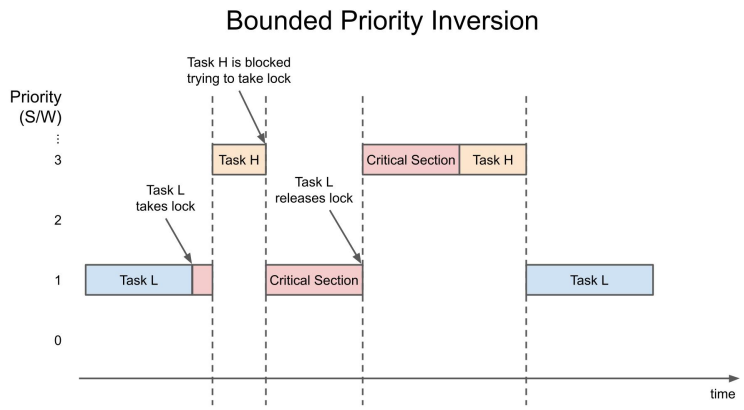
Discussion points

Priority Inversion (scenario taken from web)

Priority inversion is a bug that occurs when a high priority task is indirectly preempted by a low priority task. For example, the low priority task holds a mutex that the high priority task must wait for to continue executing. In the simple case, the high priority task (Task H) would be blocked as long as the low priority task (Task L) held the lock.

This is known as "bounded priority inversion," as the length of time of the inversion is bounded by lower task. however long the low task is in the critical section (holding the lock).

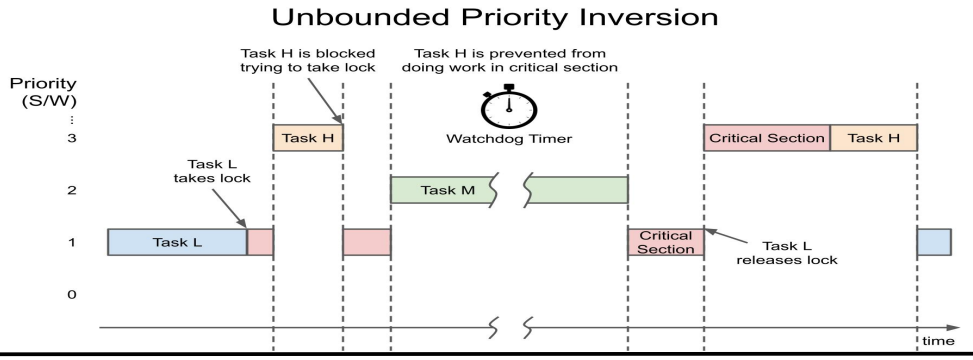


As you can see in the diagram above, Task H is blocked so long as Task L holds the lock. The priority of the tasks have been indirectly "inverted" as now Task L is running before Task H.

Unbounded priority Inversion

Unbounded priority inversion occurs when a medium priority task (Task M) interrupts Task L while it holds the lock.

It's called "unbounded" because Task M can now effectively block Task H for any amount of time, as Task M is preempting Task L (which still holds the lock).



Priority inversion nearly ended the Mars Pathfinder mission in 1997. After deploying the rover, the lander would randomly reset every few days due to an intermittent priority inversion bug that caused the watchdog timer to trigger a full system restart. Engineers eventually found the bug and sent an update patch to the lander.

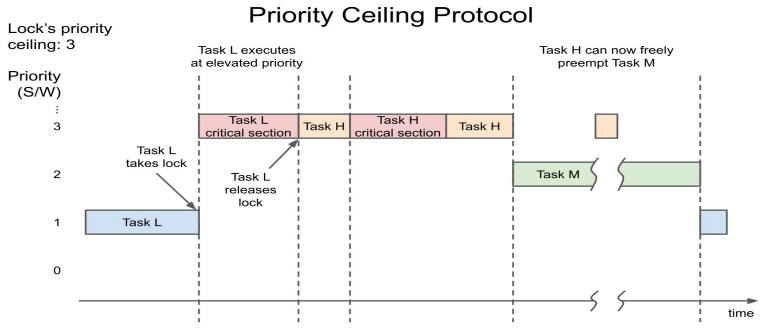
Solutions:

Priority ceiling protocol

Priority ceiling protocol involves assigning a "priority ceiling level" to each resource or lock.

Whenever a task works with a particular resource or takes a lock, the task's priority level is automatically boosted to that of the priority ceiling associated with the lock or resource.

The priority ceiling is determined by the maximum priority of any task that needs to use the resource or lock.



As the priority ceiling of the lock is 3, whenever Task L takes the lock, its priority is boosted to 3 so that it will run at the same priority as Task H. This prevents Task M (priority 2) from running until Tasks L and H are done with the lock.

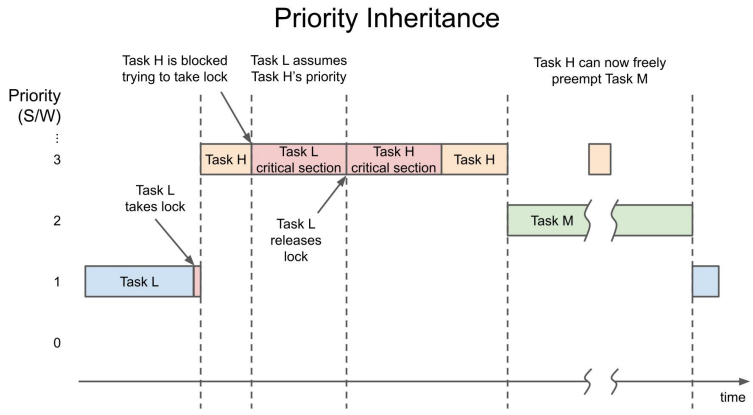
Priority inheritance

involves boosting the priority of a task holding a lock to that of any other (higher priority) task that tries to take the lock.

Task L takes the lock. Only when Task H attempts to take the lock is the priority of Task L boosted to that of Task H's.

Once again, Task M can no longer interrupt Task L until both tasks

are finished in the critical section.



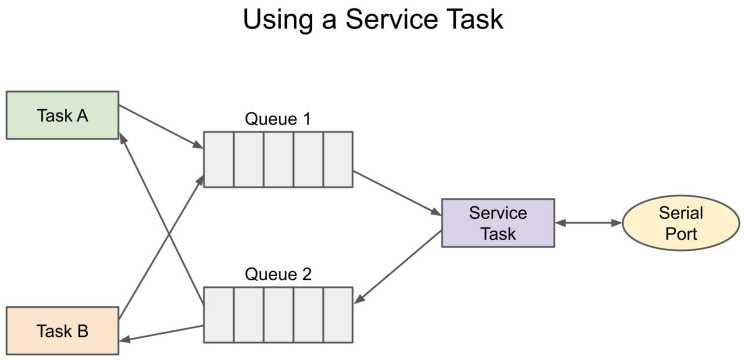
Note that in both priority ceiling protocol and priority inheritance, Task L's priority is dropped back to its original level once it releases the lock. Also note that both systems only prevent unbounded priority inversion. Bounded priority inversion can still occur.

We can only avoid or mitigate bounded priority inversion through good programming practices.

Some possible tips include (pick and choose based on your project's need):

-> Keep critical sections short to cut down on bounded priority inversion time
-> Avoid using critical sections or locking mechanisms that can block a high priority task
-> Use one task to control a shared resource to avoid the need to create locks to protect it

To demonstrate the last point, we could use a "service task" that was in charge of managing a shared resource.
For example, we could use queues to send and receive messages from a task that handled the serial port.



Interrupt management:

Refer to the Chapter 6, interrupt management.

All the interrupt that are HW interrupt, means which are supported have a specific behaviour from architecture to architecture.

And the implemened realtime kernel on these architectures might have a different implementation.

As in the priority of interrupt implementation in ARM is somewhat different than the FreeRTOS kernel

In ARM, lower the prioirty number of an interrupt, Higher is the priority
In RTOS, lower the prioirty number of Task, lower is the priority.

Furthermore, we have to understand that the kernel itself is nothing but another type of task/service/OS-service which is being triggered in timely manner based on the implementation.

As in the implementation of the freertos, the OS kernel is nothing but a combination of few services and the Hand-in hand working of the SysTick and PendSV and SVC handlers.

One has to know that at the end of day, these are hardware exceptions (CPU) and are basically interrupts.

If we check the implementation of these in port.c and the implementation of vTaskStartScheduler(), we can see that after creating the idle task, our scheduler bascially sets the prioirty of these exceptions to the lowest value (as per RTOS), means now these interrupts are having lower priority.

And this makes sense, why!!!

Because in Real-time systems there is no such thing as fair-share policy as one is there is GPOS. Hence, if a higher priority task unblocks/become ready to run then the OS/Realtime kernel itself should not be able to stop/preempt that task in order to run itself.

This point is very important in implementing interrupts.

In case you're using the interrupts without reading the FreeRTOS tutorial guide Chapter 6, you should go and give it a full reading first.

As The ARM Cortex cores, and ARM Generic Interrupt Controllers (GICs), use numerically low priority numbers to represent logically high priority interrupts. This can seem counter-intuitive, and is easy to forget.

configKERNEL_INTERRUPT_PRIORITY must always be set to the lowest possible interrupt priority. Unimplemented priority bits can be set to 1, so the constant can always be set to 255, no matter how many priority bits are actually implemented.
Cortex-M interrupts will default to a priority of zero—the highest possible priority. implementation of the Cortex-M hardware does not The permit configMAX_SYSCALL_INTERRUPT_PRIORITY to be set to 0, so the priority of an interrupt that uses the FreeRTOS API must never be left at its default value.

-Aman Kanwar

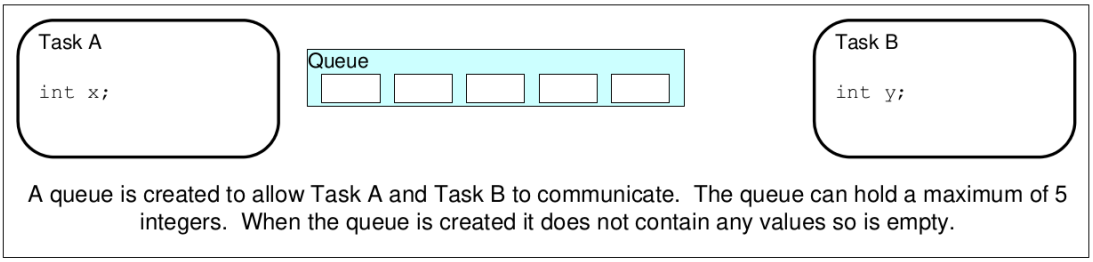
Working on Queues

Queue Creation
Queue management
Sending Data to the queue
Receiving Data from the queue
Blocking on a Queue
Clearing of Queue

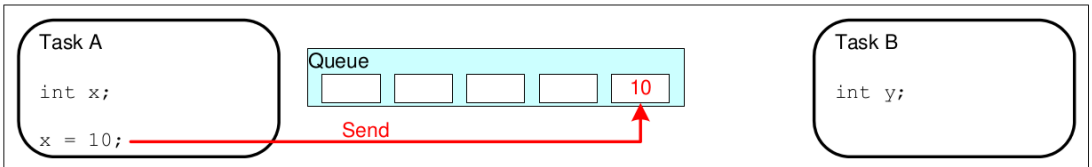
Data Storage

A queue can hold a finite number of fixed size data items. The maximum number of items a queue can hold is called its 'length'. Both the length and the size of each data item are set when the queue is created.

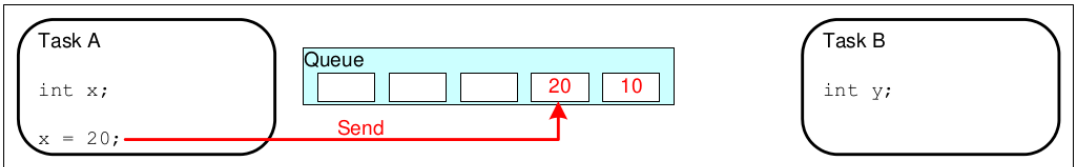
Queues are normally used as First In First Out (FIFO) buffers, where data is written to the end (tail) of the queue and removed from the front (head) of the queue.



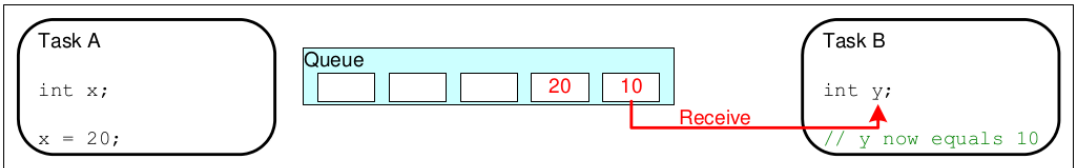
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



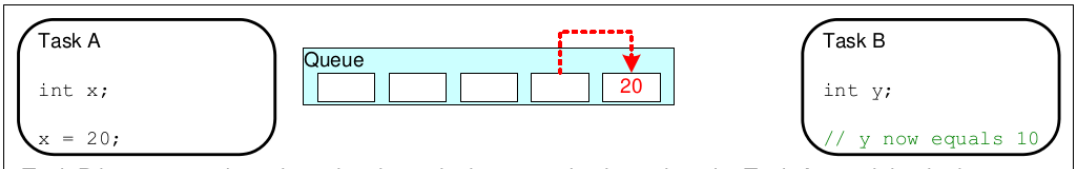
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



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.

before using the queue, we should create the queue.

Now freeRTOS provides us with following API to create the Queue.

```
#include "FreeRTOS.h"
#include "queue.h"

QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

Parameters

uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size, in bytes, of each data item that can be stored in the queue.

Return Values

NULL	The queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.
Any other value	The queue was created successfully. The returned value is a handle by which the created queue can be referenced.

Refer to disccused assignment questions on the same.