Contents

- Resource Management

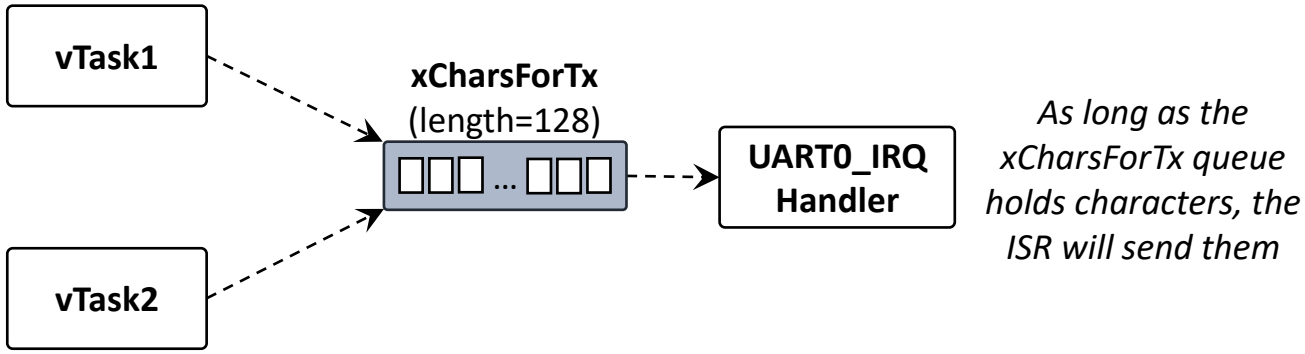
HAN Embedded Systems Engineering – MIC5

Resource Management

Barry, R. (2016). *Mastering the freertos real time kernel. Pre-release 161204 Edition*. Real Time Engineers Ltd.
Chapter 7

Example of Incorrect Resource Management

Two tasks writing characters to the xCharsForTx queue



Example of Incorrect Resource Management

Two tasks writing characters to the xCharsForTx queue

```
void vTask( void *pvParameters )
{
    // The string to print out is passed in via the parameter. Cast this to a
    // character pointer.
    char *pcTaskName = (char *)pvParameters;

    // As per most tasks, this task is implemented in an infinite loop.
    for( ;; )
    {
        // Print out the name of this task
        vSerialPutString(pcTaskName);

        // Print out the number of this task several times
        for(uint32_t i = 0; i<15; i++)
        {
            // Write the task number to the xCharsForTx
            // queue
            xSerialPutChar(pcTaskName[5], 0);
        }

        // Terminate the string
        vSerialPutString("\r\n");
    }
}
```

Expected serial output:

Task 1111111111111111
Task 2222222222222222
Task 1111111111111111
Task 2222222222222222
Task 1111111111111111
Etc.

Example of Incorrect Resource Management

Two tasks writing characters to the xCharsForTx queue

```
void vTask( void *pvParameters )
{
    // The string to print out is passed in via the parameter. Cast this to a
    // character pointer.
    char *pcTaskName = (char *)pvParameters;

    // As per most tasks, this task is implemented in an infinite loop.
    for( ;; )
    {
        // Print out the name of this task
        vSerialPutString(pcTaskName);

        // Print out the number of this task several times
        for(uint32_t i = 0; i<15; i++)
        {
            // Write the task number to the xCharsForTx
            // queue
            xSerialPutChar(pcTaskName[5], 0);
        }

        // Terminate the string
        vSerialPutString("\r\n");
    }
}
```

Actual serial output:

Task 2222222222Task 1111111111111111
Task 22221111111111111111
Task 11111111
Task 2222222222222222
Task 2222222211111111
Etc.

Example of Incorrect Resource Management

Two tasks writing characters to the xCharsForTx queue

The xCharsForTx queue is not managed properly by using the function xSerialPutChar(). It just writes to the queue as soon as a task is in Running state.

What can we do to manage shared resources in a multitasking system?

More Examples of Incorrect Resource Management

- Read, modify, write operations
- Non-atomic access to variables, for example updating multiple elements of a struct
- Function re-entrancy – is it safe to call a function from more than one task

Mutual Exclusion

Does a task start to access a shared resource?

And

Is that shared resource not **thread-safe**?

Then

Make sure that the same task has exclusive access to the resource until the resource has been returned to a consistent state.

Critical Sections

A critical section is a region of code that must not be interrupted

The crudest way to do this is by using taskENTER_CRITICAL() and taskEXIT_CRITICAL()

```
/* Ensure access to the PORTA register cannot be interrupted by placing it within a
critical section.  Enter the critical section. */
taskENTER_CRITICAL();

/* A switch to another task cannot occur between the call to taskENTER_CRITICAL() and
the call to taskEXIT_CRITICAL().  Interrupts may still execute on FreeRTOS ports that
allow interrupt nesting, but only interrupts whose logical priority is above the
value assigned to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant – and those
interrupts are not permitted to call FreeRTOS API functions. */
PORTA |= 0x01;

/* Access to PORTA has finished, so it is safe to exit the critical section. */
taskEXIT_CRITICAL();
```

Listing 114. Using a critical section to guard access to a register

Critical Sections

A critical section is a region of code that must not be interrupted

The crudest way to do this is by using taskENTER_CRITICAL() and taskEXIT_CRITICAL()

```
void vPrintString( const char *pcString )
{
    /* Write the string to stdout, using a critical section as a crude method of
    mutual exclusion. */
    taskENTER_CRITICAL();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    taskEXIT_CRITICAL();
}
```

Listing 115. A possible implementation of vPrintString()

Critical Sections

This ensures mutual exclusion, because

- It globally disables interrupts
- Pre-emptive context switches can only occur from within the tick handler

Critical sections are

- very fast to enter
- very fast to exit
- always deterministic

This makes their use ideal when the region of code being protected is very short!

Suspending the Scheduler

Suspend, or ‘lock’, the scheduler

This protects the region of code from access by other tasks
This does NOT protect the region of code from access by ISRs

```
void vPrintString( const char *pcString )
{
    /* Write the string to stdout, suspending the scheduler as a method of mutual
    exclusion. */
    vTaskSuspendScheduler();
    {
        printf( "%s", pcString );
        fflush( stdout );
    }
    xTaskResumeScheduler();
}
```

Listing 119. The implementation of vPrintString()

Suspending the Scheduler

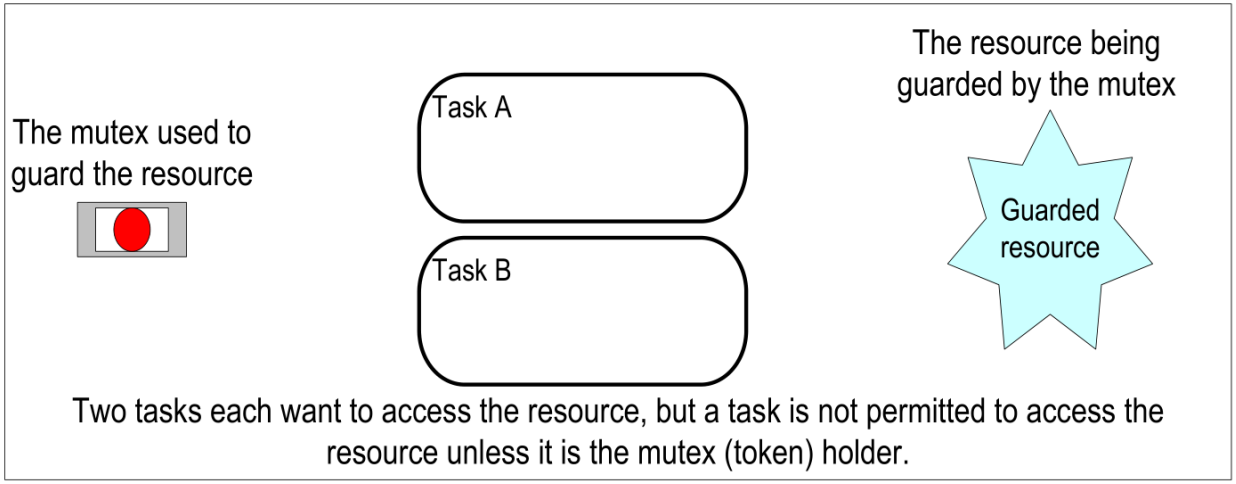
If an interrupt requests a context switch while the scheduler is suspended, then:

- the request is held pending
- The context switch is performed only when the scheduler is resumed (un-suspended).

Mutexes (and Binary Semaphores)

Mutex is an abbreviation for MUTual EXclusion

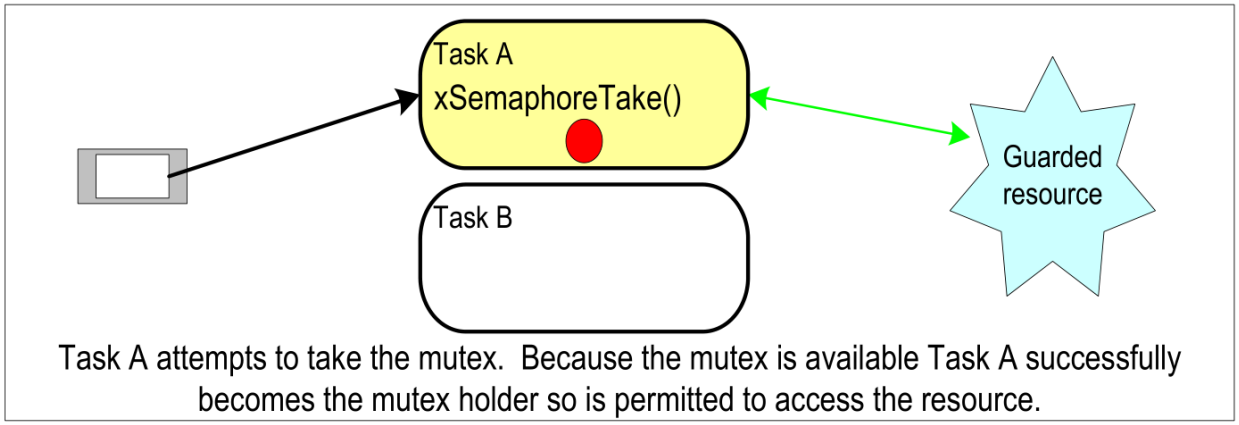
Special type of binary semaphore that can be thought of as a token that is associated with a shared resource



Mutexes (and Binary Semaphores)

Mutex is an abbreviation for MUTual EXclusion

Special type of binary semaphore that can be thought of as a token that is associated with a shared resource

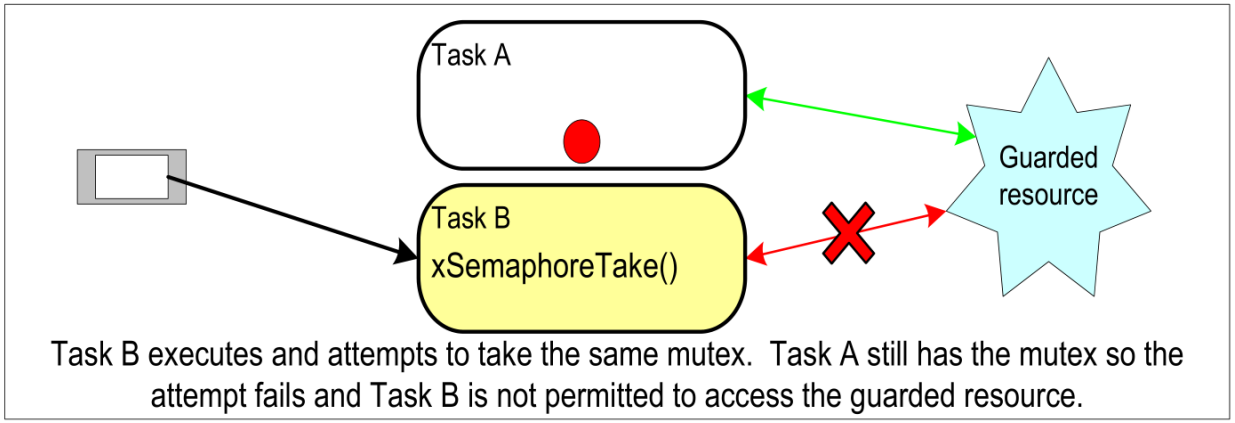


Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.

Mutexes (and Binary Semaphores)

Mutex is an abbreviation for MUTual EXclusion

Special type of binary semaphore that can be thought of as a token that is associated with a shared resource

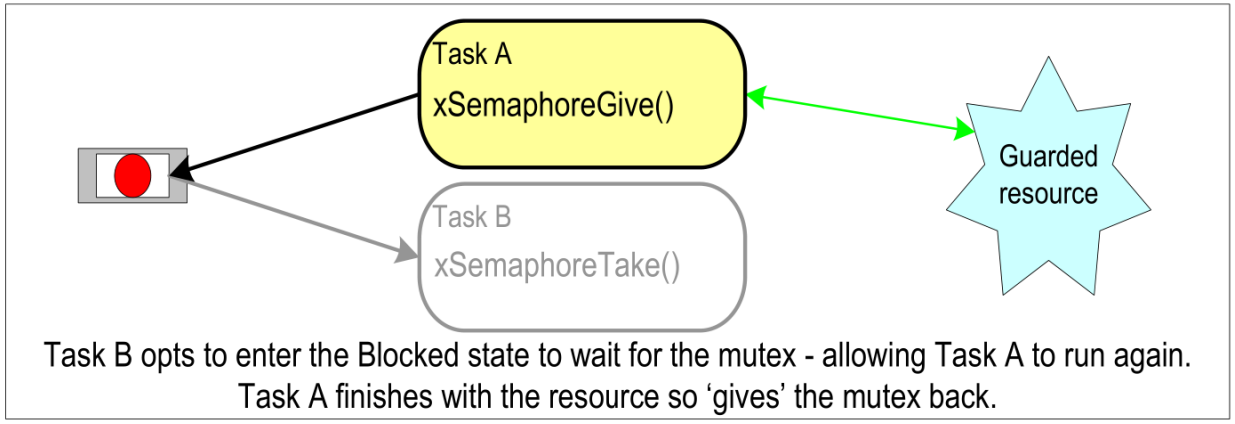


Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.

Mutexes (and Binary Semaphores)

Mutex is an abbreviation for MUTual EXclusion

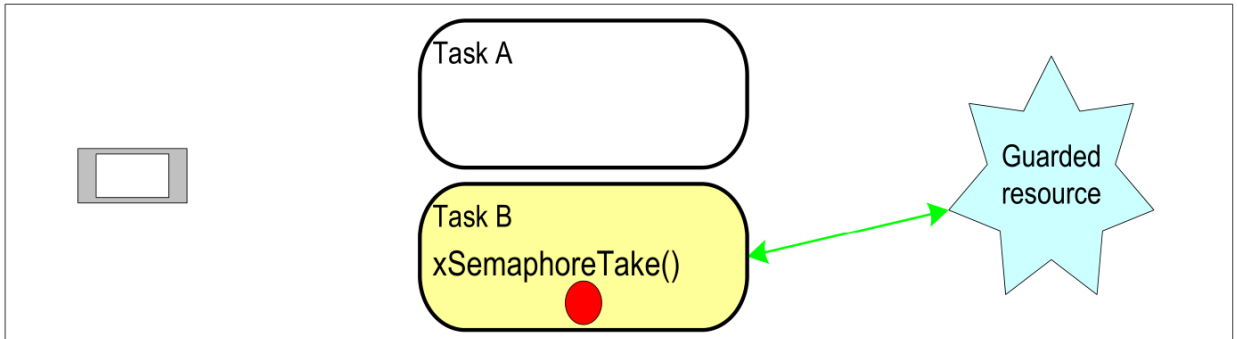
Special type of binary semaphore that can be thought of as a token that is associated with a shared resource



Mutexes (and Binary Semaphores)

Mutex is an abbreviation for MUTual EXclusion

Special type of binary semaphore that can be thought of as a token that is associated with a shared resource

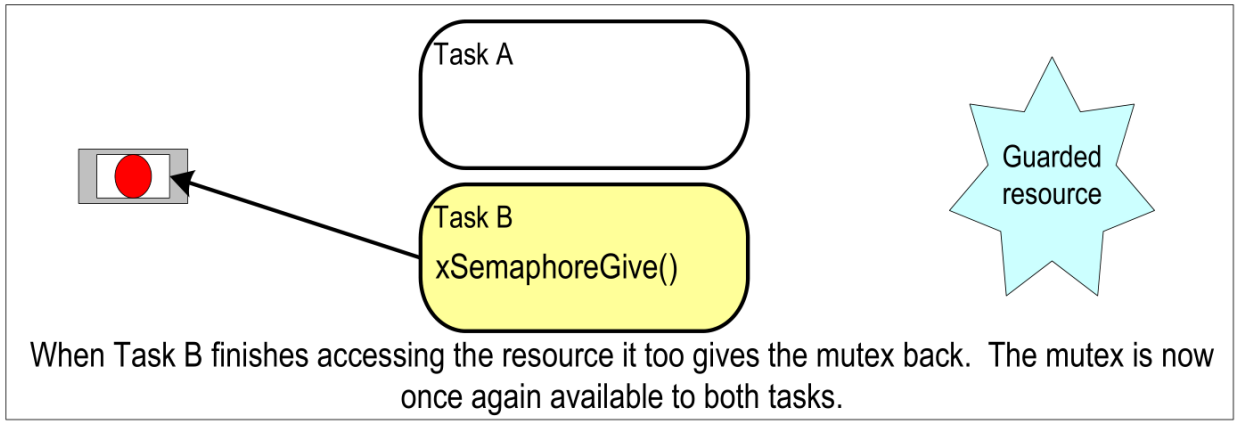


Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.

Mutexes (and Binary Semaphores)

Mutex is an abbreviation for MUTual EXclusion

Special type of binary semaphore that can be thought of as a token that is associated with a shared resource



Mutexes (and Binary Semaphores)

Mutex is an abbreviation for MUTual EXclusion

Special type of binary semaphore that can be thought of as a token that is associated with a shared resource

A mutex is a type of semaphore.

- A mutex must always be returned
- A binary semaphore, when used for synchronization, is normally discarded

The mutex mechanism purely works through the discipline of the application writer

Mutexes (and Binary Semaphores)

```
static void prvNewPrintString( const char *pcString )
{
    /* The mutex is created before the scheduler is started, so already exists by the
    time this task executes.

    Attempt to take the mutex, blocking indefinitely to wait for the mutex if it is
    not available straight away.  The call to xSemaphoreTake() will only return when
    the mutex has been successfully obtained, so there is no need to check the
    function return value.  If any other delay period was used then the code must
    check that xSemaphoreTake() returns pdTRUE before accessing the shared resource
    (which in this case is standard out).  As noted earlier in this book, indefinite
    time outs are not recommended for production code. */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute once the mutex has been successfully
        obtained.  Standard out can be accessed freely now as only one task can have
        the mutex at any one time. */
        printf( "%s", pcString );
        fflush( stdout );

        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );
}
```

Listing 121. The implementation of prvNewPrintString()

Priority Inversion

Example: an application that creates two instances, called *Task 1 (priority 1)* and *Task 2 (priority 2)* of the following task:

```
static void prvPrintTask( void *pvParameters )
{
char *pcStringToPrint;
const TickType_t xMaxBlockTimeTicks = 0x20;

    /* Two instances of this task are created.  The string printed by the task is
    passed into the task using the task's parameter.  The parameter is cast to the
    required type. */
    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        /* Print out the string using the newly defined function. */
        prvNewPrintString( pcStringToPrint );

        /* Wait a pseudo random time.  Note that rand() is not necessarily reentrant,
        but in this case it does not really matter as the code does not care what
        value is returned.  In a more secure application a version of rand() that is
        known to be reentrant should be used - or calls to rand() should be protected
        using a critical section. */
        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );
    }
}
```

Listing 122. The implementation of prvPrintTask() for Example 20

Priority Inversion

Example: an application that creates two instances, called *Task 1 (priority 1)* and *Task 2 (priority 2)* of the following task:

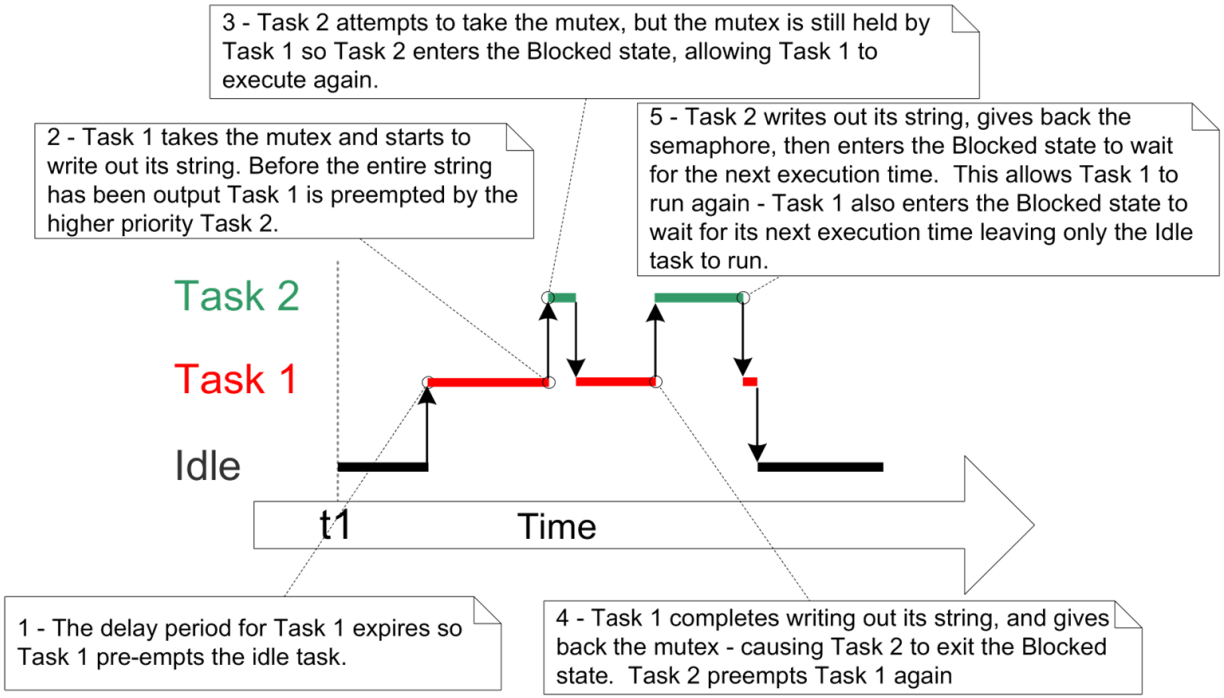


Figure 65. A possible sequence of execution for Example 20

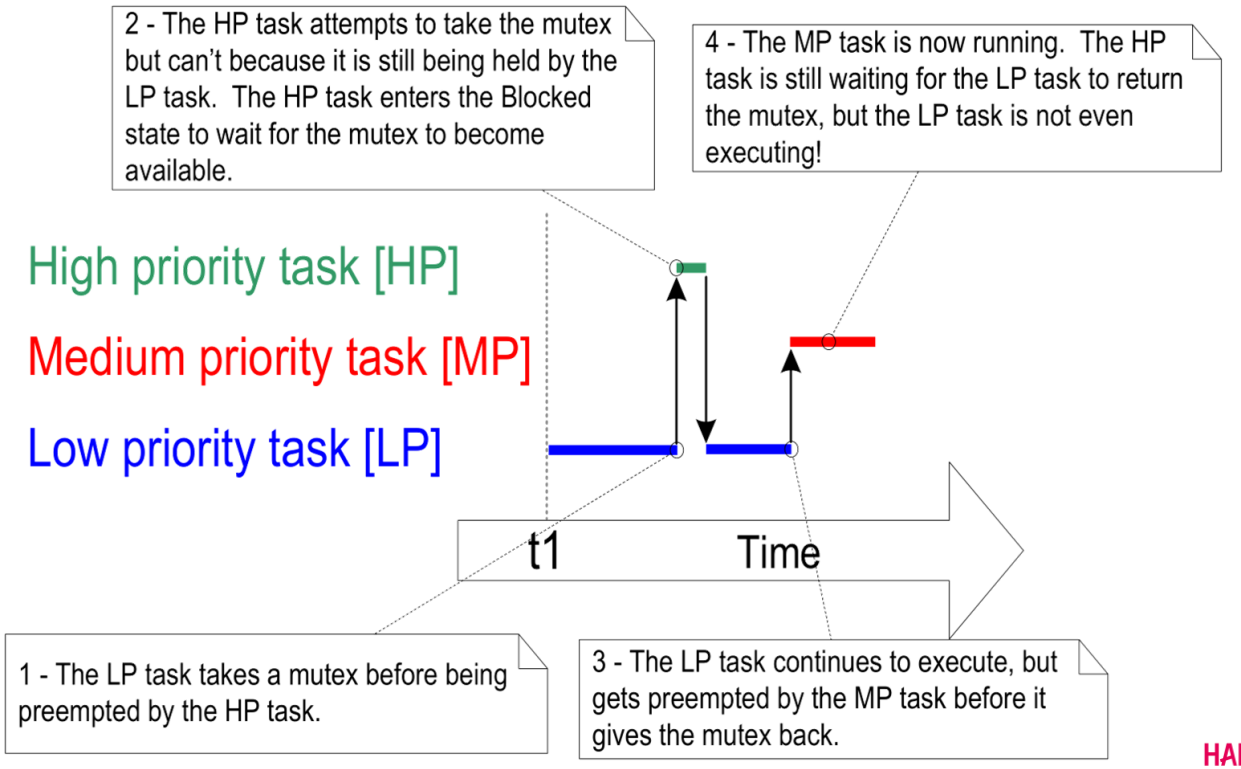
Priority Inversion

The higher priority Task 2 has to wait for the lower priority Task 1 to finish

A higher priority task being delayed by a lower priority task in this manner is called **priority inversion**.

Priority Inversion

This can be a significant problem, for example in the following theoretical worst case scenario:



Priority Inheritance

FreeRTOS mutexes include a **priority inheritance** mechanism

Is a scheme to minimize the negative effect of priority inversion by ensuring it is **time bounded**

Priority Inheritance

Priority inheritance works by temporarily raising the priority of the mutex holder to the priority of the highest priority task that is attempting to obtain the same mutex

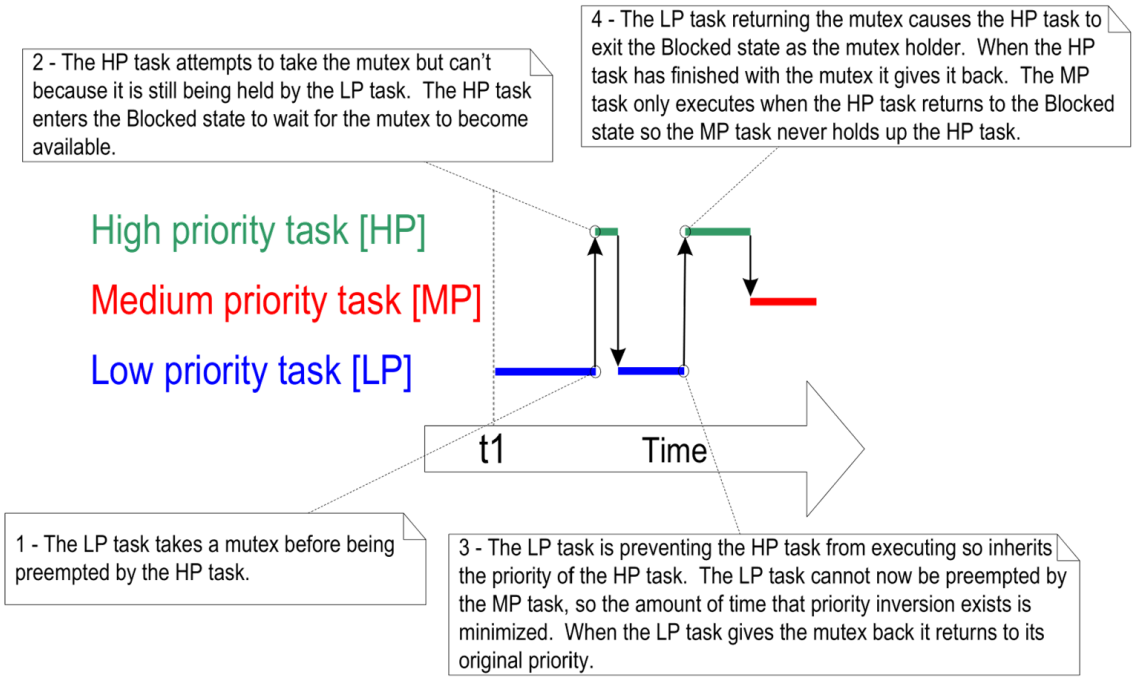


Figure 67. Priority inheritance minimizing the effect of priority inversion

Priority Inheritance

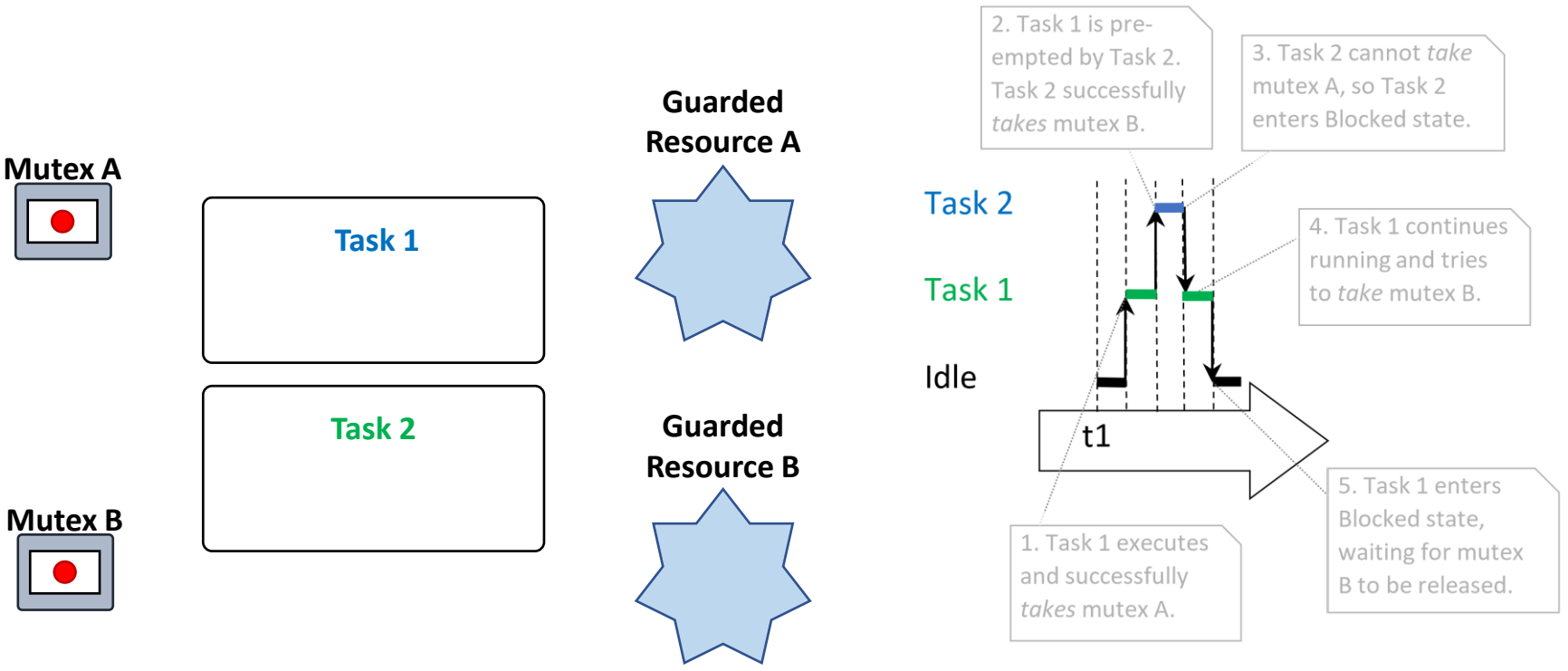
Disadvantages

Does not ‘fix’ priority inversion, but merely lessens its impact by ensuring that the inversion is always time bounded

Complicates system timing analysis

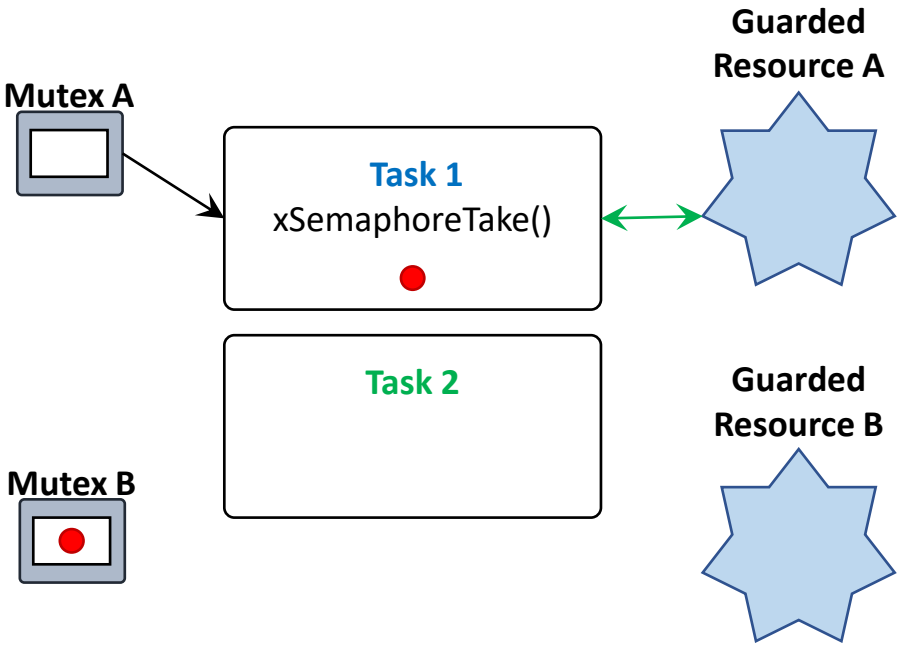
Deadlock

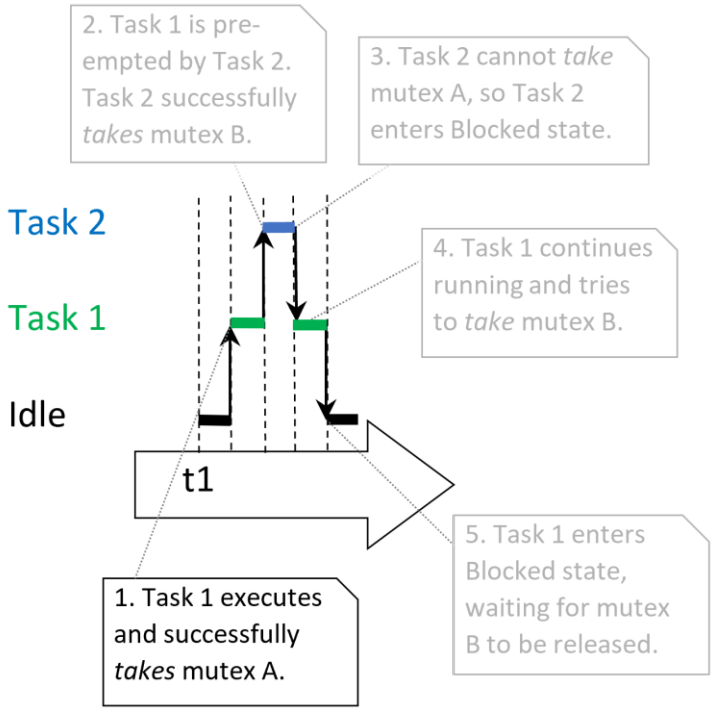
Occurs when two tasks cannot proceed, because they are both waiting for a resource held by the other



Deadlock

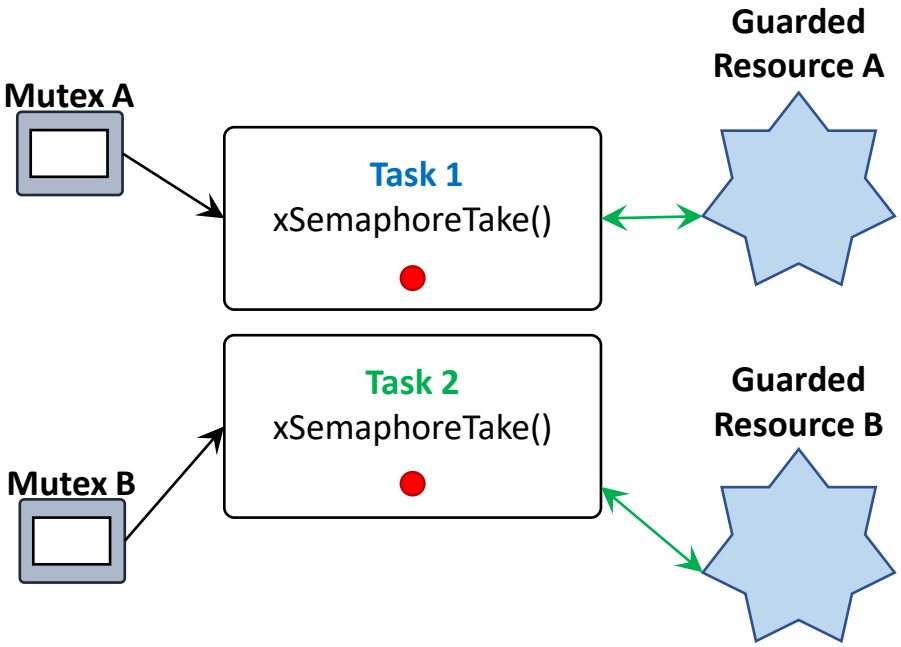
Occurs when two tasks cannot proceed, because they are both waiting for a resource held by the other

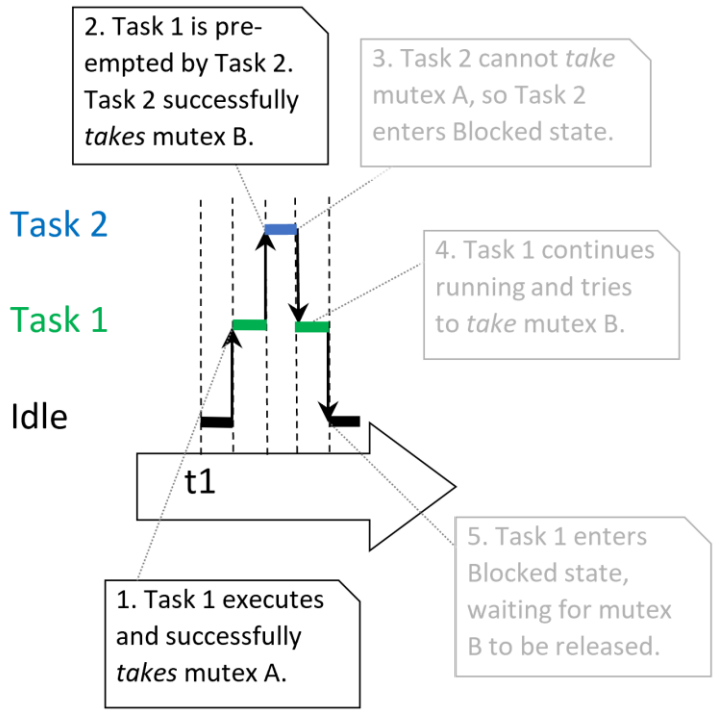




Deadlock

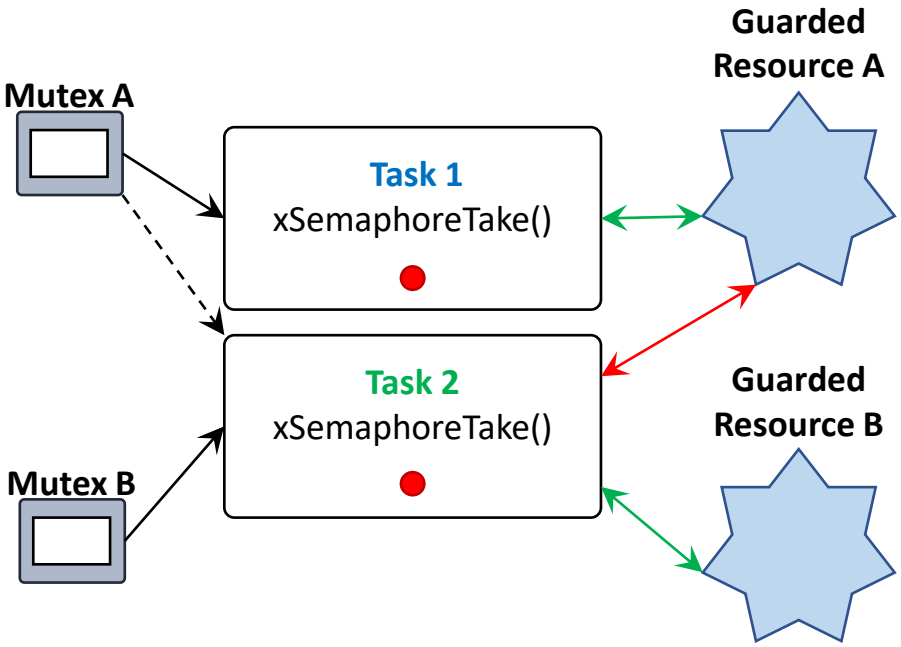
Occurs when two tasks cannot proceed, because they are both waiting for a resource held by the other

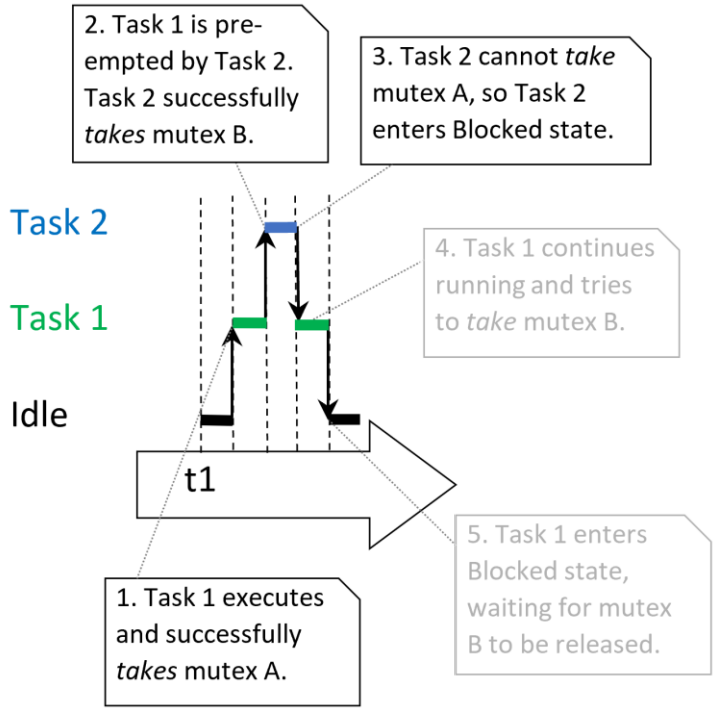




Deadlock

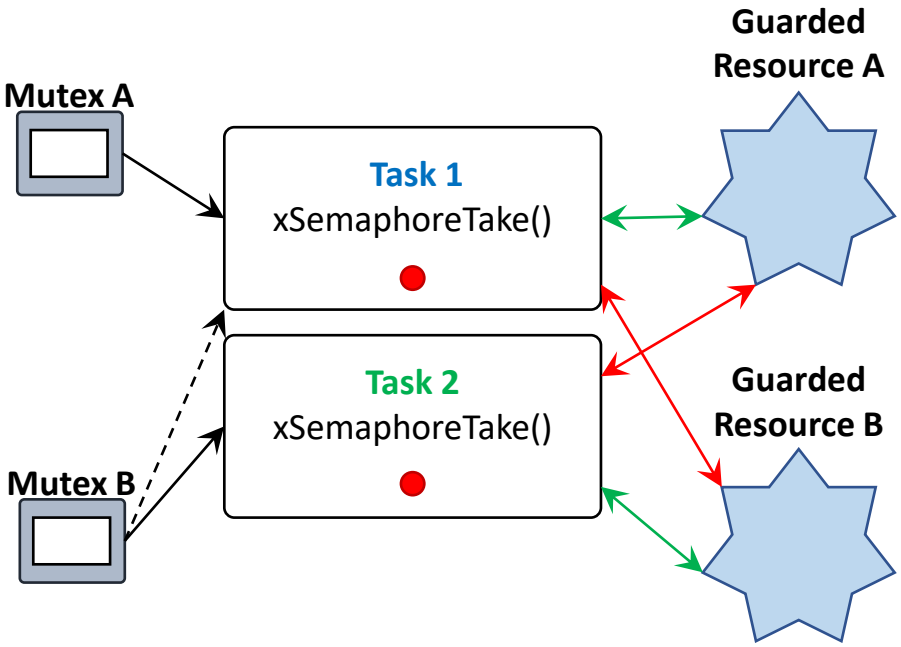
Occurs when two tasks cannot proceed, because they are both waiting for a resource held by the other

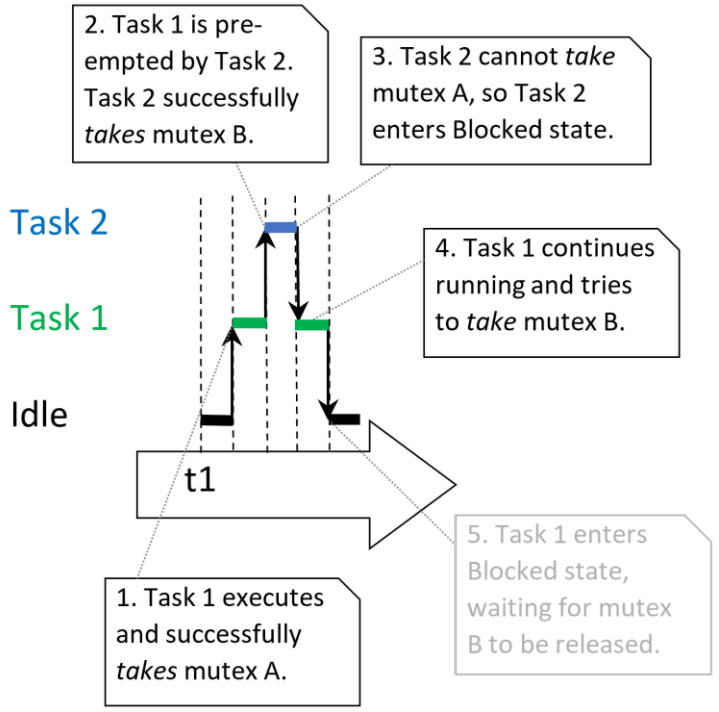




Deadlock

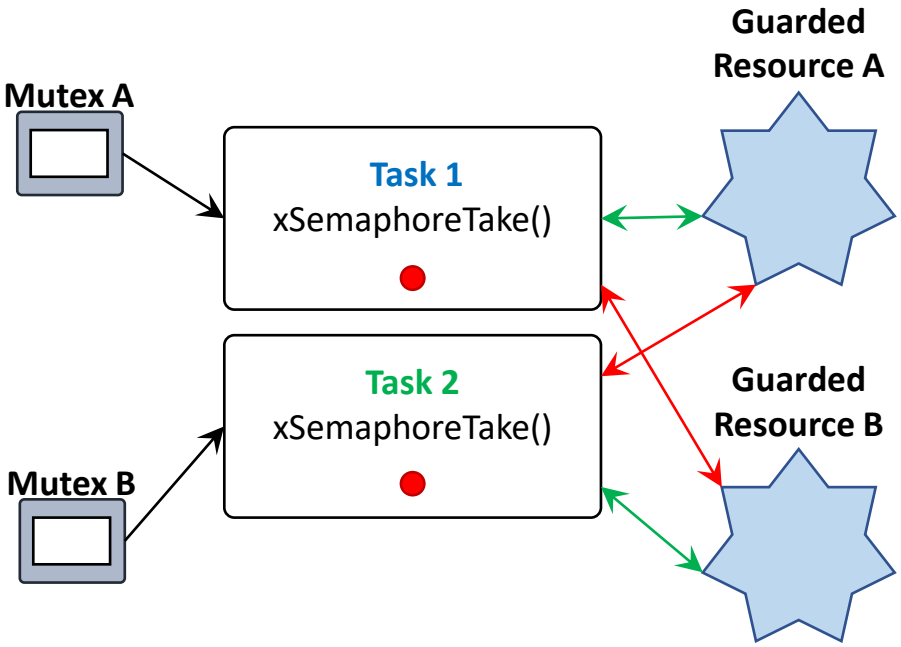
Occurs when two tasks cannot proceed, because they are both waiting for a resource held by the other

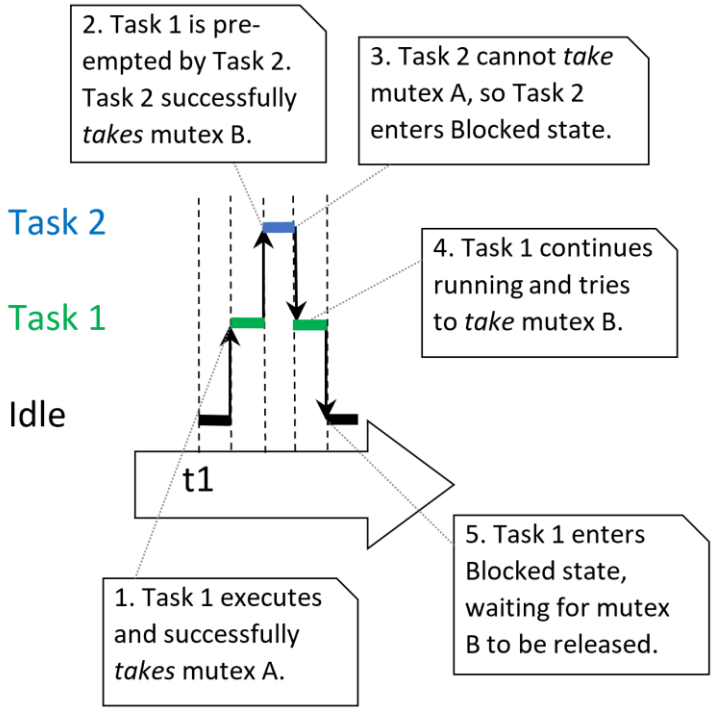




Deadlock

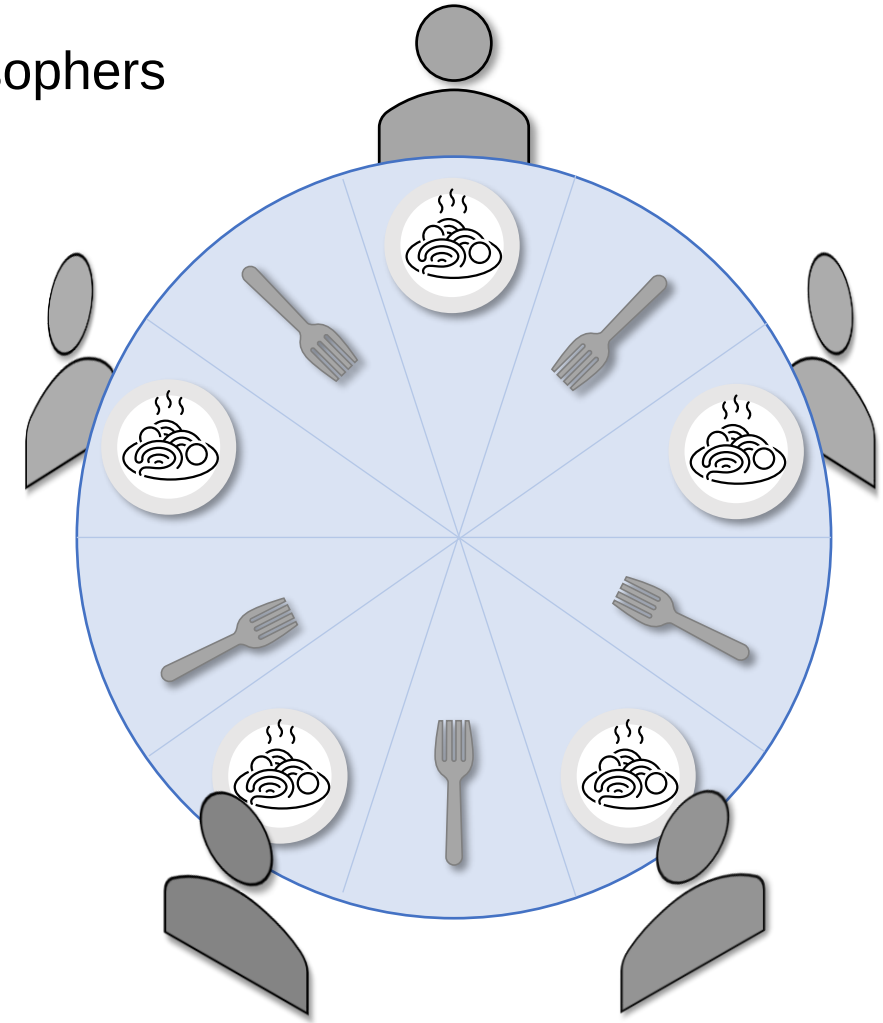
Occurs when two tasks cannot proceed, because they are both waiting for a resource held by the other





Deadlock

Dining philosophers



Deadlock

The best method of avoiding deadlock is to consider its potential at design time

That's why in small embedded systems, deadlock rarely is a problem

It is considered **bad practice** for a task to wait indefinitely to obtain a mutex

Instead, use a time out that is a little longer than the maximum time it is expected to have to wait for the mutex. Then failure to obtain the mutex within that time will be a symptom of a design error

Recursive Mutexes

A task can also deadlock itself if it attempts to take the same mutex more than once

For example:

1. A task successfully *takes* a mutex
2. While holding the mutex, the task calls a library function
3. The library function attempts to *take* the same mutex, but cannot obtain it
4. The library function then enters the Blocked state to wait for the mutex to come available

Recursive Mutexes

This type of deadlock can be avoided by using a **recursive mutex**

- A recursive mutex can be *taken* more than once by the same task
- Will be returned only when the same number of *gives* have been executed

Mutex functions

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

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void );
```

Mutex functions

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

```
BaseType_t xSemaphoreTakeRecursive(SemaphoreHandle_t xSemaphore,
                                   TickType_t xTicksToWait );
```

Mutex functions

```
BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore );
```

```
BaseType_t xSemaphoreGiveRecursive(SemaphoreHandle_t xSemaphore );
```

Mutexes and Task Scheduling

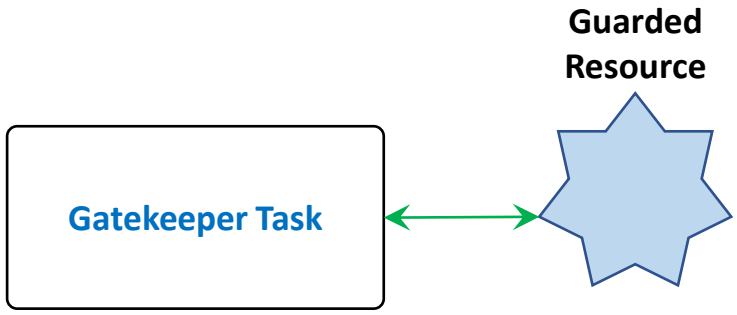
Optional

Gatekeeper Tasks

Gatekeeper tasks provide a clean method of implementing mutual exclusion without the risk of priority inversion or deadlock.

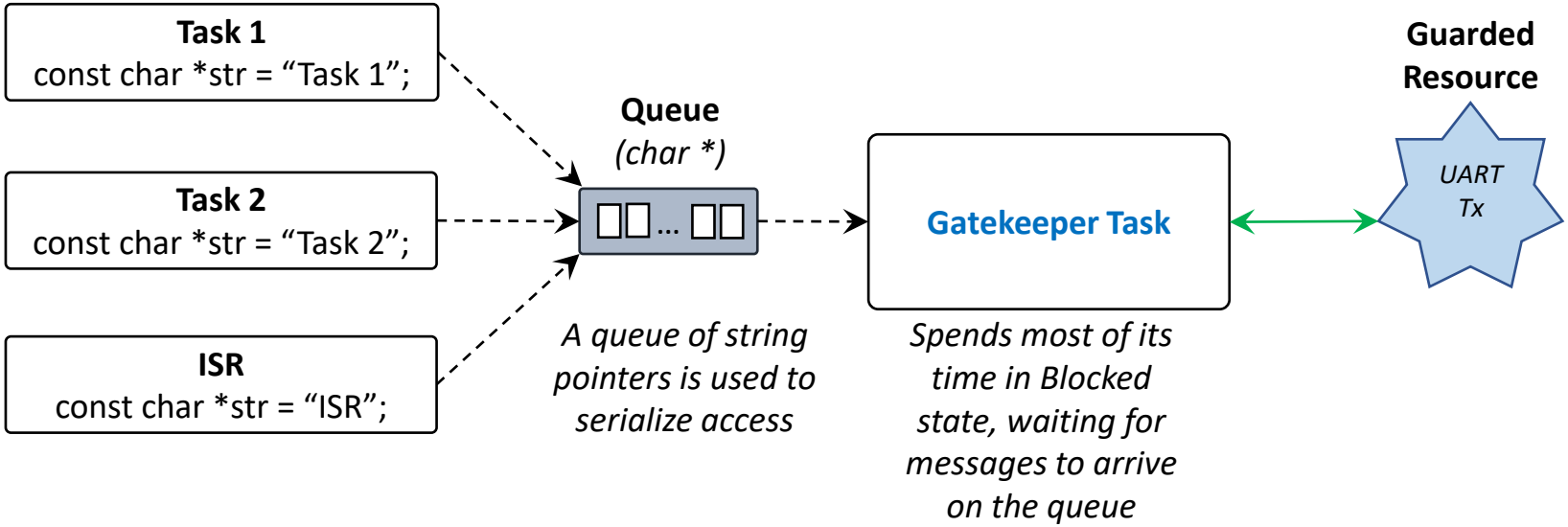
The gatekeeper tasks has sole ownership of a resource.

Other task needing to access the resource can do so only indirectly by using the services of the gatekeeper.



Gatekeeper Tasks

Gatekeeper tasks provide a clean method of implementing mutual exclusion without the risk of priority inversion or deadlock.



Summary

- When and why resource management and control is necessary.
- What a critical section is.
- What mutual exclusion means.
- What it means to suspend the scheduler.
- How to use a mutex.
- What priority inversion is, and how priority inheritance can reduce (but not remove) its impact.
- How to create and use a gatekeeper task.