Resource Management in FreeRTOS

Mojtaba Bagherzadeh, Adrien Lapointe

Royal Military College (RMC)

mojtaba@cs.queensu.ca,adrien.lapointe@rmc.ca

February 25, 2018

Overview

The Problem

In a multitasking system, there is potential for errors if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state. This issue can happen in different situations such as:

The Problem

In a multitasking system, there is potential for errors if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state. This issue can happen in different situations such as:

- Accessing peripherals devices (e.g., writing in display by multiple tasks)

The Problem

In a multitasking system, there is potential for errors if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state. This issue can happen in different situations such as:

- Accessing peripherals devices (e.g., writing in display by multiple tasks)
- Read, modify, write operations

The Problem

In a multitasking system, there is potential for errors if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state. This issue can happen in different situations such as:

- Accessing peripherals devices (e.g., writing in display by multiple tasks)
- Read, modify, write operations
- Non-atomic access to variables (e.g., updating multiple members of a structure or updating a 32-bit variable on a 16-bit machine)

The Problem

In a multitasking system, there is potential for errors if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state. This issue can happen in different situations such as:

- Accessing peripherals devices (e.g., writing in display by multiple tasks)
- Read, modify, write operations
- Non-atomic access to variables (e.g., updating multiple members of a structure or updating a 32-bit variable on a 16-bit machine)
- Function reentrancy

Mutual Exclusion

To ensure data consistency when sharing a resource, a 'mutual exclusion' technique can be used. In FreeRTOS Mutual exclusion can be implemented using several methods including

- Critical section
- Suspending (or Locking) the scheduler
- Mutexes (and binary semaphores)
- Gatekeeper tasks

Critical Section

```
taskENTER_CRITICAL(); // start critical section
 {
        // access to shared resources
 }
 taskEXIT_CRITICAL(); // start critical section
```

Critical Section

```
taskENTER_CRITICAL(); // start critical section
 {
        // access to shared resources
 }
 taskEXIT_CRITICAL(); // start critical section
```

How Does It Work?

A task that called `taskENTER_CRITICAL()` is guaranteed to remain in the Running state until the critical section is exited.

Critical Section

```
taskENTER_CRITICAL(); // start critical section
 {
        // access to shared resources
 }
 taskEXIT_CRITICAL(); // start critical section
```

How Does It Work?

A task that called `taskENTER_CRITICAL()` is guaranteed to remain in the Running state until the critical section is exited.

How Critical Section Is Implemented?

It is implemented by disabling interrupts, either completely, or increasing the task priority up to the interrupt priority set by `configMAX_SYSCALL_INTERRUPT_PRIORITY`.

Suspending (or Locking) the Scheduler

```
vTaskSuspendAll(  ); // start critical section
{
        // access to shared resources
}
xTaskResumeAll( void ); // start critical section
```

Suspending (or Locking) the Scheduler

```
vTaskSuspendAll(  ); // start critical section
{
        // access to shared resources
}
xTaskResumeAll( void ); // start critical section
```

How Does It Work?

Disable the preemption by disabling the scheduler allows the task remain in Running state until scheduler is resumed.

Mutexes (and Binary Semaphores)

Definition

The word MUTEX originates from 'MUTual EXclusion'. The mutex can be considered as a token that is associated with a resource being shared. To use the resource, a task should first take the mutex, hold it during use, and release it after use.

Mutexes (and Binary Semaphores)

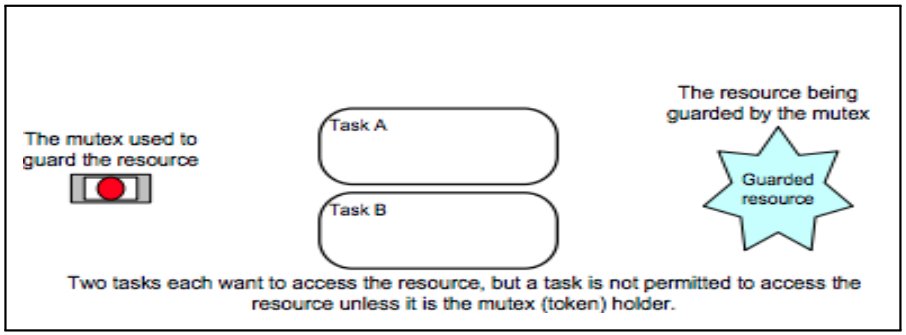
Definition

The word MUTEX originates from 'MUTual EXclusion'. The mutex can be considered as a token that is associated with a resource being shared. To use the resource, a task should first take the mutex, hold it during use, and release it after use.

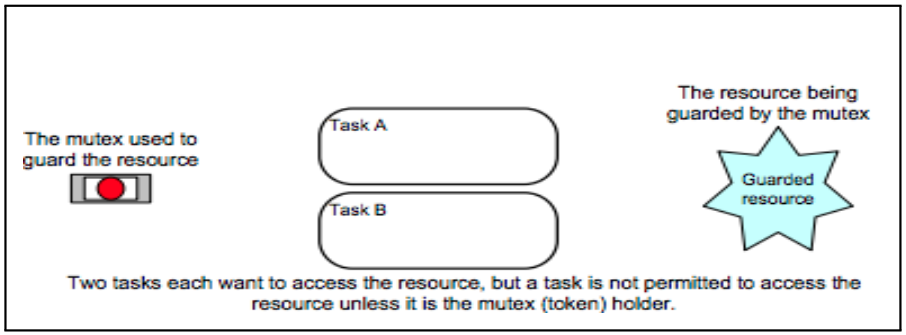
Implemnation

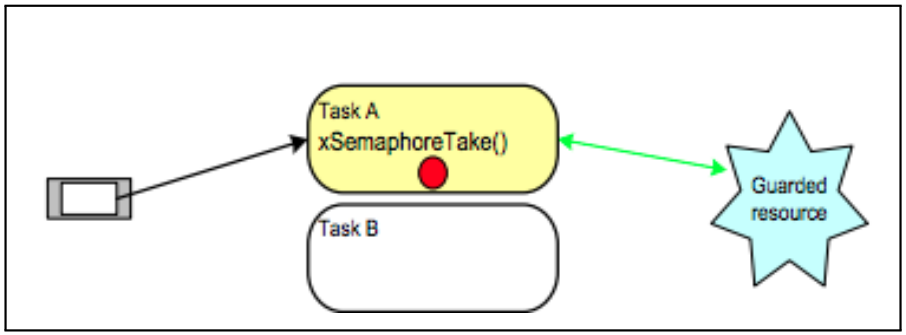
The mutex is implemented as a binary semaphore which should be returned after use. To use the mutex, the `configUSE_MUTEXES` must be set to 1.

How does mutex work?

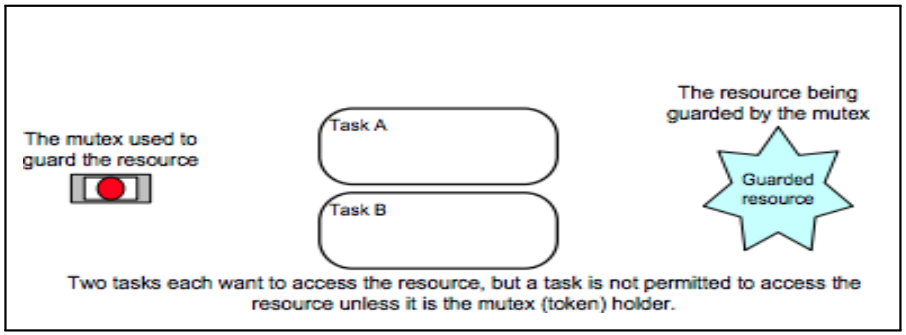


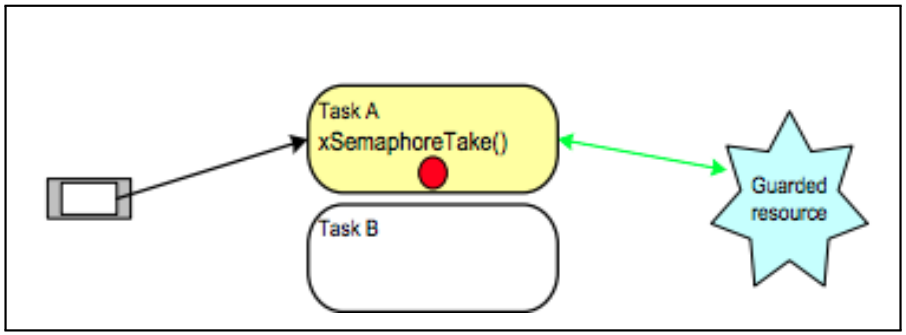
How does mutex work?

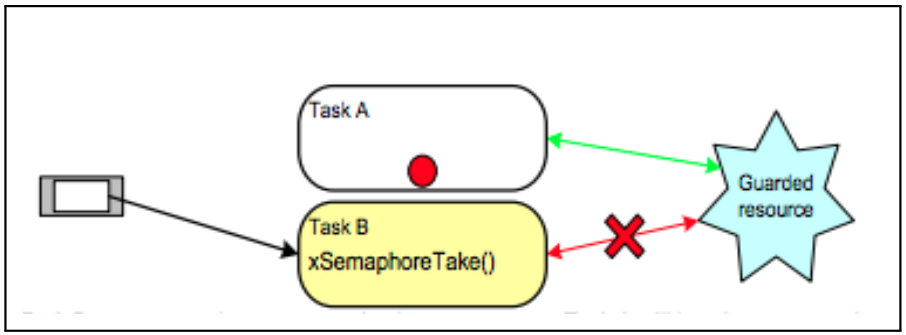




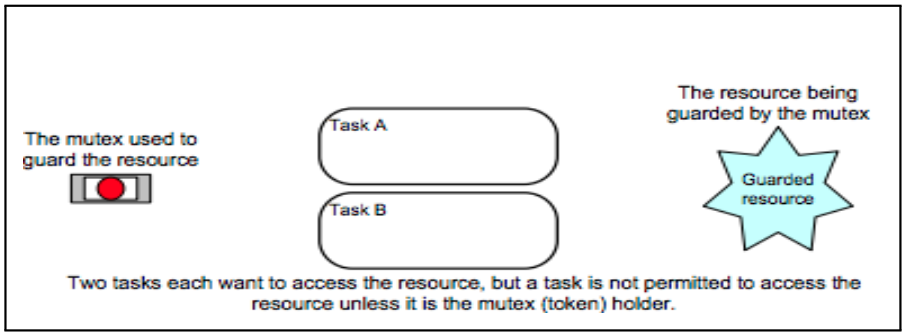
How does mutex work?

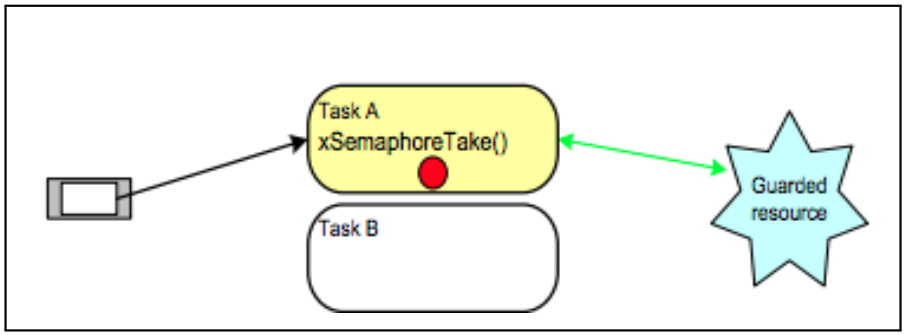


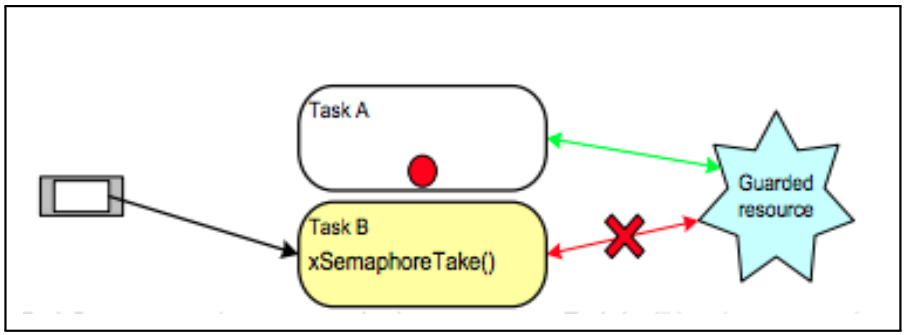


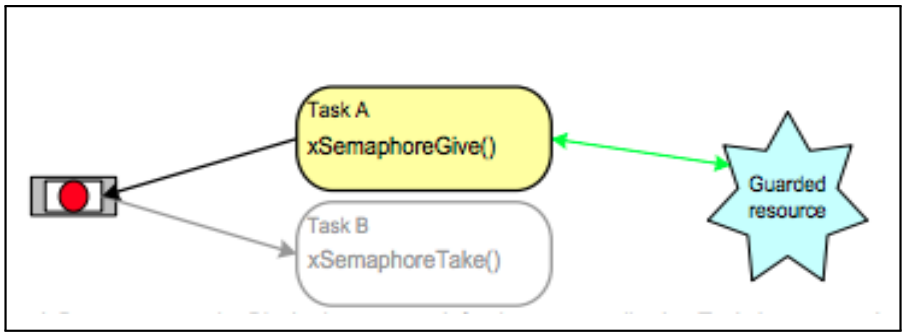


How does mutex work?

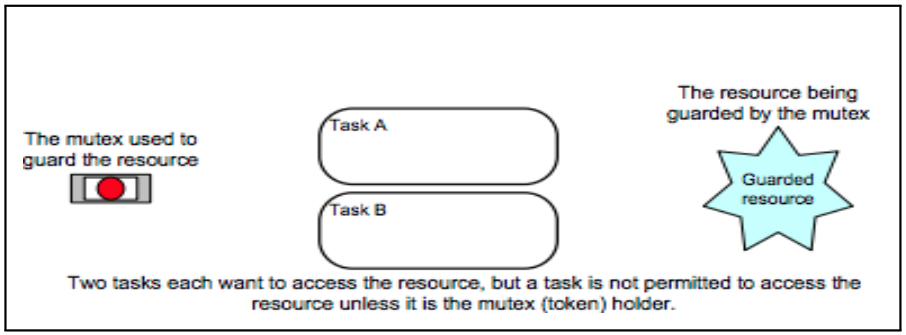


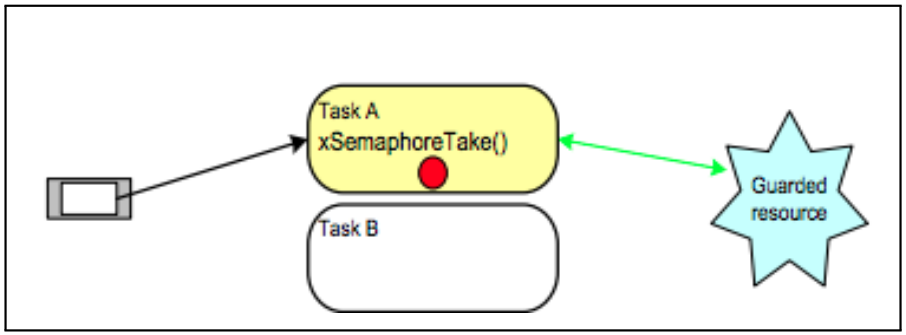


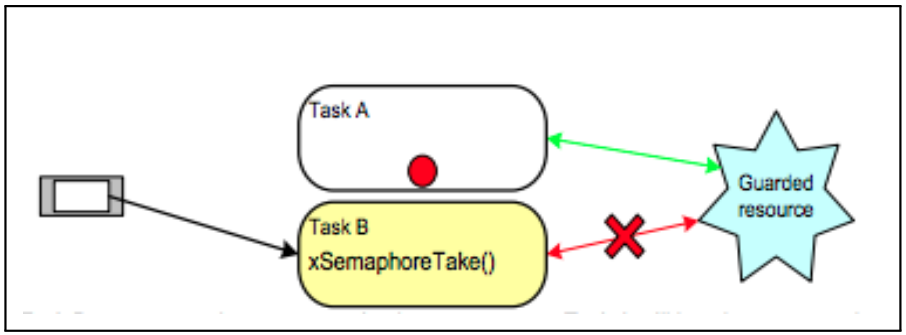


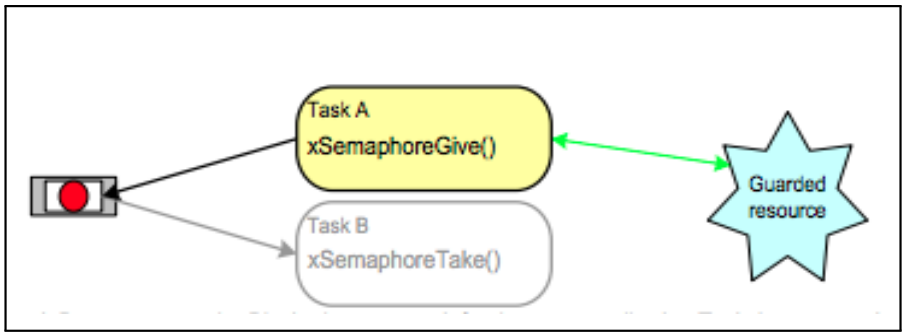


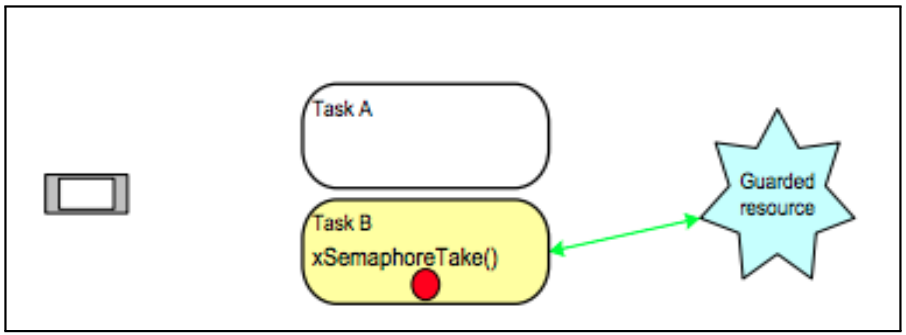
How does mutex work?



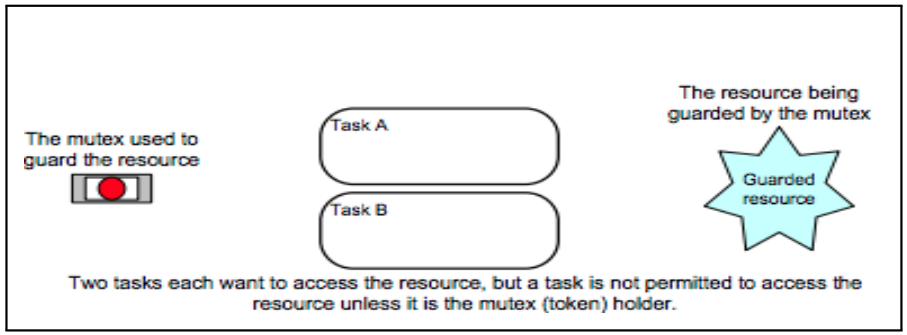


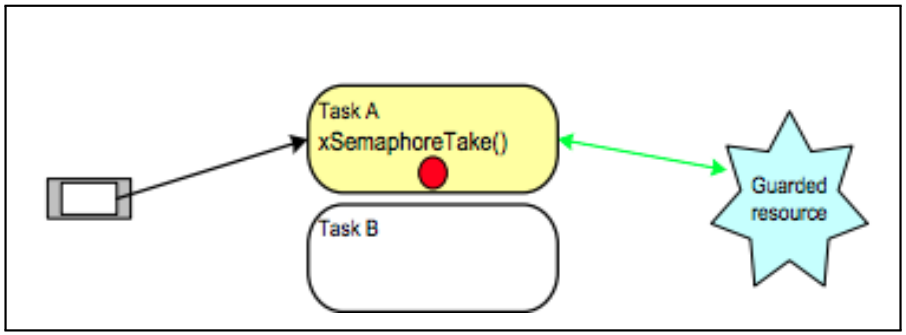


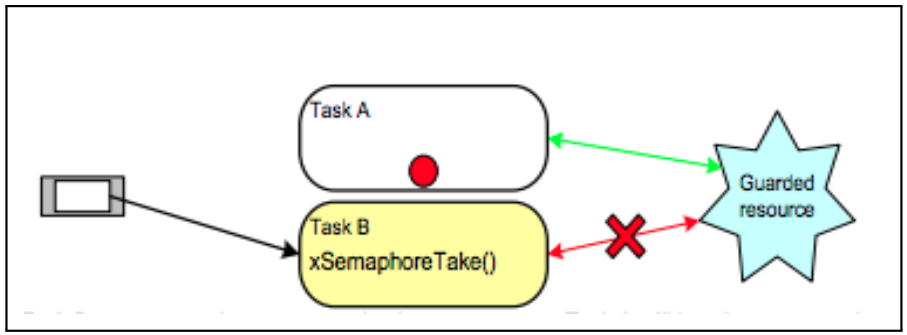


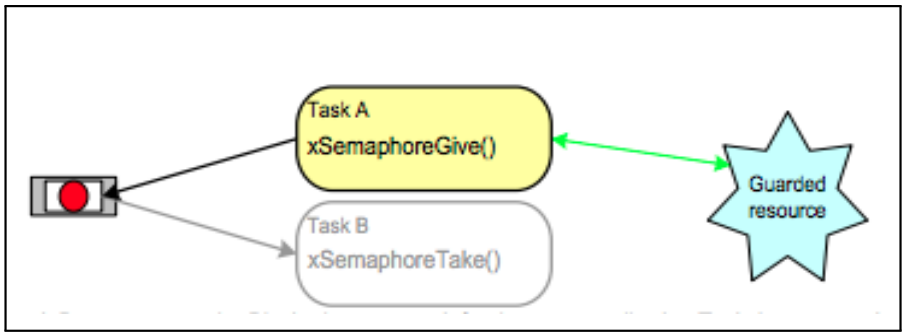


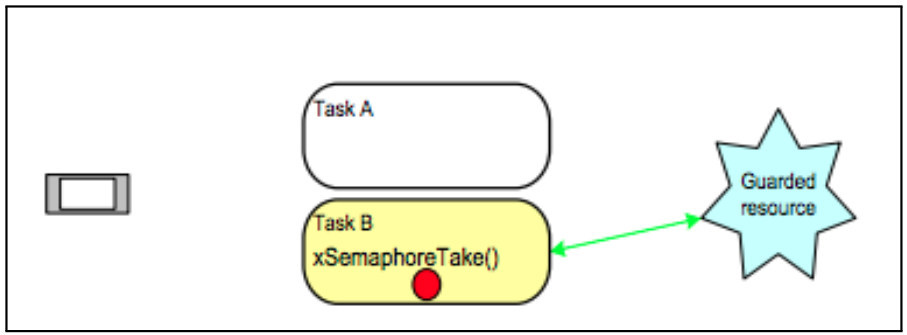
How does mutex work?

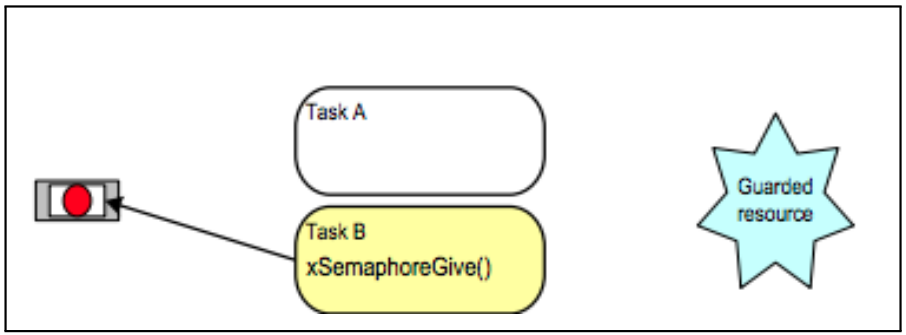












Mutex Operations

Create a Semaphore

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

`xSemaphoreCreateMutex` creates a mutex and returns its handle.

Mutex Operations

Create a Semaphore

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

`xSemaphoreCreateMutex` creates a mutex and returns its handle.

Take a Semaphore

```
xSemaphoreTake( SemaphoreHandle_t xSemaphore,TickType_t xTicksToWait )
```

`xSemaphoreTake` take the semaphore specified by `SemaphoreHandle_t`. The owner task blocks if the semaphore is taken by others. `xTicksToWait` specifies the maximum blocking time.

Mutex Operations

Create a Semaphore

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

`xSemaphoreCreateMutex` creates a mutex and returns its handle.

Take a Semaphore

```
xSemaphoreTake( SemaphoreHandle_t xSemaphore,TickType_t xTicksToWait )
```

`xSemaphoreTake` take the semaphore specified by `SemaphoreHandle_t`. The owner task blocks if the semaphore is taken by others. `xTicksToWait` specifies the maximum blocking time.

Release a Semaphore

```
xSemaphoreGive( SemaphoreHandle_t xSemaphore )
```

`xSemaphoreGive` release the semaphore which is specified by `SemaphoreHandle_t`.

Problems with Mutex

- Priority Inversion

Problems with Mutex

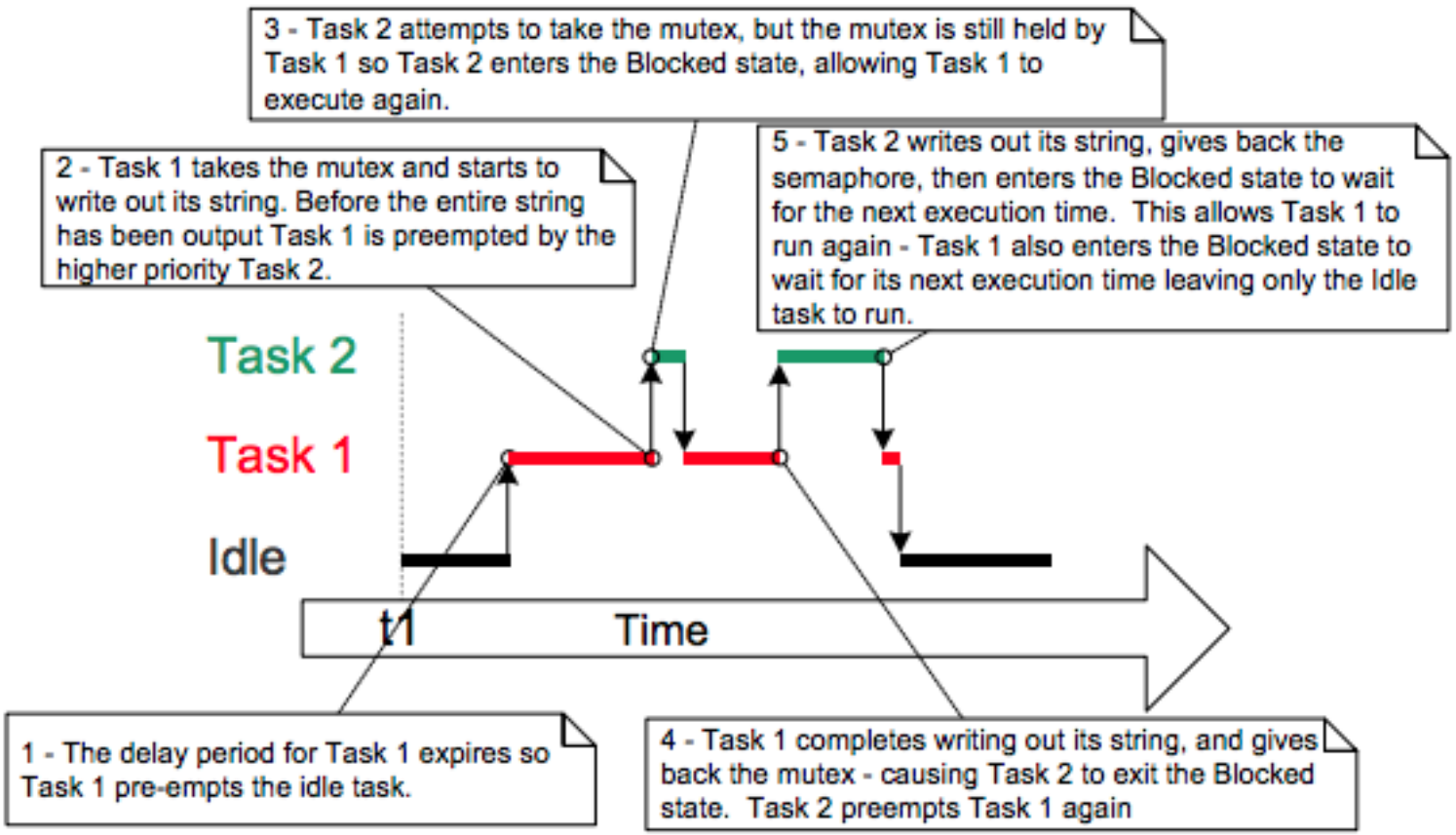
- Priority Inversion
- Deadlock

Problems with Mutex

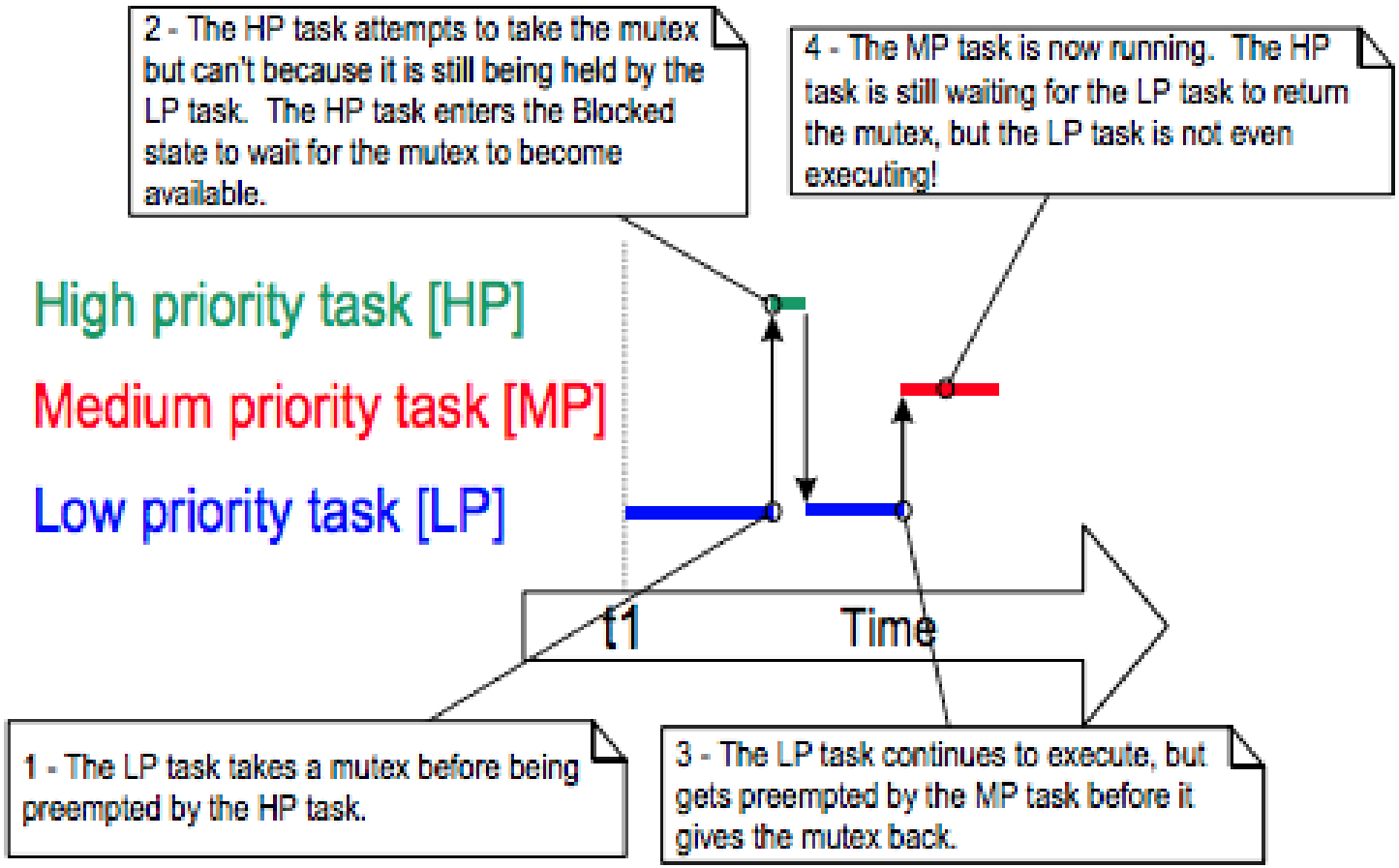
- Priority Inversion
- Deadlock
- Self-deadlock

Priority Inversion

A higher priority task is delayed by a lower priority task when sharing a resource that is taken by the low priority task before the high priority task.



Priority Inversion Worst Case Scenario



Possible Solutions for Priority Inversion

- Priority Inheritance
- Ceiling Protocols

Deadlock

Deadlock

Deadlock occurs when two tasks cannot proceed because they are both waiting for a resource that is held by the other. Using design time analysis and specify the maximum blocking time for taking mutex can help to prevent deadlock.

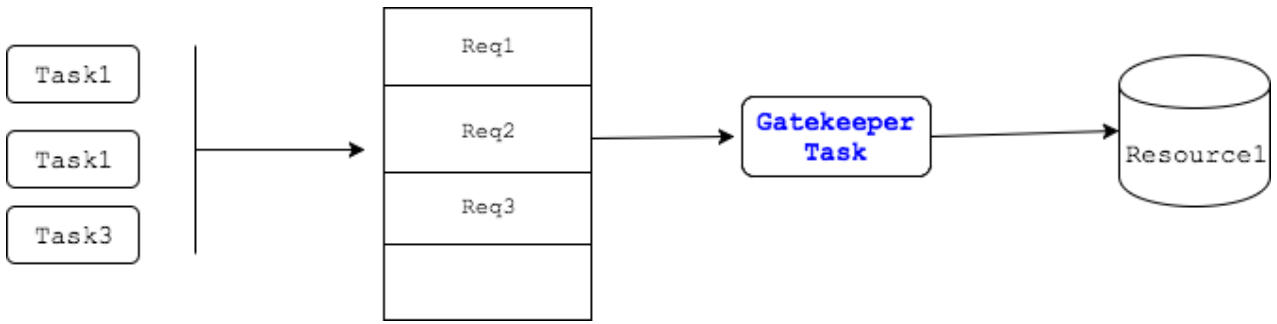
Self-deadlock

Self-deadlock happens if a task attempts to take the same mutex more than once, without first releasing the mutex. Using recursive mutexes can solve this problem. Related APIs are:

- Create semaphores using `xSemaphoreCreateRecursiveMutex()`
- Take semaphores using `xSemaphoreTakeRecursive()`
- Release semaphores using `xSemaphoreGiveRecursive()`

Gatekeeper Task

- A gatekeeper task provides a clean method of implementing mutual exclusion without the risk of priority inversion or deadlock.
- A gatekeeper task is a task that has sole ownership of a resource. Only the gatekeeper task is allowed to access the resource directly. Any other task needing to access the resource can do so only indirectly by using the services of the gatekeeper.
- A gatekeeper task gets requests using a queue and serializes requests for using the related resource.



References

Richard Barry. Mastering the FreeRTOS Real Time Kernel. FreeRTOS.org, 2016

Question?