RESOURCE MANAGEMENT

INDEX:

1. Introduction

In a multitasking system there is potential for error if one task starts to access a resource, but does not complete its access before being transitioned out of the Running state. If the task leaves the resource in an inconsistent state, then access to the same resource by any other task or interrupt could result in data corruption, or other similar issue

1.1 Examples for the situations of data corruption

1.1.1 Accessing Peripherals

Consider the following scenario where two tasks attempt to write to an Liquid Crystal Display (LCD).

1. Task A executes and starts to write the string “Hello world” to the LCD.

2. Task A is pre-empted by Task B after outputting just the beginning of the string— “Hello w”.

3. Task B writes “Abort, Retry, Fail?” to the LCD before entering the Blocked state.

4. Task A continues from the point at which it was pre-empted, and completes outputting the remaining characters of its string—“orld”.

The LCD now displays the corrupted string “Hello wAbort, Retry, Fail?orld

1.1.2 Read, Modify, Write Operations

```
/* The C code being compiled. */
PORTA |= 0x01;
/* The assembly code produced when the C code is compiled. */
LOAD R1,[#PORTA]   ; Read a value from PORTA into R1
MOVE R2,#0x01      ; Move the absolute constant 1 into R2
OR R1,R2            ; Bitwise OR R1 (PORTA) with R2 (constant
1)
STORE R1,[#PORTA]  ; Store the new value back to PORTA
```

- It can be seen that the value of PORTA is first read from memory into a register, modified within the register, and then written back to memory. This is called a read, modify, write operation
- 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 memory mapped register called PORTA

1. Task A loads the value of PORTA into a register—the read portion of the operation.
2. Task A is pre-empted by Task B before it completes the modify and write portions of the same operation.
3. Task B updates the value of PORTA, then enters the Blocked state.

4. Task A continues from the point at which it was pre-empted. It modifies the copy of the PORTA value that it already holds in a register, before writing the updated value back to PORTA. When Task A writes to PORTA, it overwrites the modification that has already been performed by Task B, effectively corrupting the PORTA register value.

1.1.3 Non-atomic Access to Variables

- Updating multiple members of a structure, or updating a variable that is larger than the natural word size of the architecture (for example, updating a 32-bit variable on a 16-bit machine), are examples of non-atomic operations. If they are interrupted, they can result in data loss or corruption.

1.1.4 Reentrant Functions

- A reentrant function is one that can be interrupted in the middle of its execution and safely called again ("re-entered") before the previous executions are complete. To achieve this, the function must meet certain conditions:

    1. **No Static or Global Data**: The function should not use or modify any static or global variables. Static and global variables retain their values between function calls, which can lead to conflicts if the function is re-entered.
    2. **No Return from Heap or Static Memory**: The function should not return pointers to static or global variables or to memory that is dynamically allocated (unless managed properly).
    3. **No I/O Operations**: Avoid performing input/output operations that affect shared resources.
    4. **Use of Local Variables**: The function should use only local variables, which are allocated on the stack and are unique to each function call.
    5. **Example of reentrant function**

```
long lAddOneHundred( 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;
return lVar2;
}
```

6. **Example of non-reentrant function**

```
long lVar1;
long lNonsenseFunction( void )
{
/* lState is static, so is not allocated on the
stack. Each task that calls this function will access
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;
   }
   }
```

- **Thread Safety**

A function is thread-safe if it can be safely called by multiple threads at the same time without causing any kind of corruption or unexpected behavior. This typically means that the function:

1. **Does Not Use Shared Resources**: Similar to reentrant functions, thread-safe functions should not use or modify any shared resources unless proper synchronization mechanisms (like mutexes) are used.
2. **Proper Synchronization**: When accessing shared resources, thread-safe functions must use synchronization techniques to ensure that the resource is accessed by only one thread at a time.

- **Reentrancy and Thread Safety Relationship**

All reentrant functions are thread-safe, but not all thread-safe functions are reentrant. The primary difference is that a reentrant function must not use any shared resources

at all (no static/global variables, no I/O), whereas a thread-safe function can use shared resources as long as it uses proper synchronization mechanisms.

1.1.5 Mutual Exclusion

- To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a 'mutual exclusion' technique. The goal is to ensure that, once a task starts to access a shared resource that is not re-entrant and not thread-safe, the same task has exclusive access to the resource until the resource has been returned to a consistent state
- best mutual exclusion method is to (whenever possible, as it is often not practical) design the application in such a way that resources are not shared, and each resource is accessed only from a single task.

1.2 Methods to avoid Data corruption due to pre-emption of tasks

1.2.1 Critical Sections

- Basic critical sections are regions of code that are surrounded by calls to the macros taskENTER_CRITICAL() and taskEXIT_CRITICAL(), respectively. Critical sections are also known as critical regions.
- taskENTER_CRITICAL() and taskEXIT_CRITICAL() do not take any parameters, or return a value

```
taskENTER_CRITICAL();
PORTA |= 0x01;
taskEXIT_CRITICAL();
```

- **Example:**

```
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();

}
```

- They work by disabling interrupts, either completely, or up to the interrupt priority set by configMAX_SYSCALL_INTERRUPT_PRIORITY—depending on the FreeRTOS port being used.
- Pre-emptive context switches can occur only from within an interrupt, so, 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
- Basic 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().
- For this reason, standard out (stdout, or the stream where a computer writes its output data) should not be protected using a critical section because writing to the terminal can be a relatively long operation
- It is safe for critical sections to become nested, because the kernel keeps a count of the nesting depth. The critical section will be exited only when the nesting depth returns to zero— which is when one call to taskEXIT_CRITICAL() has been executed for every preceding call to taskENTER_CRITICAL().
- taskENTER_CRITICAL() and taskEXIT_CRITICAL() do not end in 'FromISR', so must not be called from an interrupt service routine. taskENTER_CRITICAL_FROM_ISR() is an interrupt safe version of taskENTER_CRITICAL(), and taskEXIT_CRITICAL_FROM_ISR() is an interrupt safe version of taskEXIT_CRITICAL().
- taskENTER_CRITICAL_FROM_ISR() returns a value that must be passed into the matching call to taskEXIT_CRITICAL_FROM_ISR()
- **Example**

```
void vAnInterruptServiceRoutine( void )
{
      UBaseType_t uxSavedInterruptStatus;
      uxSavedInterruptStatus=taskENTER_CRITICAL_FROM_ISR();
      taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );
}
```

- It is wasteful to use more processing time executing the code that enters and then subsequently exits a critical section, than executing the code actually being protected

by the critical section. Basic critical sections are very fast to enter, very fast to exit, and always deterministic, making their use ideal when the region of code being protected is very short.

1.2.2 Suspending (or Locking) the Scheduler

- Critical sections can also be created by suspending the scheduler. Suspending the scheduler is sometimes also known as 'locking' 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 only protects a region of code from access by other tasks, because interrupts remain enabled.
- A critical section that is too long to be implemented by simply disabling interrupts can, instead, be implemented by suspending the scheduler. However, interrupt activity while the scheduler is suspended can make resuming (or 'un-suspending') the scheduler a relatively long operation, so consideration must be given to which is the best method to use in each case
- **vTaskSuspendAll() API Function**
  - `void vTaskSuspendAll( void );`
  - The scheduler is suspended by calling vTaskSuspendAll(). Suspending the scheduler prevents a context switch from occurring, but leaves interrupts enabled. If an interrupt requests a context switch while the scheduler is suspended, then the request is held pending, and is performed only when the scheduler is resumed (un-suspended).
  - FreeRTOS API functions must not be called while the scheduler is suspended
- **xTaskResumeAll() API Function**
  - `BaseType_t xTaskResumeAll( void );`
  - The scheduler is resumed (un-suspended) by calling xTaskResumeAll().
  - Returned value : Context switches that are requested while the scheduler is suspended are held pending and performed only as the scheduler is being resumed. If a pending context switch is performed before xTaskResumeAll() returns then pdTRUE is returned. Otherwise pdFALSE is returned

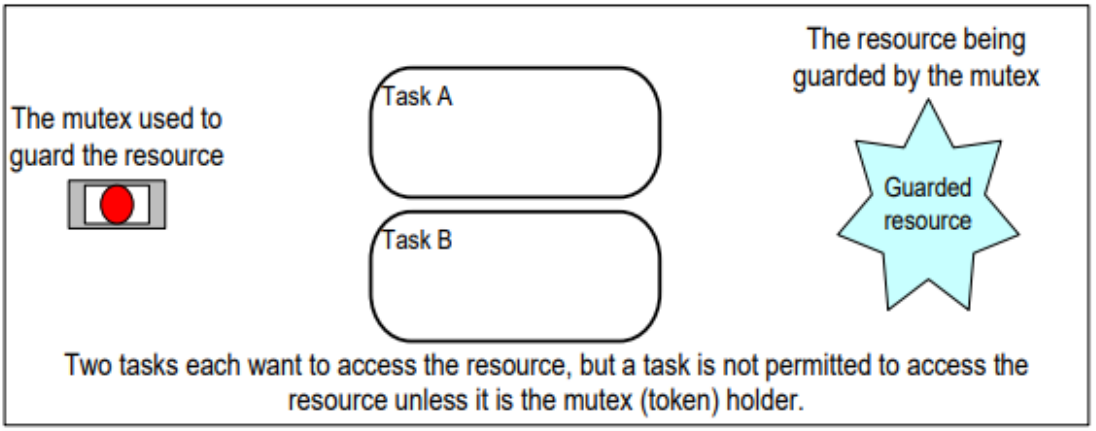
1.2.3 Use of Mutexes(MUTual EXculsion)

- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared. 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

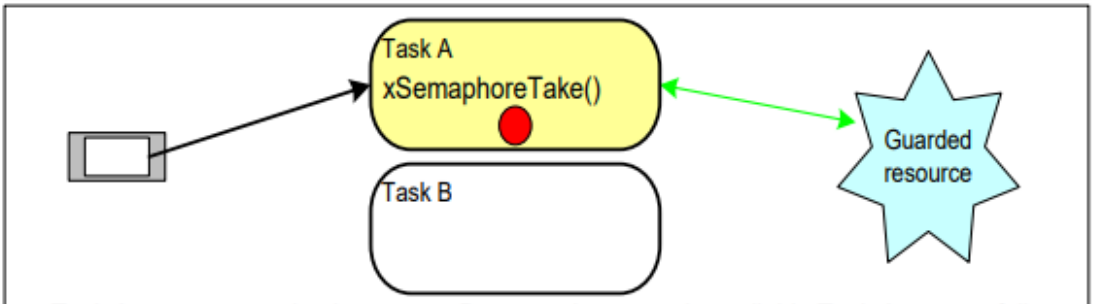
the token has been returned can another task successfully take the token, and then safely access the same shared resource

2. Mutexes (and Binary Semaphores)

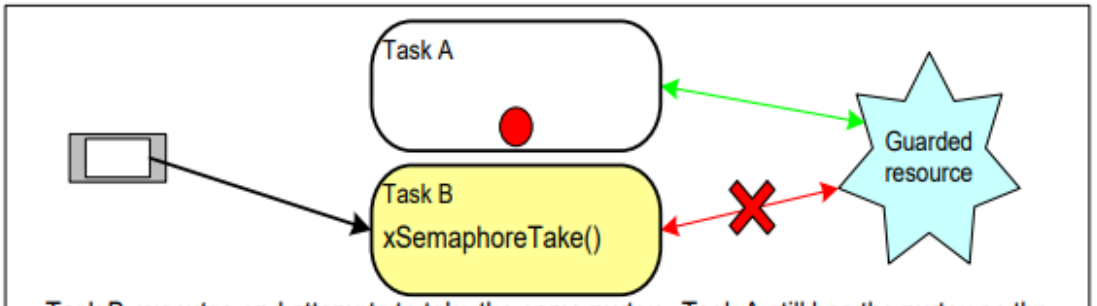
- A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks.
- The word MUTEX originates from ‘MUTual EXclusion’. configUSE_MUTEXES must be set to 1 in FreeRTOSConfig.h for mutexes to be available
- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared. 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
- The primary difference is what happens to the semaphore after it has been obtained:
  - 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.



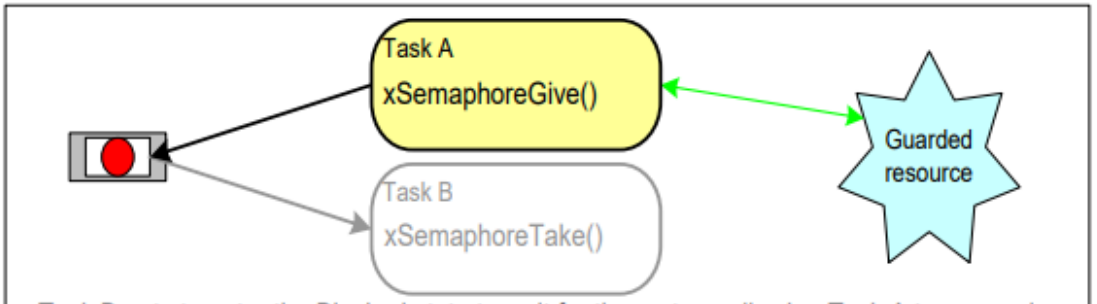
Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder.



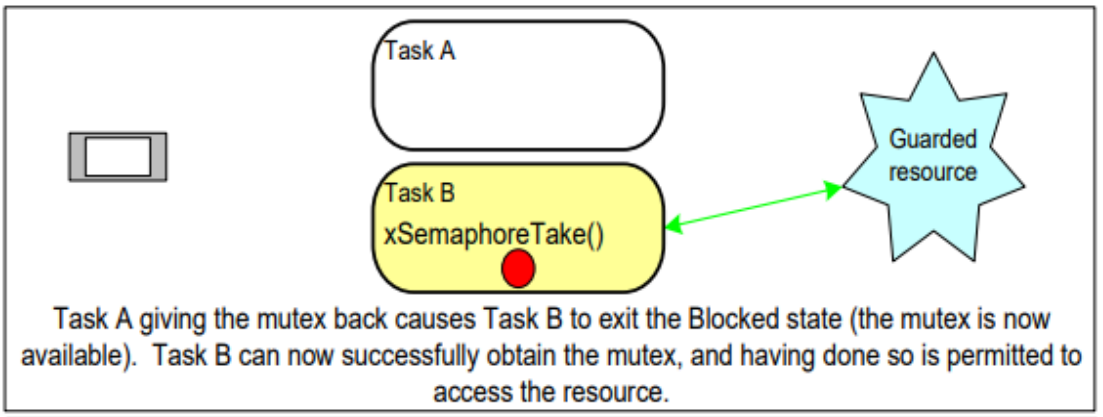
Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.



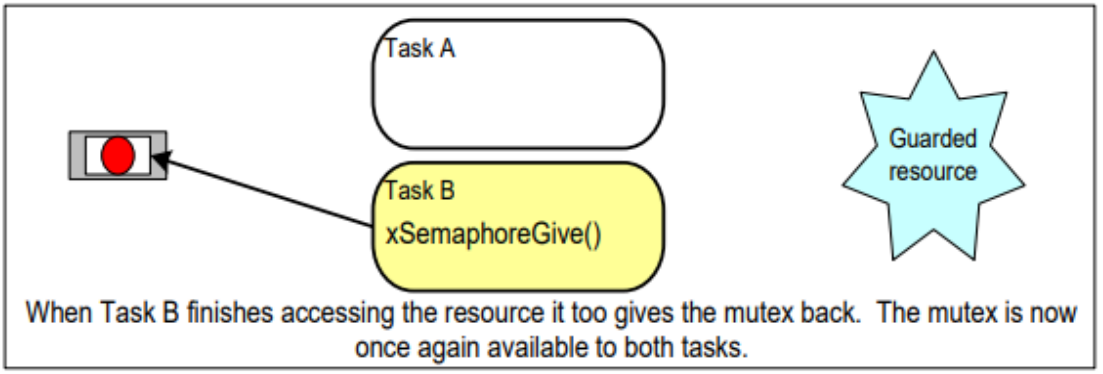
Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.



Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.



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.



When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

3. Mutex APIS

1. **xSemaphoreCreateMutex()**
    - `SemaphoreHandle_t xSemaphoreCreateMutex( void );`
    - A mutex is a type of semaphore. Handles to all the various types of FreeRTOS semaphore are stored in a variable of type SemaphoreHandle_t.
    - Before a mutex can be used, it must be created. To create a mutex type semaphore, use the xSemaphoreCreateMutex() API function.
    - Returned value
        - If NULL is returned then the mutex could not be created because there is insufficient heap memory available for FreeRTOS to allocate the mutex data structures.
        - A non-NULL return value indicates that the mutex has been created successfully. The returned value should be stored as the handle to the created mutex.

2. **xSemaphoreTake()**
    - The function xSemaphoreTake() in FreeRTOS is used to attempt to obtain a semaphore (in this case, xMutex) with blocking behavior, meaning the task will wait indefinitely until the semaphore becomes available
    - This is useful when a task must synchronize its execution with another task or interrupt handler that controls access to a shared resource, like a critical section of code or a hardware peripheral.
    - The semaphore (xMutex) helps manage exclusive access to resources. Tasks that acquire the semaphore can proceed, while others must wait until the semaphore is released (xSemaphoreGive()).
    - `xSemaphoreTake(xMutex, portMAX_DELAY);`
    - Parameters:
        - **Semaphore Handle (xMutex)**: This is the handle to the semaphore you want to take (i.e., obtain ownership of).
        - **Block Time (portMAX_DELAY)**: This parameter determines how long the task will block (wait) if the semaphore is not available immediately. portMAX_DELAY is a constant defined in FreeRTOS headers, typically set to the maximum value of the tick type (TickType_t), which effectively means wait indefinitely until the semaphore becomes available.
    - Return Value:
        - **pdTRUE**: This is the return value of xSemaphoreTake() when the semaphore was successfully obtained (the task now has ownership of the semaphore).
        - **pdFALSE**: This return value indicates that the semaphore could not be obtained immediately within the specified portMAX_DELAY time. However, since portMAX_DELAY is used, the function will block indefinitely until the semaphore is obtained or another event causes the task to unblock.

3. **xSemaphoreGive()**
    - The purpose of xSemaphoreGive(xMutex) is to release ownership of the semaphore xMutex, allowing other tasks that are blocked on this semaphore to proceed. This is typically done at the end of a critical section where a task no longer needs exclusive access to a shared resource.
    - **Parameters:**
        - **Semaphore Handle (**xMutex**)**: This is the handle to the semaphore that you want to give (release). The semaphore must have been previously created using a function like xSemaphoreCreateMutex().
    - **Return Value**: None (void) xSemaphoreGive() does not return a value.

Example:

1. create a new version of vPrintString() called prvNewPrintString(), then calls the new function from multiple tasks. prvNewPrintString() is functionally identical to vPrintString(), but controls access to standard out using a mutex, rather than by locking the scheduler

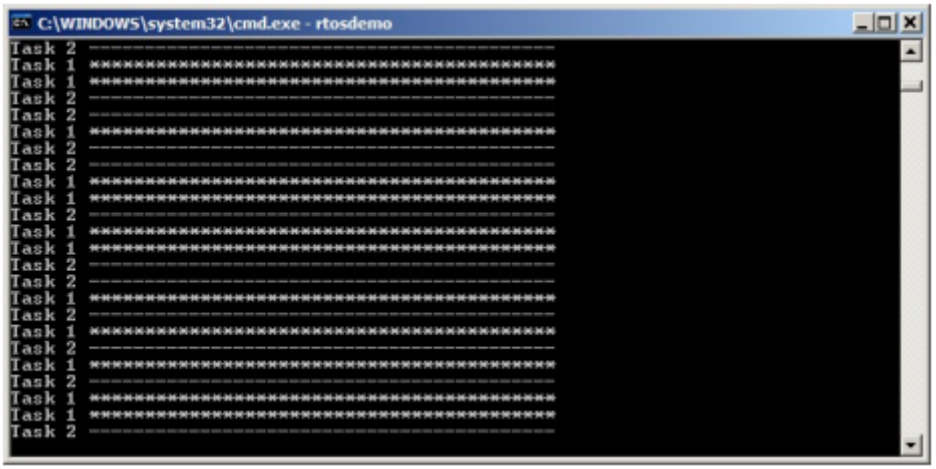
```
static void prvNewPrintString( const char *pcString )
{
    /* The mutex is created before the scheduler is started, so already exists by the
    time this task executes.

    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 );
}
```

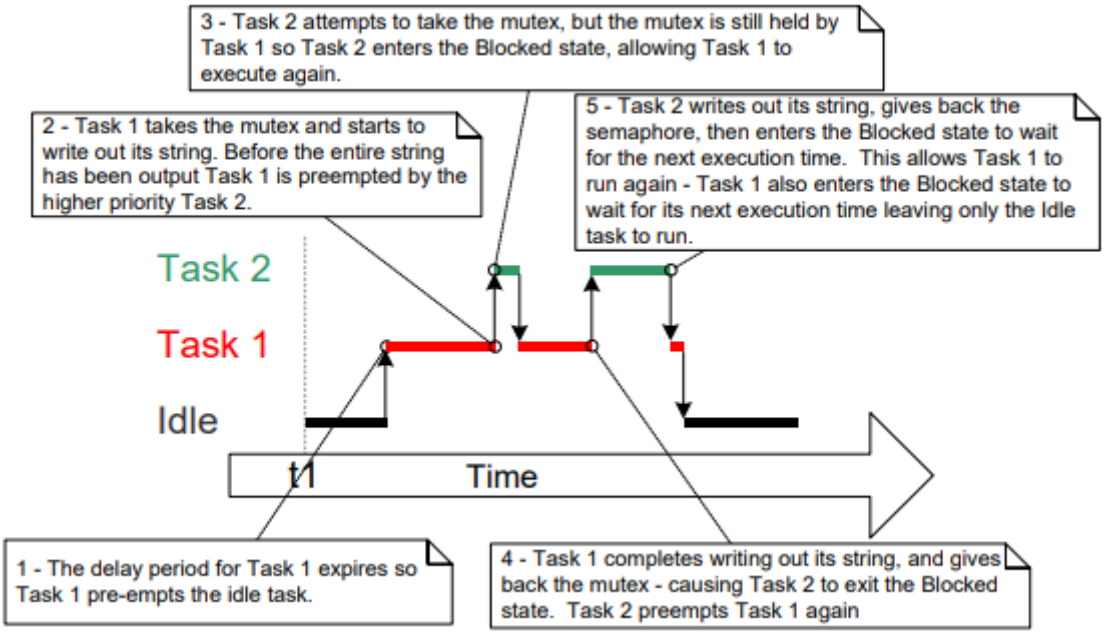
2. prvNewPrintString() is called repeatedly by two instances of a task implemented by prvPrintTask(). A random delay time is used between each call. The task parameter is used to pass a unique string into each instance of the task.

```
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 ) );
    }
}
```

3. main() simply creates the mutex, creates the tasks, then starts the scheduler.
4. The two instances of prvPrintTask() are created at different priorities, so the lower priority task will sometimes be pre-empted by the higher priority task. As a mutex is used to ensure each task gets mutually exclusive access to the terminal, even when pre-emption occurs, the strings that are displayed will be correct and in no way corrupted. The frequency of pre-emption can be increased by reducing the maximum time the tasks spend in the Blocked state, which is set by the xMaxBlockTimeTicks constant.

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

    /* Check the semaphore was created successfully before creating the tasks. */
    if( xMutex != NULL )
    {
        /* Create two instances of the tasks that write to stdout.  The string they
        write is passed in to the task as the task's parameter.  The tasks are
        created at different priorities so some pre-emption will occur. */
        xTaskCreate( prvPrintTask, "Print1", 1000,
                     "Task 1 ****************************************\r\n", 1, NULL );

        xTaskCreate( prvPrintTask, "Print2", 1000,
                     "Task 2 ----------------------------------------\r\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 2
    provides more information on heap memory management. */
    for( ;; );
}
```





4. Priority Inversion

- The sequence of execution depicted shows 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 behaviour 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.

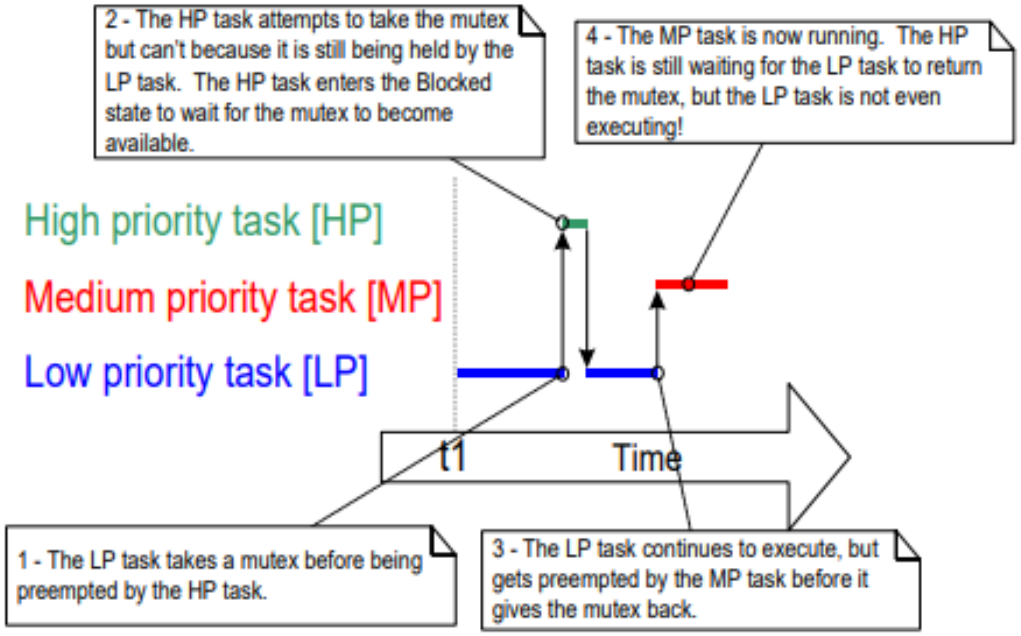


Figure 66. A worst case priority inversion scenario

- 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.

5. Priority Inheritance

- FreeRTOS mutexes and binary semaphores are very similar—the difference being that mutexes include a basic 'priority inheritance' mechanism, whereas binary semaphores do not.
- Priority inheritance is a scheme that minimizes the negative effects of priority inversion.
- It does not 'fix' priority inversion, but merely lessens its impact by ensuring that the inversion is always time bounded.
- However, priority inheritance complicates system timing analysis, and it is not good practice to rely on it for correct system operation
- 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. 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

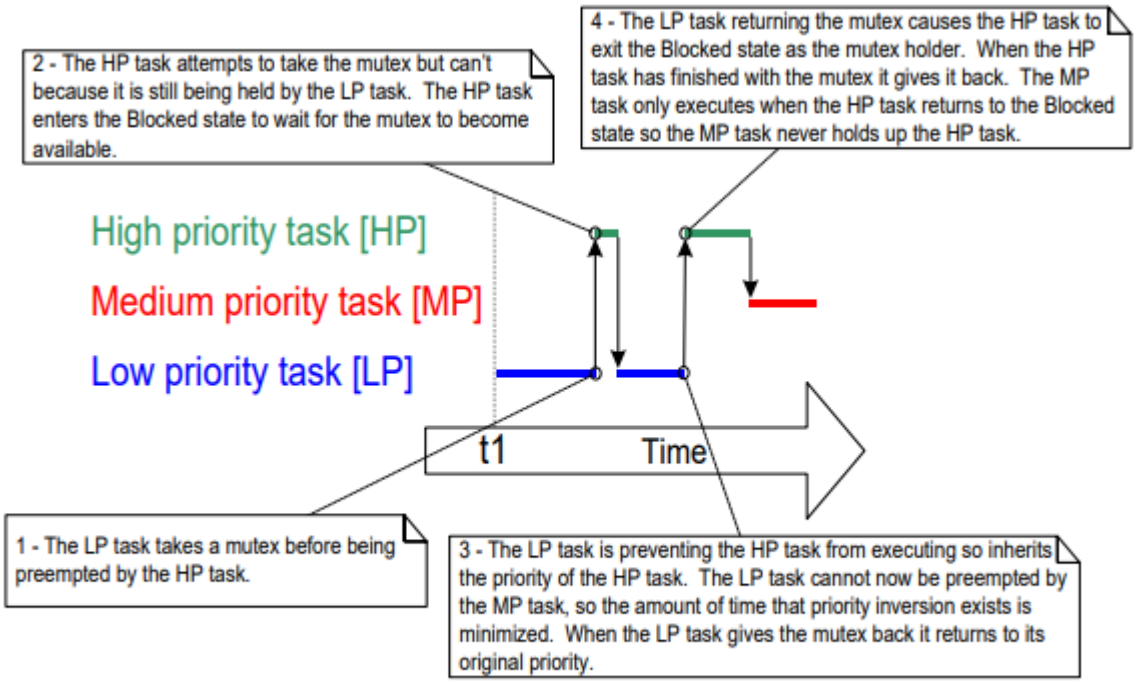


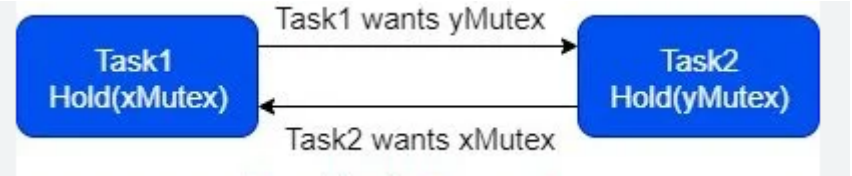
Figure 67. Priority inheritance minimizing the effect of priority inversion

- As just seen, priority inheritance functionality effects the priority of tasks that are using the mutex. For that reason, mutexes must not be used from an interrupt service routines.

6. Deadlock (or Deadly Embrace)

- 'Deadlock' is another potential pitfall of using mutexes for mutual exclusion. Deadlock is sometimes also known by the more dramatic name 'deadly embrace'.
- 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:
    - Task A executes and successfully takes mutex X.
    - Task A is pre-empted by Task B
    - 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.

- 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. Deadlock has occurred because neither task can proceed.



- 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, which might be a deadlock.
- In practice, deadlock is not a big problem in small embedded systems, because the system designers can have a good understanding of the entire application, and so can identify and remove the areas where it could occur

7. Recursive Mutexes

- It is also possible for a task to deadlock with itself. This will happen if a task attempts to take the same mutex more than once, without first returning the mutex
- Consider the following scenario:
    1. A task successfully obtains a mutex.
    2. While holding the mutex, the task calls a library function.
    3. The implementation of the library function attempts to take the same mutex, and enters the Blocked state to wait for the mutex to become available.
- At the end of this scenario the task is in the Blocked state to wait for the mutex to be returned, but the task is already the mutex holder. A deadlock has occurred because the task is in the Blocked state to wait for itself.
- This type of deadlock can be avoided by using a recursive mutex in place of a standard mutex. A recursive mutex can be ‘taken’ more than once by the same task, and will be returned only

after one call to ‘give’ the recursive mutex has been executed for every preceding call to ‘take’ the recursive mutex.

- Standard mutexes and recursive mutexes are created and used in a similar way:
  - Standard mutexes are created using xSemaphoreCreateMutex(). Recursive mutexes are created using xSemaphoreCreateRecursiveMutex(). The two API functions have the same prototype.
  - Standard mutexes are ‘taken’ using xSemaphoreTake(). Recursive mutexes are ‘taken’ using xSemaphoreTakeRecursive(). The two API functions have the same prototype.
  - Standard mutexes are ‘given’ using xSemaphoreGive(). Recursive mutexes are ‘given’ using xSemaphoreGiveRecursive(). The two API functions have the same prototype.

8. Mutexes and Task Scheduling

- If two tasks of different priority use the same mutex, then the FreeRTOS scheduling policy makes the order in which the tasks will execute clear; the highest priority task that is able to run will be selected as the task that enters the Running state. For example, if a high priority task is in the Blocked state to wait for a mutex that is held by a low priority task, then the high priority task will pre-empt the low priority task as soon as the low priority task returns the mutex. The high priority task will then become the mutex holder
- It is however common to make an incorrect assumption as to the order in which the tasks will execute when the tasks have the same priority. If Task 1 and Task 2 have the same priority, and Task 1 is in the Blocked state to wait for a mutex that is held by Task 2, then Task 1 will not pre-empt Task 2 when Task 2 ‘gives’ the mutex. Instead, Task 2 will remain in the Running state, and Task 1 will simply move from the Blocked state to the Ready state.

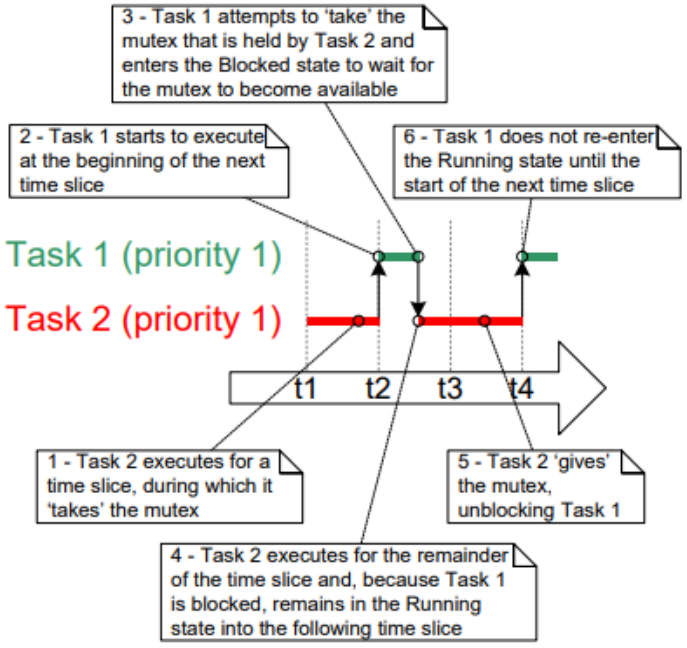


Figure 68 A possible sequence of execution when tasks that have the same priority use the same mutex

- In the scenario shown in Figure 68, the FreeRTOS scheduler does not make Task 1 the Running state task as soon as the mutex is available because:

  1. Task 1 and Task 2 have the same priority, so unless Task 2 enters the Blocked state, a switch to Task 1 should not occur until the next tick interrupt (assuming configUSE_TIME_SLICING is set to 1 in FreeRTOSConfig.h).

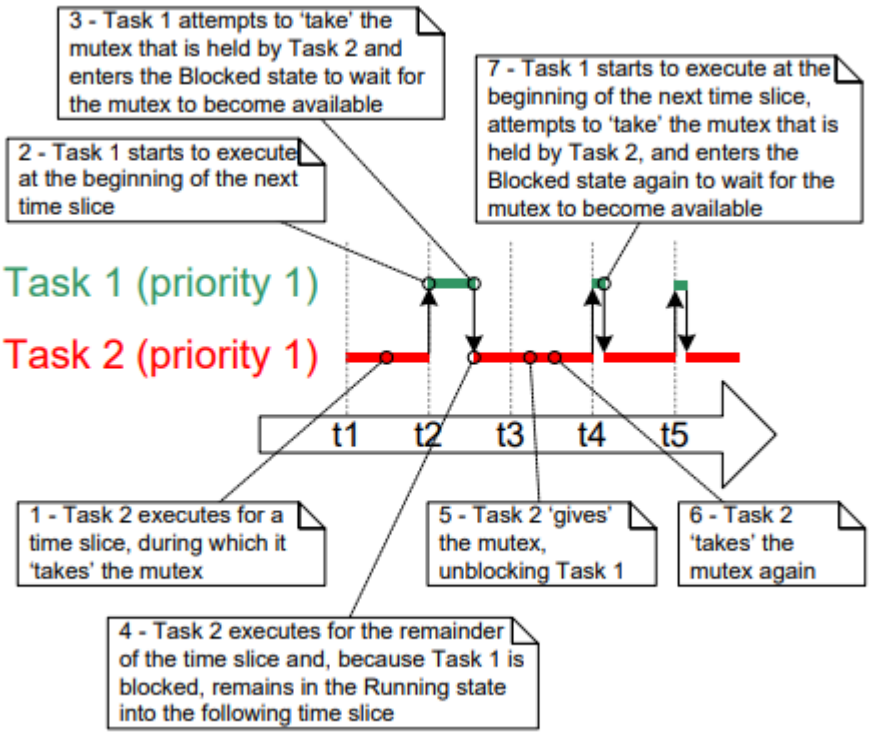
  2. If a task uses a mutex in a tight loop, and a context switch occurred each time the task 'gave' the mutex, then the task would only ever remain in the Running state for a short time. If two or more tasks used the same mutex in a tight loop, then processing time would be wasted by rapidly switching between the tasks.

- If a mutex is used in a tight loop by more than one task, and the tasks that use the mutex have the same priority, then care must be taken to ensure the tasks receive an approximately equal amount of processing time. The reason the tasks might not receive an equal amount of processing time

- **Example:**

```
/* The implementation of a task that uses a mutex in a tight loop.  The task creates
a text string in a local buffer, then writes the string to a display.  Access to the
display is protected by a mutex. */
void vATask( void *pvParameter )
{
extern SemaphoreHandle_t xMutex;
char cTextBuffer[ 128 ];

    for( ;; )
    {
        /* Generate the text string - this is a fast operation. */
        vGenerateTextInALocalBuffer( cTextBuffer );

        /* Obtain the mutex that is protecting access to the display. */
        xSemaphoreTake( xMutex, portMAX_DELAY );

        /* Write the generated text to the display - this is a slow operation. */
        vCopyTextToFrameBuffer( cTextBuffer );

        /* The text has been written to the display, so return the mutex. */
        xSemaphoreGive( xMutex );
    }
}
```

- creating the string is a fast operation, and updating the display is a slow operation. Therefore, as the mutex is held while the display is being updated, the task will hold the mutex for the majority of its run time



- Task 1 re-entering the Blocked state—that happens inside the xSemaphoreTake() API function.
- Task 1 will be prevented from obtaining the mutex until the start of a time slice coincides with one of the short periods during which Task 2 is not the mutex holder.

- The scenario can be avoided by adding a call to taskYIELD() after the call to xSemaphoreGive(), where taskYIELD() is called if the tick count changed while the task held the mutex

```
void vFunction( void *pvParameter )
{
extern SemaphoreHandle_t xMutex;
char cTextBuffer[ 128 ];
TickType_t xTimeAtWhichMutexWasTaken;

    for( ;; )
    {
        /* Generate the text string - this is a fast operation. */
        vGenerateTextInALocalBuffer( cTextBuffer );

        /* Obtain the mutex that is protecting access to the display. */
        xSemaphoreTake( xMutex, portMAX_DELAY );

        /* Record the time at which the mutex was taken. */
        xTimeAtWhichMutexWasTaken = xTaskGetTickCount();

        /* Write the generated text to the display - this is a slow operation. */
        vCopyTextToFrameBuffer( cTextBuffer );

        /* The text has been written to the display, so return the mutex. */
        xSemaphoreGive( xMutex );

        /* If taskYIELD() was called on each iteration then this task would only ever
        remain in the Running state for a short period of time, and processing time
        would be wasted by rapidly switching between tasks.  Therefore, only call
        taskYIELD() if the tick count changed while the mutex was held. */
        if( xTaskGetTickCount() != xTimeAtWhichMutexWasTaken )
        {
            taskYIELD();
        }
    }
}
```

9. Gatekeeper Tasks

- Gatekeeper tasks provide 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.

9.1 Example : Re-writing vPrintString() to use a gatekeeper task

- This time, a gatekeeper task is used to manage access to standard out. When a task wants to write a message to standard out, it does not call a print function directly but, instead, sends the message to the gatekeeper.

- The gatekeeper task uses a FreeRTOS queue to serialize access to standard out. The internal implementation of the task does not have to consider mutual exclusion because it is the only task permitted to access standard out directly.
- The gatekeeper task spends most of its time in the Blocked state, waiting for messages to arrive on the queue. When a message arrives, the gatekeeper simply writes the message to standard out, before returning to the Blocked state to wait for the next message
- Interrupts can send to queues, so interrupt service routines can also safely use the services of the gatekeeper to write messages to the terminal. In this example, a tick hook function is used to write out a message every 200 ticks.
- A tick hook (or tick callback) is a function that is called by the kernel during each tick interrupt. To use a tick hook function:

    1. Set configUSE_TICK_HOOK to 1 in FreeRTOSConfig.h.
    2. Provide the implementation of the hook function, using the exact function name and prototype

```
void vApplicationTickHook( void );
```

- Tick hook functions execute within the context of the tick interrupt, and so must be kept very short, must use only a moderate amount of stack space, and must not call any FreeRTOS API functions that do not end with 'FromISR()'.
- The scheduler will always execute immediately after the tick hook function, so interrupt safe FreeRTOS API functions called from the tick hook do not need to use their pxHigherPriorityTaskWoken parameter, and the parameter can be set to NULL.

Gatekeeper task

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
char *pcMessageToPrint;

    /* This is the only task that is allowed to write to standard out.  Any other
    task wanting to write a string to the output does not access standard out
    directly, but instead sends the string to this task.  As only this task accesses
    standard out there are no mutual exclusion or serialization issues to consider
    within the implementation of the task itself. */
    for( ;; )
    {
        /* Wait for a message to arrive.  An indefinite block time is specified so
        there is no need to check the return value - the function will only return
        when a message has been successfully received. */
        xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

        /* Output the received string. */
        printf( "%s", pcMessageToPrint );
        fflush( stdout );

        /* Loop back to wait for the next message. */
    }
}
```

- Two separate instances of the task are created, and the string the task writes to the queue is passed into the task using the task parameter.

Print task implementation

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

    /* Two instances of this task are created.  The task parameter is used to pass
    an index into an array of strings into the task.  Cast this to the required
    type. */
    iIndexToString = ( int ) pvParameters;

    for( ;; )
    {
        /* Print out the string, not directly, but instead by passing a pointer to
        the string to the gatekeeper task via a queue.  The queue is created before
        the scheduler is started so will already exist by the time this task executes
        for the first time.  A block time is not specified because there should
        always be space in the queue. */
        xQueueSendToBack( xPrintQueue, &( pcStringsToPrint[ iIndexToString ] ), 0 );

        /* 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 ) );
    }
}
```

- The tick hook function counts the number of times it is called, sending its message to the gatekeeper task each time the count reaches 200. For demonstration purposes only, the tick hook writes to the front of the queue, and the tasks write to the back of the queue

Tick hook implementation

```
void vApplicationTickHook( void )
{
static int iCount = 0;

    /* Print out a message every 200 ticks.  The message is not written out directly,
    but sent to the gatekeeper task. */
    iCount++;
    if( iCount >= 200 )
    {
        /* As xQueueSendToFrontFromISR() is being called from the tick hook, it is
        not necessary to use the xHigherPriorityTaskWoken parameter (the third
        parameter), and the parameter is set to NULL. */
        xQueueSendToFrontFromISR( xPrintQueue,
                                  &( pcStringsToPrint[ 2 ] ),
                                  NULL );

        /* Reset the count ready to print out the string again in 200 ticks time. */
        iCount = 0;
    }
}
```

- As normal, main() creates the queues and tasks necessary to run the example, then starts the scheduler

```
/* Define the strings that the tasks and interrupt will print out via the
gatekeeper. */
static char *pcStringsToPrint[] =
{
    "Task 1 ****************************************************\r\n",
    "Task 2 ----------------------------------------------------\r\n",
    "Message printed from the tick hook interrupt ##############\r\n"
};

/*-----------------------------------------------------------*/

/* Declare a variable of type QueueHandle_t.  The queue is used to send messages
from the print tasks and the tick interrupt to the gatekeeper task. */
QueueHandle_t xPrintQueue;

/*-----------------------------------------------------------*/

int main( void )
{
    /* Before a queue is used it must be explicitly created.  The queue is created
    to hold a maximum of 5 character pointers. */
    xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

    /* Check the queue was created successfully. */
    if( xPrintQueue != NULL )
    {
        /* Create two instances of the tasks that send messages to the gatekeeper.
        The index to the string the task uses is passed to the task via the task
        parameter (the 4th parameter to xTaskCreate()).  The tasks are created at
        different priorities so the higher priority task will occasionally preempt
        the lower priority task. */
        xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );
        xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

        /* Create the gatekeeper task.  This is the only task that is permitted
        to directly access standard out. */
        xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, 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 2
    provides more information on heap memory management. */
    for( ;; );
}
```

- The gatekeeper task is assigned a lower priority than the print tasks—so messages sent to the gatekeeper remain in the queue until both print tasks are in the Blocked state. In some situations, it would be appropriate to assign the gatekeeper a higher priority, so messages get processed immediately—but doing so would be at the cost of the gatekeeper delaying lower priority tasks until it has completed accessing the protected resource

10. Example codes

10.1. Use of mutexes

```
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

// Mutex handle
SemaphoreHandle_t xMutex;

void vApplicationIdleHook(void) {
    // Idle hook implementation (optional)
}

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void prvNewPrintString(const char *pcString) {
    // Attempt to take the mutex, blocking indefinitely if it's not available
    if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
            // The following line will only execute once the mutex has been
successfully obtained
        vPrintString("%s", pcString);
        vTaskDelay(1000);
        fflush(stdout); // Ensure output is flushed

        // The mutex MUST be given back!
        xSemaphoreGive(xMutex);
    } else {
        // Failed to take the mutex
        vPrintString("Failed to take the mutex\n");
    }
}

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

    // The string to print is passed as the task's parameter
    pcStringToPrint = (const char *)pvParameters;

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

        // Wait a pseudo-random time
        vTaskDelay((rand() % xMaxBlockTimeTicks));
```

```
    }
}

int main(void) {
    // Initialize the mutex before the scheduler is started
    xMutex = xSemaphoreCreateMutex();

    // Check the mutex was created successfully before creating the tasks
    if (xMutex != NULL) {
        vPrintString("Mutex created successfully\n");

        // Create two instances of the tasks that write to stdout
                  if  (xTaskCreate(prvPrintTask,  "Print1",  130,  "Task  1
**********************************\n", 1, NULL) == pdPASS) {
            vPrintString("Task 1 created successfully\n");
        } else {
            vPrintString("Failed to create Task 1\n");
        }

                  if  (xTaskCreate(prvPrintTask,  "Print2",  130,  "Task  2
----------------------------------\n", 2, NULL) == pdPASS) {
            vPrintString("Task 2 created successfully\n");
        } else {
            vPrintString("Failed to create Task 2\n");
        }

        // Start the scheduler so the created tasks start executing
        vTaskStartScheduler();
    } else {
        vPrintString("Mutex not created successfully\n");
    }

    // If all is well, main() will never reach here
    for(;;);
}
```

Output:

Mutex created successfully

Task 1 created successfully

Task 2 created successfully

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 ************************************

Task 2 ------------------------------------

Task 1 ************************************

Task 2 ------------------------------------

Task 1 ************************************

Task 2 ------------------------------------

Task 1 ************************************

Task 2 ------------------------------------

10.2 Use of gatekeeper task

```
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include <stdio.h>
#include <stdlib.h>

// Define the maximum number of messages the queue can hold
#define QUEUE_LENGTH 5
void vApplicationIdleHook(void) {
    // Idle hook implementation (optional)
}

// Queue handle for passing messages to the gatekeeper task
QueueHandle_t xPrintQueue;

// Array of strings that tasks and interrupt will print via the gatekeeper
static char *pcStringsToPrint[] = {
    "Task 1 ****************************************************\r\n",
    "Task 2 ----------------------------------------------------\r\n",
    "Message printed from the tick hook interrupt ##############\r\n"
};

// Task to manage access to standard output
static void prvStdioGatekeeperTask(void *pvParameters) {
    char *pcMessageToPrint;

    for (;;) {
        // Wait for a message to arrive in the queue
        xQueueReceive(xPrintQueue, &pcMessageToPrint, portMAX_DELAY);

        // Output the received string
        printf("%s", pcMessageToPrint);

        fflush(stdout); // Ensure output is flushed

        // Loop back to wait for the next message
    }
}
```

```
// Task to send messages to the gatekeeper task
static void prvPrintTask(void *pvParameters) {
    int iIndexToString;
    const TickType_t xMaxBlockTimeTicks = 0x20;

    // Extract the index to the string from the task parameter
    iIndexToString = (int)pvParameters;

    for (;;) {
        // Send a pointer to the string to the gatekeeper task via the queue
           xQueueSendToBack(xPrintQueue, &(pcStringsToPrint[iIndexToString]),
0);

        // Wait a pseudo random time
        vTaskDelay((rand() % xMaxBlockTimeTicks));
    }
}

// Interrupt handler to send a message to the gatekeeper task
void vApplicationTickHook(void) {
    static int iCount = 0;

    // Print a message every 200 ticks
    iCount++;
    if (iCount >= 200) {
        // Send a pointer to the message via the queue
        xQueueSendToFrontFromISR(xPrintQueue, &(pcStringsToPrint[2]), NULL);

        // Reset the count to print the message again in 200 ticks
        iCount = 0;
    }
}

// Main function
int main(void) {
    // Create the queue to pass messages to the gatekeeper task
    xPrintQueue = xQueueCreate(QUEUE_LENGTH, sizeof(char *));

    // Check if the queue was created successfully
    if (xPrintQueue != NULL) {
        // Create tasks that send messages to the gatekeeper task
        xTaskCreate(prvPrintTask, "Print1", 100, (void *)0, 1, NULL);
        xTaskCreate(prvPrintTask, "Print2", 100, (void *)1, 2, NULL);

        // Create the gatekeeper task that manages access to stdout
            xTaskCreate(prvStdioGatekeeperTask, "Gatekeeper", 100, NULL, 0,
NULL);

        // Start the FreeRTOS scheduler
        vTaskStartScheduler();
    }

    // If the scheduler starts successfully, main() will not reach here
    for (;;);

}
```

Output:

Message printed from the tick hook interrupt ###############

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 1 **********

Task 1 **********

Task 1 **********

Task 2 ----------

Task 1 **********

Task 2 ----------

Task 2 ----------

Message printed from the tick hook interrupt ###############

Task 2 ----------

Task 1 **********

Task 1 **********

Task 2 ----------

Task 2 ----------

Task 1 **********

Task 1 **********

10.3 Priority inversion

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

// Define task priorities
#define HIGH_PRIORITY   (configMAX_PRIORITIES - 1)
#define MEDIUM_PRIORITY (configMAX_PRIORITIES - 2)
#define LOW_PRIORITY    (configMAX_PRIORITIES - 3)


// Define task handles
TaskHandle_t highTaskHandle;
TaskHandle_t mediumTaskHandle;
TaskHandle_t lowTaskHandle;

// Define a mutex (resource)
SemaphoreHandle_t mutex;


void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void vApplicationIdleHook(void) {
        //vPrintString("idle...");
}

// High priority task
void vHighPriorityTask(void *pvParameters) {
    while(1) {
        vPrintString("High priority task trying to acquire mutex\n");
        if (xSemaphoreTake(mutex, portMAX_DELAY) == pdTRUE) {
            printf("High priority task acquired mutex\n");
            // Simulate some work
            vTaskDelay(pdMS_TO_TICKS(1000));
            xSemaphoreGive(mutex);
            printf("High priority task released mutex\n");
        }
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

// Medium priority task
void vMediumPriorityTask(void *pvParameters) {
    while(1) {
        vPrintString("Medium priority task trying to acquire mutex\n");
```

```
        if (xSemaphoreTake(mutex, portMAX_DELAY) == pdTRUE) {
            vPrintString("Medium priority task acquired mutex\n");
            // Simulate some work
            vTaskDelay(pdMS_TO_TICKS(500));
            vPrintString("Medium priority task releasing mutex\n");
            xSemaphoreGive(mutex);
        }
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

// Low priority task
void vLowPriorityTask(void *pvParameters) {
    while(1) {
        vPrintString("Low priority task trying to acquire mutex\n");
        if (xSemaphoreTake(mutex, portMAX_DELAY) == pdTRUE) {
            vPrintString("Low priority task acquired mutex\n");
            // Simulate some work
            vTaskDelay(pdMS_TO_TICKS(200));
            xSemaphoreGive(mutex);
            vPrintString("Low priority task released mutex\n");
        }
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}

int main() {
    // Create mutex
    mutex = xSemaphoreCreateMutex();

    // Create tasks
    xTaskCreate(vHighPriorityTask, "High", configMINIMAL_STACK_SIZE, NULL,
HIGH_PRIORITY, &highTaskHandle);
    xTaskCreate(vMediumPriorityTask, "Medium", configMINIMAL_STACK_SIZE,
NULL, MEDIUM_PRIORITY, &mediumTaskHandle);
    xTaskCreate(vLowPriorityTask, "Low", configMINIMAL_STACK_SIZE, NULL,
LOW_PRIORITY, &lowTaskHandle);

    // Start scheduler
    vTaskStartScheduler();
    for(;;);
}
```

Output:

High priority task trying to acquire mutex

High priority task acquired mutex

Medium priority task trying to acquire mutex

Low priority task trying to acquire mutex

High priority task released mutex

Medium priority task acquired mutex

Medium priority task releasing mutex

Low priority task acquired mutex

Low priority task released mutex

Medium priority task trying to acquire mutex

Medium priority task acquired mutex

Low priority task trying to acquire mutex

High priority task trying to acquire mutex

Medium priority task releasing mutex

High priority task acquired mutex

High priority task released mutex

Medium priority task trying to acquire mutex

Medium priority task acquired mutex

Medium priority task releasing mutex

Low priority task acquired mutex

Low priority task released mutex

Medium priority task trying to acquire mutex

Medium priority task acquired mutex

Low priority task trying to acquire mutex

High priority task trying to acquire mutex

Medium priority task releasing mutex

High priority task acquired mutex

High priority task released mutex

Medium priority task trying to acquire mutex

Medium priority task acquired mutex

Medium priority task releasing mutex

Low priority task acquired mutex

Low priority task released mutex

10.4 Deadlock condition

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

// Define task priorities
#define TASK1_PRIORITY   (configMAX_PRIORITIES )
#define TASK2_PRIORITY   (configMAX_PRIORITIES - 1)
#define WATCHDOG_PRIORITY (configMAX_PRIORITIES - 2)

// Define task handles
TaskHandle_t task1Handle;
TaskHandle_t task2Handle;

// Define mutexes
SemaphoreHandle_t mutex1;
SemaphoreHandle_t mutex2;

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void vApplicationIdleHook(void) {
        //vPrintString("idle...");
}

const char* getTaskStateString(eTaskState state) {
    switch (state) {
        case eRunning:   return "Running";
        case eReady:     return "Ready";
        case eBlocked:   return "Blocked";
        case eSuspended: return "Suspended";
        case eDeleted:   return "Deleted";
        default:         return "Unknown";
    }
}

// Function to print deadlock condition
void printDeadlockCondition(void) {
    vPrintString("Deadlock Condition Detected:\n");
    vPrintString("Task 1 status: %s\n",
getTaskStateString(eTaskGetState(task1Handle)));
    vPrintString("Task 2 status: %s\n",
getTaskStateString(eTaskGetState(task2Handle)));
    vPrintString("Mutex 1 count: %d\n", (int)uxSemaphoreGetCount(mutex1));
    vPrintString("Mutex 2 count: %d\n", (int)uxSemaphoreGetCount(mutex2));
}

// Watchdog task to detect deadlocks
void vWatchdogTask(void *pvParameters) {
    while(1) {
        // Check if both tasks are blocked
```

```
        if ((eTaskGetState(task1Handle) == eBlocked) &&
(eTaskGetState(task2Handle) == eBlocked)) {
            printDeadlockCondition();
        }
        vTaskDelay(pdMS_TO_TICKS(1000)); // Check every second
    }
}

// Task 1
void vTask1(void *pvParameters) {
    while(1) {
        printf("Task 1 trying to acquire mutex 1\n");
        if (xSemaphoreTake(mutex1, portMAX_DELAY) == pdTRUE) {
            printf("Task 1 acquired mutex 1\n");
            vTaskDelay(pdMS_TO_TICKS(500)); // Simulate work
            printf("Task 1 trying to acquire mutex 2\n");
            if (xSemaphoreTake(mutex2, portMAX_DELAY) == pdTRUE) {
                printf("Task 1 acquired mutex 2\n");
                // Do something with mutex 1 and mutex 2...
                xSemaphoreGive(mutex2);
                printf("Task 1 released mutex 2\n");
            } else {
                // Print deadlock condition if unable to acquire mutex 2
                printDeadlockCondition();
            }
            xSemaphoreGive(mutex1);
            printf("Task 1 released mutex 1\n");
        } else {
            // Print deadlock condition if unable to acquire mutex 1
            printDeadlockCondition();
        }
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}

// Task 2
void vTask2(void *pvParameters) {
    while(1) {
        printf("Task 2 trying to acquire mutex 2\n");
        if (xSemaphoreTake(mutex2, portMAX_DELAY) == pdTRUE) {
            printf("Task 2 acquired mutex 2\n");
            vTaskDelay(pdMS_TO_TICKS(500)); // Simulate work
            printf("Task 2 trying to acquire mutex 1\n");
            if (xSemaphoreTake(mutex1, portMAX_DELAY) == pdTRUE) {
                printf("Task 2 acquired mutex 1\n");
                // Do something with mutex 1 and mutex 2...
                xSemaphoreGive(mutex1);
                printf("Task 2 released mutex 1\n");
            } else {
                // Print deadlock condition if unable to acquire mutex 1
                printDeadlockCondition();
            }
            xSemaphoreGive(mutex2);
            printf("Task 2 released mutex 2\n");
        } else {
            // Print deadlock condition if unable to acquire mutex 2
            printDeadlockCondition();
        }
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

```
int main() {
    // Create mutexes
    mutex1 = xSemaphoreCreateMutex();
    mutex2 = xSemaphoreCreateMutex();

    // Create tasks
    xTaskCreate(vTask1, "Task1", configMINIMAL_STACK_SIZE, NULL,
TASK1_PRIORITY, &task1Handle);
    xTaskCreate(vTask2, "Task2", configMINIMAL_STACK_SIZE, NULL,
TASK2_PRIORITY, &task2Handle);
    xTaskCreate(vWatchdogTask, "Watchdog", configMINIMAL_STACK_SIZE, NULL,
WATCHDOG_PRIORITY, NULL);

    // Start scheduler
    vTaskStartScheduler();

    for(;;);
}
```

Output:

Task 2 trying to acquire mutex 2

Task 2 acquired mutex 2

Task 1 trying to acquire mutex 1

Task 1 acquired mutex 1

Deadlock Condition Detected:

Task 1 status: Blocked

Task 2 status: Blocked

Mutex 1 count: 0

Mutex 2 count: 0

Task 2 trying to acquire mutex 1

Task 1 trying to acquire mutex 2

Deadlock Condition Detected:

Task 1 status: Blocked

Task 2 status: Blocked

Mutex 1 count: 0

Mutex 2 count: 0

Deadlock Condition Detected:

Task 1 status: Blocked

Task 2 status: Blocked

Mutex 1 count: 0

Mutex 2 count: 0

Deadlock Condition Detected:

Task 1 status: Blocked

Task 2 status: Blocked

Mutex 1 count: 0

Mutex 2 count: 0

Deadlock Condition Detected:

Task 1 status: Blocked

Task 2 status: Blocked

Mutex 1 count: 0

Mutex 2 count: 0

10.5 Use of Recursive mutex

```
#include <stdio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>

// Define task priorities
#define TASK1_PRIORITY   (configMAX_PRIORITIES )

// Define task handles
TaskHandle_t task1Handle;

// Define a recursive mutex
SemaphoreHandle_t recursiveMutex;

void vPrintString(const char *format, ...) {
    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
    fflush(stdout);
}

void vApplicationIdleHook(void) {
        //vPrintString("idle...");
}

// Function 1
void function1(void) {
```

```
    vPrintString("Function1 trying to acquire recursive mutex\n");
    if (xSemaphoreTakeRecursive(recursiveMutex, portMAX_DELAY) == pdTRUE) {
        vPrintString("Function1 acquired recursive mutex\n");
        // Perform some task
        xSemaphoreGiveRecursive(recursiveMutex);
        vPrintString("Function1 released recursive mutex\n");
    }
}

// Task 1
void vTask1(void *pvParameters) {
    while(1) {
        vPrintString("Task 1 trying to acquire recursive mutex\n");
        if (xSemaphoreTakeRecursive(recursiveMutex, portMAX_DELAY) == pdTRUE)
{
            vPrintString("Task 1 acquired recursive mutex\n");

            // Nested function call acquiring the same mutex
            vPrintString("Task 1 calling function1()\n");
            function1();

            xSemaphoreGiveRecursive(recursiveMutex);
            vPrintString("Task 1 released recursive mutex\n");
        }
        vTaskDelay(pdMS_TO_TICKS(1000)); // Delay for 1 second
    }
}



int main() {
    // Create a recursive mutex
    recursiveMutex = xSemaphoreCreateRecursiveMutex();

    // Create task
    xTaskCreate(vTask1, "Task1", configMINIMAL_STACK_SIZE, NULL,
TASK1_PRIORITY, &task1Handle);

    // Start scheduler
    vTaskStartScheduler();

    for(;;);

}
```

Output:

Task 1 trying to acquire recursive mutex

Task 1 acquired recursive mutex

Task 1 calling function1()

Function1 trying to acquire recursive mutex

Function1 acquired recursive mutex

Function1 released recursive mutex

Task 1 released recursive mutex

Task 1 trying to acquire recursive mutex

Task 1 acquired recursive mutex

Task 1 calling function1()

Function1 trying to acquire recursive mutex

Function1 acquired recursive mutex

Function1 released recursive mutex

Task 1 released recursive mutex