Real Time Operating System
"FreeRTOS"
Resource Management

Sherif Hammad

Using the FreeRTOS Real Time Kernel - a Practical Guide - Cortex M3 Edition (FreeRTOS Tutorial Books) by Richard Barry

Agenda

- 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.
- What is DeadLock?

Why Resource Management

- **In a multitasking system, there is potential for conflict if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state.**

- "Resource Data" could be corrupted in the following cases:
  - **Accessing Peripherals:** Consider the following scenario where two tasks attempt to write to an LCD:
    - Task A executes and starts to write the string "Hello world" to the LCD.
    - Task A is pre-empted by Task B after outputting just the beginning of the string—"Hello w".
    - Task B writes "Abort, Retry, Fail?" to the LCD before entering the Blocked state.
    - Task A continues from the point at which it was pre-empted and completes outputting the remaining characters—"orld".
    - The LCD now displays the corrupted string "Hello wAbort, Retry, Fail?orld".

Why Resource Management

```
/* The C code being compiled. */
GlobalVar |= 0x01;

/* The assembly code produced. */
LDR       r4,[pc,#284]
LDR       r0,[r4,#0x08] /* Load the value of GlobalVar into r0. */
ORR       r0,r0,#0x01   /* Set bit 0 of r0. */
STR       r0,[r4,#0x08] /* Write the new r0 value back to GlobalVar. */
```

Listing 59. An example read, modify, write sequence

- Read, Modify, Write Operations: This is a 'non-atomic' operation because it takes more than one instruction to complete and can be interrupted. Consider the following scenario where two tasks attempt to update a variable called GlobalVar:
  - Task A loads the value of GlobalVar into a register—the read portion of the operation.
  - Task A is pre-empted by Task B before it completes the modify and write portions of the same operation (before ORR instruction)
  - Task B updates the value of GlobalVar, then enters the Blocked state.
  - Task A continues from the point at which it was pre-empted. It modifies/corrupts the copy of the GlobalVar value calculated by Task B
- Non-atomic Access to Variables:
  - Updating multiple members of a structure
  - Updating a variable that is larger than the natural word size of the architecture (for example, updating a 64-bit variable on a 32-bit machine)

Why Resource Management

- **Function Reentrancy**
  - A function is reentrant if it is safe to call the function from more than one task, or from both tasks and interrupts.
  - Each task maintains its own stack and its own set of core register values. If a function does not access any data other than data stored on the stack or held in a register, then the function is reentrant.

```
/* A parameter is passed into the function.  This will either be
passed on the stack or in a CPU register.  Either way is safe as
each task maintains its own stack and its own set of register
values.  */
long lAddOneHundered( long lVar1 )
{
/* This function scope variable will also be allocated to the stack
or a register, depending on the compiler and optimization level.  Each
task or interrupt that calls this function will have its own copy
of lVar2.  */
long lVar2;

    lVar2 = lVar1 + 100;

    /* Most likely the return value will be placed in a CPU register,
    although it too could be placed on the stack.  */
    return lVar2;
}
/* In this case lVar1 is a global variable so every task that calls
the function will be accessing the same single copy of the variable.  */
long lVar1;

long lNonsenseFunction( void )
{
/* This variable is static so is not allocated on the stack.  Each task
that calls the function will be accessing the same single copy of the
variable.  */
static long lState = 0;
long lReturn;

    switch( lState )
    {
        case 0 :    lReturn = lVar1 + 10;
                    lState = 1;
                    break;

        case 1 :    lReturn = lVar1 + 20;
                    lState = 0;
                    break;
    }
```

Listing 60. An example of a reentrant function

Listing 61. An example of a function that is not reentrant

Basic Critical Sections

```
/* Ensure access to the GlobalVar variable 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, but only interrupts whose priority is above the
value assigned to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant
- and those interrupts are not permitted to call FreeRTOS API
functions. */
GlobalVar |= 0x01;

/* Access to GlobalVar is complete so the critical section can be exited. */
taskEXIT_CRITICAL();
```

Listing 62. Using a critical section to guard access to a variable

```
void vPrintString( const char *pcString )
{
static char cBuffer[ ioMAX_MSG_LEN ];

    /* Write the string to stdout, using a critical section as a crude method
    of mutual exclusion. */
    taskENTER_CRITICAL();
    {
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );
    }
    taskEXIT_CRITICAL();
}
```

Listing 63. A possible implementation of vPrintString()

Basic Critical Sections

- Critical sections implemented in this way are a very crude method of providing **mutual exclusion**.
- They work by **disabling interrupts** up to the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY.
- Pre-emptive context switches can occur only from within an interrupt.
- As long as interrupts remain disabled, the task that called taskENTER_CRITICAL() is guaranteed to remain in the Running state until the critical section is exited.
- **Critical sections must be kept very short;** otherwise, they will adversely affect interrupt response times.
- Every call to taskENTER_CRITICAL() must be closely paired with a call to taskEXIT_CRITICAL().

Suspending (or Locking) the Scheduler

- Critical sections can also be created by suspending the scheduler.
- Basic critical sections protect a region of code from access by other tasks and by interrupts.
- A critical section implemented by suspending the scheduler protects a region of code only from access by other tasks because interrupts remain enabled.
- FreeRTOS API functions should not be called while the scheduler is suspended.

Suspending (or Locking) the Scheduler

The vTaskSuspendAll() API Function

```
void vTaskSuspendAll( void );
```

Listing 64. The vTaskSuspendAll() API function prototype

The xTaskResumeAll() API Function

```
portBASE_TYPE xTaskResumeAll( void );
```

Listing 65. The xTaskResumeAll() API function prototype

```
void vPrintString( const char *pcString )
{
static char cBuffer[ ioMAX_MSG_LEN ];

    /* Write the string to stdout, suspending the scheduler as a method
    of mutual exclusion. */
    vTaskSuspendScheduler();
    {
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );
    }
    xTaskResumeScheduler();
}
```

Listing 66. The implementation of vPrintString()

Table 19. xTaskResumeAll() return value

Returned Value	Description
Returned value	Context switches that are requested while the scheduler is suspended are held pending and performed only as the scheduler is being resumed. A previously pending context switch being performed before xTaskResumeAll() returns results in the function returning pdTRUE. In all other cases, xTaskResumeAll() returns pdFALSE.

Mutexes (and Binary Semaphores)

- In a mutual exclusion scenario: the mutex can be thought of as a token associated with the shared resource.
- For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder).
- When the token holder has finished with the resource, it must 'give' the token back.
- Only when the token has been returned can another task successfully take the token and then safely access the same shared resource.
- A task is not permitted to access the shared resource unless it holds the token.
- A semaphore that is used for mutual exclusion must always be returned.
- A semaphore that is used for synchronization is normally discarded and not returned.
- There is no reason why a task cannot access the resource at any time, but each task 'agrees' not to do so, unless it is able to become the mutex holder.

Mutexes (and Binary Semaphores)

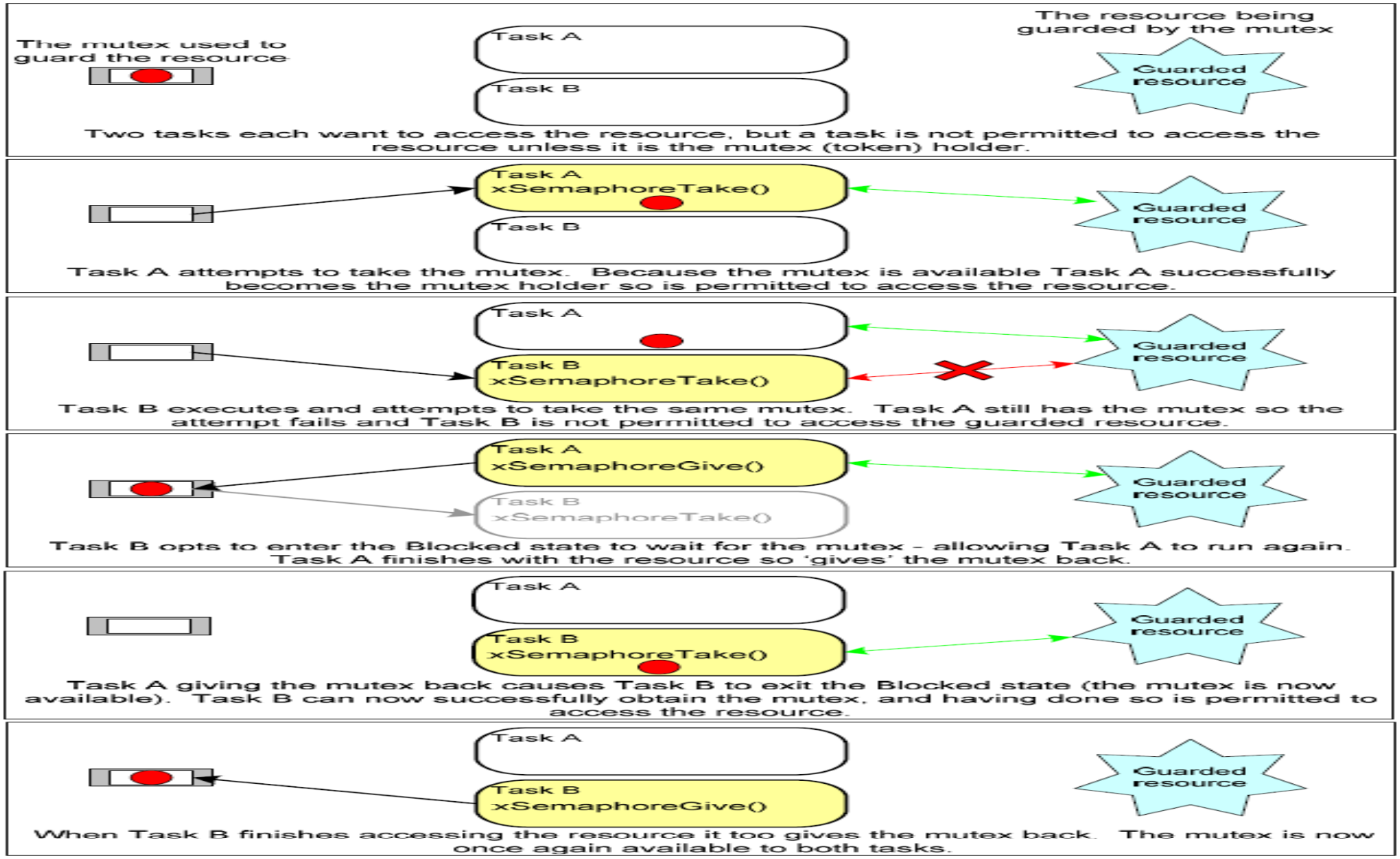


Figure 35. Mutual exclusion implemented using a mutex

Mutexes (and Binary Semaphores)

```
int main( void )
{
    /* Before a semaphore is used it must be explicitly created.  In this example
    a mutex type semaphore is created. */
    xMutex = xSemaphoreCreateMutex();

    /* The tasks are going to use a pseudo random delay, seed the random number
    generator. */
    srand( 567 );

    /* Only create the tasks if the semaphore was created successfully. */
    if( xMutex != NULL )
    {
        /* Create two instances of the tasks that write to stdout.  The string
        they write is passed in as the task parameter.  The tasks are created
        at different priorities so some pre-emption will occur. */
        xTaskCreate( prvPrintTask, "Print1", 240,
                     "Task 1 ****************************************\n", 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 240,
                     "Task 2 ----------------------------------------\n", 2, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 5 provides more information on memory management. */
    for( ;; );
}
```

Mutexes (and Binary Semaphores)

```
static void prvNewPrintString( const char *pcString )
{
static char cBuffer[ mainMAX_MSG_LEN ];

    /* The mutex is created before the scheduler is started so already
    exists by the time this task first 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). */
    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. */
        sprintf( cBuffer, "%s", pcString );
        consoleprint( cBuffer );

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

Listing 68. The implementation of prvNewPrintString()

```
static void prvPrintTask( void *pvParameters )
{
char *pcStringToPrint;

    /* Two instances of this task are created so the string the task will send
    to prvNewPrintString() is passed into the task using the task parameter.
    Cast this 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() & 0x1FF ) );
    }
}
```

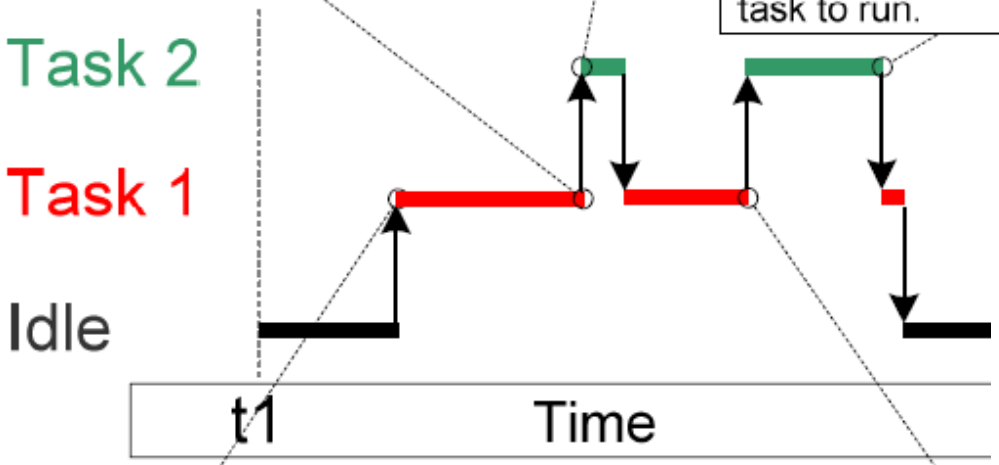
Listing 69. The implementation of prvPrintTask() for Example 15

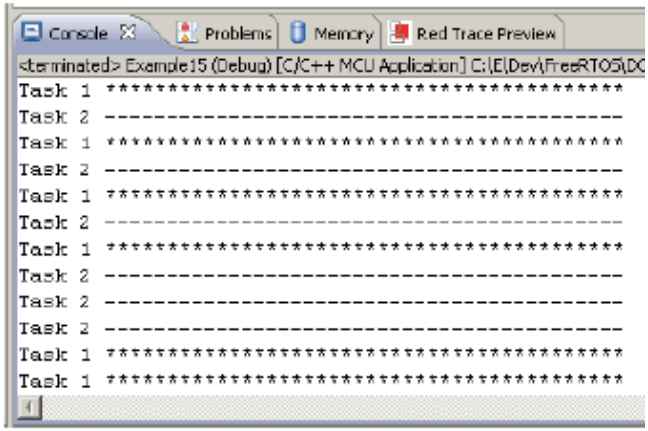
Mutexes (and Binary Semaphores)

3 - Task 2 attempts to take the mutex, but the mutex is still held by Task 1 so Task 2 enters the Blocked state, allowing Task 1 to execute again.

2 - Task 1 takes the mutex and starts to write out its string. Before the entire string has been output Task 1 is preempted by the higher priority Task 2.

5 - Task 2 writes out its string, gives back the semaphore, then enters the Blocked state to wait for the next execution time. This allows Task 1 to run again - Task 1 also enters the Blocked state to wait for its next execution time leaving only the Idle task to run.





1 - The delay period for Task 1 expires so Task 1 pre-empts the idle task.

4 - Task 1 completes writing out its string, and gives back the mutex - causing Task 2 to exit the Blocked state. Task 2 preempts Task 1 again.

Figure 37. A possible sequence of execution for Example 15

Mutexes (and Binary Semaphores)
Priority Inversion

- The higher priority Task 2 having to wait for the lower priority Task 1 to give up control of the mutex.
- A higher priority task being delayed by a lower priority task in this manner is called ‘priority inversion’.
- This undesirable behavior would be exaggerated further if a medium priority task started to execute while the high priority task was waiting for the semaphore—the result would be a high priority task waiting for a low priority task without the low priority task even being able to execute.
- Priority inversion can be a significant problem, but in small embedded systems it can often be avoided at system design time, by considering how resources are accessed.

Mutexes (and Binary Semaphores)

Priority Inversion

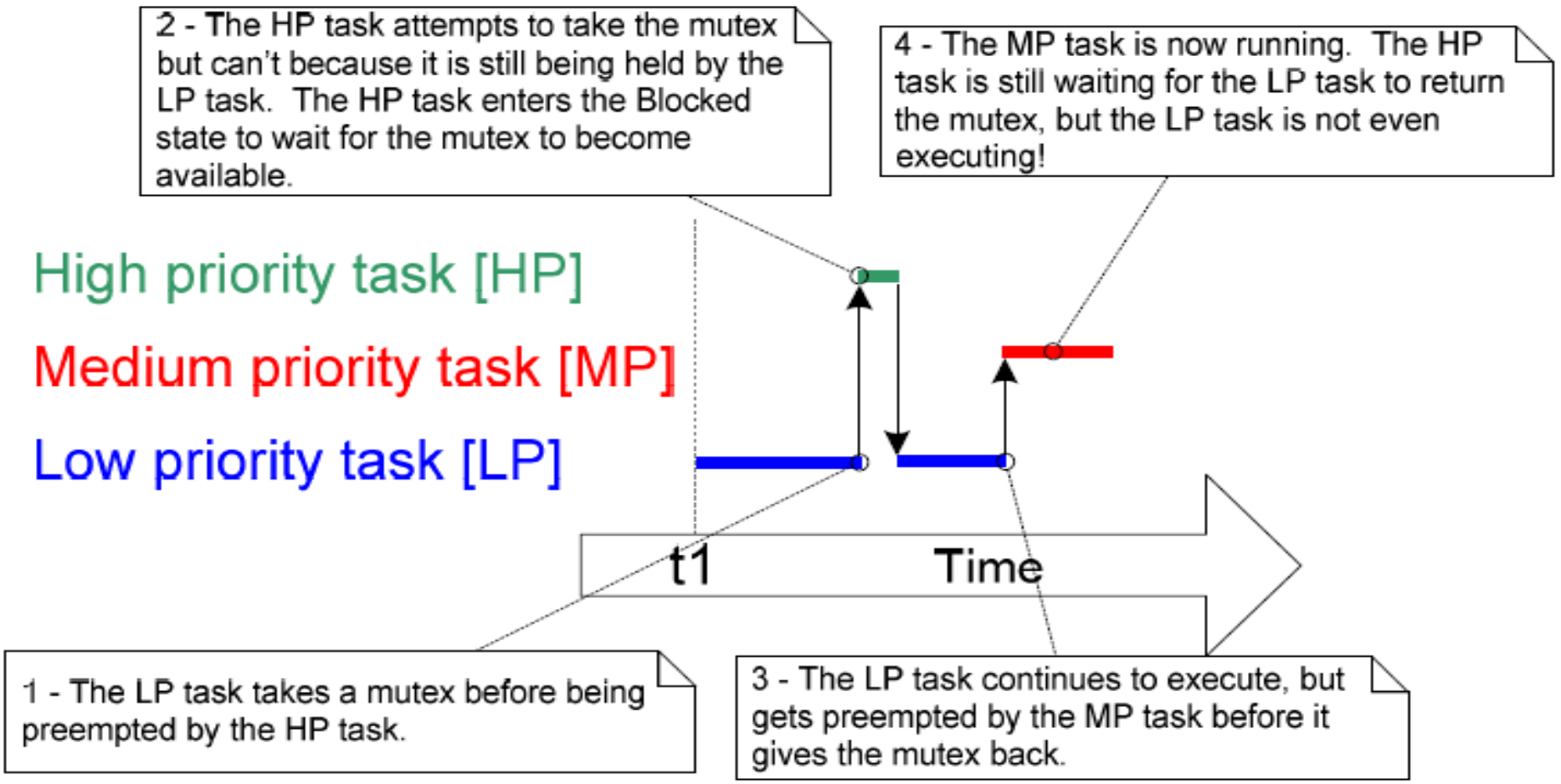


Figure 38. A worst case priority inversion scenario

Mutexes (and Binary Semaphores)
Priority Inheritance

- FreeRTOS mutexes and binary semaphores are very similar “BUT”
- Mutexes include a basic ‘priority inheritance’ mechanism
- Binary semaphores do not.
- Priority inheritance is a scheme that minimizes the negative effects of priority inversion but does not ‘fix’ priority inversion
- Priority inheritance works by temporarily raising the priority of the mutex holder to that of the highest priority task that is attempting to obtain the same mutex.
- The low priority task that holds the mutex ‘inherits’ the priority of the task waiting for the mutex.
- The priority of the mutex holder is reset automatically to its original value when it gives the mutex back.

Mutexes (and Binary Semaphores)
Priority Inheritance

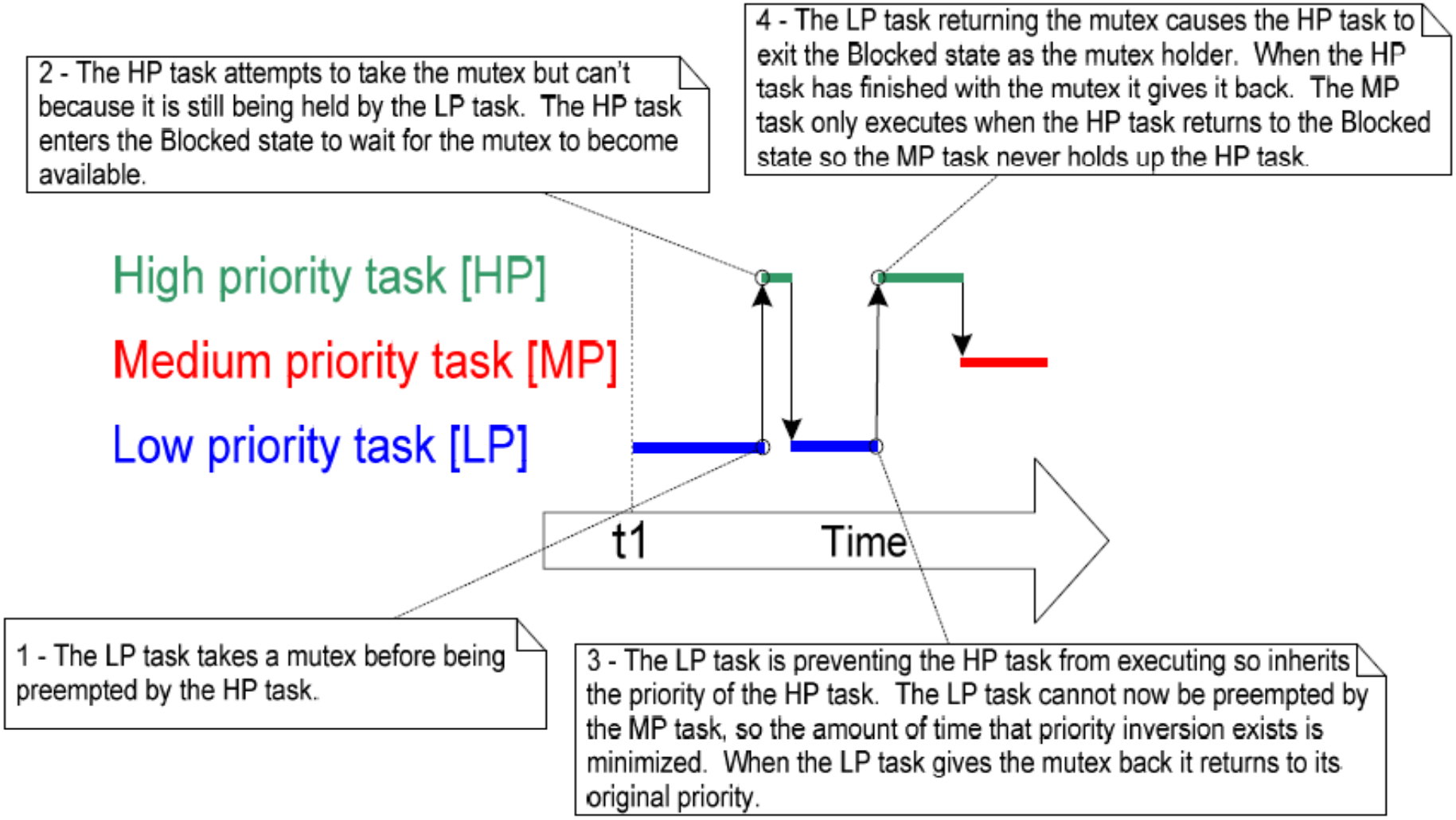


Figure 39. Priority inheritance minimizing the effect of priority inversion

Mutexes (and Binary Semaphores) Deadlock (or Deadly Embrace)

- 'Deadlock' ('deadly embrace') is another potential pitfall that can occur when using mutexes for mutual exclusion.

- Deadlock occurs when two tasks cannot proceed because they are both waiting for a resource that is held by the other.

- Consider the following scenario where Task A and Task B both need to acquire mutex X and mutex Y in order to perform an action:
  1. Task A executes and successfully takes mutex X.
  2. Task A is pre-empted by Task B.
  3. Task B successfully takes mutex Y before attempting to also take mutex X—but mutex X is held by Task A, so is not available to Task B. Task B opts to enter the Blocked state to wait for mutex X to be released.
  4. Task A continues executing. It attempts to take mutex Y—but mutex Y is held by Task B, so is not available to Task A. Task A opts to enter the Blocked state to wait for mutex Y to be released. At the end of this scenario, Task A is waiting for a mutex held by Task B, and Task B is waiting for a mutex held by Task A.