Semaphores and Mutexes

INTRODUCTION

There is often ambiguity around semaphores and mutexes. The most obvious difference is that mutexes include a priority inheritance mechanism, and binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

COUNTING SEMAPHORES

Just as binary semaphores can be thought of as queues of length one, counting semaphores can be thought of as queues of length greater than one. Again, users of the semaphore are not interested in the data that is stored in the queue - just whether the queue is empty or not. Counting semaphores are typically used for two things:

1. Counting events.

In this usage scenario an event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the count value to be zero when the semaphore is created.

2. Resource management.

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the count value to be equal to the maximum count value when the semaphore is created.

BIINARY SEMAPHORES

Think of a binary semaphore as a queue that can only hold one item. The queue can therefore only be empty, or full (hence binary). Tasks and interrupts using the queue don't care what the queue holds - they only want to know if the queue is empty or full. This mechanism can be exploited to synchronise, for example, a task with an interrupt.

Consider the case where a task is used to service a peripheral. Polling the peripheral would be wasteful of CPU resources, and prevent other tasks from executing. It is therefore preferable that the task spends most of its time in the Blocked state (allowing other tasks to execute) and only executes itself when there is actually something for it to do.

This is achieved by having the task Block while attempting to 'take' the semaphore. An interrupt routine is then written for the peripheral that just 'gives' the semaphore when the peripheral requires servicing. The task always 'takes' the semaphore (reads from the queue to make the queue empty), but never 'gives' it. The interrupt always 'gives' the semaphore (writes to the queue to make it full) but never takes it.

Semaphore API functions permit a block time to be specified. The block time indicates the maximum number of 'ticks' that a task should spend in the Blocked state when attempting to 'take' a semaphore, should the semaphore not be immediately available. If more than one task blocks on the same semaphore then the task with the highest priority will be the task that is unblocked the next time the semaphore becomes available.

OPEN**RTOS** and SAFE**RTOS** contain a Task Notification feature that can be used as a faster and lighter weight binary semaphore alternative in some situations.

high**integrity**systems

email: sales@high**integrity**systems.com
web: www.high**integrity**systems.com

RECURSIVE MUTEXES

A mutex used recursively can be 'taken' repeatedly by the owner. The mutex doesn't become available again until the owner has given back the mutex for each successful mutex take request. For example, if a task successfully 'takes' the same mutex 5 times then the mutex will not be available to any other task until it has also 'given' the mutex back exactly five times.

Recursive mutexes use a priority inheritance mechanism so a task 'taking' a semaphore must always 'give' the semaphore back once the semaphore is no longer required.

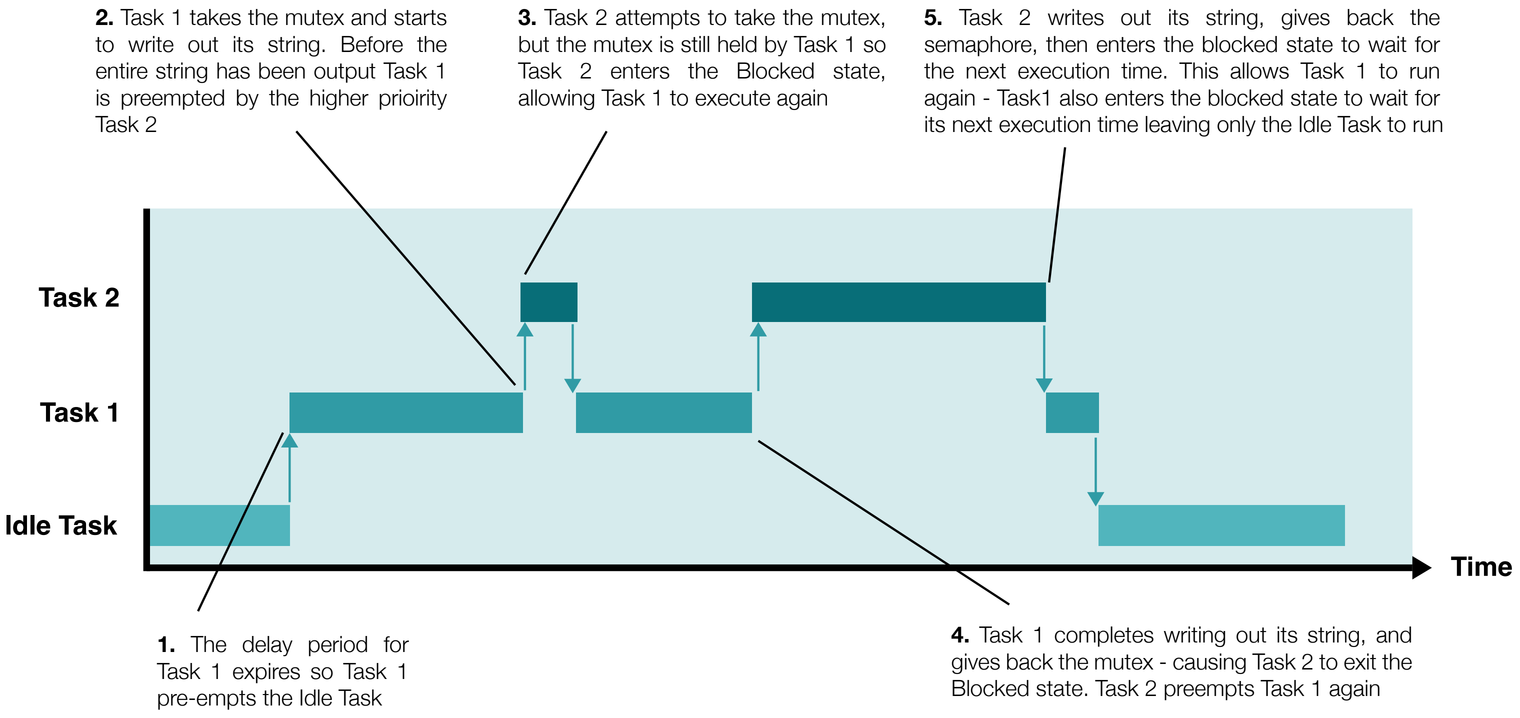
Mutex type semaphores cannot be used from within interrupt service routines. This is because they include a priority inheritance mechanism which only makes sense if the mutex is given and taken from a task, not an interrupt. Further, an interrupt cannot block to wait for a resource that is guarded by a mutex to become available

MUTEXES

Whereas binary semaphores are the better choice for implementing synchronization (between tasks or between tasks and an interrupt), mutexes are the better choice for implementing simple mutual exclusion (hence 'MUT'ual 'EX'clusion).

When used for mutual exclusion the mutex acts like a token that is used to guard a resource. When a task wishes to access the resource it must first obtain ('take') the token. When it has finished with the resource it must 'give' the token back - allowing other tasks the opportunity to access the same resource.

Mutexes also permit a block time to be specified. The block time indicates the maximum number of 'ticks' that a task should spend in the Blocked state when attempting to 'take' a mutex if the mutex is not immediately available. Unlike binary semaphores, mutexes employ priority inheritance. This means that if a high priority task blocks while attempting to obtain a mutex (token) that is currently held by a lower priority task, then the priority of the task holding the token is temporarily raised to that of the blocking task. This mechanism is designed to ensure the higher priority task is kept in the blocked state for the shortest time possible, and in so doing minimises the 'priority inversion' that has already occurred.



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

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 prioirity Task 2

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

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

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 - Task1 also enters the blocked state to wait for its next execution time leaving only the Idle Task to run

Figure. A Possible Sequence of Execution Involving Mutexes