More Basics and Advanced Features of FreeRTOS

ENCE464 Embedded Software and Advanced Computing

Course coordinator: Steve Weddell (steve.weddell@canterbury.ac.nz)

Lecturer: Le Yang (le.yang@canterbury.ac.nz)

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

From The Little Book of Semaphores.pdf

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!

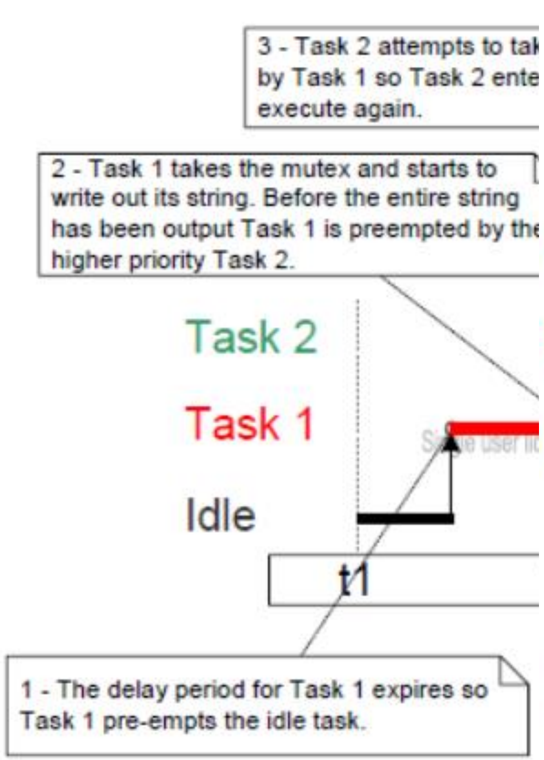
- Solution?
  - Allows one thread to pass at a time

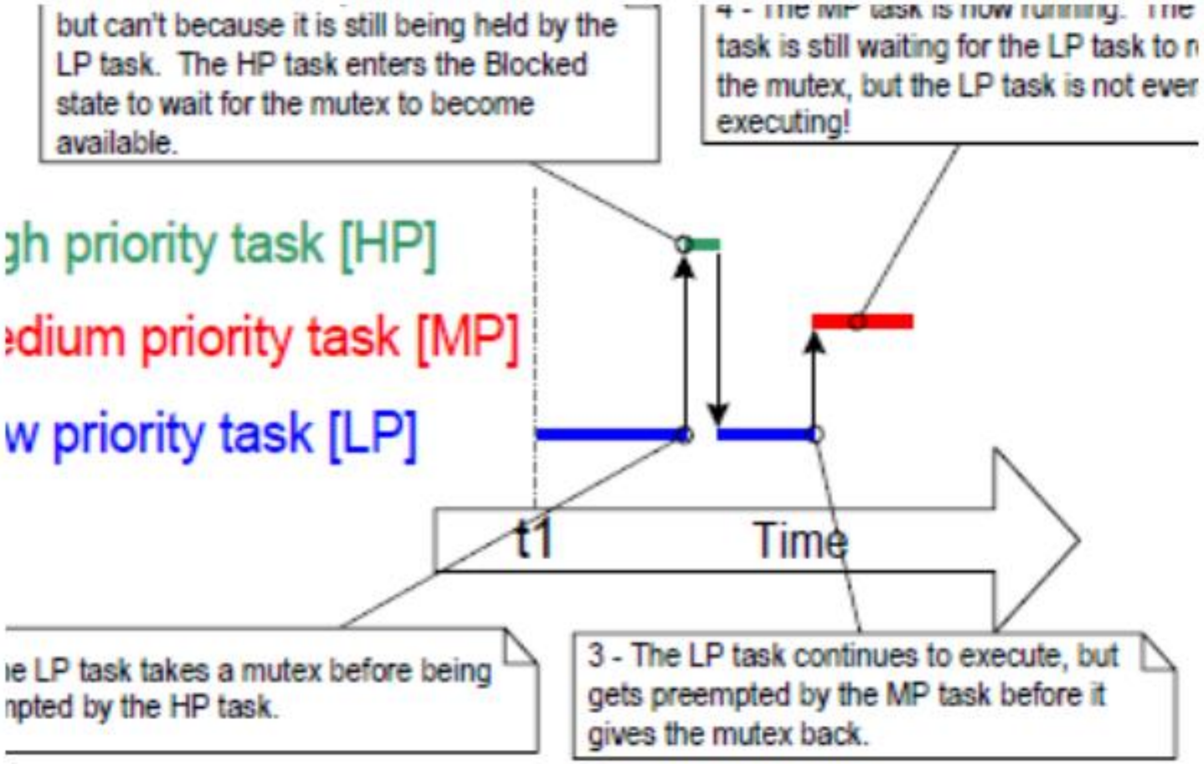
Thread

```
# rendevous

mutex.take()
    count = count + 1
mutex.give()

if count == N:  barrier.give()

barrier.take()
barrier.give()
# critical point
```

Priority Inversion

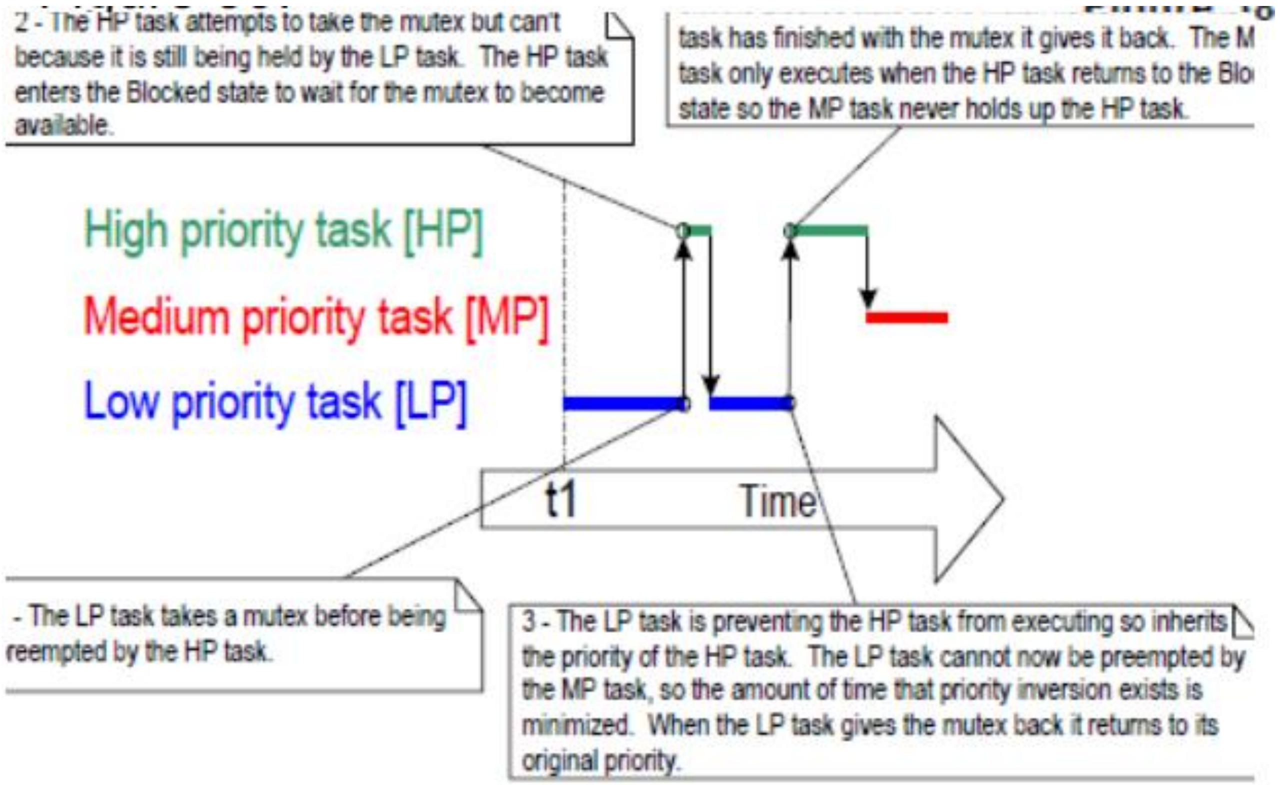
- In FreeRTOS, mutexes are realized as a special form of semaphores
- Priority inversion: higher priority task is blocked due to lower priority task holding the mutex





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



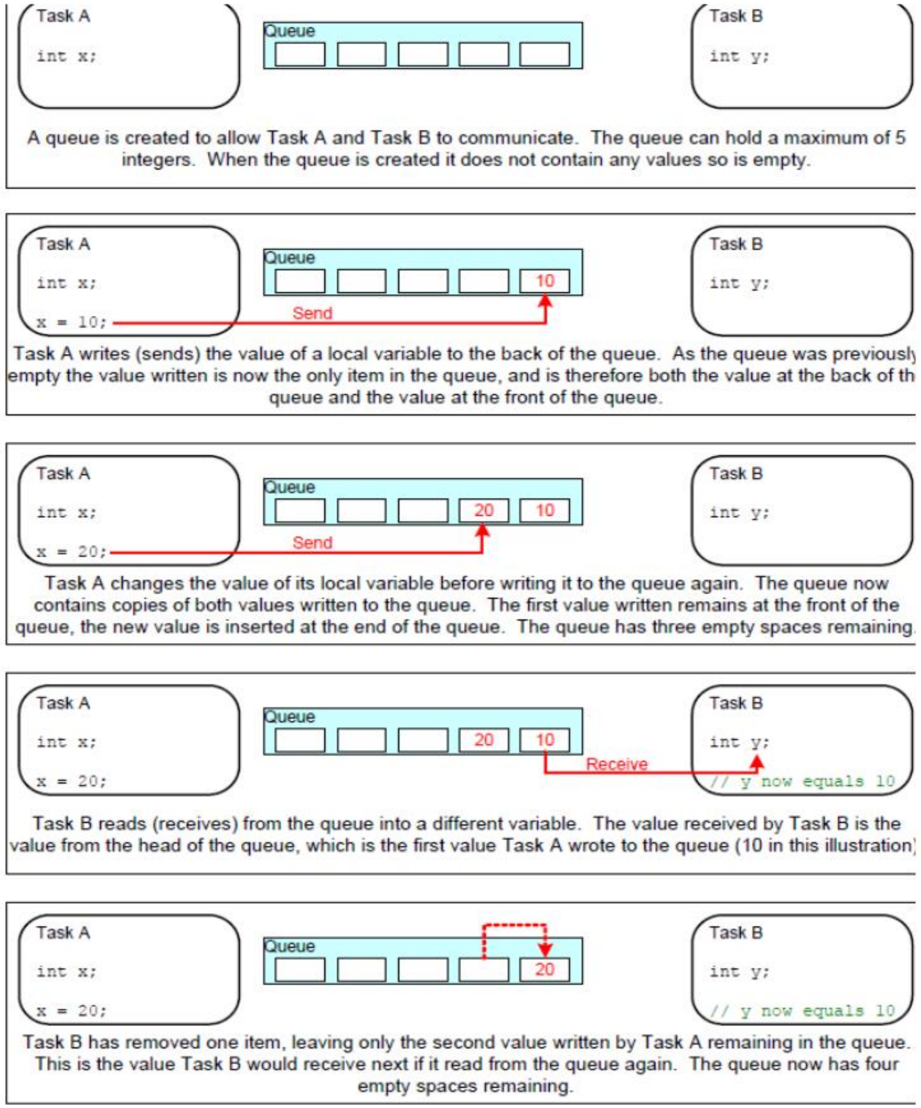
Where we’re going today

- 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

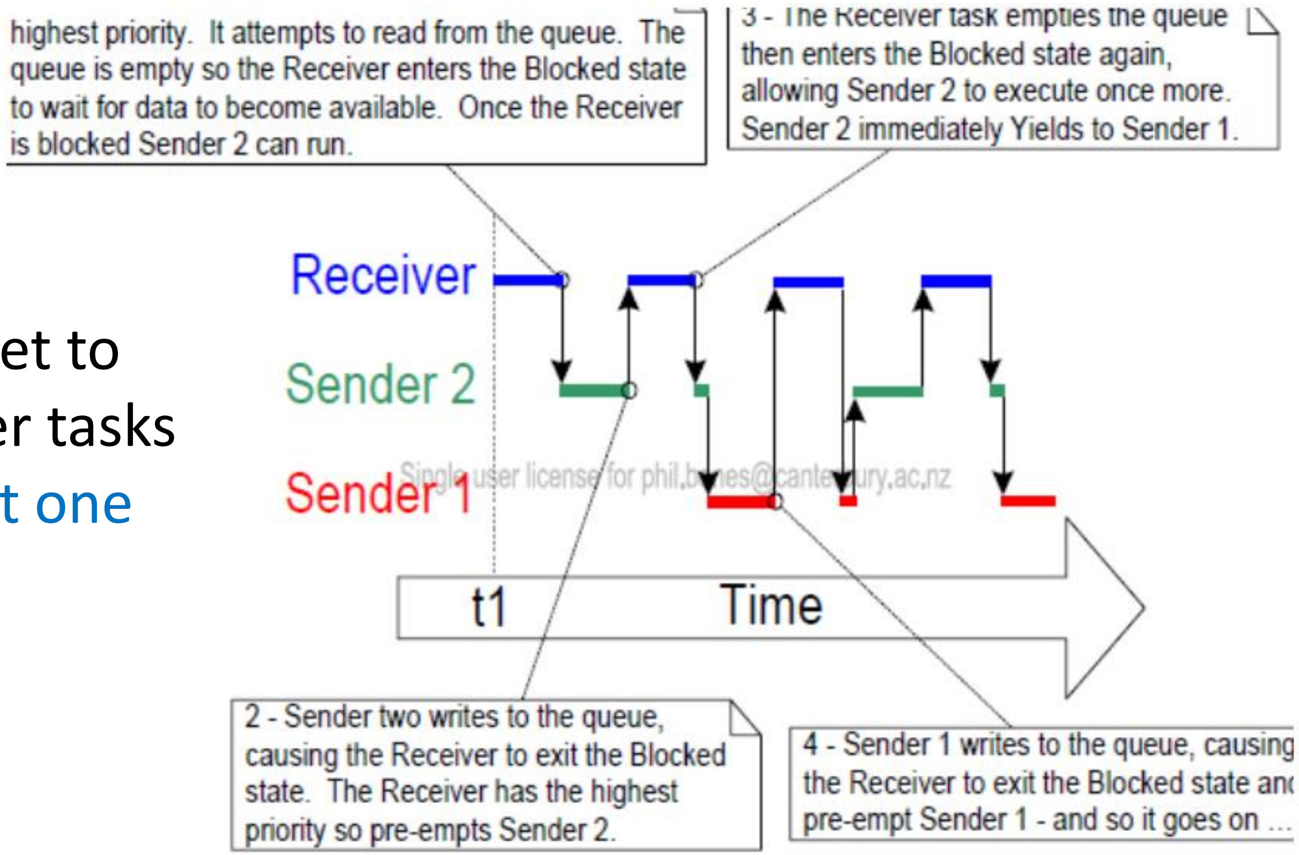


An Application Example

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

‘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

```
leaders = followers = 0
mutex = Semaphore(1)
leaderQueue = Semaphore(0)
followerQueue = Semaphore(0)
rendezvous = Semaphore(0)
```

- leaders, followers: numbers of leaders and followers in the queues
- 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()
if followers > 0:
    followers--
    followerQueue.signal()
else:
    leaders++
    mutex.signal()
    leaderQueue.wait()

dance()
rendezvous.wait()
mutex.signal()
```

Listing 3.19: Queue solution (followers)

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

dance()
rendezvous.signal()
```

Example of Semaphore and Queue Usage: The "Heli Rig" project – 2013 3rd Pro Project

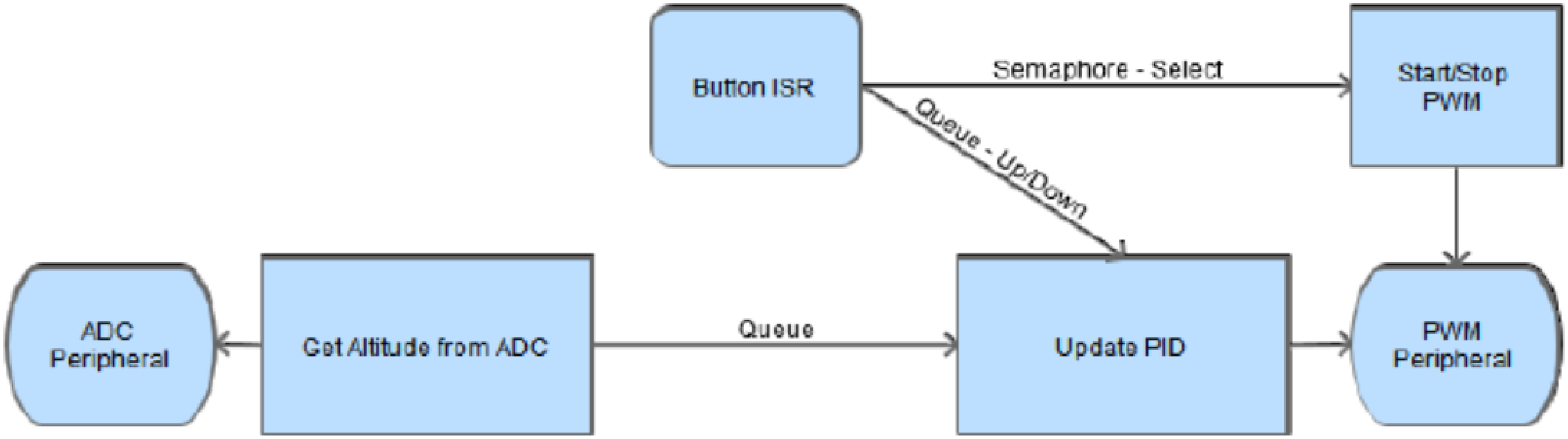


Figure 7: The helicopter control program was broken down into a number of FreeRTOS tasks.

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.