Lecture 7

Embedded Systems

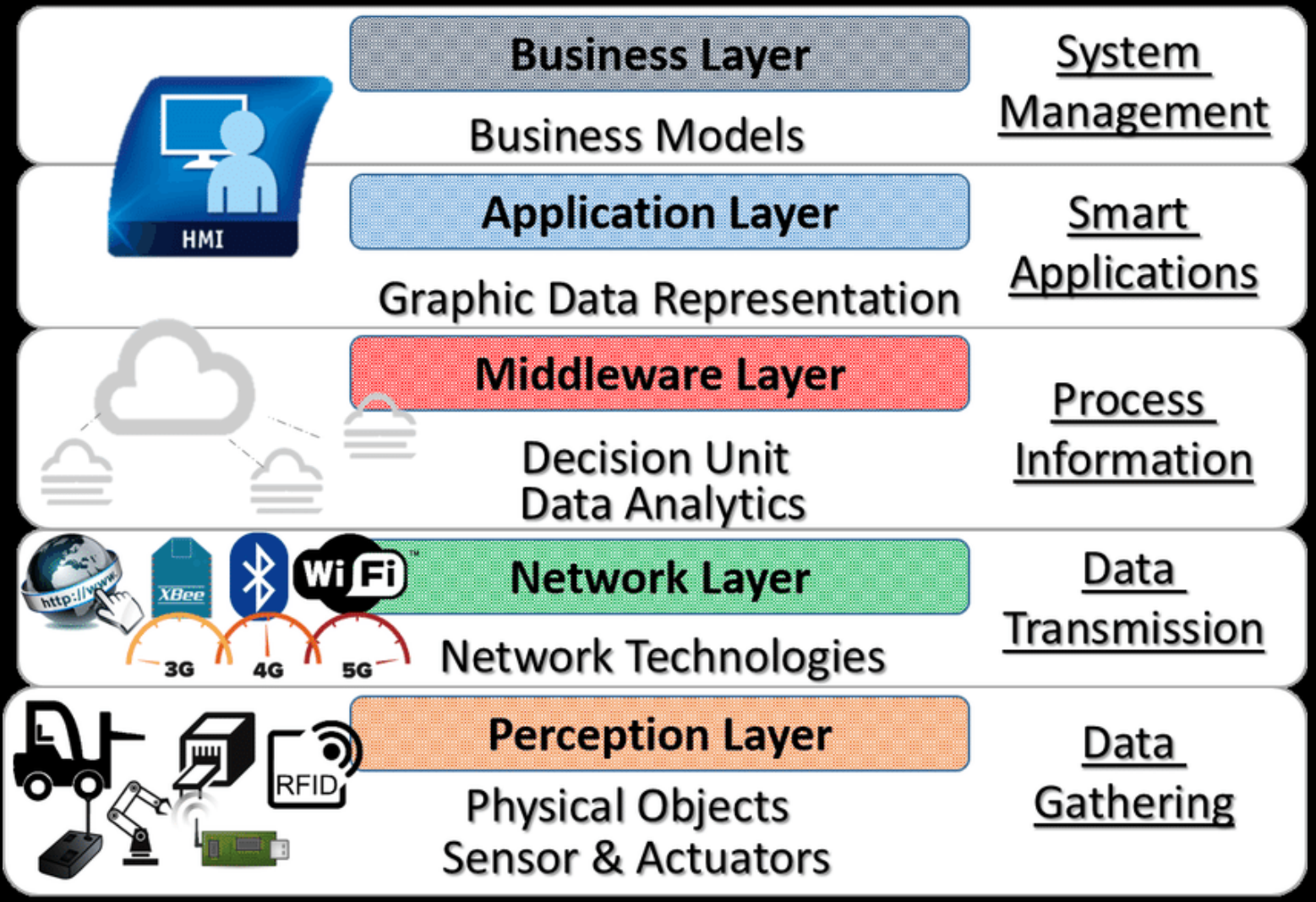
RTOS II

Associated Prof. Wafaa Shalash

Lect. 7

Class Rules

- Be in class on time,
- Listen to instructions and explanations.
- Talk to your classmates only when there is an activity.
- Use appropriate and professional language.
- Keep your mobile silent.


Business Layer
Business Models
System Management
HMI
Application Layer
Graphic Data Representation
Smart Applications
Middleware Layer
Decision Unit
Data Analytics
Process Information
http://www
XBee
Wi Fi
3G
4G
5G
Network Layer
Network Technologies
Data Transmission
RFID
Perception Layer
Physical Objects
Sensor & Actuators
Data Gathering

Lecture Topics

- The Heart of RTOS
- Preemptive vs Non-Preemptive Scheduling

Why Are Task Properties Important?

- **Predictability** – Ensures real-time deadlines are met.
- **Resource Management** – Prevents stack overflows and deadlocks.
- **Efficiency** – Optimizes CPU usage and task switching.

Example of an RTOS Task (FreeRTOS):

```c
void vTask1(void *pvParameters) {
    for (;;) {
        // Perform task 1 functionality
        printf("Task 1 is running\n");
        vTaskDelay(1000 / portTICK_PERIOD_MS);  // Delay for 1 second
    }
}

void vTask2(void *pvParameters) {
    for (;;) {
        // Perform task 2 functionality
        printf("Task 2 is running\n");
        vTaskDelay(500 / portTICK_PERIOD_MS);  // Delay for 0.5 second
    }
}

int main(void) {
    // Create tasks with different priorities
    xTaskCreate(vTask1, "Task1", 1000, NULL, 2, NULL);  // Task 1, higher priority
    xTaskCreate(vTask2, "Task2", 1000, NULL, 1, NULL);  // Task 2, lower priority
    vTaskStartScheduler();  // Start the RTOS scheduler
    while (1);
}
```

- **Task 1** has a priority of 2, while **Task 2** has a priority of 1.
- **Task 1** runs every 1 second, and **Task 2** runs every 0.5 second.
- The tasks are scheduled based on their priorities and delays using vTaskDelay(), which tells the RTOS when to pause a task.

Let me know if you need additional

Scheduler: The Heart of RTOS

- A **scheduler** is a system software component or module within an operating system (OS) responsible for determining the order in which multiple tasks, processes, or threads are executed by the CPU.
- In Real-Time Operating Systems (RTOS), the scheduler plays a critical role in managing tasks to ensure they meet their timing constraints.
- The scheduler implements various scheduling algorithms to decide which task gets CPU time and for how long.

Key Functions of a Scheduler

1. **Task Selection**: Selects which task should run next based on the scheduling algorithm.
2. **Preemption and Context Switching**: Switches between tasks if the chosen algorithm allows preemption.
3. **Time Management**: Keeps track of task deadlines, execution times, and periodic intervals.
4. **Prioritization**: Assigns and manages priorities to ensure high-priority tasks receive appropriate CPU time.
5. **Resource Management**: Allocates and deallocates CPU resources efficiently among tasks.

Common RTOS Scheduling Requirements

- **Predictability**: Ability to predict task execution times and meet deadlines.
- **Resource Utilization**: Efficiently using CPU time and other resources.
- **Scalability**: Handle varying numbers of tasks without significant performance degradation.

Scheduling Algorithms in RTOS

Algorithm	Description	Suitability
Rate Monotonic Scheduling (RMS)	Priority is based on task period (shorter = higher priority)	Good for periodic tasks
Earliest Deadline First (EDF)	Priority is based on the earliest deadline	Optimal for dynamic deadlines
Fixed Priority Scheduling	Priorities are set manually	Simple and predictable
Round Robin	Tasks of equal priority share CPU time	Used in hybrid RTOS

How the Scheduler Works (Core Steps)

1. **Task Creation**
   Tasks are created and assigned priorities. Each task has:
   1. Stack space
   2. Program counter (PC)
   3. Task Control Block (TCB)
2. **Task States**
   Tasks move between different states:
   1. **Ready**: Waiting to be scheduled
   2. **Running**: Currently executing
   3. **Blocked**: Waiting for an event (e.g., I/O, semaphore)
   4. **Suspended**: Paused until resumed
3. **Scheduling Decision**
   1. When an event occurs (timer tick, interrupt, task finishes), the scheduler runs.
   2. It checks the **ready list** and selects the **highest-priority task**.
   3. **Context switching** is performed if a new task is chosen.
4. **Context Switching**
   1. Saves the state (registers, PC, stack) of the current task
   2. Restores the state of the new task
   3. Switches execution to the new task

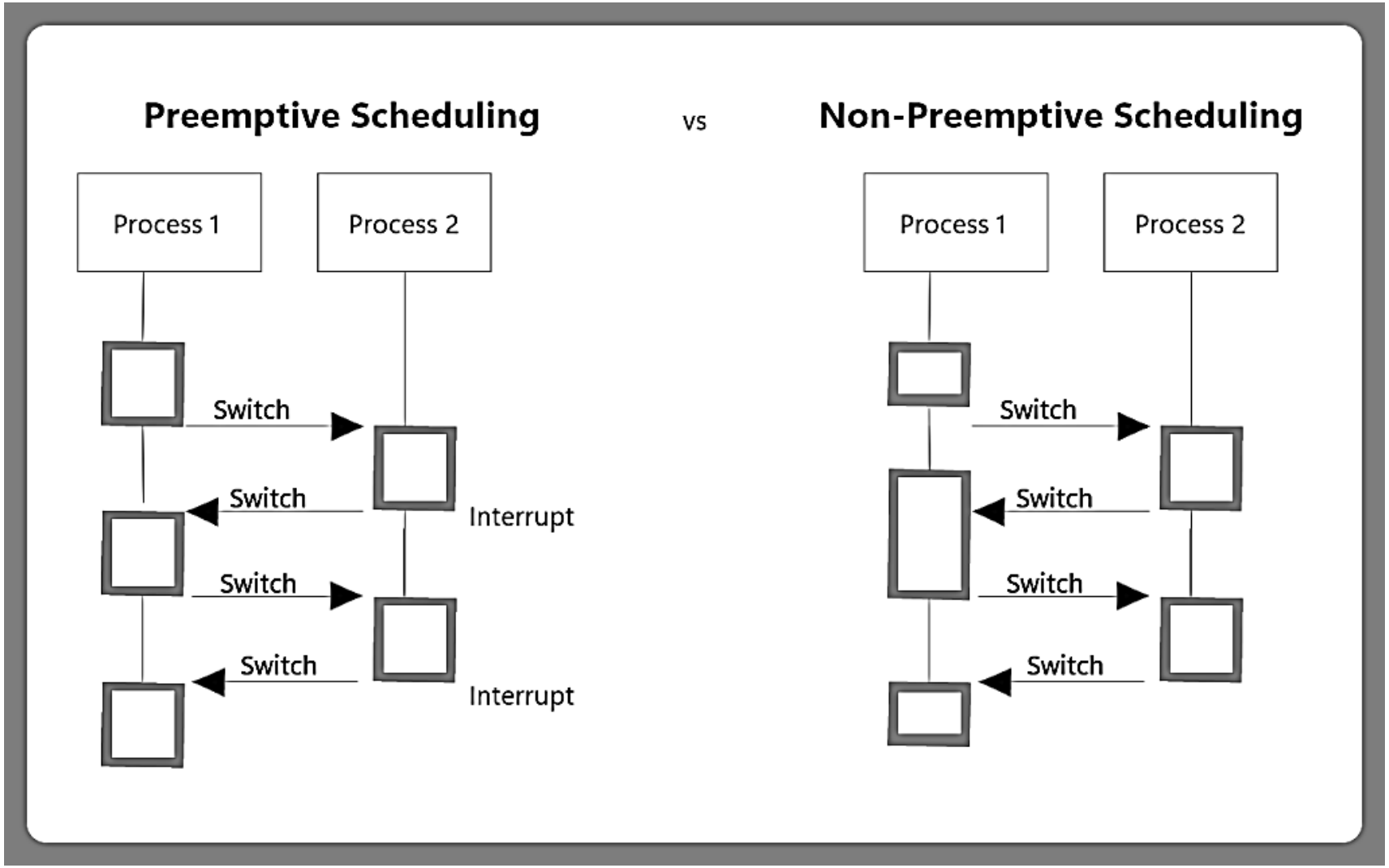
RTOS Scheduler Characteristics

- **Deterministic behavior**: Same input → same output every time
- **Low latency**: Interrupt and task switching are fast
- **Priority-based**: Not all tasks are equal
- **Real-time constraints**: Tasks must meet timing deadlines

Preemptive vs Non-Preemptive Scheduling

- Scheduling in RTOS can be categorized into two types based on the ability of the system to preempt or interrupt a task.
- **In preemptive scheduling,** the operating system can interrupt a task to give way to a higher-priority task. This type of scheduling is more suitable for real-time systems as it allows critical tasks to be executed immediately.
- **In contrast, non-preemptive** scheduling does not allow tasks to be interrupted once they have started executing.
  - This can lead to delays in the execution of high-priority tasks and is generally not suitable for real-time systems.
  - However, it can be used in systems where task execution time is predictable and short.

Preemptive vs Non-Preemptive Scheduling



Preemptive Scheduling

- **Task Switching Triggered by Priority**:
  - The RTOS continuously monitors tasks.
  - If a higher-priority task becomes ready (e.g., due to an interrupt), it **preempts** (interrupts) the currently running lower-priority task.
- **Responsive to Real-Time Events**:
  - Ideal for systems where some tasks must respond immediately (e.g., handling sensor input).
- **Example in the diagram**:
  - Task B starts running, but is interrupted when Task A (higher priority) becomes ready. Task A preempts B.

Non-Preemptive Scheduling (Right Side)

- **Task Switching Only When Task Voluntarily Yields or Completes**:
  - Even if a higher-priority task becomes ready, it must **wait** until the current task finishes.
- **Simpler and Safer for Shared Resources**:
  - No unexpected task switches, easier debugging.
- **Less responsive**:
  - May delay important tasks if a long-running task doesn’t yield.
- **Example in the diagram**:
  - Task B continues running until it finishes, even though Task A becomes ready.

Feature	Preemptive	Non-Preemptive
Task interruption	Yes, by higher priority	No, runs to completion
Responsiveness	High	Low
Resource handling	Needs careful management	Safer and simpler
Complexity	More complex	Easier to implement

FreeRTOS Scheduler

- In your **FreeRTOS example**, the **scheduler type** being used is the **preemptive priority-based scheduler**, which is the **default scheduler** in FreeRTOS (unless configured otherwise in FreeRTOSConfig.h).

Scheduler Type: Preemptive Priority-Based

Features:

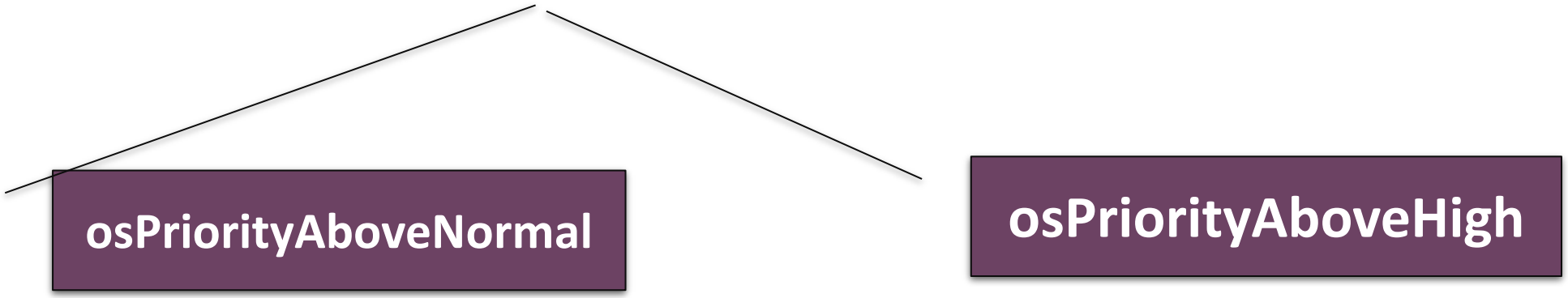
•**Preemptive:** The highest priority task that is ready to run will always execute.

•**Priority-based:** Tasks are scheduled based on their assigned priority (osPriorityHigh, osPriorityNormal, etc.).

- •Context switching happens automatically when:
- •A higher priority task becomes ready.
- •A task blocks (e.g., waits for delay or I/O).

Example

```
const osThreadAttr_t tempTaskAttr = {
 .name = "TempTask",
.priority = osPriorityHigh};
const osThreadAttr_t uartTaskAttr = {
 .name = "UARTTask",
.priority = osPriorityNormal};
```



1. Preemptive Scheduling Example (Default in FreeRTOS)

🔧 Configuration:

#define configUSE_PREEMPTION 1 // Enables preemptive scheduling

```c
void vTaskA(void *pvParameters) {
    for (;;) {
        printf("Task A (high priority)\n");
        vTaskDelay(500 / portTICK_PERIOD_MS);
    }
}

void vTaskB(void *pvParameters) {
    for (;;) {
        printf("Task B (low priority)\n");
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

int main(void) {
    xTaskCreate(vTaskA, "TaskA", 1000, NULL, 2, NULL); // Higher priority
    xTaskCreate(vTaskB, "TaskB", 1000, NULL, 1, NULL); // Lower priority
    vTaskStartScheduler();
    while(1);
}
```

Behavior:

- Task B starts running.
- When Task A becomes ready, it **preempts** Task B.
- Task A runs first every time it's ready.

2. Non-Preemptive Scheduling Example

- 🔧 **Configuration:**

#define configUSE_PREEMPTION 0 // Disables preemptive scheduling

```c
void vTaskA(void *pvParameters) {
    for (;;) {
        printf("Task A (high priority)\n");
        vTaskDelay(500 / portTICK_PERIOD_MS);  // Voluntary yield
    }
}

void vTaskB(void *pvParameters) {
    for (;;) {
        printf("Task B (low priority)\n");
        vTaskDelay(1000 / portTICK_PERIOD_MS); // Voluntary yield
    }
}

int main(void) {
    xTaskCreate(vTaskA, "TaskA", 1000, NULL, 2, NULL); // Higher priority
    xTaskCreate(vTaskB, "TaskB", 1000, NULL, 1, NULL); // Lower priority
    vTaskStartScheduler();
    while(1);
}
```

Behavior:
Tasks only switch when they call vTaskDelay or yield.
Even if Task A is higher priority, it must wait for Task B to yield.
Task switching is cooperative.

Summary:

Behavior	Preemptive	Non-Preemptive
Task Switch Trigger	RTOS preempts based on priority	Tasks voluntarily yield
Responsiveness	High (good for real-time)	Low (task must give up CPU)
Use Case	Sensor input, real-time control	Simpler systems, UI apps, teaching

Comparison

Feature	**Preemptive**	**Non-Preemptive (Cooperative)**
Task switch trigger	Automatically by the RTOS based on priority	Manually by the task (e.g., delay/yield)
Control over CPU	RTOS has full control	Tasks control when to yield

Feature	**Preemptive**	**Non-Preemptive (Cooperative)**
Response to high-priority tasks	Immediate	Delayed until current task yields
Real-time performance	Better, more deterministic	May suffer under long-running tasks

Feature	**Preemptive**	**Non-Preemptive (Cooperative)**
Risk of task starvation	Higher (low-priority tasks can be preempted often)	Lower (all tasks must yield for others to run)
Fairness	Less fair unless priority inversion is managed	More fair if tasks yield appropriately

Comparison

Feature	**Preemptive**	**Non-Preemptive (Cooperative)**
Scheduler complexity	Higher	Lower
Debugging difficulty	More complex due to context switching anytime	Easier to trace, more predictable flow
Scenario	**Preemptive**	**Non-Preemptive (Cooperative)**
Ideal for	Real-time systems with strict deadlines	Simple systems, limited resources
Example	Industrial automation, robotics	Small embedded systems, simple appliances

Parameter	Preemptive Scheduling	Non-Preemptive Scheduling
Basic	In this resources (CPU Cycle) are allocated to a process for a limited time.	Once resources (CPU Cycle) are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process cannot be interrupted until it terminates itself or its time is up.
Starvation	If a process having high priority frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then later coming process with less CPU burst time may starve.
Overhead	It has overheads of scheduling the processes.	It does not have overheads.
Flexibility	flexible	rigid
Cost	cost associated	no cost associated
CPU Utilization	In preemptive scheduling, CPU utilization is high.	It is low in non preemptive scheduling.
Examples	Examples of preemptive scheduling are Round Robin and Shortest Remaining Time First.	Examples of non-preemptive scheduling are First Come First Serve and Shortest Job First.

NEXT TIME