RTOS

REAL TIME OPERATING SYSTEM

"TASKS COMMUNICATIONS AND SYNCHRONIZATION"

SESSION 4

AMIT

SOME REAL TIME PROBLEMS

1. SHARED
2. RESOURCES
3. CRITICAL SECTION
4. RACE CONDITION

REENTERANCY

1 SHARED RESOURCES

it software or hardware resources that can be accessed by more than one task at the same time which may make a confliction in the data.

AN EXAMPLE :

- **SOFTWARE SHARED RESOURCES**

  as the global variable that is used in a function and also in ISR handler.

- **HARDWARE SHARED RESOURCES**

  as the shared LCD between more than two functions and ISRs that can access that LCD and print data on it.

How to avoid Data confliction caused by sharing s/w and h/w resources ?

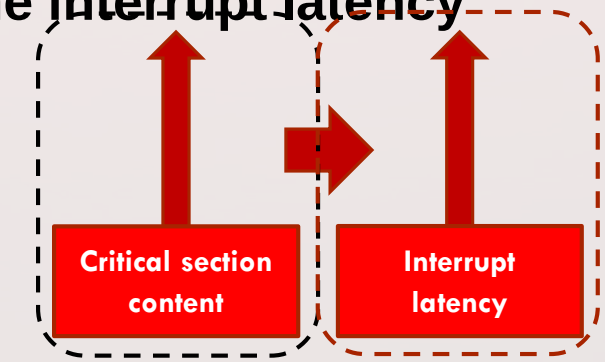
making a **critical section** is one of solutions to this Problem

2 CRITICAL SECTION

- is a function or a group of functions in the code that must be executed without any interruption from any interrupt.

- the higher the number of statements of the critical section → the higher the time of executing the

critical section → the higher the interrupt latency → the lower the system performance and reliability



How to create the critical section?

1- Enabling and disabling the GLOBAL INTERRUPT at fore/background systems

```
SREG = 0X00; //disable global interrupt.
//our critical section
SREG = 0X80; //enable global interrupt.
```

2- End the scheduler before entering the critical section, and after start it again at Realtime systems

```
//Critical Section
//end scheduler
vTaskEndScheduler();
/*
  code of Critical Section
 */
//start the scheduler again
vTaskStartScheduler();
```

3- Enter task to critical state then Exit After Execution.

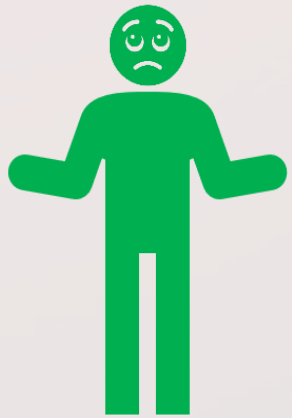
```
void vTask2 (void * para)
{
        while (1)
        {
        taskENTER_CRITICAL ();
         x ++;
         y ++;
        taskEXIT_CRITICAL ();
        }
}
```

3 RACE CONDITION

- *The race condition* occurs when two or more threads able to access a shared data then they try to modify it at the same time.

- How it happen ? the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access this shared data. Therefore, the result of the change in data is unexpected results .

  i.e both threads are "RACING" to access and change this data.



AMIT

x = 2

Task 1: R1=2	Task 2: R1=2

mov x,R1;
add R1,1;
	mov x,R1;
	add R1,1;
	mov R1,x;

4 REENTERANCY

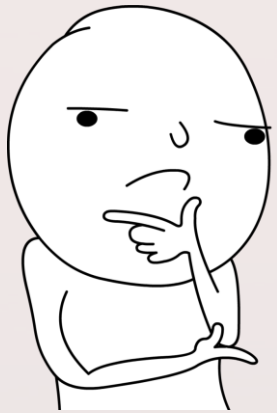
- ***The REENTERANCY* is the availability of interrupting a Task in the middle of its execution and then safely be called again without fearing of data corruption**

- ***When should you fear of non-reentrancy of a function/task?***

if this function able to access A non-constant Global Data (a shared Resource) while this Data is accessed by

- ✓ ***Software Module (ISR , another Task,…)***
- ✓ ***Hardware Module***

A shared Task between other tasks and ISRs in multitasking System must be REENTRANT or NON-REENTRANT ?

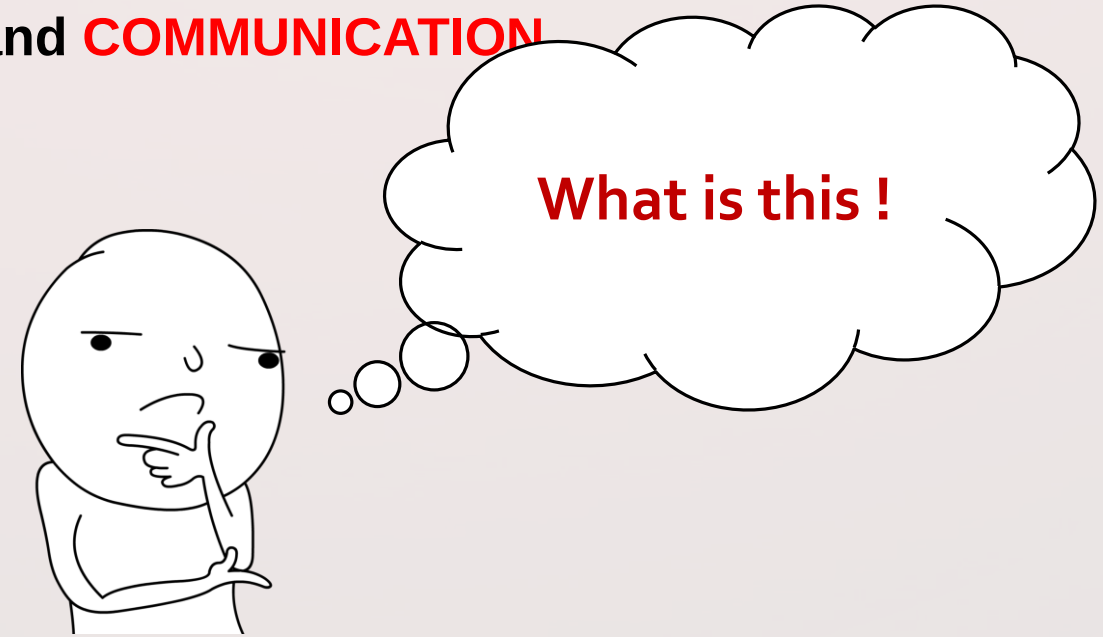


How To Avoid non-reentrancy of a function/task?

- ✓ **Hold only Local Variable inside task**
- ✓ **Disable the Interrupts before Task and Enable it After (critical Section technique)**
- ✓ **Not to call non-reentrant Function**
- ✓ **Use task Synchronization objects (Semaphore , Mutex ,..)**

Solution of previous problem at RT Operating System

- here in RTOS, the problem is much worse as not just ISR that can interrupt the normal execution, but also the Tasks can interrupt each other which means if there are two Tasks that use the same shared resource, each one of them can interrupt the operation that is executing on the other Task causes a corruption of that shared resource.
- ***To solve this problem*** : we use a new concept in RTOS called **INTER-TASK SYNCHRONIZATION** and **COMMUNICATION**



TASK SYNCHRONIZATION

Ordering operations between two or more tasks to Prevent Data Corruption and Safely Sharing Resources.

RTOS kernel uses the following objects to achieve tasks synchronization:

- ✓ **Semaphore**
- ✓ **Mutex**
- ✓ **Event flags**

SEMAPHORE

a kernel object (simply a variable or abstract data type) used for resources Management so ,that one or more tasks can acquire or release for the purposes of synchronization

Also considered as a protocol mechanism used by most Multitasking Kernels

There are two types of semaphores:

- **Counting semaphore**: are used for counting events and resource management.
- **Binary semaphore**: are used for synchronization

SEMAPHORE

For example:

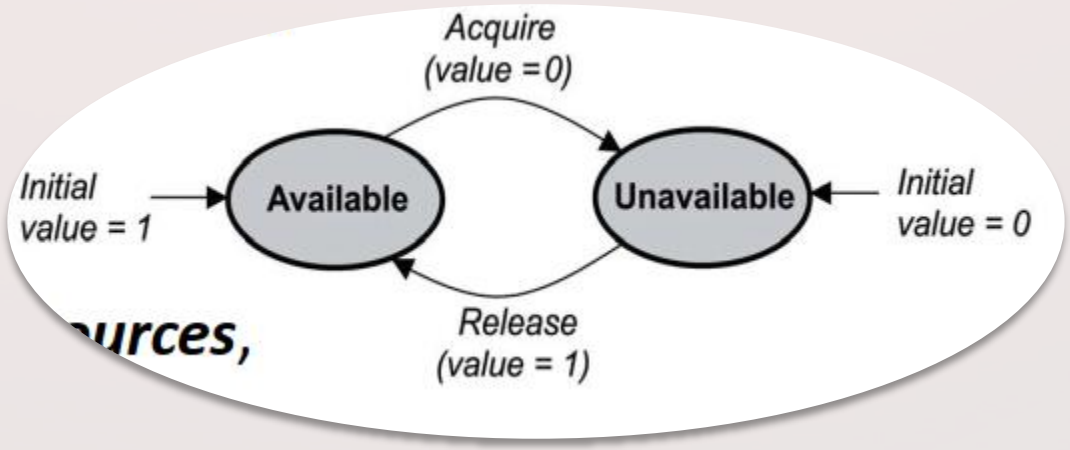
the railway is considered as shared resource between all trains,
so if the railway is busy because of a train, the semaphore will be risen to indicate the shared railway is busy and the next train must stop until the semaphore becomes available again.

Binary semaphore

A binary semaphore can have a value of either 0 or 1

- **0, semaphore is unavailable (*acquired /taken*)**
- **1, the semaphore is available (*released*)**

treated as global resource, they are shared among all tasks that need them. Making the semaphore a global resource allows any task to release it, even if the task did not initially acquire it



COUNTING SEMAPHORE

uses a count to allow it to be acquired or released multiple times.

If the initial count is:

- **0, the counting semaphore is created in the unavailable state.**
- **1 or more, the semaphore is created in the available state, and the number of tokens it has equals its count**

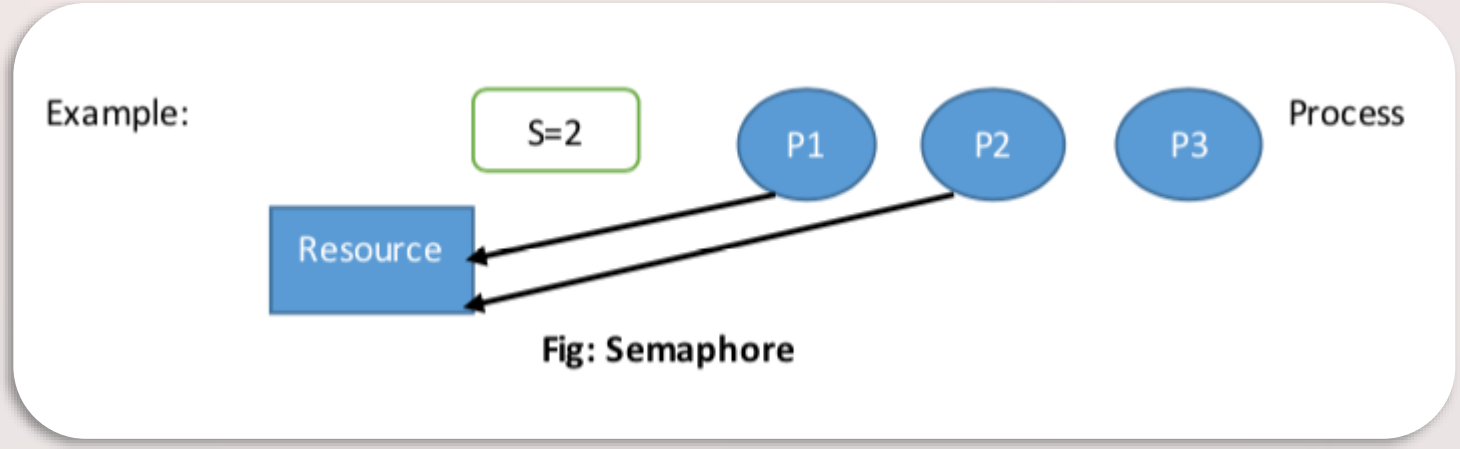


Fig: Semaphore

SEMAPHORES APIS AT FREERTOS :

At first, to use the semaphore APIs, you must first include semaphore library "**semphr.h**":

- ***Declare a Semaphore Variable:***
  semaphore variable type is "**xSemaphoreHandle**". So it declared as following:

```
xSemaphoreHandle LCD_semphr;
```

- ***Create The Semaphore:***

(1) vSemaphoreCreateBinary (LCD_semphr);

It creates a binary semaphore , the initial state of this semaphore is available.

The states of semaphore is available or not available only.

(2) LCD_semphr = xSemaphoreCreateCounting (Max_value , initial_value);

It creates several semaphores according to the value of "Max_value" parameter, the initial, state of them is determined according to the

- ***Taking a Semaphore:***
  To take the semaphore, we will use:

```
state = xSemaphoreTake ( LCD_semphr , Max_blockage_ticks );
```

This API tries to take the semaphore, if it is available the semaphore will be decreased,
if not, it will wait some ticks according to the value of "**Max_blockage_ticks**" parameter.

This API returns "**pdPASS**" if it succeeds to take the semaphore, and returns "**pdFAIL**" if it fails to take the semaphore.

- ***Giving a Semaphore:***

state = xSemaphoreGive (LCD_semphr);

This API tries to give the semaphore, the semaphore will be increased,
if the semaphore reaches to its max value, it will return “**pdFAIL**”, because there is no semaphores were taken to give them.
This API returns “**pdPASS**” if it succeeds to give the semaphore,
and returns “**pdFAIL**” if there is no semaphores were taken to give them.

FREERTOS APIs EXERCISE :

Now ,we will do two tasks, the system will run as a pre-emptive kernel, the first task will write on LCD “TASK1”, the second task will writes on LCD “TASK2 ”, using semaphore.

FREERTOS APIs EXERCISE :

First task:

```
void LCD_1 (void* para)
{
    while (1)   {
        if ( pdPASS == xSemaphoreTake(  LCD_semphr, 10)  ) {
            LCD_Write_String ( “TASK1 ”);
            xSemaphoreGive(  LCD_semphr  );
        }
        vTaskDelay(150); // the task will be in blockage state about 150 ticks
    }
}
```

FREERTOS APIs EXERCISE :

second task:

```
void LCD_2 (void* para)
{
        while (1)   {
             if ( pdPASS == xSemaphoreTake(  LCD_semphr, 10)  ) {
                 LCD_Write_String ( “TASK2 ”);
                 xSemaphoreGive(  LCD_semphr  );
              }
              vTaskDelay(150); // the task will be in blockage state about 150
ticks
        }
}
```

FREERTOS APIs EXERCISE :

- **Main Function:**

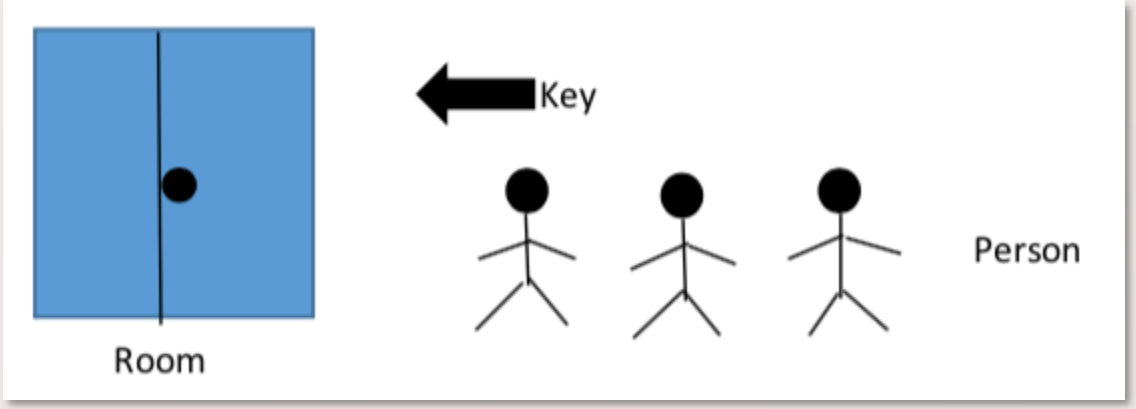
```
xSemaphoreHandle   LCD_semphr;
int main (void)
{
    LCD_init( );
    xTaskCreate(LCD1,  NULL,  250,  NULL,  1,  NULL);
    xTaskCreate(LCD2,  NULL,  250,  NULL,  2,  NULL);
    /*it is a counting semaphore that has one semaphore
maximum, and                its initial state is available*/
    LCD_semphr  =  xSemaphoreCreateCounting ( 1, 1 );
    vTaskStartScheduler();
}
```

MUTEXES

a special type of binary semaphore ,The word MUTEX originates from “mutual exclusion".

mutex can be conceptually thought of as a key that is associated with the resource being shared.

A task is not permitted to access the shared resource unless it holds the key.



- ***Declare a Semaphore Variable:***

  semaphore variable type is "**xSemaphoreHandle**". So it declared as following:

```
xSemaphoreHandle  LCD_semphr;
```

- ***Create The Mutex Semaphore:***

```
LCD_semphr = xSemaphoreCreateMutex ( );
```

  It creates a mutex semaphore, it is a binary semaphore that its initial state of this semaphore is available.

  The states of semaphore is available or not available only.

- **Taking a Mutex Semaphore**:
  To take the semaphore, we will use:

```
state = xSemaphoreTake ( LCD_semphr , Max_blockage_ticks
);
```

- **Giving a Mutex Semaphore**:
  To Give the semaphore, we will use:

```
state = xSemaphoreGive ( LCD_semphr );
```

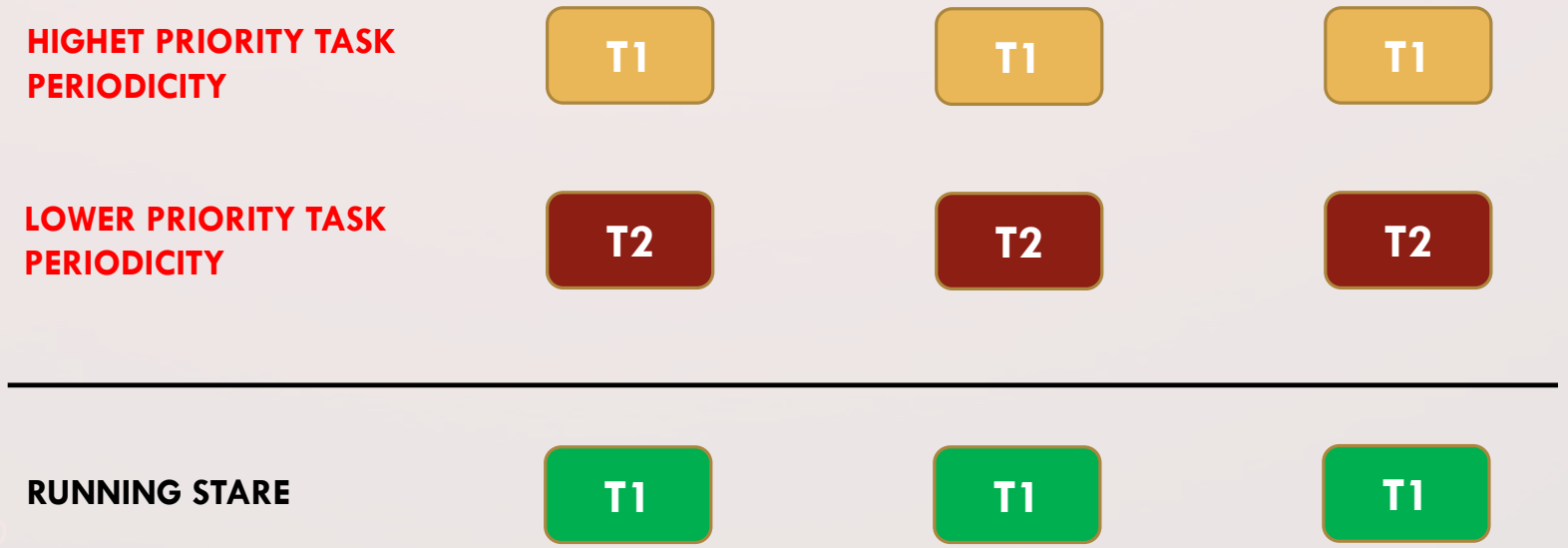
COMMON ISSUES DUE TO USAGE OF SEMAPHORES :

- Priority inversion
- Deal locks
- Starvation

COMMON ISSUES DUE TO USAGE OF SEMAPHORES :

- **Starvation**:
  it is happened when two tasks has the same periodicity and different priorities, so every time the task with the lowest priority will be ready at the same time with the task with higher Priority ,so the lowest will never go to Running State which will cause a starvation.

  because every time the higher priority task will take the semaphore first, and the lower priority task will not take this semaphore.



COMMON ISSUES DUE TO USAGE OF SEMAPHORES :

- **Dead Lock:**

In the next example, we have two tasks have the same periodicity and priority, they work in a pre-emptive kernel, so the Round-Robin algorithm will be run, so if the task1 will take “SA”, then task2 will take “SB”, then if task1 tries to take “SB”, it will fail, and if task2 tries to take “SA”, it will fail, and so on, so the code will be locked, so the arrangement of taking and giving must be same to avoid this blocking.

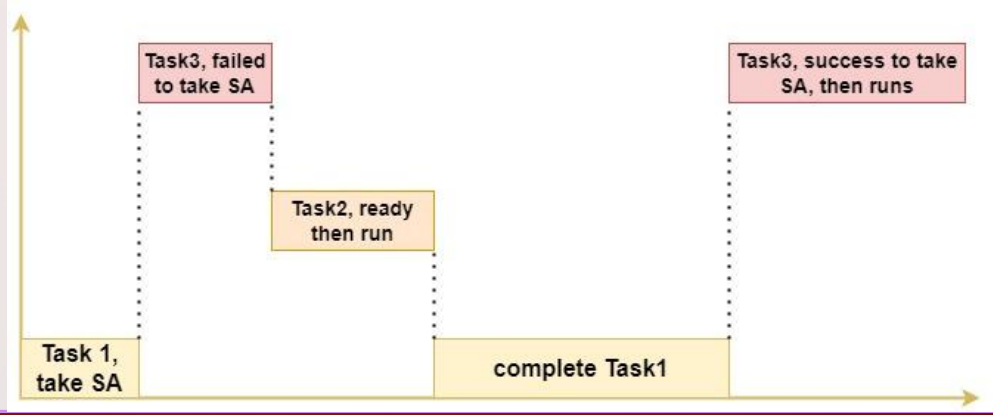
Task1	Task2
Take SA	Take SB
Take SB	Take SA
//code	//code
Give SB	Give SA
Give SA	Give SB

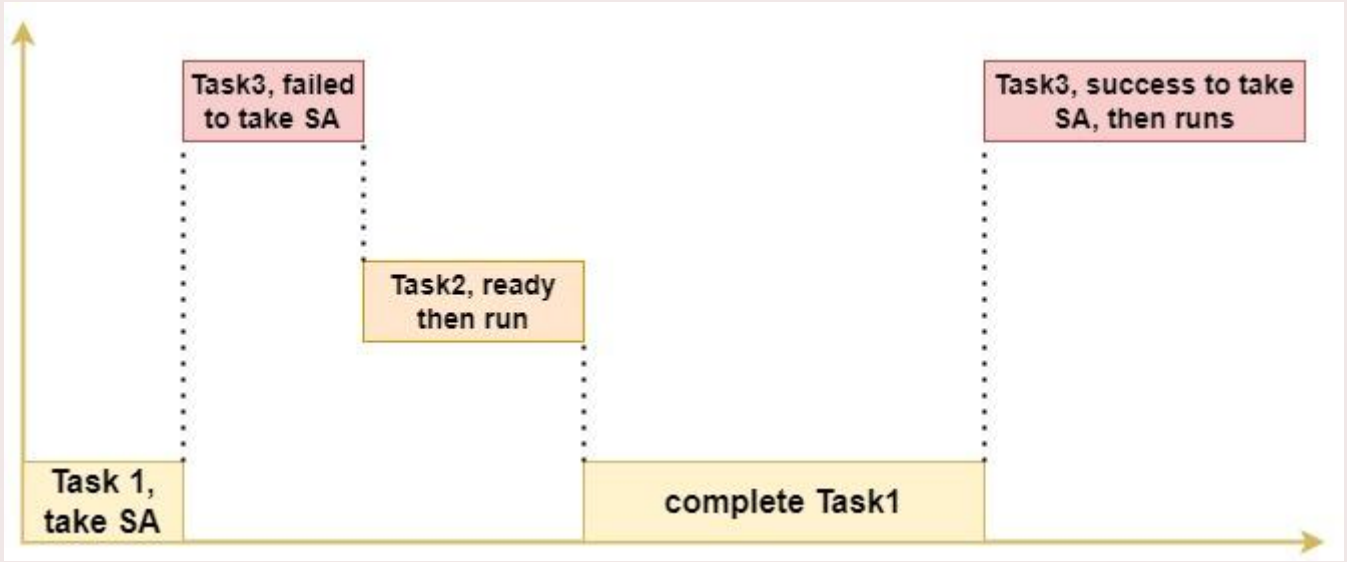
COMMON ISSUES DUE TO USAGE OF SEMAPHORES :

Priority inversion:

Assume that we has three tasks

1. **Task 1 is the lowest priority and needs to take Semaphor “SA”.**
2. **Task 2 has a medium priority and does not need semaphores.**
3. **Task 3 has the highest priority, and needs to take the same semaphore “SA”.**
4. **Task 1 runs and takes the semaphore.**
5. **Then Task 2 & Task 3 become ready.**
6. **Task 3 will run and try to take the semaphore but it will fail, then it will be in blockage state**
7. **The scheduler will find that task2 is ready and task1 wants to complete.**
8. **Task 2 will run first.**
9. **After running of Task 2.**
10. **the scheduler will complete Task 1, until Task 1 releases the semaphore.**
11. **the Task 3 will be ready and run to take the semaphore, then it will run.**

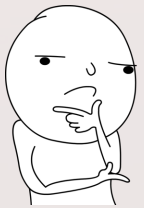




SO,WHAT IS WRONG THAT WAS HAPPENED IN THIS EXAMPLE ?..

Answer:

Task2 ran before task3 although task3 has a higher priority than task2, this is called “Priority Inversion” and it happens because of semaphore.

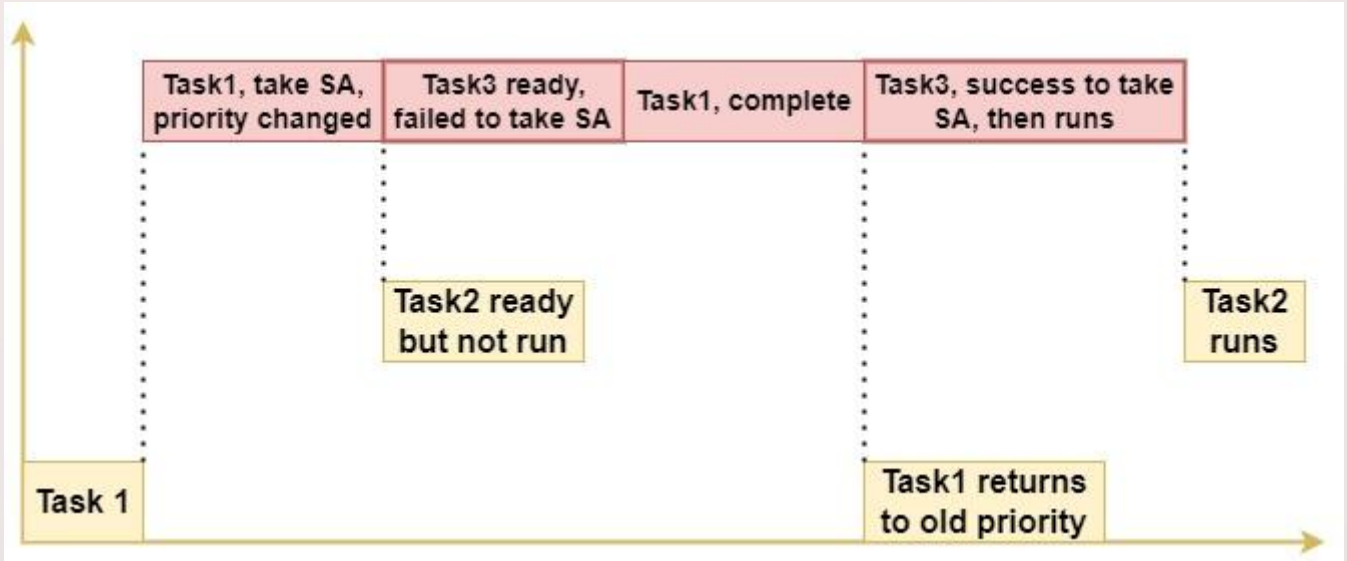


SO, HOW TO AVOID THIS PROBLEM?...

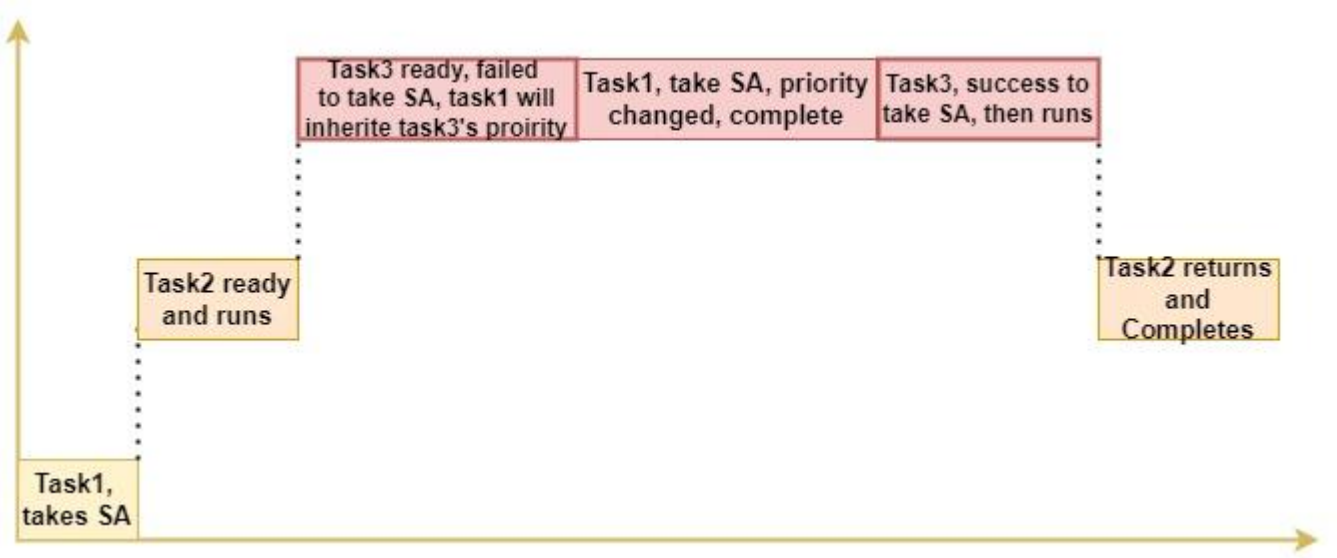
Solutions of priority inversion:

1. PRIORITY CEILING
2. PRIORITY INHERITANCE

- **Priority Ceiling:**
  It means that the semaphore will be created having a priority, and any task take this semaphore its priority will be temporary changed to the semaphore priority, so when task2 and task3 become ready, task2 will not run because task1 has a higher priority than task2, but task3 will be run using Round-Robin algorithm, but task3 will be blocked because of the semaphore, so task1 will complete until it releases the semaphore, so task3 will be run, then task2 will be run. This feature is not supported by freeRTOS.



- **Priority inheritance**:
  It means that the task1 runs, it will take the semaphore but without any priority upgrade, so when task2 becomes ready, it will interrupt the task1 and run, so when task3 becomes ready it will interrupt task2, but it will fail to take the semaphore, so it will be blocked, but at this point task1 will inherit the priority of task3, so task1 will complete because task1 priority becomes higher than task2 priority, then task3 will complete first, then task2 will complete, also.
  To use this feature is supported by freeRTOS by creating semaphore from MUTEX type.



Semaphore	Mutex
Semaphore is a signaling mechanism	Mutex is a locking mechanism.
allow multiple threads to access multiple resources	allow multiple thread to access a single resource but not simultaneously.
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.
Semaphore types are counting and binary semaphore.	Mutex is type of binary semaphore
semaphore works in kernel space	Mutex works in user space
	applies the mechanism of priority inheritance

THANK YOU !

AMIT