Real Time Operating System

## Lecture (2)
FreeRTOS

Free RTOS

FreeRTOS is solely owned, developed and maintained by Real Time Engineers Ltd. Real Time Engineers Ltd. have been working in close partnership with the world's leading chip companies for well over a decade to provide you award winning, commercial grade, and completely free high quality software.

- ✓ **FreeRTOS** is ideally suited to deeply embedded real-time applications that use microcontrollers or small microprocessors.
- ✓ This type of application normally includes a mix of both hard and soft real-time requirements.
- ✓ **FreeRTOS** is a real-time kernel (or real-time scheduler) on top of which embedded applications can be built to meet their hard real-time requirements.
- ✓ It allows applications to be organized as a collection of independent threads of execution.
- ✓ On a processor that has only one core, only a single thread can be executing at any one time.

FreeRTOS Features
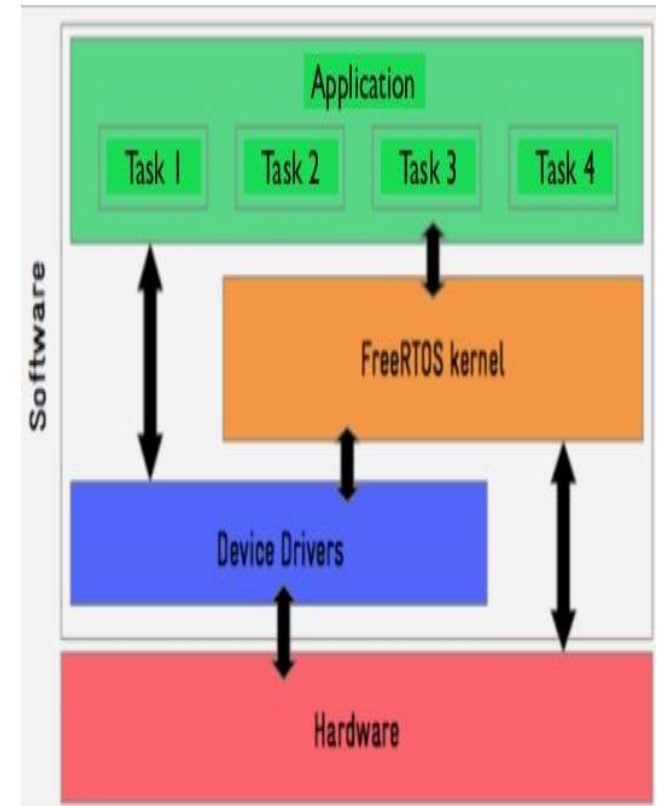
- ✓ Pre-emptive or co-operative operation
- ✓ Flexible, fast and light weight task notification mechanism
- ✓ Queues
- ✓ Binary semaphores
- ✓ Counting semaphores
- ✓ Mutexes
- ✓ Recursive Mutexes
- ✓ Software timers
- ✓ Event groups
- ✓ Tick hook functions
- ✓ Idle hook functions
- ✓ Stack overflow checking
- ✓ Trace recording
- ✓ Task run-time statistics gathering

FreeRTOS source file

You can download freeRTOS for your target H.W from this link : www.freertos.org

Then you can use freeRTOS by including these source and header files into your project

Quality RTOS & Embedded Software
About Contact Support FAQ Download

| Quick Start | Supported MCUs | PDF Books | Trace Tools | Ecosystem | Email List + |

Home
MIT License
FreeRTOS Books and Manuals
⊞ FreeRTOS
⊞ FreeRTOS Interactive!

Quick Start Guide
Support Forum
⇓ Download Source ⇓

FreeRTOS+ Ecosystem
FreeRTOS+TCP:
Thread safe TCP/IP stack
SafeRTOS:
TUV certified RTOS
OpenRTOS:
Commercial Licensed RTOS
Fail Safe File System:
Ensures data integrity
FreeRTOS BSPs:
3rd party driver packages
Trace & Visualisation:
Tracealyzer for FreeRTOS

RTOS Source Code Download Instructions

Follow the steps below to download the latest FreeRTOS versions:

1. Please keep up to date by joining the announcements mailing list for infrequent comprehensive notifications. We respect your privacy, so do not provide email addresses to any third party. Every email sent contains unsubscribe instructions.

2. Download the latest official release or a previous release from SourceForge, both of which are available as a standard zip file (.zip), and as a self extracting zip file (.exe). Alternatively obtain the source files directly from SVN.

   Download Source Code and Projects

   The FreeRTOS kernel is also available from AWS and via Github as part of the Amazon FreeRTOS operating system, and from various partner distributions.

3. Unzip the source code into a suitable directory - taking care to ensure the directory structure within the zip file is maintained. Please read the source code organisation and quick start guide pages to understand the directory structure and get up and running quickly.

4. The monitored support forum, known issue list, FAQ and version change description, all provide additional resources.

- croutine.c
- croutine.h
- FreeRTOS.h
- FreeRTOSConfig.h
- heap_1.c
- list.c
- list.h
- macros.h
- mpu_wrappers.h
- port.c
- portable.h
- portmacro.h
- projdefs.h
- queue.c
- queue.h
- semphr.h
- StackMacros.h
- task.h
- tasks.c
- timers.c
- timers.h

FreeRTOSConfig.h

FreeRTOS is configured by a header file called FreeRTOSConfig.h.

```
#define configUSE_PREEMPTION           1
#define configUSE_IDLE_HOOK            0
#define configUSE_TICK_HOOK            0
#define configCPU_CLOCK_HZ             ( ( unsigned long ) 8000000 )
#define configTICK_RATE_HZ             ( ( portTickType ) 1000 )
#define configMAX_PRIORITIES           ( ( unsigned portBASE_TYPE ) 3)
#define configMINIMAL_STACK_SIZE       ( ( unsigned short ) 85 )
#define configTOTAL_HEAP_SIZE          ( (size_t ) ( 600 ) )
#define configMAX_TASK_NAME_LEN        ( 8 )
#define configUSE_TRACE_FACILITY       0
#define configUSE_16_BIT_TICKS         1
#define configIDLE_SHOULD_YIELD        0
#define configQUEUE_REGISTRY_SIZE      0
```

xTaskCreate() API

```
BaseType_t xTaskCreate(TaskFunction_t pvTaskCode,
                       const char * const pcName,
                       uint16_t usStackDepth,
                       void *pvParameters,
                       UBaseType_t uxPriority,
                       TaskHandle_t *pxCreatedTask
                      );
```

Parameters:

1. **pvTaskCode** (pointer to the function that implements the task)
2. **pcName**   (Identifying a task by a human readable name is much simpler than attempting to identify it by its handle.)
3. **usStackDepth** (Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack)
4. ***pvParameters** (Task functions accept a parameter of type pointer to void (void*).
5. **uxPriority** (Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1))
6. ***pxCreatedTask** (pxCreatedTask can be used to pass out a handle to the task being created, for example, change the task priority or delete the task.)

EX

```
xTaskCreate( Blink_200ms_task,NULL, configMINIMAL_STACK_SIZE, NULL, 1, NULL );
xTaskCreate( Blink_1000ms_task,NULL, configMINIMAL_STACK_SIZE, NULL, 2, NULL);
```

vTaskDelay() API

void vTaskDelay(TickType_t xTicksToDelay);

Parameters:

1. xTicksToDelay (The number of tick interrupts that the calling task will remain in the Blocked state before being transitioned back into the Ready state.)

For example, if a task called vTaskDelay(100) when the tick count was 10,000, then it would immediately enter the Blocked state, and remain in the Blocked state until the tick count reached 10,100.

EX

```
void Blink_1000ms_task( void *pvParameters)
{
    DDRB |=(1<<0); //PB.0 is output

    while(1)
    {

        PORTB ^= (1<<0); //toggle PB.0
        vTaskDelay(1000); //OS Delay
    }
}
```

vTaskStartScheduler() API

void vTaskStartScheduler(void);

- ✓ Starts the FreeRTOS scheduler running.
- ✓ Typically, before the scheduler has been started, main() (or a function called by main()) will be executing. After the scheduler has been started, only tasks and interrupts will ever execute.
- ✓ The Idle task is created automatically when the scheduler is started.

Idle task : that is executed when none of the other tasks are ready to ran. The idle task is always set to the lowest priority

Note

will only return if there is not enough FreeRTOS heap memory available for the Idle task to be created.

Lab 1

Using FreeRTOS :
- we have two tasks. First task blinks a LED every 200 msec and the other one blinks a LED every 1000 msec.
- The two tasks run in parallel.

Time To Code

Lab 2

Using FreeRTOS create Two TASKS :

1. Write on LCD "I am TASK 1 " for 1 sec.
2. Write on LCD "I am TASK 2 " for 1 sec.

Shared Resource

Critical section :

Code segment which need to be treated indivisibly .After a section of code starts executing ,it must not be interrupted .

Shared Resources :

A shared resource is a resource that can be used by more than one task .Each task should gain exclusive access to the shared resources to prevent data corruption.

Process 2

Process 1

Process 3

Shared resource (LCD)

Interrupt are typically disabled before critical section is executing and enabled when the critical code is finished
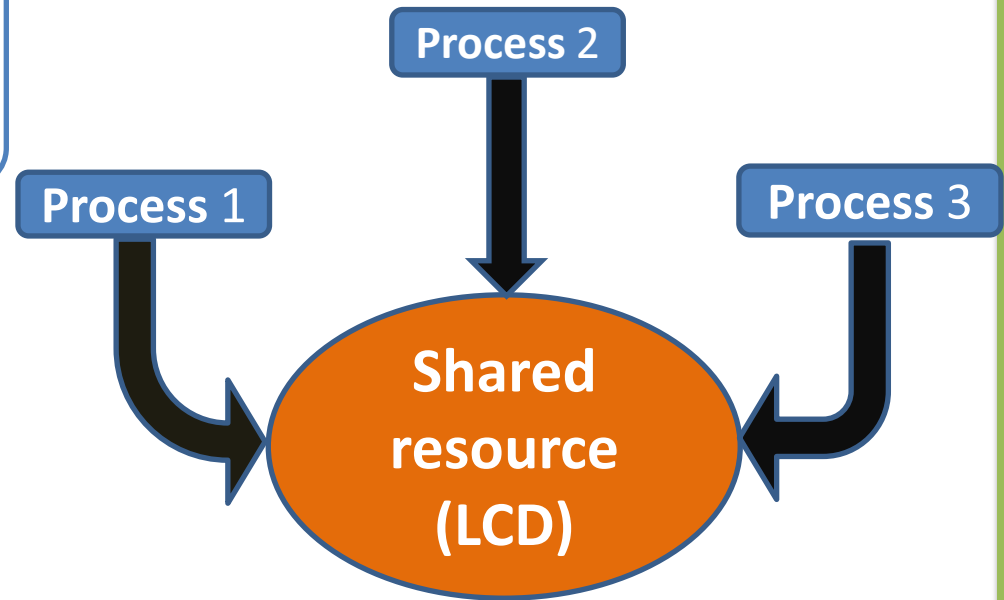
Semaphores

Imagine what would happen if two tasks were allowed to send characters to a printer at the same time. The printer would contain inter-leaved data from each task. For instance, the printout from Task 1 printing "I am Task 1!" and Task 2 printing "I am Task 2!" could result in:

I la amm T Tasask kl !2!

Semaphores are especially useful when tasks share I/O devices.

A *semaphore* is a kernel object that one or more threads of execution can acquire or release for the purpose of synchronization or mutual exclusion.

In this case, use a semaphore and initialize it to 1 (binary semaphore). The rule is simple: to access the printer, each task first must obtain the resource's semaphore.
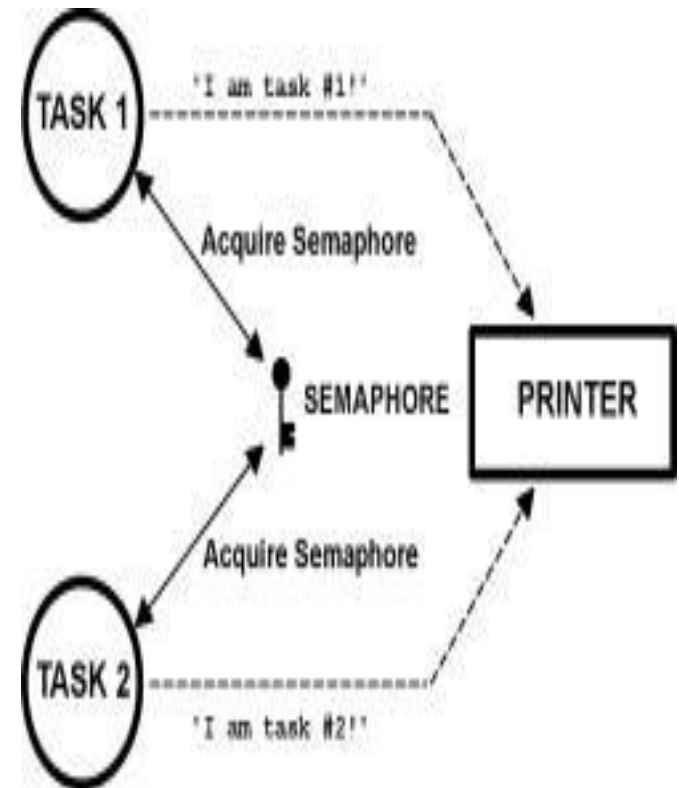
Semaphores

- ❑ A binary semaphore is like a *key* that allows a task to carry out some operation or to access a resource.
- ❑ There maybe several keys for each semaphore
- ❑ If the task can acquire the semaphore, it can carry out the intended operation or access the resource.
- ❑ Otherwise the task is blocked, if it chooses to wait for the release of the semaphore

- ❑ Blocked tasks are kept in a task-waiting list.
- ❑ When the semaphore is released, one task of the task waiting list gets access to the semaphore and is put into ready state
- ❑ The exact implementation of the task-waiting list depends on the RTOS kernel.

Binary Semaphores
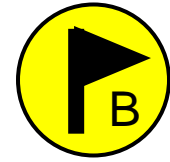
> *A* **binary semaphore** is used when a resource can be used by only one task at a time.

- ❑ A binary semaphore has either a value of:
  - 0 (unavailable)
  - 1 (available)
- ❑ The task WaitTask has to wait until the task SignalTask releases the binary semaphore.
- ❑ A binary semaphore is a shared resource
  - Any task can release it!

SignalTask

WaitTask
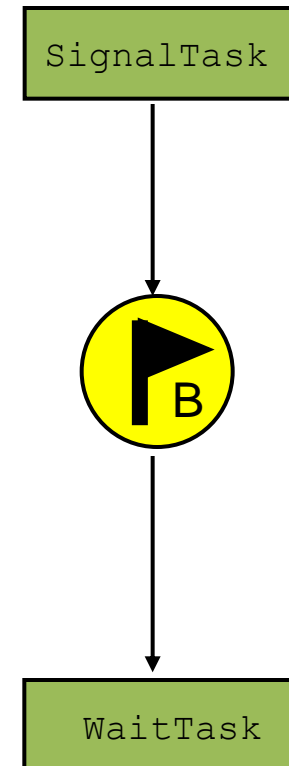
Counting Semaphores
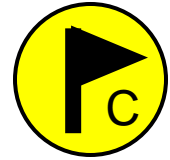
A **counting semaphore** is used when a resource can be used by more than one task at the same time

- ❑ A counting semaphore uses a count to be able to give more than one task access :
    - • 0 (unavailable)
    - • > 0 (available)
- ❑ with the semaphore count initialized to the number of free resources.
- ❑ Threads then atomically increment the count when resources are added and atomically decrement the count when resources are removed.
- ❑ When the semaphore count becomes zero, indicating that no more resources are present, threads trying to decrement the semaphore block wait until the count becomes greater than zero.
- ❑ A counting semaphore is a shared resource
    - • Any task can release it!
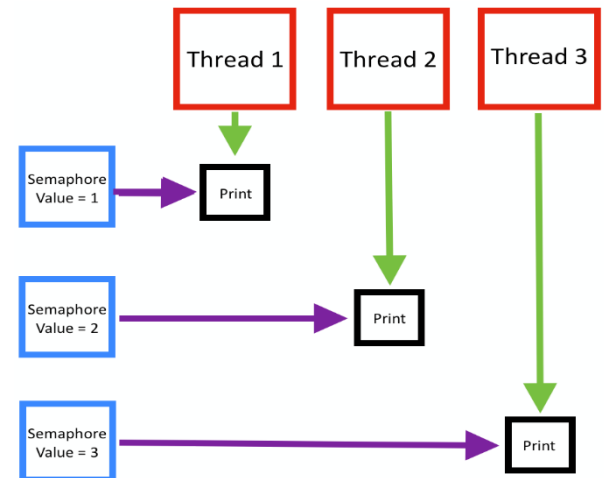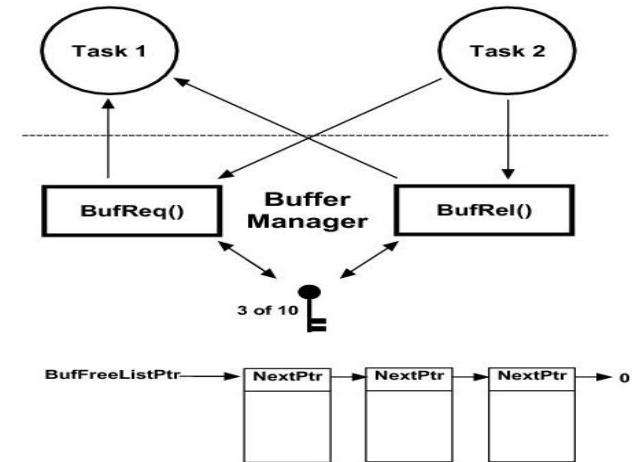
binary Semaphores creation

Defining a semaphore

A semaphore is referenced by a global variable of type xSemaphoreHandle

```
/*create a semaphore handle*/
xSemaphoreHandle binary_semaphore ;
```

Creating a binary semaphore

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

➢ Creates a binary semaphore, and returns a handle by which the semaphore can be referenced.
➢ Each binary semaphore requires a small amount of RAM that is used to hold the semaphore's state.

Return Values:

NULL :The semaphore could not be created because there was insufficient heap memory available for FreeRTOS to allocate the semaphore data structure.

Any other value :The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

EX
```
//creating binary semaphore
binary_semaphore = xSemaphoreCreatBinary();
```

Counting Semaphores creation

Defining a semaphore

```
/*create a semaphore handle*/
xSemaphoreHandle Counting_semaphore ;
```

Creating a counting semaphore

```
SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount,
                                            UBaseType_t uxInitialCount);
```

> Creates a counting semaphore, and returns a handle by which the semaphore can be referenced.

Parameters:

uxMaxCount :The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.

uxInitialCount :The count value assigned to the semaphore when it is created.

Return Values :

NULL :Returned if the semaphore cannot be created.

Any other value :The count value assigned to the semaphore when it is created.

EX

```
/*Create counting semaphore with initial value 1
counting_semaphore = xSemaphoreCreateCounting(5,1)
```

xSemaphoreTake() API

xSemaphoreTake proto type

BaseType_t xSemaphoreTake(**SemaphoreHandle_t** xSemaphore, **TickType_t** xTicksToWait);

'Takes' (or obtains) a semaphore that has previously been created using a call to vSemaphoreCreateBinary(),

Parameters:

xSemaphore :The Semaphore being 'Taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

xTicksToWait :The maximum amount of time the task should remain in the Blocked state to wait for the semaphore to become available, if the semaphore is not available immediately.

Return Values :

pdPASS: Returned only if the call to xSemaphoreTake() was successful in obtaining the semaphore.

pdFAIL: Returned if the call to xSemaphoreTake() did not successfully obtain the semaphore.

xSemaphoreGive() API

> **xSemaphoreGive proto type**

`BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );`

'Gives' (or releases) a semaphore that has previously been created using a call to vSemaphoreCreateBinary(),

Parameters:
 xSemaphore :The Semaphore being 'given'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

Return Values :
 pdPASS: The semaphore 'give' operation was successful.
 pdFAIL: The semaphore 'give' operation was not successful

Note

the task calling xSemaphoreGive() is not the semaphore holder. A task must successfully 'take' a semaphore before it can successfully 'give' it back.

Lab 3

We have two tasks, the communicate with each other using a binary semaphore:

1. First task: Scans a push button every 200msec and gives the semaphore when it's pressed.
2. Second Task: Waits for that semaphore and toggle a LED once when getting it.

Time To
Code

Lab 4

We have two tasks, the communicate with each other using a counting semaphore:

1. A tactile switch which is connected to external interrupts EXT0 .when ISR takes place it gives the semaphore.
2. Second Task: Waits for that semaphore and write "button is pressed" on LCD.

Time To Code

Priority Inversions

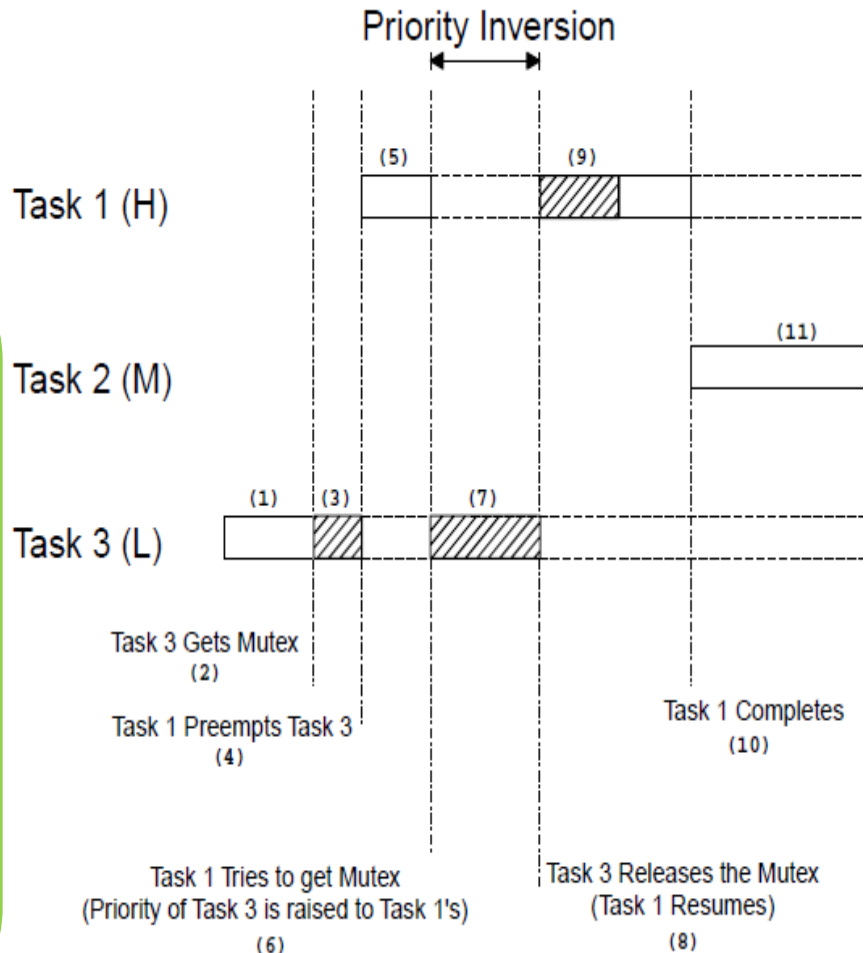Priority inversion is a problem in real-time systems and occurs mostly when you use a real-time kernel.

- In *Priority Inversion*, higher priority task (H) ends up waiting for middle priority task (M).
- H is sharing critical section with lower priority task (L).
- L is already in critical section.
- Effectively, H waiting for M results in inverted priority
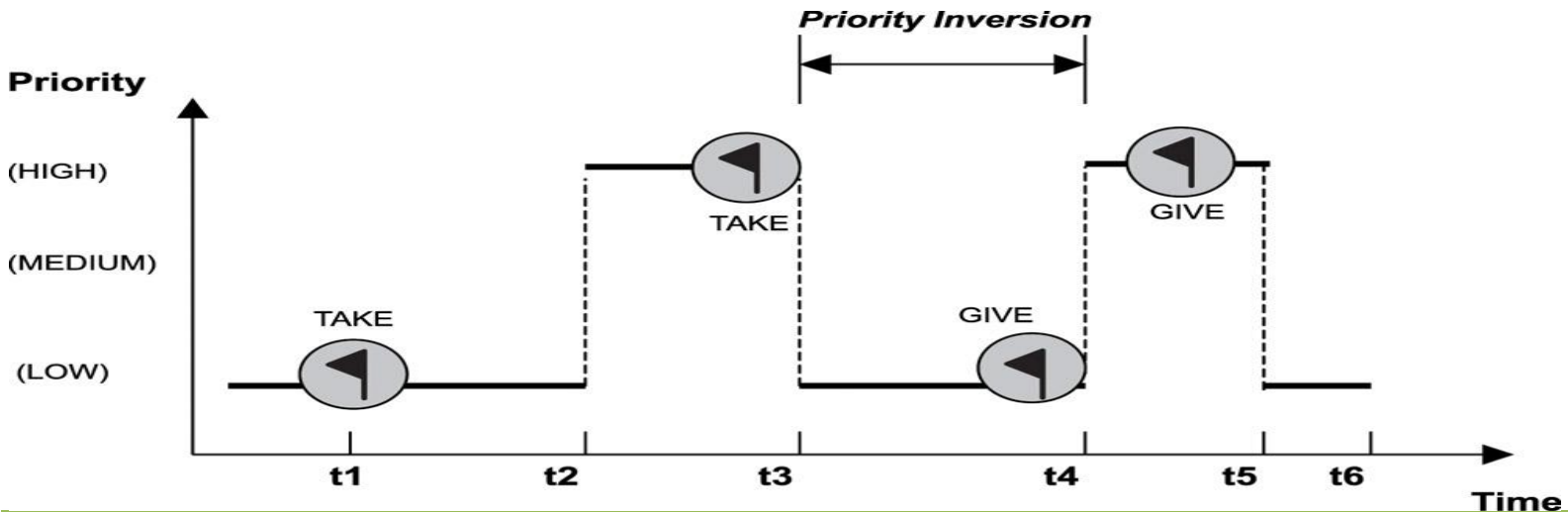
Priority Inheritance

One of the solution for this problem is *Priority Inheritance*..

- ❑ In *Priority Inheritance*, when L is in critical section, L inherits priority of H at the time when H starts pending for critical section. By doing so, M doesn't interrupt L and H doesn't wait for M to finish.
- ❑ L goes back to its old priority when L comes out of critical section.
- ❑ Inheriting of priority is done temporarily .

Priority Inversion

Task 1 (H)   (5)   (9)

Task 2 (M)   (11)

Task 3 (L)   (1) (3)   (7)

Task 3 Gets Mutex
(2)

Task 1 Preempts Task 3
(4)

Task 1 Tries to get Mutex
(Priority of Task 3 is raised to Task 1's)
(6)

Task 1 Completes
(10)

Task 3 Releases the Mutex
(Task 1 Resumes)
(8)

Deadlock

A *deadlock* , is a situation in which two tasks are each unknowingly wait-ing for resources held by the other. Assume process 1 has exclusive access to Resource1 and process 2 has exclusive access to Resource 2 . If process 1 needs exclusive access to Resource1 and process 2 needs exclusive access to Resource2. neither task can continue. They are deadlocked.
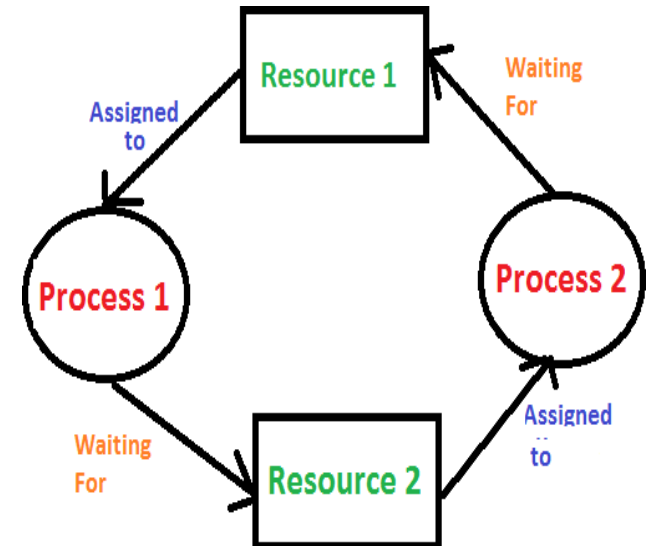
The simplest way to avoid a deadlock is for tasks:
1.   acquire all resources before proceeding.
2.   acquire the resources in the same order.
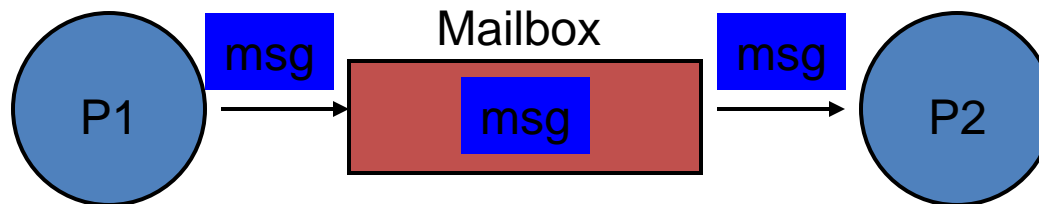3.   release the resources in the reverse order.
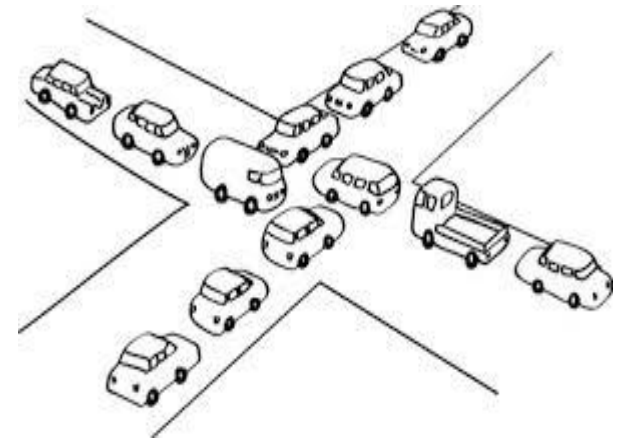
Message passing

Message passing between processes:

- ❑ Message passing is an abstract communication scheme.
- ❑ Processes communicate by sending and receiving messages.
- ❑ A received message can initiate new actions (like an interrupt).
- ❑ Message passing can also be implemented on shared memory architectures.

Message queue

- ❑ A *message queue* is a buffer-like object through which tasks and ISRs send and receive messages to communicate and synchronize with data.
- ❑ A message queue holds temporariliy messages from a sender until the intended receiver is ready to read them
- ❑ Temporary buffering decouples sending from receiving
- ❑ A message queue with a single buffer is called a *mailbox*
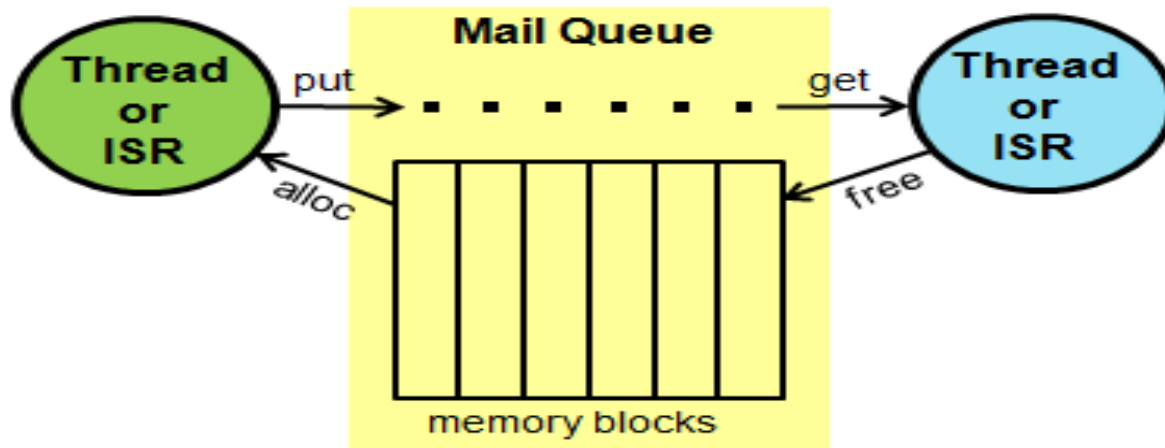
message queue creation

Defining a queue

```
/*create Message Queue Handle*/
xQueueHandle buffer;
```

Creating a queue

```
xQueueHandle xQueueCreate( UBaseType_t uxQueueLength,
                           UBaseType_t uxItemSize );
```

➡ Creates a new queue and returns a handle by which the queue can be referenced.

Parameters:
- **uxQueueLength**:The maximum number of items that the queue being created can hold at any one time.
- **uxItemSize** :The size, in bytes, of each data item that can be stored in the queue.

Return Values:
- **NULL**:The queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.

- **Any other value** :The queue was created successfully. The returned value is a handle by which the created queue can be referenced.

EX
```
/*create message queue */
buffer = xQueueCreate(6,sizeof(unsigned char));
```

xQueueSend() API

```
BaseType_t xQueueSend( QueueHandle_t xQueue,
                       const void * pvItemToQueue,
                       TickType_t xTicksToWait );
```

Sends (writes) an item to the front or the back of a queue.

Parameters:
- **xQueue:** The handle of the queue to which the data is being senT written
  pvItemToQueue : A pointer to the data to be copied into the queue.
- **xTicksToWait   :** The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue.

Return Values:
- **pdPASS :** Returned if data was successfully sent to the queue.
- **errQUEUE_FULL :** Returned if data could not be written to the queue because the queue was already full.

EX

```
/*if a key is pressed send the value to buffer" message queue"*/
xQueueSend(buffer,&val,portMAX_DELAY);
```

xQueueReceive () API

BaseType_t   **xQueueReceive(QueueHandle_t xQueue,**
                                 void *pvBuffer,
                                 TickType_t xTicksToWait);

➡ Receive (read) an item from a queue

Parameters:
- **xQueue:** The handle of the queue to which the data is being received (read).
- **pvBuffer:** A pointer to the data to be received data will be copied .
- **xTicksToWait** :The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue.

Return Values:
- **pdPASS :** Returned if data was successfully read from the queue.
- **errQUEUE_FULL :** Returned if data could not be read from the queue because the queue is already empty.

EX
```
//Receive a message from the queue
ret = xQueueReceive(buffer,&val,portMAX_DELAY);
```

Lab 5

We have two tasks, the communicate with each other using a Message queue:
1. First task : check on keypad every 50ms and send the number of pressed button in the queue.
2. Second task : receive a the key no from the queue and display it on LCD .

Time To
Code

www.imtschool.com

ww.facebook.com/imaketechnologyschool/