

# More Basics and Advanced Features of FreeRTOS

#### **ENCE464 Embedded Software and Advanced Computing**

Course coordinator: Steve Weddell (<a href="mailto:steve.weddell@canterbury.ac.nz">steve.weddell@canterbury.ac.nz</a>)

Lecturer: Le Yang (<a href="mailto:le.yang@canterbury.ac.nz">le.yang@canterbury.ac.nz</a>)

Department of Electrical and Computer Engineering

#### Where we're going today

More on mutexes

More on queues

#### Mutexes as Token

- Mutexes are used to achieve mutual exclusion, controlling concurrent access to shared resources
  - Only the task that has the mutex can access the shared resource at a time
- Realize mutual exclusion via semaphore 'mutex' initialized to 1

	Thread A	
1	mutex.take()	
2	# critical section	
3	count = count + 1	
4	mutex.give()	

	Thread B
1	mutex.take()
2	# critical section
3	count = count + 1
4	mutex.give()

This solution generalizes to arbitrary number of tasks

#### Extension to Multiplex

Mutex as token allows only one task to access the shared resource.

- Multiplex allows up to N tasks to access the share resource
  - In the context of embedded systems, manage a finite number of buffers within a shared memory space
  - Implementation: initialize a semaphore 'multiplex' to N

	Thread
1	multiplex.take()
2	# critical section
3	multiplex.give()

#### Synchronizing More than 2 Tasks (1)

- Use of a barrier
  - There are N tasks, all of which need to reach some critical point before proceeding further
- Implementation of a barrier using semaphores

	initialise
1	N = number of threads
2	count = 0
3	mutex = createSemaphore(1)
4	barrier = createSemaphore(0)

- count: keep track how many tasks have arrived critical point
- mutex: token (semaphore) for mutually exclusive access to shared variable count
- barrier: binary semaphore for blocking until all tasks have arrived critical point

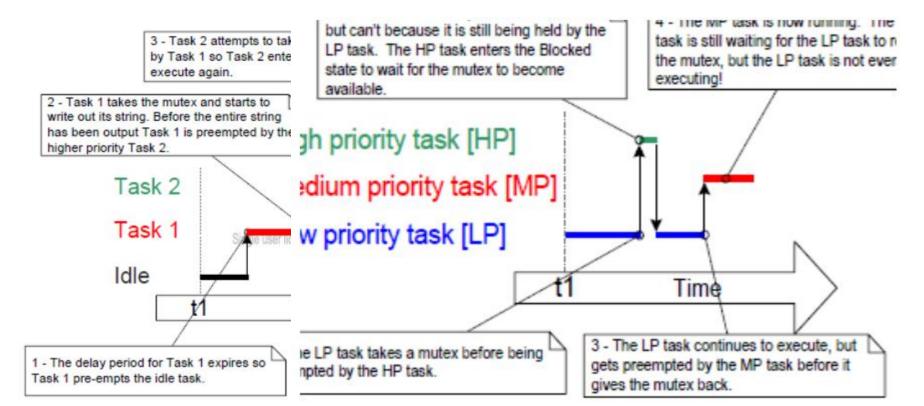
#### Synchronizing More than 2 Tasks (2)

- Is this a correct solution?
  - No
  - Only one task can proceed to critical point
    - Deadlock!
- Solution?
  - Allows one thread to pass at a time

```
Thread
     # rendevous
3
     mutex.take()
4
       count = count + 1
     mutex.give()
5
6
     if count == N: barrier.give()
     barrier.take()
     barrier.give()
10
     # critical point
11
```

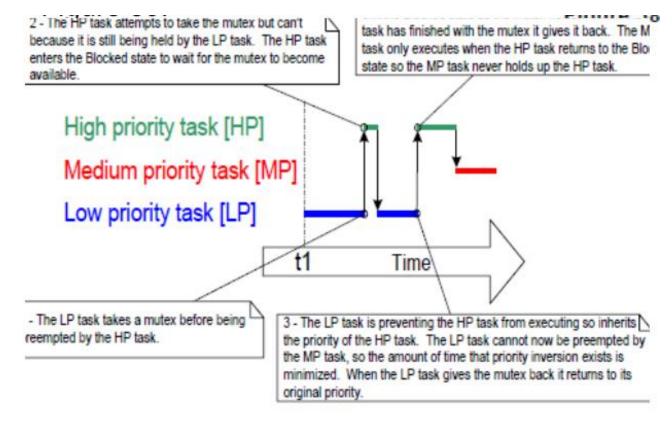
#### **Priority Inversion**

- In FreeRTOS, mutexes are realized as a special form of semaphores
- Priority inversion: higher priority task is blocked due to lower priority task holding the mutex



#### **Priority Inheritance**

- In FreeRTOS, priority inheritance is introduced to mitigate priority inversion
  - The task holding the mutex temporarily receives the priority of the highest priority task blocked due to waiting for the mutex



#### Where we're going today

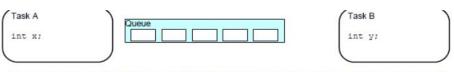
More on mutexes

More on queues

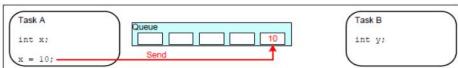
#### Queues

Semaphores are used to communicate events

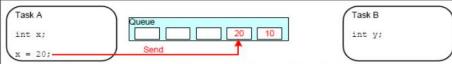
- Queues are used to communicate events and transfer data
  - First-in-first-out buffers
    - Writing to the tail and reading from the head



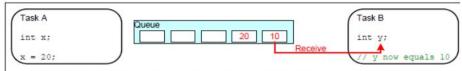
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



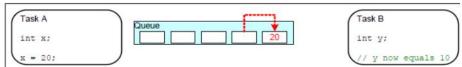
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration)



Task B has removed one item, leaving only the second value written by Task A remaining in the queue.

This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

#### An Application Example

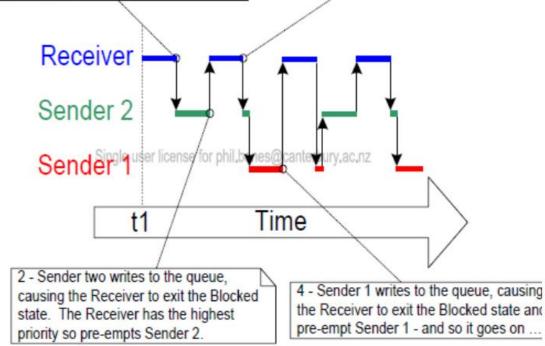
• Two tasks, 'Sender 1' and 'Sender 2', write data to a queue while a single task, 'Receiver', retrieves data items from it

highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Once the Receiver is blocked Sender 2 can run.

then enters the Blocked state again, allowing Sender 2 to execute once more. Sender 2 immediately Yields to Sender 1.

'Receiver' task is deliberately set to have higher priority that sender tasks

Queue can only have at most one entry



## Implement Queues using Semaphores (1)

- Ballroom dancer example
  - Two tasks represent two types of dancers: leaders and followers
  - Wait in two queues before entering dancing floor
  - When a leader arrives, it checks to see if there is a follower waiting
    - If yes, they proceed to dance
    - If no, the leader waits
  - Similar for the follower

#### Implement Queues using Semaphores (2)

Initialization

Listing 3.17: Queue hint

```
1 leaders = followers = 0
2 mutex = Semaphore(1)
3 leaderQueue = Semaphore(0)
4 followerQueue = Semaphore(0)
5 rendezvous = Semaphore(0)
```

- leaders, followers. mannaers or readers and romowers in the quedes
- mutex: token for shared variables leaders, followers access
- leaderQueue, followerQueue: semaphores for the queues
- rendezvous: barrier before critical section dance()

## Implement Queues using Semaphores (3)

Listing 3.18: Queue solution (leaders)

```
mutex.wait()
2 if followers > 0:
      followers--
      followerQueue.signal()
5 else:
      leaders++
      mutex.signal()
      leaderQueue.wait()
  dance()
  rendezvous.wait()
  mutex.signal()
```

Listing 3.19: Queue solution (followers)

```
mutex.wait()
 2 if leaders > 0:
       leaders--
       leaderQueue.signal()
 5 else:
       followers++
       mutex.signal()
       followerQueue.wait()
   dance()
11 rendezvous.signal()
```

# Example of Semaphore and Queue Usage: The "Heli Rig" project – 2013 3<sup>rd</sup> Pro Project

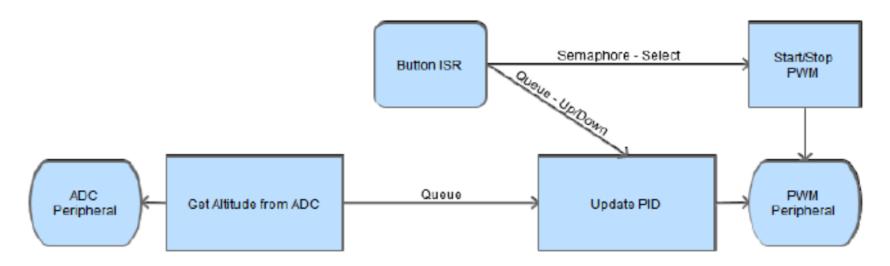


Figure 7: The helicopter control program was broken down into a number of FreeR-TOS tasks.

A real-time operating system, FreeRTOS, was used to modularise the Heli-Rig code into a number of tasks as shown in Fig. 7. Communication between tasks can use queues and semaphores, but many combinations exist.

**Exercise**: Justify the use of Queues and a Semaphore for use in this application.