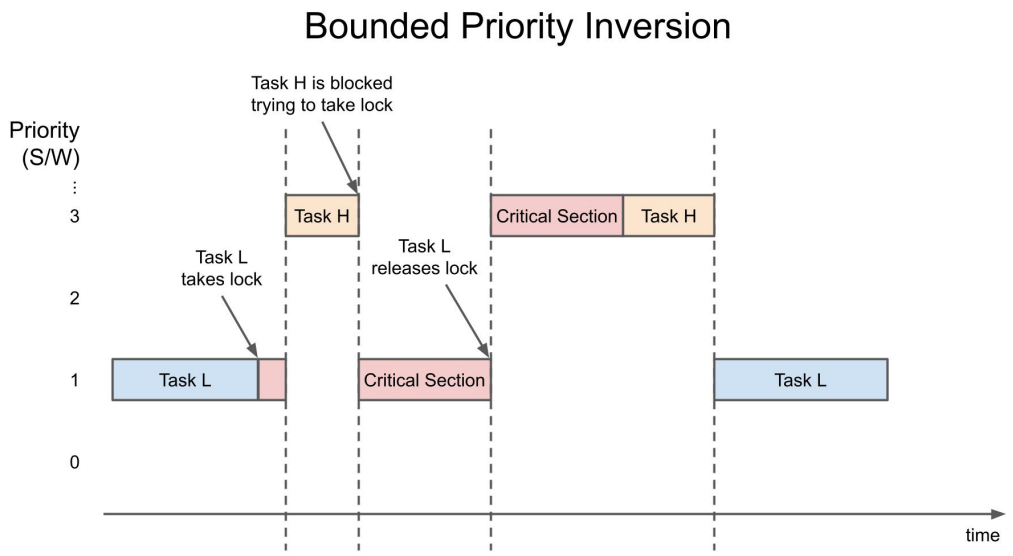
Introduction to RTOS - Solution to Part 11 (Priority Inversion)

By ShawnHymel

Concepts

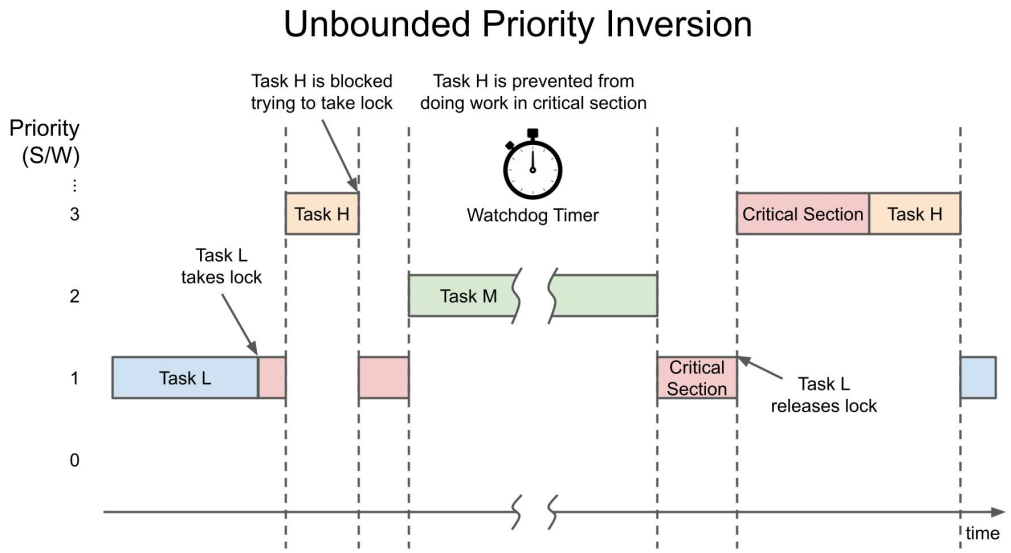
Priority inversion is a bug that occurs when a high priority task is indirectly preempted by a low priority task. For example, the low priority task holds a mutex that the high priority task must wait for to continue executing.

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 however long the low priority 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.

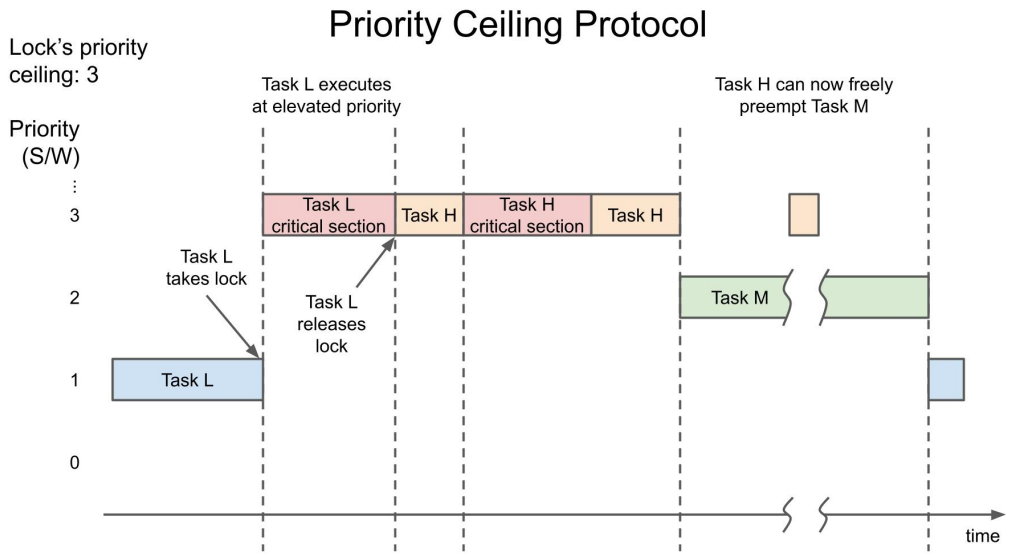
Unbounded priority inversion occurs when a medium priority task (Task M) interrupts Task L while it holds the lock. It’s called “unbounded” because Task M can now effectively block Task H for any amount of time, as Task M is preempting Task L (which still holds the lock).



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. You can read about [the mission and bug here](#).

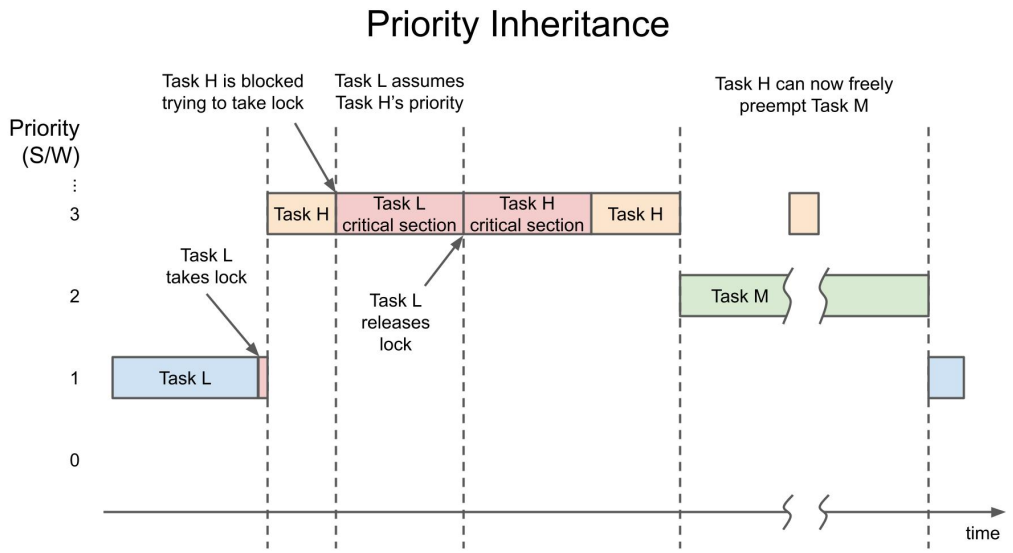
There are a few ways to combat unbounded priority inversion. Two popular methods include priority ceiling protocol and priority inheritance.

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.

Another method, known as “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.



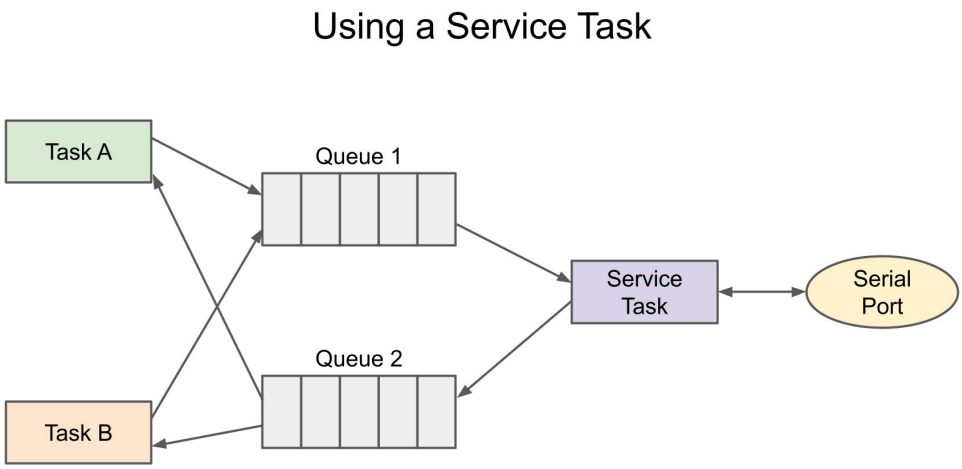
Task L takes the lock. Only when Task H attempts to take the lock is the priority of Task L boosted to that of Task H's. Once again, Task M can no longer interrupt Task L until both tasks are finished in the critical section.

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.



While it requires extra overhead to implement another task, we can effectively avoid needing critical sections for the serial port (or other resource) using this method.

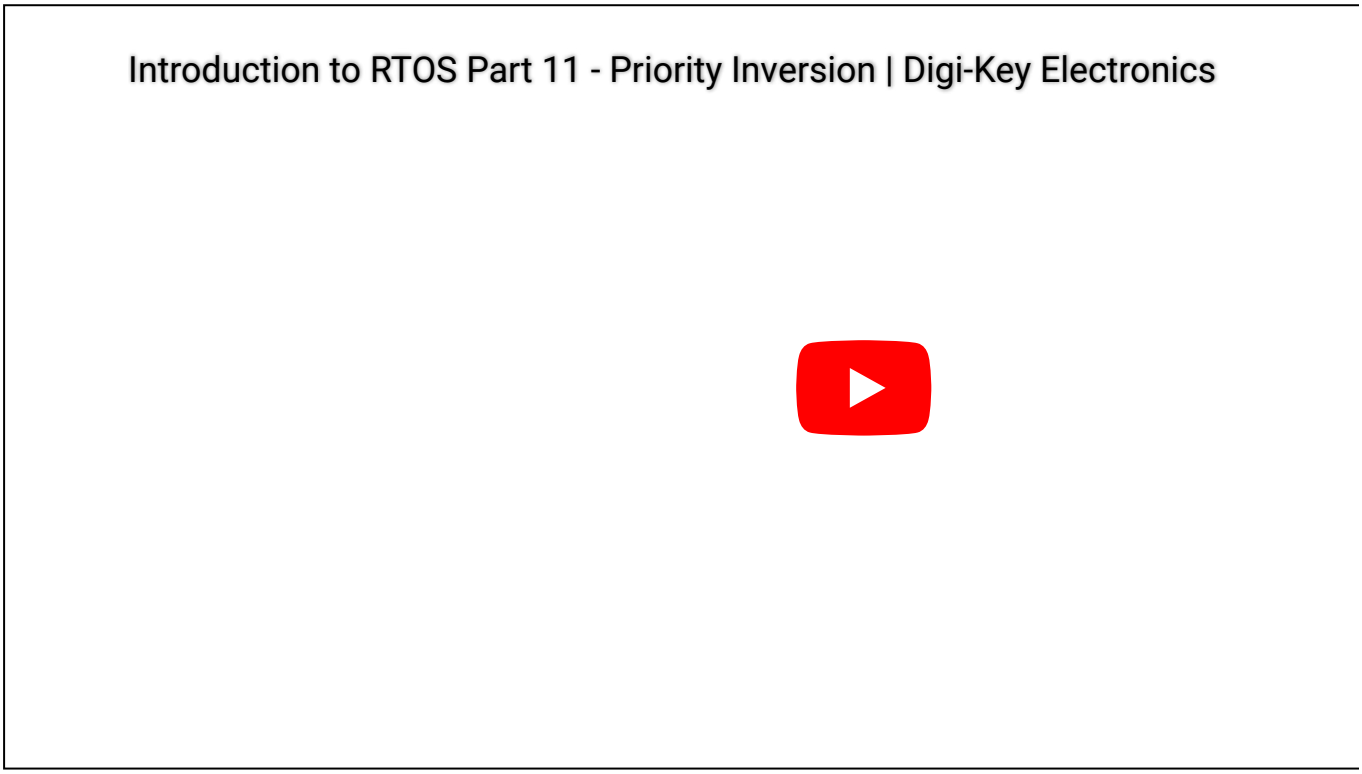
Required Hardware

Any ESP32 development board should work, so long as it's supported in the Arduino IDE. See here for a list of supported ESP32 boards. You may also use any development board capable of running FreeRTOS, although my solution will likely vary some (as the ESP32 runs a modified version of FreeRTOS called ESP-IDF).

This solution uses the Adafruit Feather HUZZAH32.

Video

If you have not done so, please watch the following video, which provides the steps necessary to creating tasks and assigning priorities. It also demonstrates a working version of the challenge:



Challenge

Start with the priority inversion code below. If you run it, you'll find that Task H is blocked for around 5 seconds waiting for a mutex to be released from Task L. Task M interrupts task L for those 5 seconds, causing unbounded priority inversion.

Your job is to use the critical section guards to prevent the scheduler from interrupting during the critical section (empty while loop) in Task H and Task L. These guards include portENTER_CRITICAL() and portEXIT_CRITICAL() for the ESP32 or taskENTER_CRITICAL() and taskEXIT_CRITICAL() for vanilla FreeRTOS.

Copy Code

```
/**
 * ESP32 Priority Inversion Demo
 *
 * Demonstrate priority inversion.
 *
 * Date: February 12, 2021
 * Author: Shawn Hymel
 * License: 0BSD
 */

// You'll likely need this on vanilla FreeRTOS
//#include semphr.h

// Use only core 1 for demo purposes
#if CONFIG_FREERTOS_UNICORE
  static const BaseType_t app_cpu = 0;
#else
```

```
  static const BaseType_t app_cpu = 1;
#endif

// Settings
TickType_t cs_wait = 250;   // Time spent in critical section (ms)
TickType_t med_wait = 5000; // Time medium task spends working (ms)

// Globals
static SemaphoreHandle_t lock;

//*****************************************************************************
// Tasks

// Task L (low priority)
void doTaskL(void *parameters) {

  TickType_t timestamp;

  // Do forever
  while (1) {

    // Take lock
    Serial.println("Task L trying to take lock...");
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    xSemaphoreTake(lock, portMAX_DELAY);

    // Say how long we spend waiting for a lock
    Serial.print("Task L got lock. Spent ");
    Serial.print((xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp);
    Serial.println(" ms waiting for lock. Doing some work...");

    // Hog the processor for a while doing nothing
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    while ( (xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp < cs_wait);

    // Release lock
    Serial.println("Task L releasing lock.");
    xSemaphoreGive(lock);

    // Go to sleep
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

// Task M (medium priority)
void doTaskM(void *parameters) {

  TickType_t timestamp;

  // Do forever
  while (1) {

    // Hog the processor for a while doing nothing
    Serial.println("Task M doing some work...");
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    while ( (xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp < med_wait);

    // Go to sleep
    Serial.println("Task M done!");
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

// Task H (high priority)
void doTaskH(void *parameters) {

  TickType_t timestamp;

  // Do forever
  while (1) {

    // Take lock
    Serial.println("Task H trying to take lock...");
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    xSemaphoreTake(lock, portMAX_DELAY);
```

```
    // Say how long we spend waiting for a lock
    Serial.print("Task H got lock. Spent ");
    Serial.print((xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp);
    Serial.println(" ms waiting for lock. Doing some work...");

    // Hog the processor for a while doing nothing
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    while ( (xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp < cs_wait);

    // Release lock
    Serial.println("Task H releasing lock.");
    xSemaphoreGive(lock);

    // Go to sleep
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

//*****************************************************************************
// Main (runs as its own task with priority 1 on core 1)

void setup() {

  // Configure Serial
  Serial.begin(115200);

  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);
  Serial.println();
  Serial.println("---FreeRTOS Priority Inversion Demo---");

  // Create semaphores and mutexes before starting tasks
  lock = xSemaphoreCreateBinary();
  xSemaphoreGive(lock); // Make sure binary semaphore starts at 1

  // The order of starting the tasks matters to force priority inversion

  // Start Task L (low priority)
  xTaskCreatePinnedToCore(doTaskL,
                          "Task L",
                          1024,
                          NULL,
                          1,
                          NULL,
                          app_cpu);

  // Introduce a delay to force priority inversion
  vTaskDelay(1 / portTICK_PERIOD_MS);

  // Start Task H (high priority)
  xTaskCreatePinnedToCore(doTaskH,
                          "Task H",
                          1024,
                          NULL,
                          3,
                          NULL,
                          app_cpu);

  // Start Task M (medium priority)
  xTaskCreatePinnedToCore(doTaskM,
                          "Task M",
                          1024,
                          NULL,
                          2,
                          NULL,
                          app_cpu);

  // Delete "setup and loop" task
  vTaskDelete(NULL);
}

void loop() {
```

```
  // Execution should never get here
}
```

Solution

Here is one possible solution to the challenge:

Copy Code

```
/**
 * ESP32 Critical Section Demo
 *
 * Demonstrate how priority inversion can be avoided through the use of
 * critical sections.
 *
 * Date: February 12, 2021
 * Author: Shawn Hymel
 * License: 0BSD
 */

// You'll likely need this on vanilla FreeRTOS
//#include semphr.h

// Use only core 1 for demo purposes
#if CONFIG_FREERTOS_UNICORE
  static const BaseType_t app_cpu = 0;
#else
  static const BaseType_t app_cpu = 1;
#endif

// Settings
TickType_t cs_wait = 250;   // Time spent in critical section (ms)
TickType_t med_wait = 5000; // Time medium task spends working (ms)

// Globals
static portMUX_TYPE spinlock = portMUX_INITIALIZER_UNLOCKED;

//*****************************************************************************
// Tasks

// Task L (low priority)
void doTaskL(void *parameters) {

  TickType_t timestamp;

  // Do forever
  while (1) {

    // Take lock
    Serial.println("Task L trying to take lock...");
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    portENTER_CRITICAL(&spinlock);  // taskENTER_CRITICAL() in vanilla FreeRTOS

    // Say how long we spend waiting for a lock
    Serial.print("Task L got lock. Spent ");
    Serial.print((xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp);
    Serial.println(" ms waiting for lock. Doing some work...");

    // Hog the processor for a while doing nothing
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    while ( (xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp < cs_wait);

    // Release lock
    Serial.println("Task L releasing lock.");
    portEXIT_CRITICAL(&spinlock);  // taskEXIT_CRITICAL() in vanilla FreeRTOS

    // Go to sleep
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

// Task M (medium priority)
void doTaskM(void *parameters) {
```

```
  TickType_t timestamp;

  // Do forever
  while (1) {

    // Hog the processor for a while doing nothing
    Serial.println("Task M doing some work...");
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    while ( (xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp < med_wait);

    // Go to sleep
    Serial.println("Task M done!");
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

// Task H (high priority)
void doTaskH(void *parameters) {

  TickType_t timestamp;

  // Do forever
  while (1) {

    // Take lock
    Serial.println("Task H trying to take lock...");
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    portENTER_CRITICAL(&spinlock);  // taskENTER_CRITICAL() in vanilla FreeRTOS

    // Say how long we spend waiting for a lock
    Serial.print("Task H got lock. Spent ");
    Serial.print((xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp);
    Serial.println(" ms waiting for lock. Doing some work...");

    // Hog the processor for a while doing nothing
    timestamp = xTaskGetTickCount() * portTICK_PERIOD_MS;
    while ( (xTaskGetTickCount() * portTICK_PERIOD_MS) - timestamp < cs_wait);

    // Release lock
    Serial.println("Task H releasing lock.");
    portEXIT_CRITICAL(&spinlock);  // taskEXIT_CRITICAL() in vanilla FreeRTOS

    // Go to sleep
    vTaskDelay(500 / portTICK_PERIOD_MS);
  }
}

//*****************************************************************************
// Main (runs as its own task with priority 1 on core 1)

void setup() {

  // Configure Serial
  Serial.begin(115200);

  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);
  Serial.println();
  Serial.println("---FreeRTOS Critical Section Demo---");

  // The order of starting the tasks matters to force priority inversion

  // Start Task L (low priority)
  xTaskCreatePinnedToCore(doTaskL,
                          "Task L",
                          1024,
                          NULL,
                          1,
                          NULL,
                          app_cpu);

  // Introduce a delay to force priority inversion
  vTaskDelay(1 / portTICK_PERIOD_MS);
```

```
  // Start Task H (high priority)
  xTaskCreatePinnedToCore(doTaskH,
                          "Task H",
                          1024,
                          NULL,
                          3,
                          NULL,
                          app_cpu);

  // Start Task M (medium priority)
  xTaskCreatePinnedToCore(doTaskM,
                          "Task M",
                          1024,
                          NULL,
                          2,
                          NULL,
                          app_cpu);

  // Delete "setup and loop" task
  vTaskDelete(NULL);
}

void loop() {
  // Execution should never get here
}
```

Explanation

For the ESP32, note that we need to create spinlock as a global variable:

Copy Code

```
static portMUX_TYPE spinlock = portMUX_INITIALIZER_UNLOCKED;
```

This spinlock is used to prevent the other core from entering the critical section (assuming we're using the other core).

In both Task L and Task H, we add the following line when entering the critical section:

Copy Code

```
portENTER_CRITICAL(&spinlock);  // taskENTER_CRITICAL() in vanilla FreeRTOS
```

This line disables the scheduler for that core and disables hardware interrupts (up to a certain priority level). It also "takes" a spinlock. If another task (in the other core) attempts to take the spinlock, that task will wait doing nothing (busy wait or spin) until the lock is made available.

We run the critical section, which prints out how long it took us to get the lock and busy-waits with an empty while loop for a while. When it's done, we exit the critical section with the following:

Copy Code

```
portEXIT_CRITICAL(&spinlock);  // taskEXIT_CRITICAL() in vanilla FreeRTOS
```

This technique prevents unbounded priority inversion, but it may not be the best approach, as it disables interrupts (in a core) to do so. However, it's one possible approach if you cannot use priority inheritance (as covered in the video). Sometimes, an RTOS does not allow for dynamic priority assignment, so priority inheritance will not work.

Recommended Reading

All demonstrations and solutions for this course can be found in this GitHub repository.

- How to use priority inheritance: https://www.embedded.com/how-to-use-priority-inheritance/
- Priority inversion: https://barrgroup.com/embedded-systems/how-to/rtos-priority-inversion

- What really happened on Mars rover Pathfinder: http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html
- What is a Real-Time Operating System Part 1(RTOS)?
- Introduction to RTOS - Solution to Part 2 (FreeRTOS)
- Introduction to RTOS - Solution to Part 3 (Task Scheduling)
- Introduction to RTOS - Solution to Part 4 (Memory Management)
- Introduction to RTOS - Solution to Part 5 (FreeRTOS Queue Example)
- Introduction to RTOS - Solution to Part 6 (FreeRTOS Mutex Example)
- Introduction to RTOS - Solution to Part 7 (FreeRTOS Semaphore Example)
- Introduction to RTOS - Solution to Part 8 (Software Timers)
- Introduction to RTOS - Solution to Part 9 (Hardware Interrupts)
- Introduction to RTOS - Solution to Part 10 (Deadlock and Starvation)
- Introduction to RTOS - Solution to Part 12 (Multicore Systems)

Project Details

Platforms

Arduino

Development

C

C++

Tags

Arduino

RTOS

License

Attribution

Get Involved

Like

Save