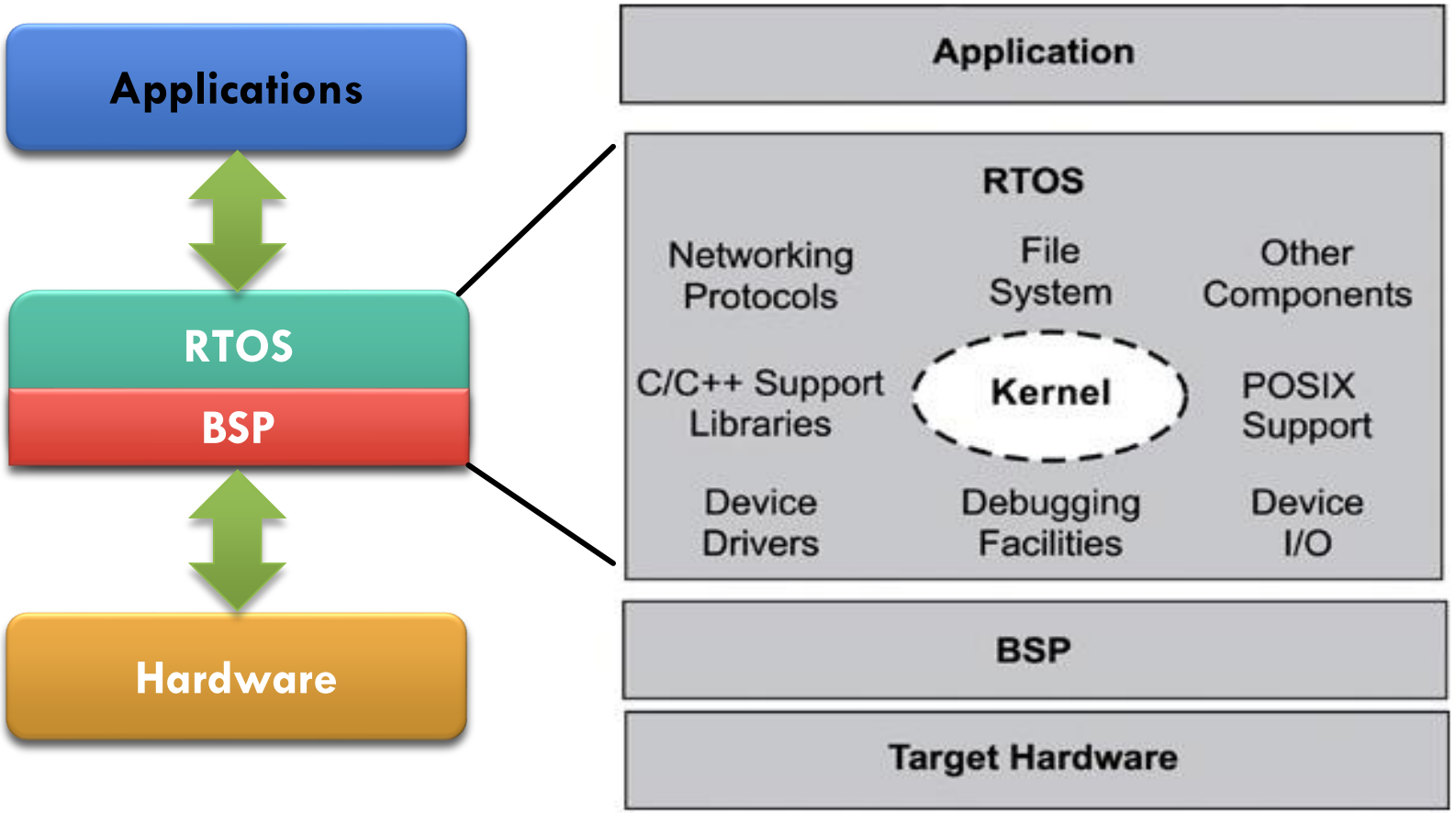
RTOS - INTRODUCTION

MOST COMMONLY USED RTOS

RTOS - INTERNALS

RTOS - ARCHITECTURE

RTOS – ARCHITECTURE



RTOS - ARCHITECTURE

KERNEL

- An RTOS generally avoids implementing the kernel as a monolithic program.
- The kernel is developed instead as a micro-kernel with added configurable functionalities like scheduling, Managing memory protection and IPC.
- All other basic services can be made part of user space and can be run in the form of servers.
- This implementation gives resulting benefit in increase system configurability, as each embedded application requires a specific set of system services with respect to its characteristics.
- QNX, VxWorks OS follows the Microkernel approach

RTOS - ARCHITECTURE

BSP

- In embedded systems, a board support package (BSP) is the layer of software containing hardware-specific drivers and other routines
- This allow a particular operating system (traditionally a RTOS) to function in a particular hardware environment integrated with the RTOS itself.
- Third-party hardware developers who wish to support a particular RTOS must create a BSP that allows that RTOS to run on their platform.
- BSPs are typically customizable, allowing the user to specify which drivers and routines should be included in the build based on their selection of hardware and software options.

RTOS - ARCHITECTURE

BSP

- For instance, a particular single-board computer might be paired with any of several graphics cards; in that case the BSP might include a driver for each.

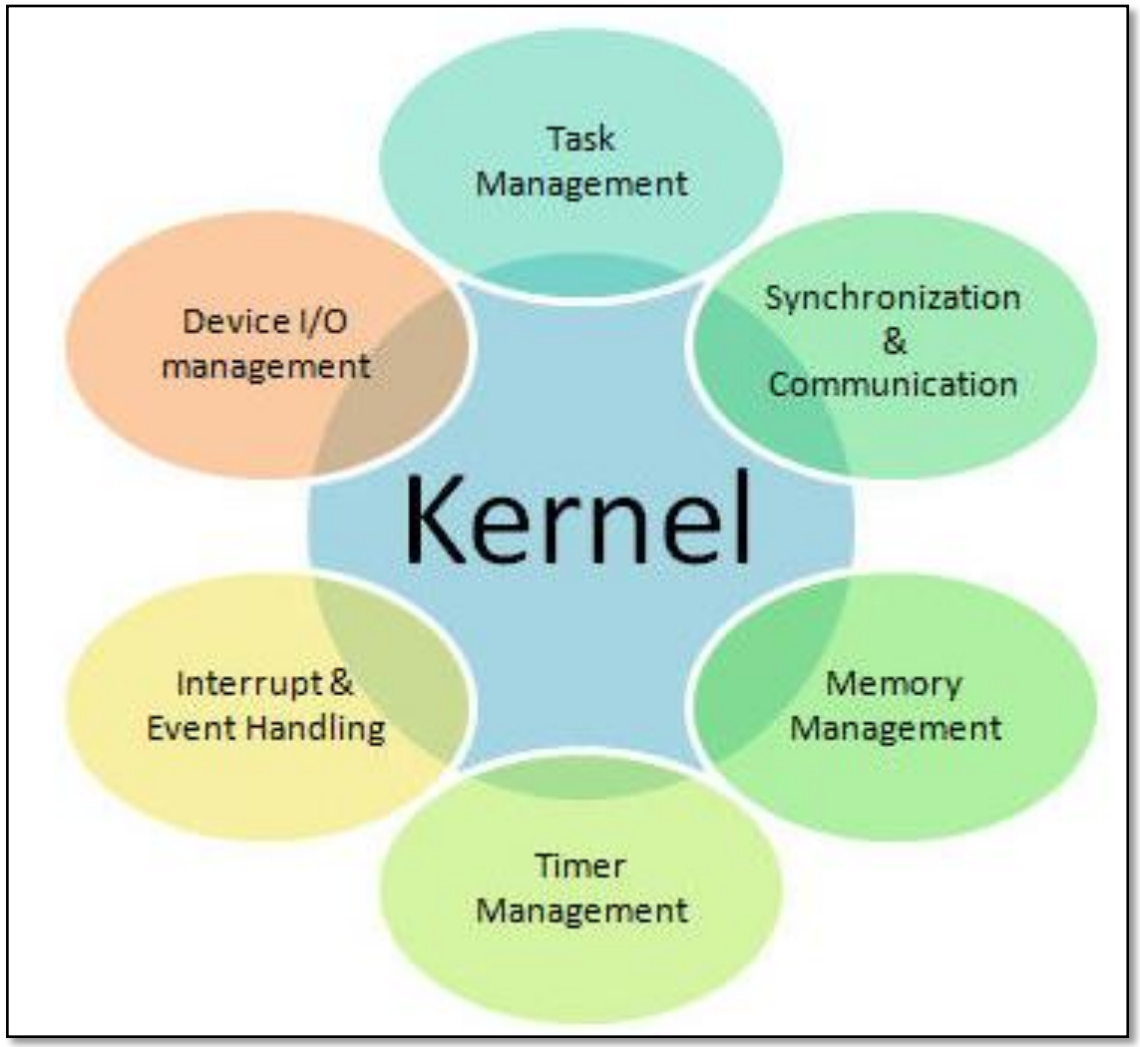
- While building the BSP image the user would specify which graphics driver to include based in his choice of hardware.

- BSP is supposed to perform the following operations
  - Initialize the processor
  - Initialize the bus
  - Initialize the interrupt controller
  - Initialize the clock
  - Initialize the RAM settings
  - Load and run boot loader from flash

RTOS – KERNEL SERVICES

RTOS – KERNEL SERVICES

RTOS – KERNEL SERVICES



RTOS – KERNEL SERVICES

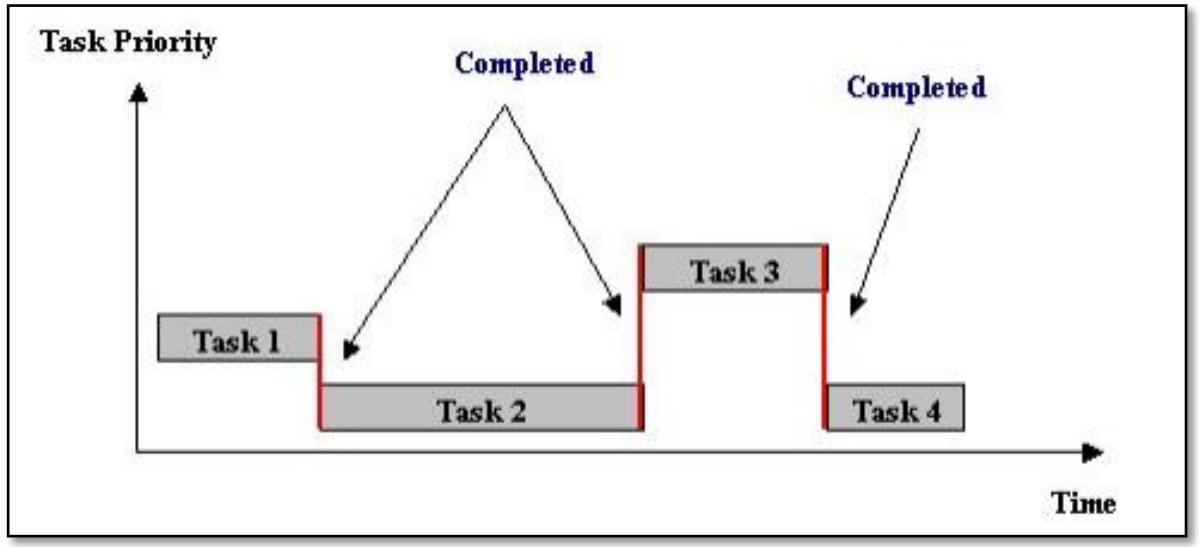
TASK MANAGEMENT

- To achieve concurrency in embedded application program, it is decompose into small, schedulable, and sequential program units known as "Task".
- Task management allows programmers to design their software as a number of separate "chunks" of codes with each having a distinct goal and deadline.
- This service encompasses mechanism such as scheduler and dispatcher that creates and maintain task objects.
- Scheduler : It keeps record of the state of each task and selects from among them that are ready to execute and allocates the CPU to one of them.
- Two types of schedulers: non-preemptive and priority-based preemptive.

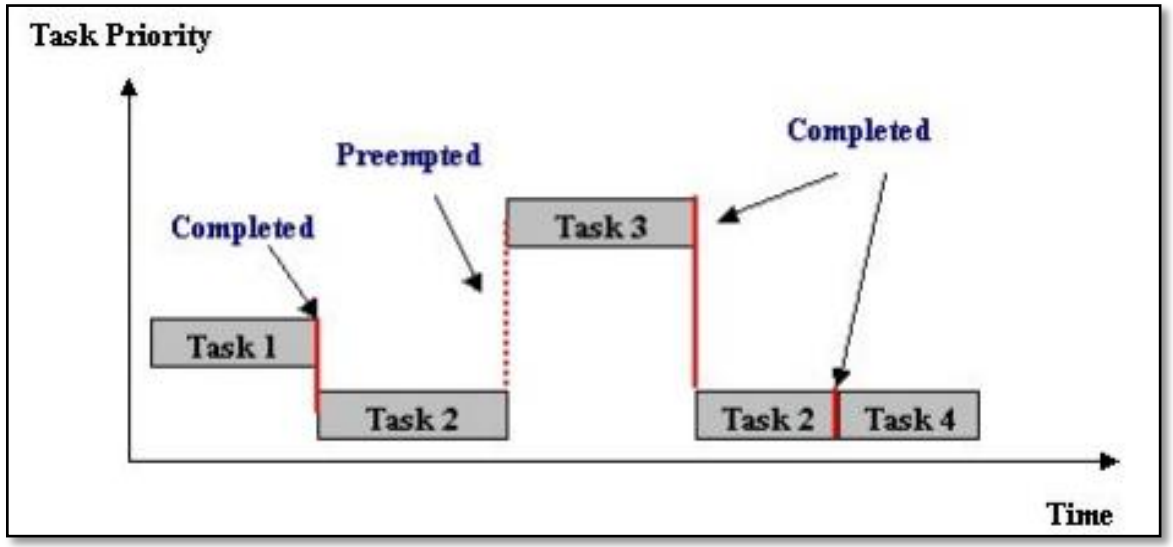
RTOS – KERNEL SERVICES

TASK MANAGEMENT

- A scheduler helps to maximize CPU utilization among different tasks in a multi-tasking program and to minimize waiting time.
- Dispatcher: It gives control of the CPU to the task selected by the scheduler by performing context switching and changes the flow of execution.



Non-Preemptive



Preemptive

RTOS – KERNEL SERVICES

SYNCHRONIZATION & COMMUNICATION

- Task Synchronization & Inter Task Communication (IPC) serves to enable information to be transmitted safely from one task to another.
- Synchronization is essential for tasks to share mutually exclusive resources (devices, buffers, etc.,) and/or allow multiple concurrent tasks to be executed.
- Most commonly used task synchronizations are Semaphore and Mutex
- Inter task communication involves sharing of data among tasks through common memory space, transmission of data, etc.
- Inter task communications is achieved in EOS by Shared memory, Message queue, Pipe, Mailbox etc.,

RTOS – KERNEL SERVICES

MEMORY MANAGEMENT

- An embedded OS usually strive to achieve small footprint by including only the functionality needed for the user's applications.
- Two types of memory managements are provided in EOS – Stack and Heap.
- In a multi-tasking EOS, each task needs to be allocated with an amount of memory for storing their contexts (such as registers contents, program counter) for context switching.
- This allocation of memory is done using task-control block model and this set of memory is commonly known as kernel stack and the management process termed Stack Management.

RTOS – KERNEL SERVICES

MEMORY MANAGEMENT

- Upon the completion of a program initialization, physical memory of the MCU or MPU will usually be occupied with program code, program data and system stack. The remaining physical memory is called heap.
- This heap memory is typically used by the kernel for dynamic memory allocation of data space for tasks.
- The memory is divided into fixed size memory blocks, which can be requested by tasks.
- When a task finishes using a memory block it must return it to the pool. This process of managing the heap memory is known as Heap management.

RTOS – KERNEL SERVICES

INTERRUPT AND EVENT HANDLING

- A fundamental challenge in RTOS design is supporting interrupts and thereby allowing asynchronous access to internal RTOS data structures.

- The interrupt and event handling mechanism provides following functions:
  - Defining interrupt handler
  - Creation and deletion of ISR
  - Referencing the state of an ISR
  - Enabling and disabling of an interrupt
  - Changing and referencing of an interrupt mask

RTOS – KERNEL SERVICES

INTERRUPT AND EVENT HANDLING

- The interrupt and event handling mechanism of an RTOS helps to ensure:
  - ✓ Data integrity by restricting interrupts from occurring when modifying a data structure
  - ✓ Minimum interrupt latencies due to disabling of interrupts when RTOS is performing critical operations
  - ✓ Fastest possible interrupt responses that marked the preemptive performance of an RTOS
  - ✓ Shortest possible interrupt completion time with minimum overheads

RTOS – KERNEL SERVICES

TIMER MANAGEMENT

- In embedded systems, system and user tasks are often scheduled to perform after a specified duration.
- For example,
  - Ensure execution fairness in scheduling algorithm
  - Software-based memory refresh mechanism in dynamic memory
  - To schedule communication protocols related activities
- To implement above functions, there is a need for a periodical interrupt to keep track of time delays and timeout. This is taken care by Timer Management unit.

RTOS – KERNEL SERVICES

DEVICE I/O MANAGEMENT

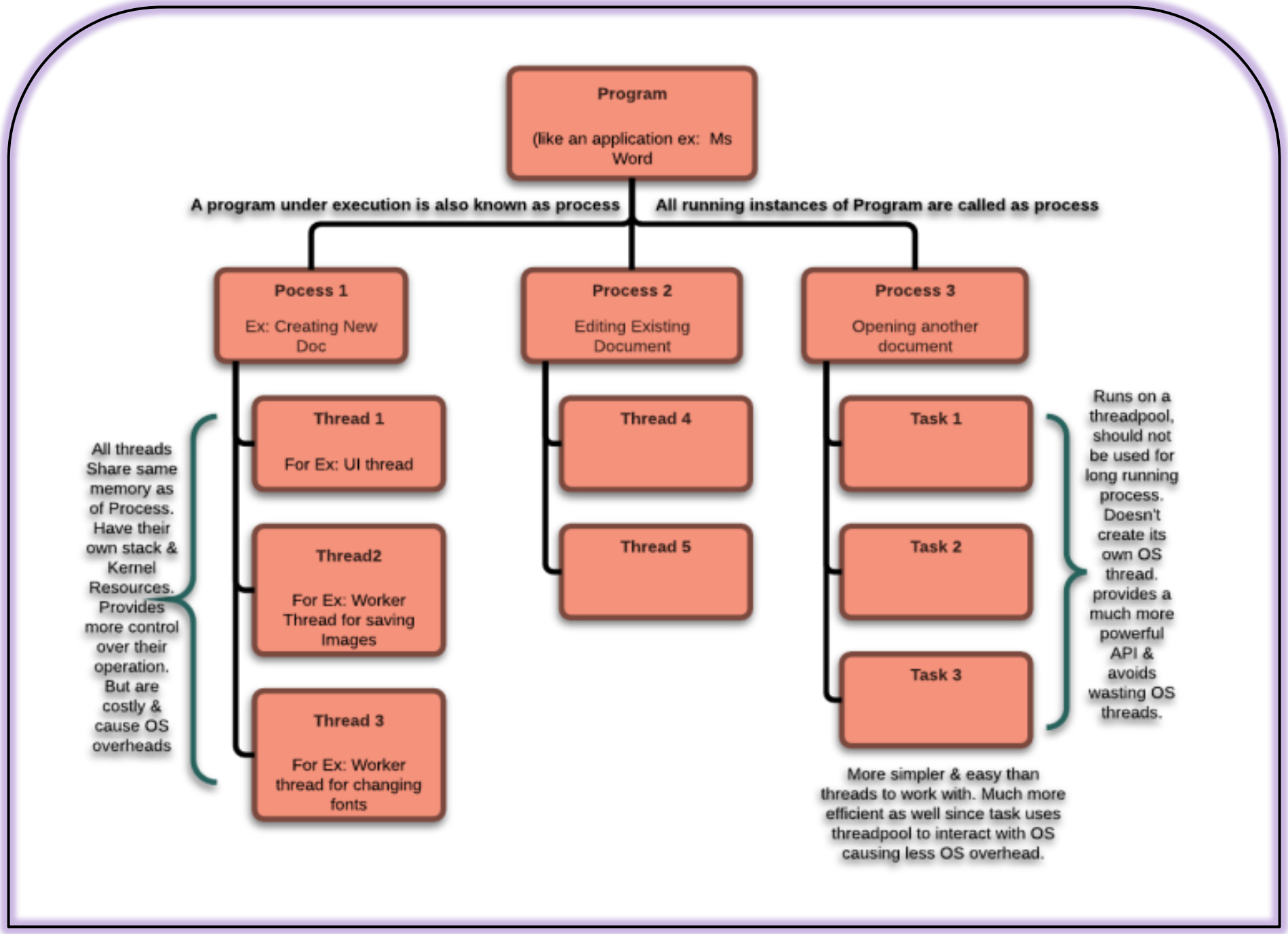
- This service provide a uniform framework ("API") and supervision facility via the available kernel system calls for an embedded system to organize and access large numbers of diverse hardware device drivers.

- An RTOS device I/O management also needs to
  - Manage asynchronous communication (event-driven requests)
  - Manage synchronous communication (data transfers)
  - Providing protection to I/O devices
  - Managing a fair and efficient I/O sharing scheme

TASK, PROCESS & THREAD

PROCESS MANAGEMENT

- A software is a collection of programs, libraries and related data in a system.
- A program is simply a passive, static sequence of instructions that could represent a system's hardware and software resources which performs a specific task when executed.
- A process (a.k.a task) is basically a program in execution and it is created by an OS to encapsulate all the information that is involved in the executing of a program (stack, PC, source code, data, etc.).
- In a multitasking EOS, system activities are divided up into simpler, separate components known as threads (lightweight processes) which has smallest sequence of instructions that can be managed independently by a scheduler.


Program
(like an application ex: Ms Word
A program under execution is also known as process
All running instances of Program are called as process
Pocess 1
Ex: Creating New Doc
Process 2
Editing Existing Document
Process 3
Opening another document
Thread 1
For Ex: UI thread
Thread2
For Ex: Worker Thread for saving Images
Thread 3
For Ex: Worker thread for changing fonts
Thread 4
Thread 5
Task 1
Task 2
Task 3
All threads Share same memory as of Process. Have their own stack & Kernel Resources. Provides more control over their operation. But are costly & cause OS overheads
Runs on a threadpool, should not be used for long running process. Doesn't create its own OS thread. provides a much more powerful API & avoids wasting OS threads.
More simpler & easy than threads to work with. Much more efficient as well since task uses threadpool to interact with OS causing less OS overhead.

TASK, PROCESS & THREAD

- The terms "task," "thread," and "process" are often used in the context of concurrent programming, but they refer to different concepts.

- Task:

  - A task is a unit of work to be completed by a program or system.

  - It can represent any discrete piece of work, such as processing data, handling user input, or executing a specific function.

  - Tasks can be sequential or concurrent, depending on whether they can be executed independently or need to wait for other tasks to complete.

TASK, PROCESS & THREAD

- Thread:
  - A thread is the smallest unit of execution within a process.
  - Threads share the same memory space and resources within a process, allowing them to communicate and coordinate directly.
  - Multiple threads within a single process can execute concurrently, enabling parallelism and multitasking.
  - Threads are lightweight compared to processes, as they share resources and can be created and destroyed more efficiently.

TASK, PROCESS & THREAD

- Process:
  - A process is an instance of a running program.
  - Each process has its own memory space, resources, and execution environment, providing isolation and protection from other processes.
  - Processes can contain multiple threads, allowing for parallel execution of tasks within the same program.
  - Processes communicate and coordinate with each other through inter-process communication (IPC) mechanisms.

PROCESS MANAGEMENT

PROCESS vs THREAD

Comparison Basis	Process	Thread
Definition	A process is a **program under execution** i.e an active program.	A thread is a **lightweight process** that can be managed independently by a scheduler.
Context switching time	Processes require **more time** for context switching as they are more heavy.	Threads require **less time** for context switching as they are lighter than processes.
Memory Sharing	Processes are independent and **don't share** memory.	A thread may **share** some memory with its peer threads.
Communication	Communication between processes **requires more time** than between threads.	Communication between threads **requires less time** than between processes .
Blocked	If a process gets blocked, remaining processes can **continue execution**.	If a user level thread gets blocked, all of its **peer threads also get blocked**.
Resource Consumption	Processes require **more resources** than threads.	Threads generally need **less resources** than processes.
Dependency	Individual processes are **independent** of each other.	Threads are parts of a process and so are **dependent.**
Data and Code sharing	Processes have **independent** data and code segments.	A thread **shares** the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are **treated separately** by the operating system.	All user level peer threads are **treated as a single task** by the operating system.

MULTITHREADING & MULTIPROCESSING

OS – KERNEL FEATURES

MULTITASKING SYSTEM

- What is Multitasking?
  - Separate tasks share one processor (or processors)
  - Tasks may interact to execute as a whole program
  - Each task executes within its own context
    - Owns processor
    - Sees its own variables
    - May be interrupted

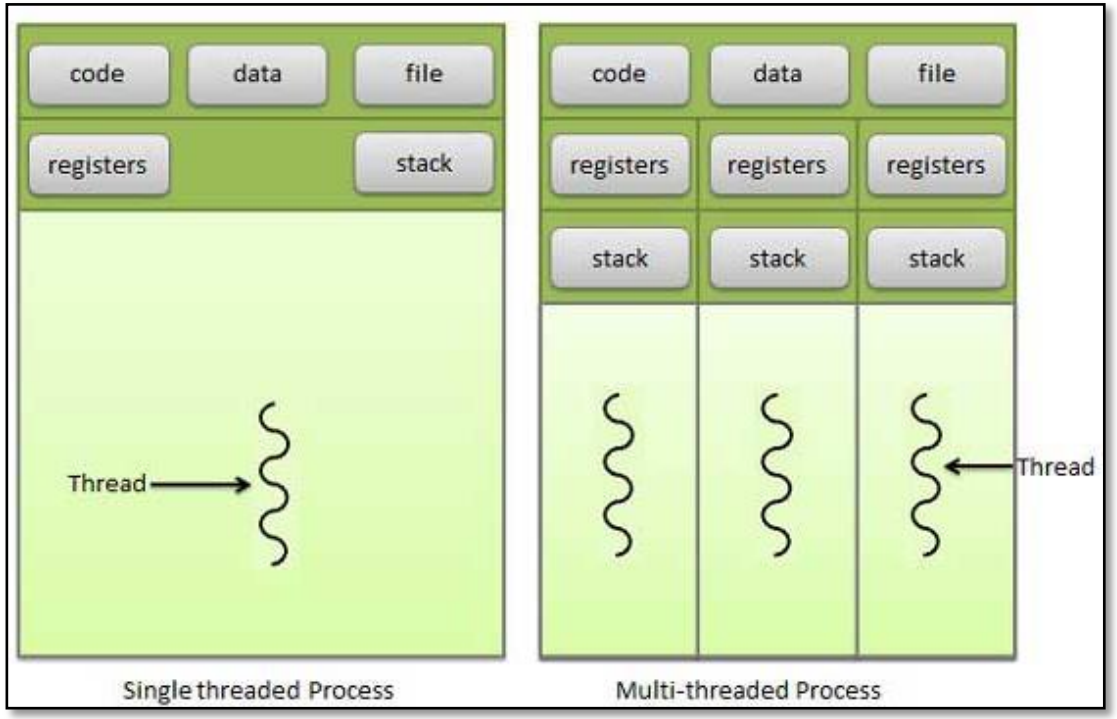
- Context switching: When the CPU switches from running one task to running another, it is said to have switched contexts.

- Save the MINIMUM needed to restore the interrupted process such as contents of registers, program counter, special variables, memory page registers etc.,

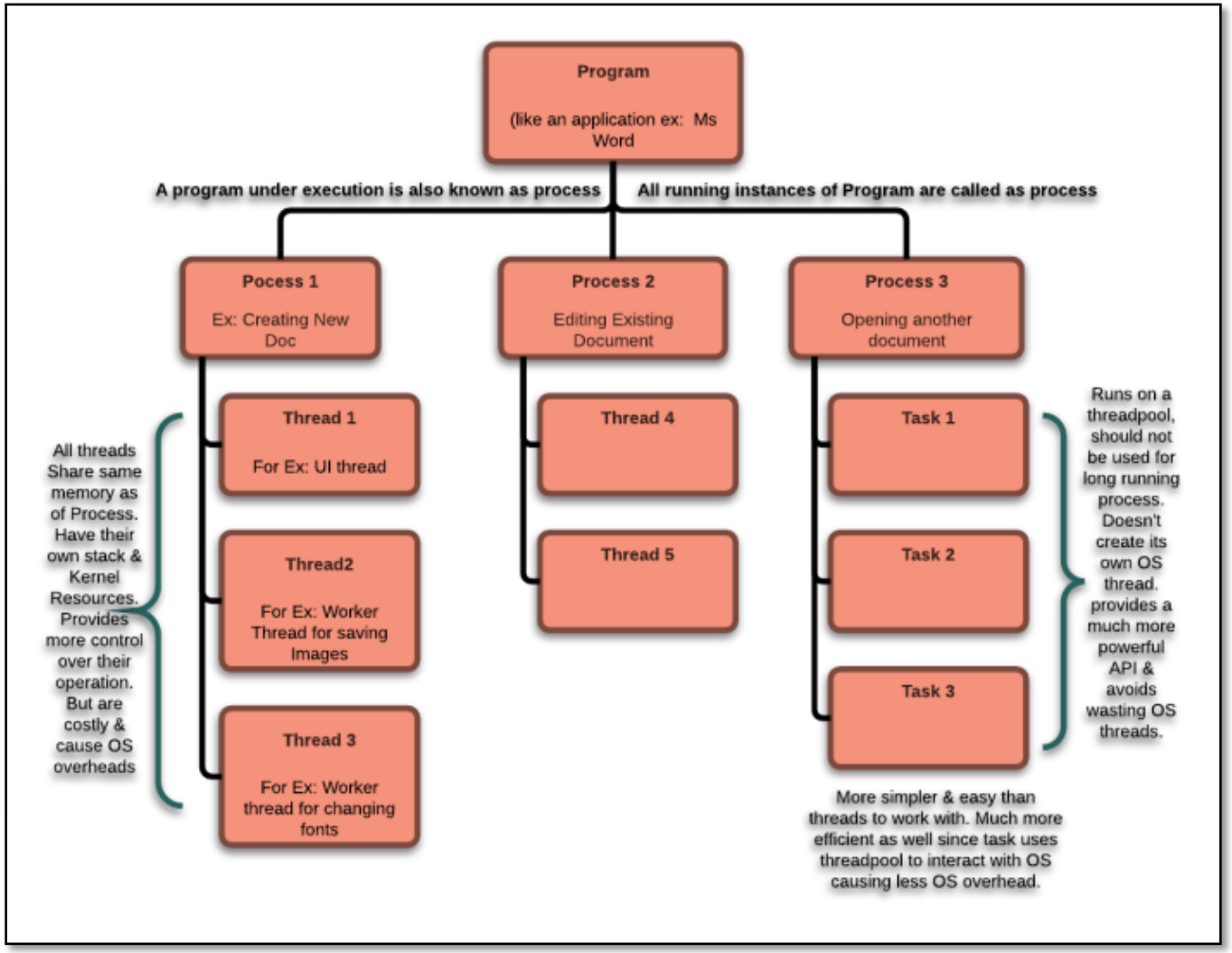
MULTITHREADING

- Multithreading is a programming technique that enables a single process to perform multiple tasks concurrently.
- It allows a program to execute multiple threads of execution within the same process, where each thread represents a separate flow of control.
- Multithreading can improve the performance and responsiveness of applications, especially in tasks that involve I/O operations or parallel computation.
- By utilizing multiple threads, a program can make better use of available CPU resources and handle multiple tasks simultaneously, enhancing efficiency and responsiveness.

PROCESS MANAGEMENT



Single Vs Multi-threaded Process



Example for Process and Thread

MUTLIPROCESSING

- Multiprocessing refers to the capability of a system to execute multiple processes concurrently using multiple CPUs or CPU cores.
- Unlike multithreading, which involves executing multiple threads within a single process, multiprocessing involves running multiple processes simultaneously, each with its own memory space.
- This allows for true parallelism, as each process can execute independently and utilize separate resources.
- Multiprocessing is commonly used in modern computer systems, especially in servers, high-performance computing clusters, and systems that require intensive computational tasks.

MUTLIPROCESSING

- It offers advantages such as improved performance, scalability, and fault tolerance by distributing tasks across multiple processors or cores.
- Additionally, multiprocessing can enhance system responsiveness by allowing tasks to run concurrently without blocking each other.
- However, multiprocessing also introduces challenges related to communication and coordination between processes, as well as managing shared resources and avoiding conflicts.
- Techniques such as inter-process communication (IPC) and synchronization mechanisms are used to address these challenges and ensure proper coordination among processes.

SCHEDULABILITY ANALYSIS

SCHEDULABILITY ANALYSIS

SCHEDULING CRITERIA

1. CPU utilization: CPU should be working most of the time (Ideally 100% all the time)
2. Throughput: total number of processes completed per unit time(10 tasks/second)
3. Turnaround time (TAT): amount of time taken to execute a particular process, TATi=CTi–ATi (Where CTi → Completion Time, ATi → Arrival Time)
4. Waiting time(WT): time periods spent waiting in the ready queue by a process to acquire get control on the CPU, WTi = TATi – BTi (Where BTi → CPU burst time)
5. Load average: average number of processes residing in the ready queue waiting for their turn to get into the CPU
6. Response time: Amount of time it takes from when a request was submitted until the first response is produced

SCHEDULING OBJECTIVE: **Max →CPU utilization, Throughput**
Min → Turnaround time, Waiting time, Load average, Response time

SCHEDULABILITY ANALYSIS

TASK MODEL

- A task = (C, P)
  - C: worst case execution time/computing time (C<=P!)
  - P: period (D=P)
  - C/P is CPU utilization of a task

U>1 (overload): some task will fail to meet its deadline
U<1 : it will depend on the scheduling algorithms
U=1 : CPU is kept busy, all deadlines will be met

- A task set: (Ci,Pi)
  - All tasks are independent
  - The periods of tasks start at 0 simultaneously
  - $U=\Sigma(Ci/Pi)$ is CPU utilization of a task set

- CPU utilization is a measure on how busy the processor could be during the shortest repeating cycle: P1*P2*...*Pn

SCHEDULABILITY ANALYSIS

SCHEDULABILITY TEST

- Schedulability test determine whether a given task set is feasible to schedule?
- Necessary test:
  - If test is passed, tasks may be schedulable but not necessarily
  - If test is not passed, tasks are definitely not schedulable
- Sufficient test:
  - If test is passed, then task are definitely schedulable
  - If test is not passed, tasks may be schedulable but not necessarily
- Exact test: (Necessary test + Sufficient test)
  - The task set is schedulable if and only if it passes the test

SCHEDULABILITY ANALYSIS

SCHEDULABILITY TEST

- Necessary test

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{P_i} \right) \leq 1$$

- Sufficient test (Utilization Bound test)

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{PT_i} \right) \leq N(2^{\frac{1}{N}} - 1)$$

N	B(N)
1	1.0
2	0.828
3	0.779
4	0.756
5	0.743
6	0.734
∞	0.693

REAL TIME SCHEDULING

REAL TIME SCHEDULING

- Scheduling refers to the way processes are assigned to run on the available CPUs, since there are typically many more processes running on the system.
- Real-time scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU in RTOS.
- By switching the CPU among processes, the operating system can make the embedded system is more productive.
- Scheduling algorithm is the method by which threads, processes or data flows are given access to system resources
- The need for a scheduling algorithm arises from requirement for most modern systems to perform multitasking.

REAL TIME SCHEDULING

REAL-TIME SCHEDULING ALGORITHM

- The purpose of a real-time scheduling algorithm is to ensure that critical timing constraints, such as deadlines and response time, are met.
- Real-time systems use preemptive multitasking with priorities are assigned to tasks, and the RTOS always executes the task with highest priority.
- Most algorithms are classified as
  - Fixed-priority: Algorithm assigns priorities at design time, and those priorities remain constant for the lifetime of the task. Ex.: Rate-Monotonic Scheduling (RMS)
  - Dynamic-priority: Assigns priorities dynamically at runtime, based on execution parameters of tasks, such as upcoming deadlines. Ex.: Earliest Deadline First (EDF)
  - Mixed-priority: This algorithm has both static and dynamic components.

REAL TIME SCHEDULING

TYPES OF SCHEDULING ALGORITHM

- Non pre-emptive Algorithm:
  - First come First served (FCFS)
  - Priority(Non-pre-emptive)
  - Shortest Job First(SJF)

- Pre-emptive Algorithm:
  - Shortest remaining Time First(SRTF)
  - Round Robin(RR)
  - Priority(Pre-emptive)
  - Rate Monotonic Scheduling (RMS)
  - Earliest Deadline First (EDF)

REAL TIME SCHEDULING

RATE MONOTONIC SCHEDULING(RMS)

- **Concept:** Task with the shortest period executes with the highest priority.
- **Working :**
  - Rate-monotonic is a fixed priority based scheduling.
  - The scheduling scheme is pre-emptive; it ensures that a task is pre-empted if another task with a shorter period is expected to run.
  - Used in embedded systems where the nature of the scheduling is deterministic.
  - Consider three tasks with a period Task-1(10ms), Task-2(15ms), Task-3 (20ms), then as per RMS priority of the tasks can be assigned as:

priority (task1) > priority (task2) > priority (task3)

REAL TIME SCHEDULING

RATE MONOTONIC SCHEDULING(RMS) – EXAMPLE-1

- Consider set of task running on a Automotive control system as follows
  - Speed measurement Task (T1): C=20ms, P=100ms, D=100ms
  - ABS control Task (T2): C=40ms, P=150ms, D=150ms
  - Fuel injection Task (T3): C=100ms, P=350ms, D=350ms
  - Other software with soft deadlines e.g. audio, air condition etc.,

Necessary Test

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{P_i}\right) \leq 1$$

U=(20/100)+(40/150)+(100/350)
= 0.2 + 0.2666 + 0.2857 = 0.7517 ≤ 1

Necessary Test is passed hence given task set must be tested under sufficient test to conclude the Schedulability. But, if necessary test is failed we may conclude given task set is definitely not schedulable by any scheduling algorithm.

Sufficient Test

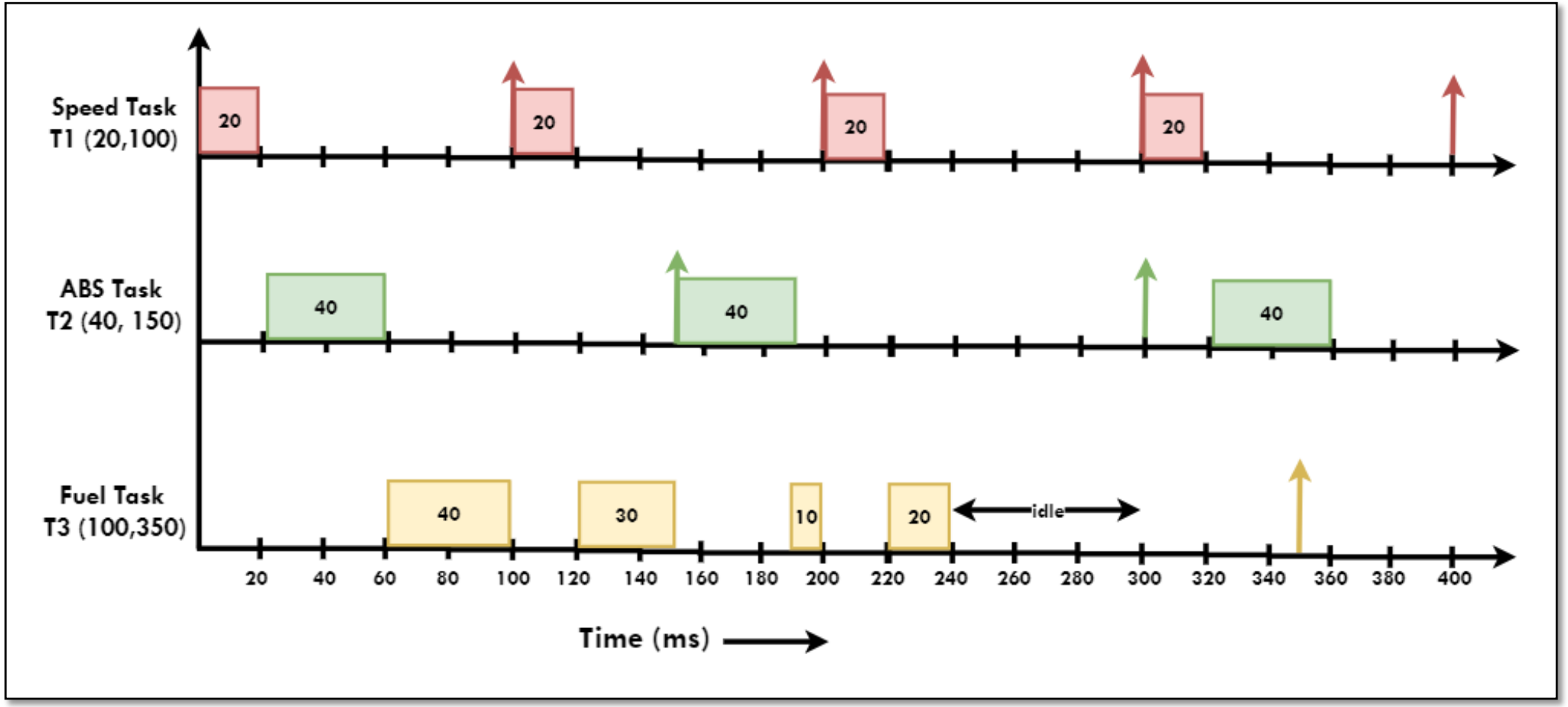
$$U = \sum_{i=1}^{N} \left(\frac{C_i}{PT_i}\right) \leq N(2^{\frac{1}{N}} - 1)$$

B(3)=0.779
U=0.7517 ≤ B(3) ≤ 1

Since given task set passes Sufficient test we may conclude it is definitely schedulable under RMS

REAL TIME SCHEDULING

RATE MONOTONIC SCHEDULING(RMS) – EXAMPLE-1



This is only for the first period. But this is enough to conclude that the task set is schedulable

REAL TIME SCHEDULING

RATE MONOTONIC SCHEDULING(RMS) – EXAMPLE-2

- Consider set of four task (Ci, Pi) running on embedded system as follows, T1= (1, 3) , T2= (1, 5) , T3= (1, 6) , T4= (2, 20)

Necessary Test

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{P_i}\right) \leq 1$$

U=(1/3)+(1/5)+(1/6)+(2/10)=0.899 ≤ 1

Necessary Test is passed hence given task set must be tested under sufficient test to conclude the Schedulability. But, if necessary test is failed we may conclude given task set is definitely not schedulable by any scheduling algorithm.

Sufficient Test

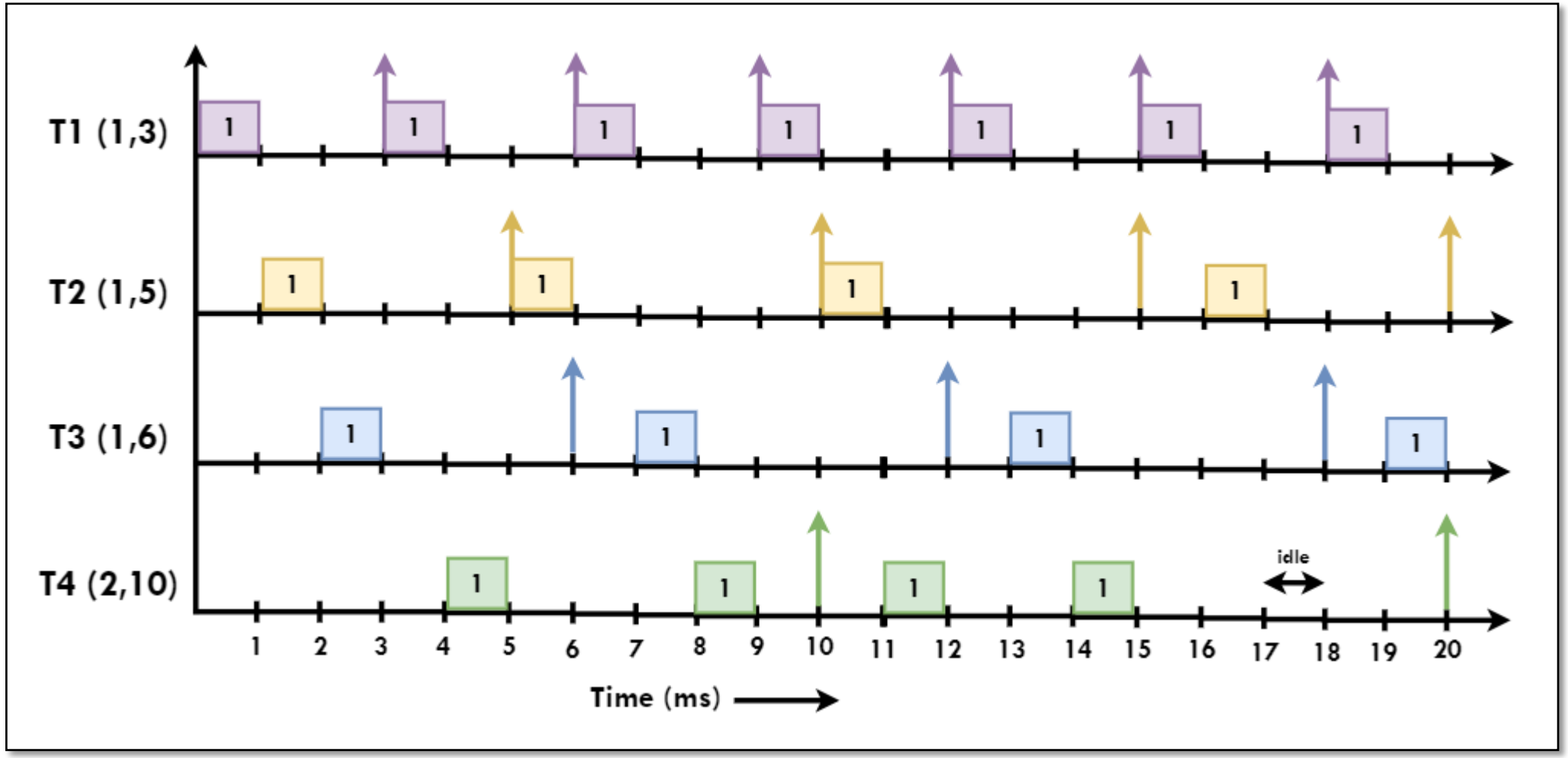
$$U = \sum_{i=1}^{N} \left(\frac{C_i}{PT_i}\right) \leq N(2^{\frac{1}{N}} - 1)$$

B(4)=0.756

The given task set fails in the Sufficient test due to U=0.899 > B(4) we can't conclude precisely whether given task set is schedulable or not under RMS

REAL TIME SCHEDULING

RATE MONOTONIC SCHEDULING(RMS) – EXAMPLE-2



REAL TIME SCHEDULING

RATE MONOTONIC SCHEDULING(RMS) – EXAMPLE-3

- Consider set of three task (Ci, Pi) running on embedded system as follows, T1= (10, 30) , T2= (15, 40) , T3= (5, 50)

Necessary Test

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{P_i}\right) \leq 1$$

U=(10/30)+(15/40)+(5/50)=0.808 ≤ 1

Necessary Test is passed hence given task set must be tested under sufficient test to conclude the Schedulability. But, if necessary test is failed we may conclude given task set is definitely not schedulable by any scheduling algorithm.

Sufficient Test

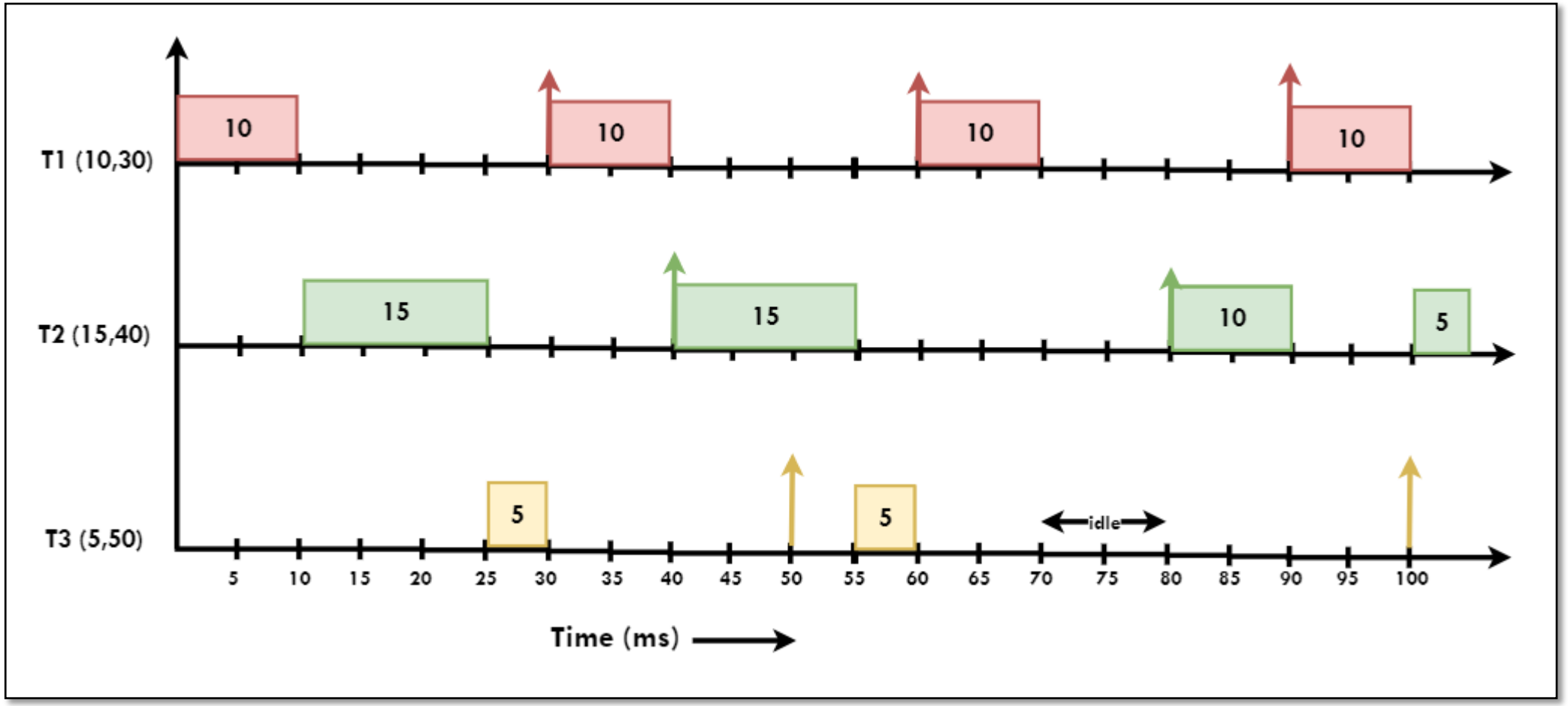
$$U = \sum_{i=1}^{N} \left(\frac{C_i}{PT_i}\right) \leq N(2^{\frac{1}{N}} - 1)$$

B(3)=0.779

The given task set fails in the Sufficient test due to U=0.808 > B(3) we can't conclude precisely whether given task set is schedulable or not under RMS

REAL TIME SCHEDULING

RATE MONOTONIC SCHEDULING(RMS) – EXAMPLE-3



REAL TIME SCHEDULING

RATE MONOTONIC SCHEDULING(RMS) - PROs & CONs

- PROs:
  - Simple to understand
  - Easy to implement (static/fixed priority assignment)
  - Stable: though some of the lower priority tasks fail to meet deadlines, others may meet deadlines
- CONs:
  - Lower CPU utilization
  - Requires D=T
  - Only deal with independent tasks
  - Non-precise Schedulability analysis

REAL TIME SCHEDULING

EARLIEST DEADLINE FIRST (EDF)

- Concept: Task closest to the end of its period assigned the highest priority
- Working :
  - Earliest deadline first is a dynamic priority based scheduling.
  - The scheduling scheme is pre-emptive; it ensures that a task is pre-empted if another task having a nearest deadline is expected to run.
  - EDF is optimal scheduling i.e it can schedule the task set if any other scheduling algorithm can schedule
  - If two task have the same deadlines, need to chose one if the two at random but in most of the case it proceed with the same task which is currently executing on the CPU to avoid context switching overhead.

REAL TIME SCHEDULING

EARLIEST DEADLINE FIRST (EDF) – EXAMPLE-1

- Consider set of task running on a weather monitoring station as follows
  - Temperature measurement Task (T1): C=1ms, P=4ms
  - Humidity measurement Task (T2): C=2ms, P=5ms
  - Co2 measurement Task (T3): C=2ms, P=7ms

Necessary Test

$$U = \sum_{i=1}^{N} \left(\frac{C_i}{P_i}\right) \leq 1$$

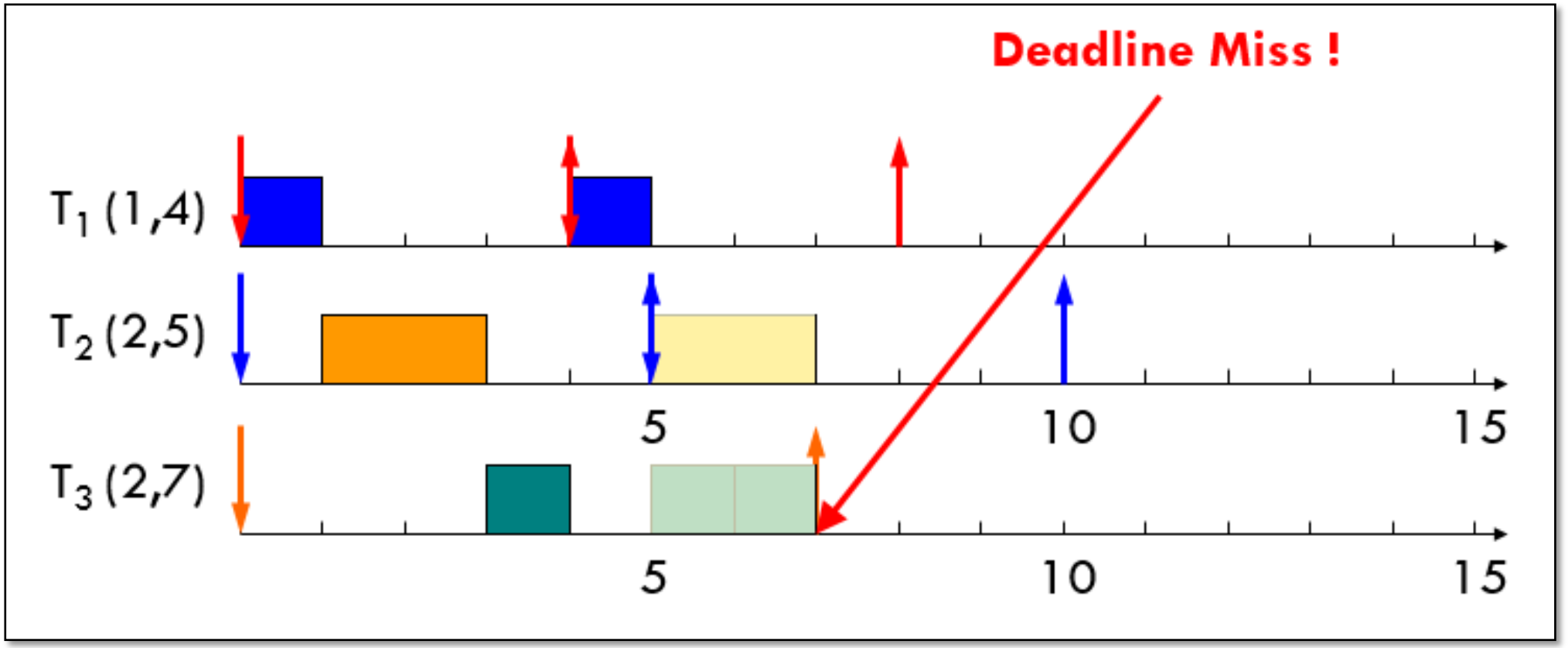
U=(1/4)+(2/5)+(2/7)
= 0.25 + 0.4+ 0.2857
= 0.935 ≤ 1

Necessary Test is passed hence given task set must be schedulable by EDF

REAL TIME SCHEDULING

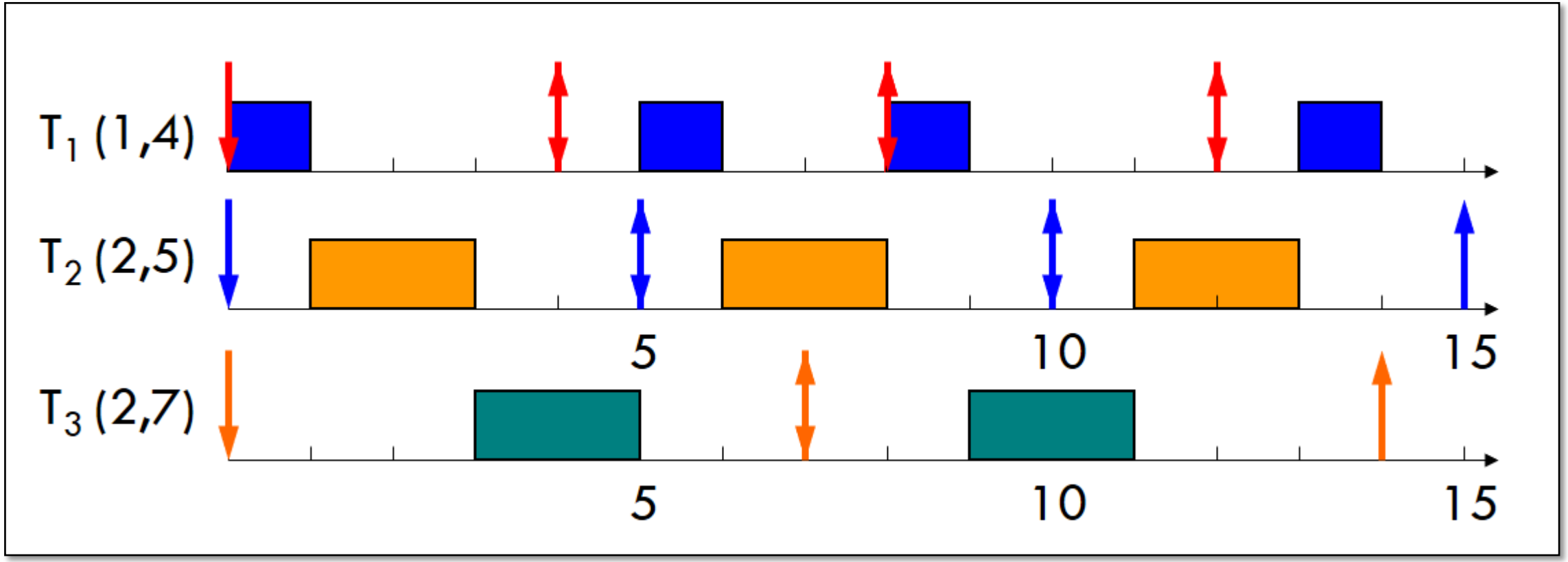
EARLIEST DEADLINE FIRST (EDF) – EXAMPLE-1

Let's first try with RMS



REAL TIME SCHEDULING

EARLIEST DEADLINE FIRST (EDF) – EXAMPLE-1



Although given task set failed to schedule under RMS, it can scheduled by EDF

REAL TIME SCHEDULING

EARLIEST DEADLINE FIRST (EDF) – EXAMPLE-2

- Consider set of four task (Ci, Pi) running on embedded system as follows, T1= (1, 3) , T2= (1, 5) , T3= (1, 6) , T4= (2, 20)

Necessary Test

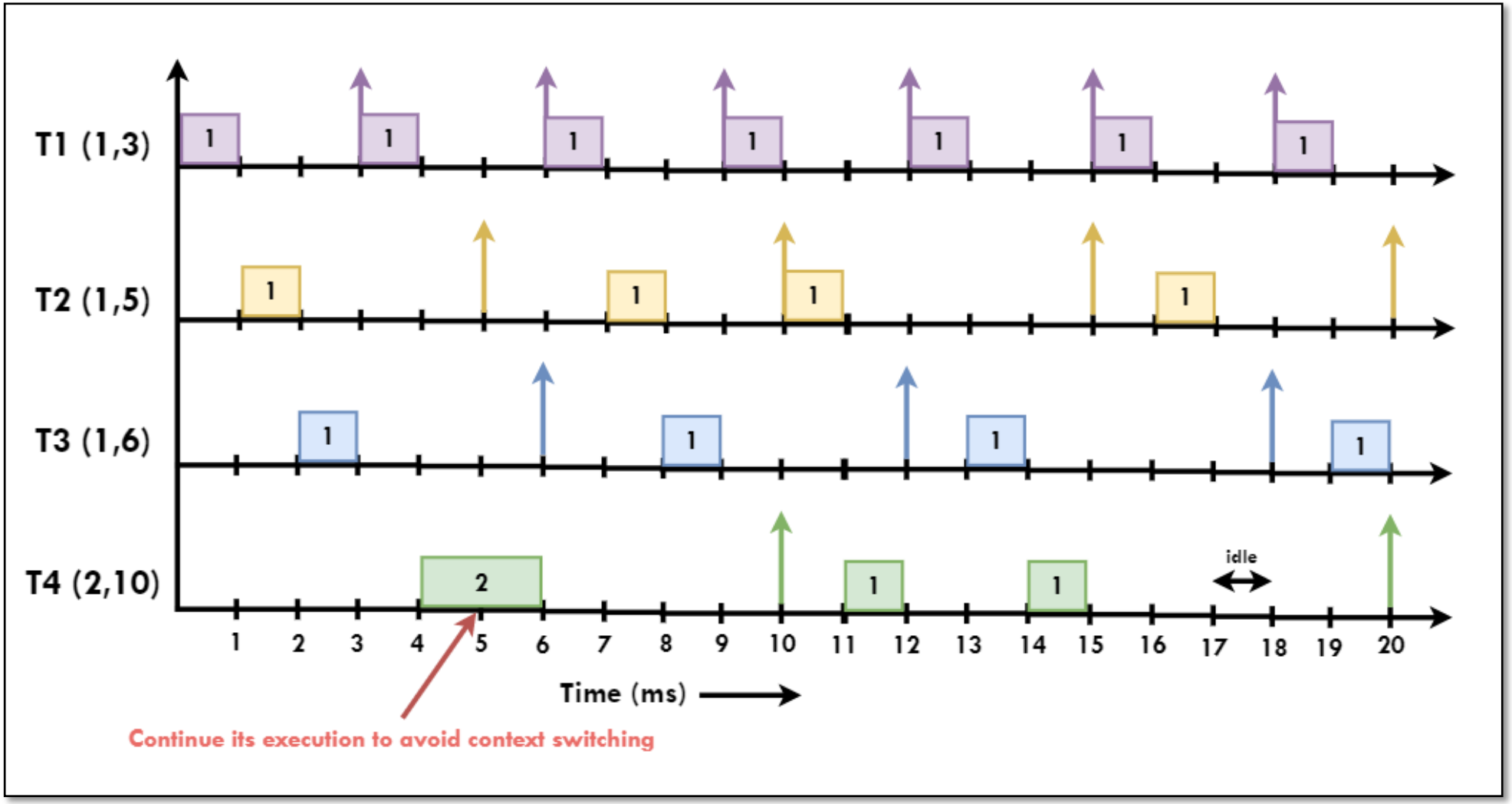
$$U = \sum_{i=1}^{N} \left(\frac{C_i}{P_i}\right) \leq 1$$

$U=(1/3)+(1/5)+(1/6)+(2/10)=0.899 \leq 1$

Necessary Test is passed hence given task set must be schedulable by EDF

REAL TIME SCHEDULING

EARLIEST DEADLINE FIRST (EDF) – EXAMPLE-2



REAL TIME SCHEDULING

EARLIEST DEADLINE FIRST (EDF) – EXAMPLE-3

- Consider set of three task (Ci, Pi) running on embedded system as follows, T1= (5, 20) , T2= (10, 40) , T3= (40, 80)

Necessary Test

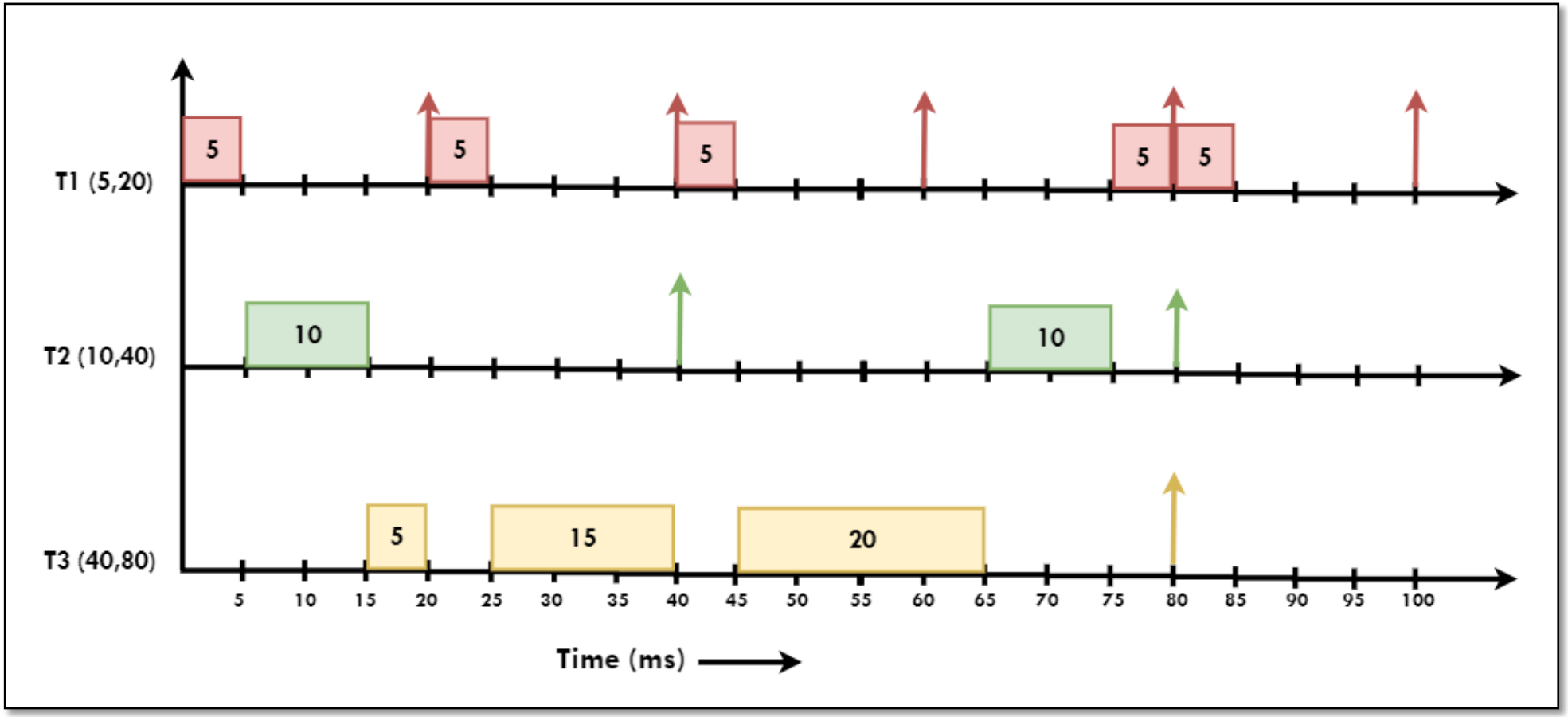
$$U = \sum_{i=1}^{N} \left(\frac{C_i}{P_i}\right) \leq 1$$

U= (5/20)+(10/40)+(40/80)
= 0.25 + 0.25 + 0.5 = 1 ≤ 1

If cumulative CPU utilization is 1 then the CPU will be utilized 100% without idle condition

REAL TIME SCHEDULING

EARLIEST DEADLINE FIRST (EDF) – EXAMPLE-3



At time instance 80ms all 3 tasks are having same deadline. When multiple tasks having deadline at same instance of a time then one of the task will be chosen randomly and get executed.

REAL TIME SCHEDULING

EARLIEST DEADLINE FIRST (EDF) - PROs & CONs

- PROs:
  - It works for all types of tasks: periodic or non periodic
  - Simple Schedulability test
  - Best CPU utilization
  - Optimal scheduling algorithm
- CONs:
  - Difficult to implement due to the dynamic priority-assignment.
  - Any task could get the highest priority even if the task is not important
  - Non stable because if any task fails to meet its deadline, the system is not predictable

INTER PROCESS COMMUNICATION (SYNCHRONIZATION)

IPC - SYNCHRONIZATION

IPC SYNCHRONIZATION TECHNIQUES

- Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other.
- This involves synchronizing their actions and managing shared data or resources.
- Synchronization involves the orderly sharing of system resources by processes, threads must use locking mechanisms to coordinate access.
- Some of the IPC synchronization techniques are, Barrier, Conditional Variables, Semaphore, Mutex, Signals, Read/write lock etc.,

IPC - SYNCHRONIZATION

IPC TECHNIQUES

Method	Provided by (Operating systems or other environments)
File	All operating systems.
Signal	Most operating systems; some systems, such as Windows, only implement signals in the C run-time library and do not actually provide support for their use as an IPC technique.
Socket	Most operating systems.
Pipe	All POSIX systems.
Named pipe	All POSIX systems.
Semaphore	All POSIX systems.
Shared memory	All POSIX systems.
Message passing (shared nothing)	Used in MPI paradigm, Java RMI, CORBA and others.
memory-mapped file	All POSIX systems; may carry race condition risk if a temporary file is used. Windows also supports this technique but the APIs used are platform specific.
Message queue	Most operating systems.
Mailbox	Some operating systems.

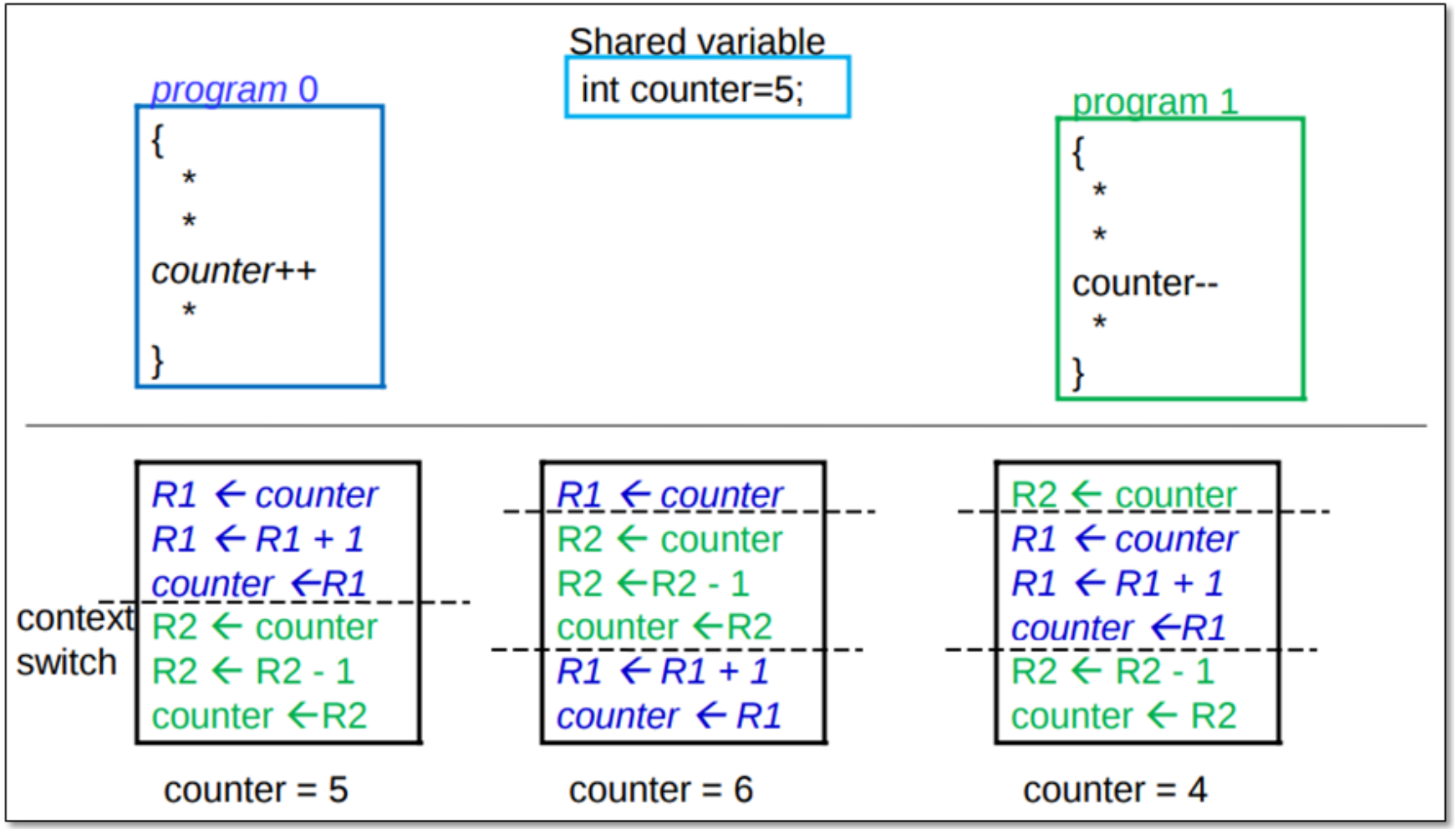
IPC - SYNCHRONIZATION

IPC SYNCHRONIZATION TECHNIQUES

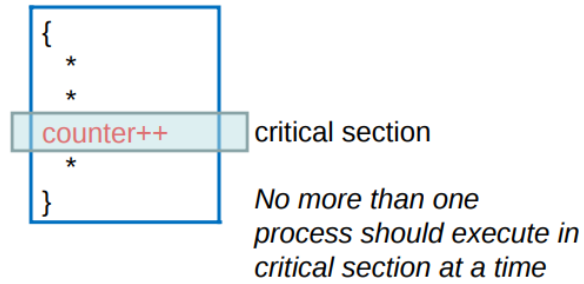
- Few definitions related to Inter Process Communication (IPC):
  - Critical Resource: A resource shared with constraints on its use (e.g., memory, files, printers, etc.)
  - Critical Section: Code that accesses a critical resource
  - Race condition: A situation where several processes access and manipulate the same data (critical section) and the outcome depends on the order in which the access take place
  - Mutual Exclusion: At most one process may be executing a Critical Section with respect to a particular critical resource simultaneously

IPC - SYNCHRONIZATION

IPC SYNCHRONIZATION TECHNIQUES



Prevent race conditions by synchronization which ensure only one process at a time manipulates the critical data



Race Condition - Example

IPC - SYNCHRONIZATION

CRITICAL SECTION - EXAMPLE

- Race condition occurs mainly due to context switching in critical section. Consider a machine with a single printer running a time-sharing operation system.
- If a process needs to print its results, it must request that the operating system give it access to the printer's device driver.
- The operating system must decide whether to grant this request, depending upon whether the printer is already being used by another process.
- If it is not used, the operating system should grant the request and allow the process to continue.

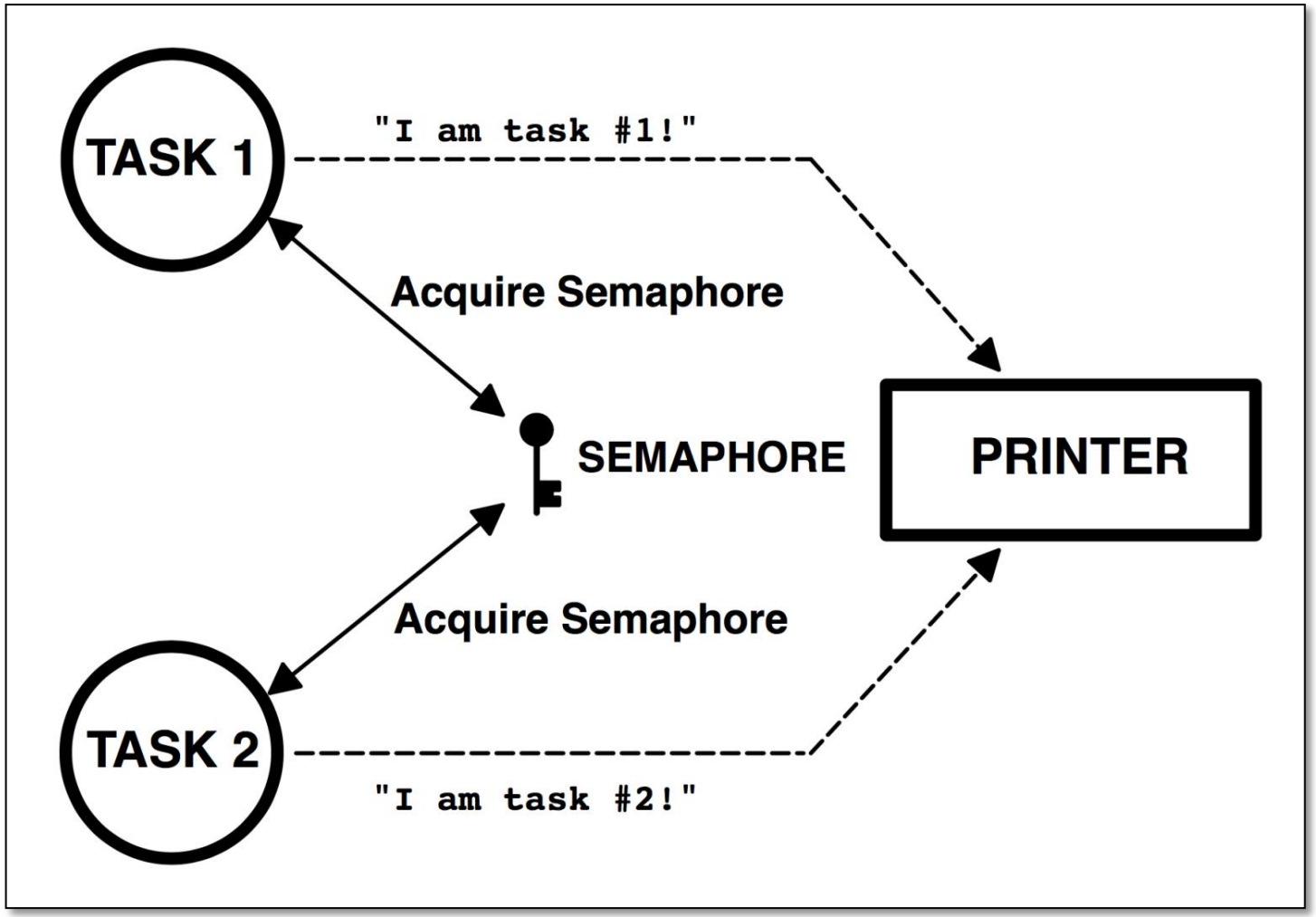
IPC - SYNCHRONIZATION

CRITICAL SECTION - EXAMPLE

- Otherwise, the operating system should deny the request and perhaps classify the process as a waiting process until the printer becomes available.
- Indeed, if two processes were given simultaneous access to the machine's printer, the results would be worthless to both.
- Here the printer is the critical resource, and the critical sections of process A and process B are the sections of the code which issue the print command.
- In order to ensure that both processes do not attempt to use the printer at the same, they must be granted mutually exclusive access to the printer driver by using SEMAPHORE or MUTEX.

IPC - SYNCHRONIZATION

CRITICAL SECTION - EXAMPLE



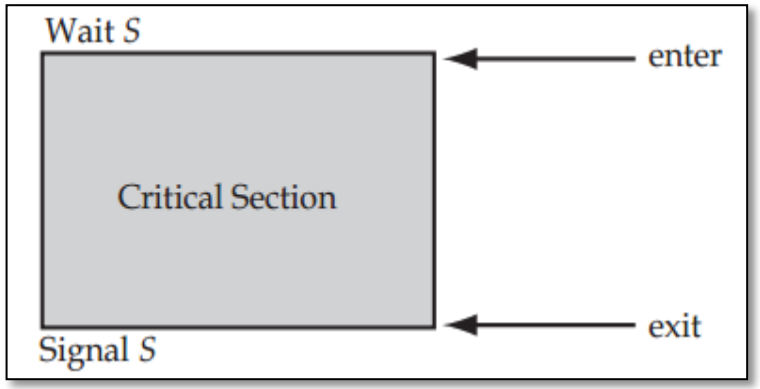
IPC - SYNCHRONIZATION

CRITICAL SECTION - PROBLEM

- The key to preventing trouble involving shared storage is find a way to restrict more than one process access the shared data or resource simultaneously.
- A part of the program where the shared memory is accessed is called the Critical Section. To avoid race conditions and flawed results, one must identify codes in Critical Sections in each thread.
- The important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section.
- Thus, execution of critical sections by the processes is mutually exclusive in time.

IPC - SYNCHRONIZATION

CRITICAL SECTION - PROBLEM



shared variable

```
int counter=5;
lock_t L;
```

program 0

```
{
  *
  *
lock(L)
counter++
unlock(L)
  *
}
```

program 1

```
{
  *
  *
lock(L)
counter--
unlock(L)
  *
}
```

- lock(L) : acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

IPC - SYNCHRONIZATION

CRITICAL SECTION - PROBLEM

- Four conditions to have a good solution for the critical section problem (mutual exclusion).

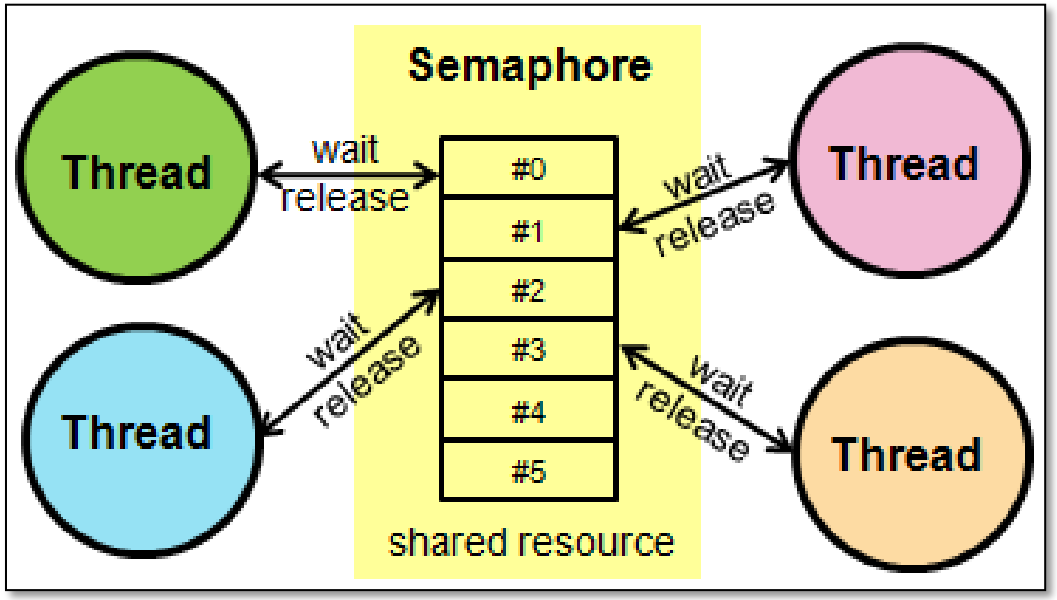
1. No two processes allowed inside their critical sections at the same moment.
2. No assumptions are made about relative execution speed of processes or number of CPUs.
3. No process outside its critical section should block other processes.
4. No process should wait arbitrary long to enter its critical section.

IPC - SYNCHRONIZATION

SEMAPHORE

- SEMAPHORE is signalling mechanism ("I am done, you can carry on" kind of signal) and no task owns semaphores.

- The semaphore advertises that a resource is available, and it provides the mechanism to wait until it is signalled as being available.



IPC – SYNCHRONIZATION

SEMAPHORE

- Semaphore is a simply a variable which is used to achieve process synchronization in the multi processing environment.
- The two most common kinds of semaphores:
  - Binary semaphore - can take the value 0 & 1 only
  - Counting semaphore - can take non-negative integer values
- Two standard operations, *wait and signal* are defined on the semaphore. Entry to the critical section is controlled by the wait operation and exit from a critical region is taken care by signal operation.
- Mutual exclusion on the semaphore is enforced within P(S) and V(S). The wait, signal operations are also called P and V operations.

IPC - SYNCHRONIZATION

SEMAPHORE

- A critical section is surrounded by both operations to implement process synchronization, both operations are atomic.

- If a number of processes attempt P(S) simultaneously, only one process will be allowed to proceed & the other processes will be waiting.

```
Process P

// Some code
P(s);
 // critical section
V(s);
 // remainder section
```

```
void P(int S)
{
    while (S == TRUE);
     S=TRUE;
}

void V(int S)
{
     S=FALSE;
}
```

Semaphore – Pseudo Code

IPC - SYNCHRONIZATION

SEMAPHORE

- For example, consider the C code for Task_A and Task_B. The problem can be solved by bracketing the output statements with semaphore operations as follows:

```
void Task_A(void)
{
    P(S);
    printf("I am Task_A");
    V(S);
}
```

```
void Task_B(void)
{
    P(S);
    printf("I am Task_B");
    V(S);
{
```

Semaphore – Example

- Assume that S is within the scope of both Task_A and Task_B and that It is initialized to FALSE by the system.

IPC - SYNCHRONIZATION

COUNTING SEMAPHORE

- Counting semaphore which can take values greater than one which can be used to protect pools of resources, or to keep track of free resources.
- The semaphore must be initialized to the total number of free resources before real-time processing can commence.
- The new wait and signal semaphore primitives, MP and MV, are designed to present access to a semaphore-protected region
- Whenever process wants that resource it calls P or wait function and when it is done it calls V or signal function.

IPC - SYNCHRONIZATION

COUNTING SEMAPHORE

- Now suppose there is a resource whose number of instance is 4. Now we initialize S = 4 and rest is same as for binary semaphore.
- If value of S becomes zero than a process has to wait until S becomes positive.
- For example, Suppose there are 4 process P1, P2, P3, P4 and they all call wait operation on S(initialized with 4).
- If another process P5 wants the resource then it should wait until one of the four process calls signal function and value of semaphore becomes positive.

counting wait becomes

```
void MP(int S)
{
    S=S-1;
    while (S < 0);
}
```

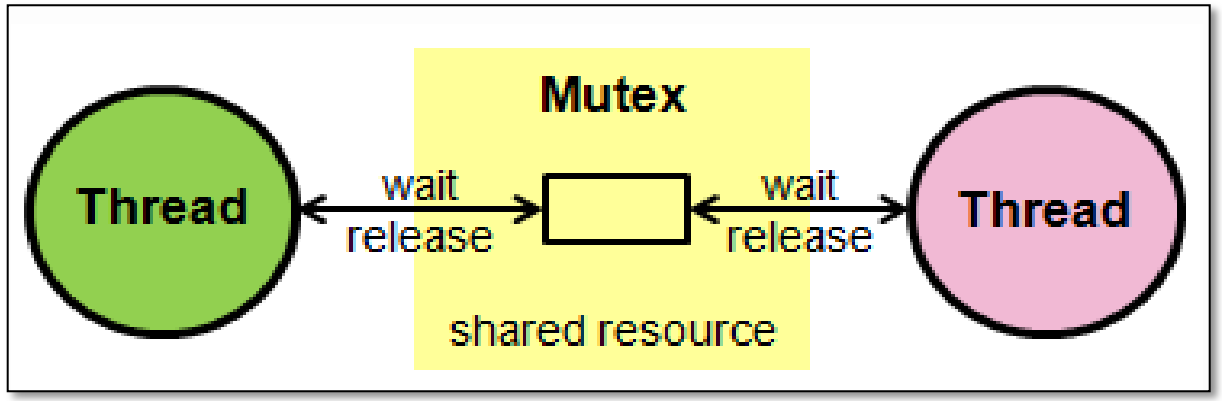
and signal becomes

```
void MV(int S)
{
    S=S+1
}
```

IPC - SYNCHRONIZATION

MUTEX

- MUTEX (Mutual Exclusion) is a locking mechanism used to synchronize access to a resource. Only one task can acquire the Mutex.
- It means there will be ownership associated with Mutex, and only the owner can release the lock (Mutex).
- Mutexes are binary semaphores that include a priority inheritance mechanism.



IPC - SYNCHRONIZATION

MUTEX

- Mutex variables are one of the primary means of implementing thread synchronization and for protecting shared data when multiple writes occur.
- A Mutex variable acts like a "lock" protecting access to a shared data resource. Thus, even if several threads try to lock a Mutex only one thread will be successful.
- No other thread can own that Mutex until the owning thread unlocks that Mutex. Threads must "take turns" accessing protected data.
- Mutexes can be used to prevent "race" conditions.

IPC - SYNCHRONIZATION

MUTEX

- In this case the Mutex is the variable s and x is a resource shared among the threads. All the concurrent threads can see s.

- When one thread "Take" on s, it changes the state of s so that all the other threads know that another thread is executing that code.

```
do_something(x)
   Take_mutex(s) // "Take" mutex using variable `s`
   f(x) // CONCURRENT ACCESS ON THIS CODE IS PREVENTED
    Give_mutex(s) // "Give" mutex
do_something_else(x)
```

IPC - SYNCHRONIZATION

MUTEX

- Later it "Give" s to indicate to the other threads that the code can be safely run. So, in order for a thread to call f(x) in this code block, it needs to wait for s to indicate that it is free and it needs to acquire the lock.

- When used for mutual exclusion the Mutex acts like a token that is used to guard a resource. When a task wishes to access the resource it must first obtain ('take') the token.

- When it has finished with the resource it must 'give' the token back - allowing other tasks the opportunity to access the same resource.

- Unlike binary semaphores however - Mutexes employ priority inheritance.

IPC - SYNCHRONIZATION

SEMAPHORE vs MUTEX

KEY POINTS	SEMAPHORE	MUTEX
Basic	Semaphore is a **signaling** mechanism.	Mutex is a **locking** mechanism.
Existence	Semaphore is an **integer** variable.	Mutex is an **object.**
Function	Semaphore **allow multiple program threads** to access a finite instance of resources.	Mutex allow multiple program thread to access a single resource but **not simultaneously.**
Ownership	Semaphore value **can be changed by any process** acquiring or releasing the resource.	Mutex object lock is **released only by the process that has acquired** the lock on it.
Categorize	**counting** semaphore and **binary** semaphore.	Mutex is **not categorized further.**
Operation	Semaphore value is modified using **wait() and signal()** operation.	Mutex object is locked or unlocked by the **process requesting or releasing the resource.**
Resources Occupied	If all resources are being used, the process requesting for resource performs wait() operation and block itself till semaphore **count become greater than one.**	If a Mutex object is already locked, the process requesting for resources waits and **queued by the system till lock is released.**
Priority Inheritance	**Not Available**	**Available**

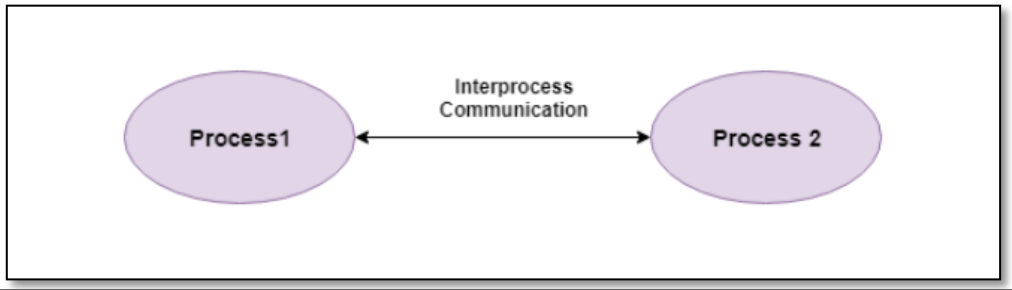
INTER PROCESS COMMUNICATION (DATA EXCHANGE)

IPC – DATA EXCHANGE

- Inter Process Communication (IPC) is a mechanism that involves communication of one process with another process. This usually occurs only in one system.

- This communication could involve a process letting another process know that some event has occurred or transferring of data from one process to another.

- Communication can be of two types
  - Between related processes initiating from only one process, such as parent and child processes.
  - Between unrelated processes, or two or more different processes.



IPC – DATA EXCHANGE

- Following are some of the important IPC data exchange techniques.
  - **Pipes:**
    - Communication between two related processes.
    - The mechanism is half duplex meaning the first process communicates with the second process.
    - To achieve a full duplex i.e., for the second process to communicate with the first process another pipe is required.
  - **FIFO**
    - Communication between two unrelated processes.
    - FIFO is a full duplex, meaning the first process can communicate with the second process and vice versa at the same time.

IPC – DATA EXCHANGE

- Shared Memory
  - Communication between two or more processes is achieved through a shared piece of memory among all processes.
  - The shared memory needs to be protected from each other by synchronizing access to all the processes.
- Message Queues
  - Communication between two or more processes with full duplex capacity.
  - The processes will communicate with each other by posting a message and retrieving it out of the queue.
  - Once retrieved, the message is no longer available in the queue.

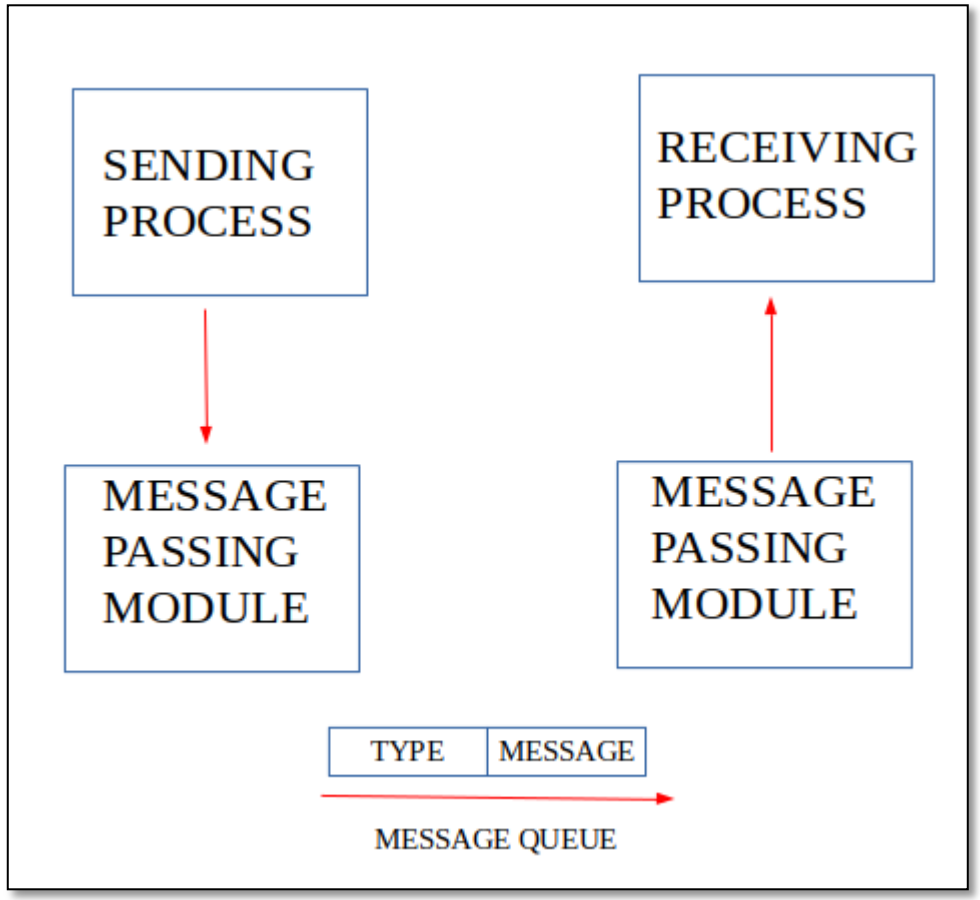
IPC – DATA EXCHANGE

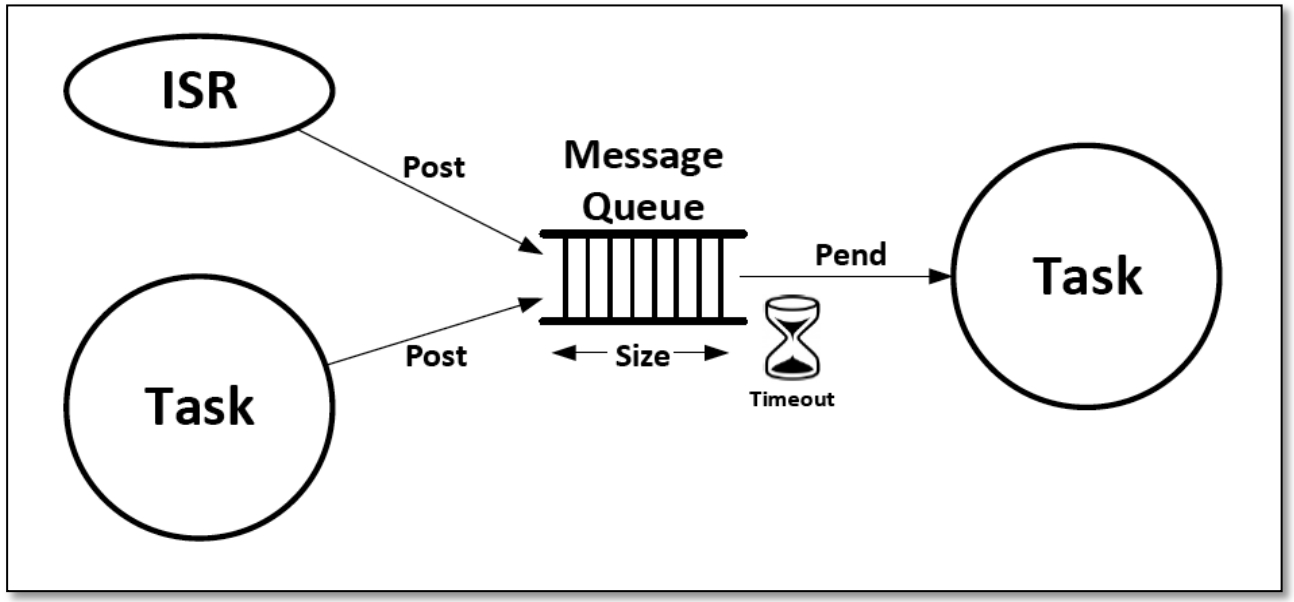
MESSAGE QUEUE

- Message queues provide an asynchronous communications, meaning that the sender and receiver of the message do not need to interact at the same time.
- Messages placed onto the queue are stored until the recipient retrieves them.
- Message queues have limits on the size of data that may be transmitted in a single message and the number of messages that may remain on the queue.
- The sending process places via some message-passing module a message onto a queue which can be read by another process.
- Each message is given an identification or type so that processes can select the appropriate message.

IPC – DATA EXCHANGE

MESSAGE QUEUE





IPC – DATA EXCHANGE

MESSAGE QUEUE

- A message queue is typically implemented as FIFO, meaning that the first message received will be the first message extracted from the queue.
- However, some kernels allow you to send messages at the head of the queue, like LIFO, making that message the first one to be extracted by the task.
- In many implementations of message queues in RTOS, a message being sent to a queue is discarded if the queue is already full.
- If a task pends (i.e., waits) for a message and there are no messages in the queue, then the task will block until a message is posted (i.e., sent) to the queue.

IPC – DATA EXCHANGE

MESSAGE QUEUE

- Message queues can be used in a number of different ways. In fact, you can write fairly complex applications in which you might only use message queues.
- Using only message queues could reduce the size of your code (i.e., footprint) because many of the other services can be simulated (semaphores, time delays, and event flags).
- Most real-time operating systems (RTOSes), such as VxWorks and QNX, encourage the use of message queueing as the primary inter-process or inter-thread communication mechanism.

PERFORMANCE METRICS OF RTOS

PERFORMANCE METRICS OF RTOS

RTOS PERFORMANCE METRICS

- There are three areas of interest when looking at the performance and usage characteristics of an RTOS: (1) Memory (2) Latency (3) Kernel services

- Memory:
  - How much ROM and RAM does the kernel need and how is this affected by options and configuration?
  - Factors that affect the memory footprint includes static or dynamic configuration of Kernel, number of instruction related to processor, global variable declaration etc.,
  - With the help of optimization setting in embedded compilers code size can be reduced, but that will most likely affect performance.
  - Most RTOS kernels are scalable, but some RTOSes, scalability only applies to the kernel. For others, scalability is extended to the rest of the middleware.

PERFORMANCE METRICS OF RTOS

RTOS PERFORMANCE METRICS

- Latency:
  - Interrupt latency: It is the sum of the hardware dependent time, which depends on the interrupt controller as well as the type of the interrupt, and the OS induced overhead.
  - Ideally, it should include the best and worst case scenarios.
  - To measure a time interval, like interrupt latency, with any accuracy, requires a suitable instrument and the best tool to use is an oscilloscope.
  - Scheduling latency: Being real time, the efficiency at which threads or tasks are scheduled is of some importance and the scheduler is at the core of an RTOS.
  - It is hard to get a clear picture of performance, as there is a wide variation in the techniques employed to make measurements and in the interpretation of the results.
  - There are really two separate measurements to consider: (1) The context switch time (2) The time overhead that the RTOS introduces when scheduling a task

PERFORMANCE METRICS OF RTOS

RTOS PERFORMANCE METRICS

- Performance of kernel services:
  - An RTOS is likely to have many API (application program interface) calls, probably numbering into the hundreds.
  - To assess timing, it is not useful to try to analyse the timing of every single call. It makes more sense to focus on the frequently used services.
  - How long does it take to perform specific actions?
  - For most RTOSes, there are four key categories of service call:
    - Threading services
    - Synchronization services
    - Inter-process communication services
    - Memory services

PERFORMANCE METRICS OF RTOS

RTOS CONSIDERATIONS

- What are the real-time capabilities? Is it soft or hard real-time interrupt handling
- What are the preemptive scheduling services?
- What is the target processor, and does it support the RTOS you have in mind?
- How large is the RTOS memory footprint?
- What are the RTOS interrupt latencies?
- How long does it take the RTOS to switch contexts?
- Does your processor development board have a BSP with your RTOS?

PERFORMANCE METRICS OF RTOS

RTOS CONSIDERATIONS

- Which development environments and debugging tools work with the RTOS?
- How much does the RTOS cost, is it open source or royalty-free?
- Does the RTOS have good documentation and/or forums?
- What is the RTOS supplier's reputation?
- How flexible is the choice of scheduling algorithms in the RTOS? E.g., FIFO, round Robin, rate-monotonic, sporadic, etc.
- Are there tools for remote diagnostics?

THANK YOU