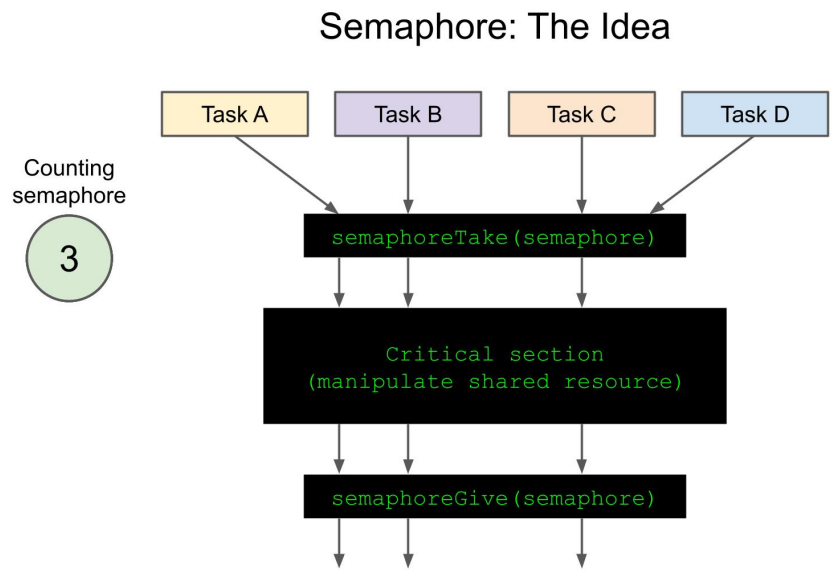
Introduction to RTOS - Solution to Part 7 (FreeRTOS Semaphore Example)

By ShawnHymel

Concepts

In programming, a semaphore is a variable used to control access to a common, shared resource that needs to be accessed by multiple threads or processes. It is similar to a mutex in that it can prevent other threads from accessing a shared resource or critical section.

However, where a mutex implies a level of ownership of the lock (i.e. a single thread is said to have the lock while executing in a critical section), a semaphore is a counter that can allow multiple threads to enter a critical section.



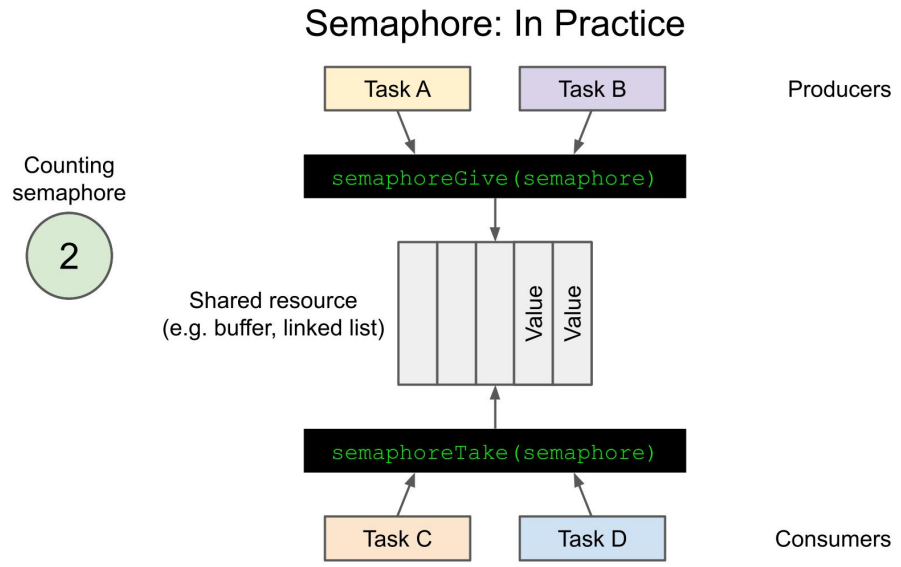
In theory, a semaphore is a shared counter that can be incremented and decremented atomically. For example, Tasks A, B, and C wish to enter the critical section in the image above. They each call semaphoreTake(), which decrements the counting semaphore. At this point, all 3 tasks are inside the critical section and the semaphore's value is 0.

If another task, like Task D, attempts to enter the critical section, it must first call semaphoreTake() as well. However, since the semaphore is 0, semaphoreTake() will tell Task D to wait. When any of the other tasks leave the critical section, they must call semaphoreGive(), which atomically increments the semaphore. At this point, Task D may call semaphoreTake() again and enter the critical section.

This demonstrates that semaphores work like a generalized mutex capable of counting to numbers greater than 1. However, in practice, you rarely use semaphores like this, as even within the critical section, there is no way to protect individual resources with just that one semaphore; you would need to use other protection mechanisms, like queues or mutexes to supplement the semaphore.

In practice, semaphores are often used to signal to other threads that some common resource is available to use. This works well in producer/consumer scenarios where one or more tasks generate data and one or

more tasks use that data.



In the example above, Tasks A and B (producers) create data to be put into a shared resource, such as a buffer or linked list. Each time they do, they call semaphoreGive() to increment the semaphore. Tasks C and D can read values from that resource, removing the values as they do so (consumers). Each time a task reads from the resource, it calls semaphoreTake(), which decrements the semaphore value.

Note that the shared resource must still be protected to prevent overwrites from the producer threads. A semaphore is then used as an additional signal to the consumer tasks that values are ready. By limiting the maximum value of the semaphore, we can control how much information can be put into or read from the resource at a time (i.e. the size of the buffer or maximum length of the linked list).

You can have a binary semaphore, which only counts to 1. Hopefully, you can see how a binary semaphore differs from a mutex in its use cases. While they might have similar implementations, the use cases for mutexes and semaphores differ.

A mutex, as mentioned earlier, implies that the same thread must “take” and “give” the mutex, which means the thread “owns” the mutex during critical section execution. A semaphore is given and taken by different threads, which means it’s better used as a signaling mechanism to synchronize threads.

Because you do not want an interrupt service routine (ISR) to block (hogging the CPU), you generally do not want to use a mutex in an ISR. However, you could use a semaphore in an ISR to signal to other threads that it has executed and some data is ready for consumption.

In FreeRTOS (and some other RTOSes), a mutex involves priority inheritance for the calling thread. If a high priority thread is waiting for a lock from a low priority thread, the low priority thread's priority is raised to be equal to or above the waiting thread so that it can quickly finish executing the critical section and release the lock. This helps alleviate the problem of priority inversion.

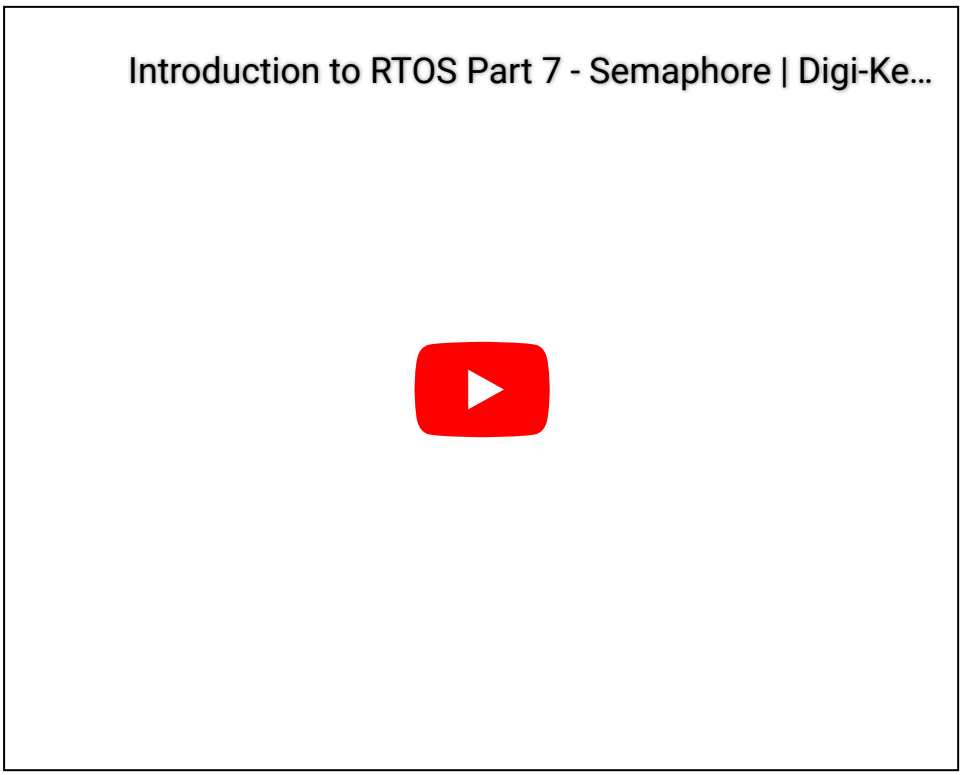
Required Hardware

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

Starting with the code below, add 2 semaphores and mutex to protect and synchronize the circular buffer.

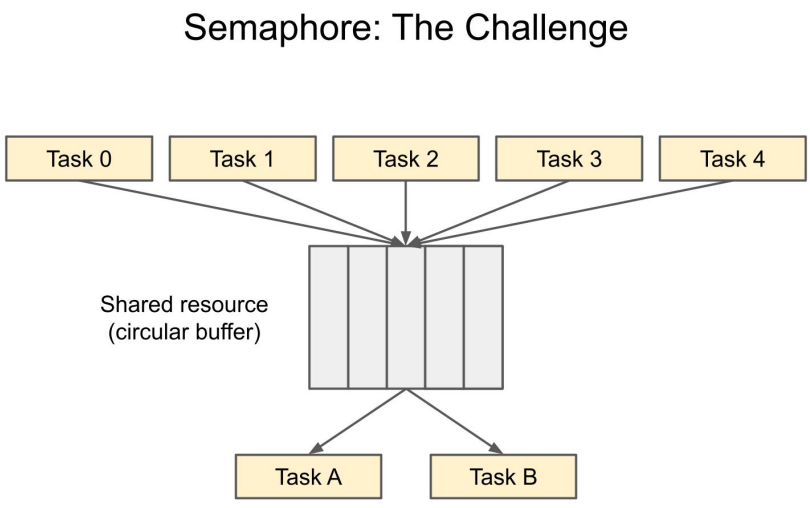
Copy Code

```
/**
 * FreeRTOS Counting Semaphore Challenge
 *
 * Challenge: use a mutex and counting semaphores to protect the shared buffer
 * so that each number (0 throguh 4) is printed exactly 3 times to the Serial
 * monitor (in any order). Do not use queues to do this!
 *
 * Hint: you will need 2 counting semaphores in addition to the mutex, one for
 * remembering number of filled slots in the buffer and another for
 * remembering the number of empty slots in the buffer.
 *
 * Date: January 24, 2021
 * Author: Shawn Hymel
 * License: 0BSD
 */
```

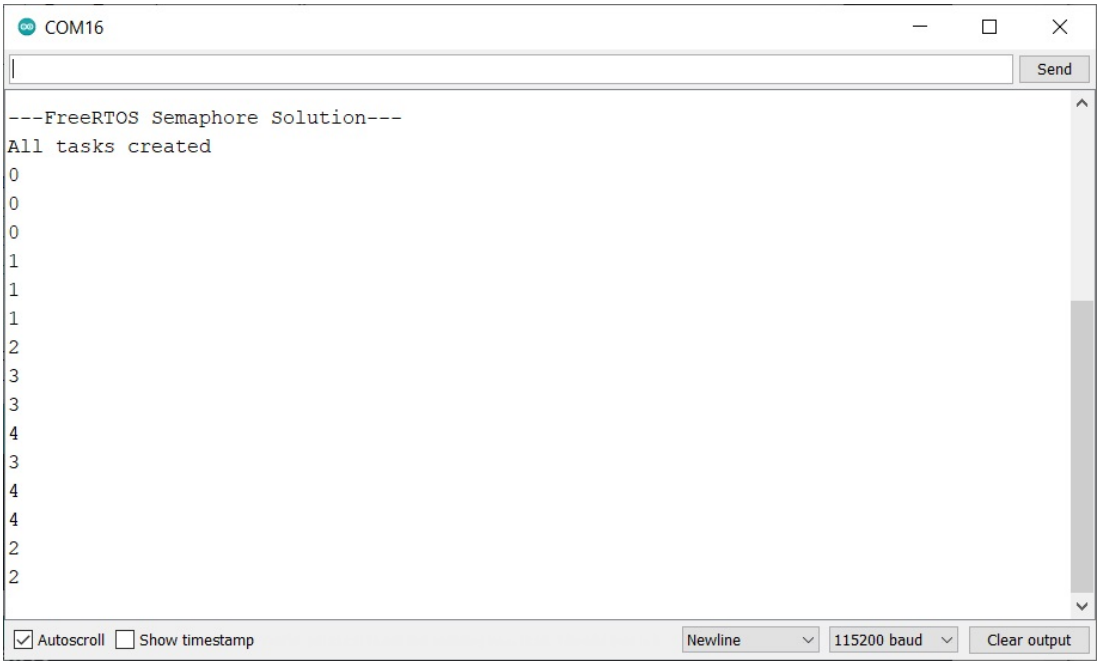
```
// You'll likely need this on vanilla FreeRTOS
//#include semphr.h

// 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
enum {BUF_SIZE = 5};                    // Size of buffer array
static const int num_prod_tasks = 5;  // Number of producer tasks
static const int num_cons_tasks = 2;  // Number of consumer tasks
static const int num_writes = 3;      // Num times each producer writes to buf

// Globals
static int buf[BUF_SIZE];             // Shared buffer
static int head = 0;                  // Writing index to buffer
static int tail = 0;                  // Reading index to buffer
static SemaphoreHandle_t bin_sem;     // Waits for parameter to be read

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

// Producer: write a given number of times to shared buffer
void producer(void *parameters) {

  // Copy the parameters into a local variable
  int num = *(int *)parameters;

  // Release the binary semaphore
  xSemaphoreGive(bin_sem);

  // Fill shared buffer with task number
  for (int i = 0; i < num_writes; i++) {

    // Critical section (accessing shared buffer)
    buf[head] = num;
    head = (head + 1) % BUF_SIZE;
  }

  // Delete self task
  vTaskDelete(NULL);
}

// Consumer: continuously read from shared buffer
void consumer(void *parameters) {

  int val;

  // Read from buffer
  while (1) {

    // Critical section (accessing shared buffer and Serial)
    val = buf[tail];
    tail = (tail + 1) % BUF_SIZE;
    Serial.println(val);
  }
}

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

void setup() {

  char task_name[12];

  // 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 Semaphore Alternate Solution---");
```

```
  // Create mutexes and semaphores before starting tasks
  bin_sem = xSemaphoreCreateBinary();

  // Start producer tasks (wait for each to read argument)
  for (int i = 0; i < num_prod_tasks; i++) {
    sprintf(task_name, "Producer %i", i);
    xTaskCreatePinnedToCore(producer,
                            task_name,
                            1024,
                            (void *)&i,
                            1,
                            NULL,
                            app_cpu);
    xSemaphoreTake(bin_sem, portMAX_DELAY);
  }

  // Start consumer tasks
  for (int i = 0; i < num_cons_tasks; i++) {
    sprintf(task_name, "Consumer %i", i);
    xTaskCreatePinnedToCore(consumer,
                            task_name,
                            1024,
                            NULL,
                            1,
                            NULL,
                            app_cpu);
  }

  // Notify that all tasks have been created
  Serial.println("All tasks created");
}

void loop() {

  // Do nothing but allow yielding to lower-priority tasks
  vTaskDelay(1000 / portTICK_PERIOD_MS);
}
```

Tasks 0-4 want to write values to the ring buffer, and Tasks A and B wish to read values from it. The ring buffer has already been implemented (in a basic form--there is nothing to prevent the head or tail from overwriting each other, as that's your job to accomplish using semaphores).



When it's done, the numbers 0, 1, 2, 3, 4 should be printed out exactly 3 times each to the serial console (in no particular order).



Solution

Spoilers below! I highly encourage you to try the challenge on your own before comparing your answer to mine. Note that my solution may not be the only way to solve the challenge.

Copy Code

```
/**
 * FreeRTOS Counting Semaphore Solution
 *
 * Use producer tasks (writing to shared memory) and consumer tasks (reading
 * from shared memory) to demonstrate counting semaphores.
 *
 * Date: January 24, 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
enum {BUF_SIZE = 5};                  // Size of buffer array
static const int num_prod_tasks = 5;  // Number of producer tasks
static const int num_cons_tasks = 2;  // Number of consumer tasks
static const int num_writes = 3;      // Num times each producer writes to buf

// Globals
static int buf[BUF_SIZE];             // Shared buffer
static int head = 0;                  // Writing index to buffer
static int tail = 0;                  // Reading index to buffer
static SemaphoreHandle_t bin_sem;     // Waits for parameter to be read
static SemaphoreHandle_t mutex;       // Lock access to buffer and Serial
static SemaphoreHandle_t sem_empty;   // Counts number of empty slots in buf
static SemaphoreHandle_t sem_filled;  // Counts number of filled slots in buf

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

// Producer: write a given number of times to shared buffer
void producer(void *parameters) {
```

```
  // Copy the parameters into a local variable
  int num = *(int *)parameters;

  // Release the binary semaphore
  xSemaphoreGive(bin_sem);

  // Fill shared buffer with task number
  for (int i = 0; i < num_writes; i++) {

    // Wait for empty slot in buffer to be available
    xSemaphoreTake(sem_empty, portMAX_DELAY);

    // Lock critical section with a mutex
    xSemaphoreTake(mutex, portMAX_DELAY);
    buf[head] = num;
    head = (head + 1) % BUF_SIZE;
    xSemaphoreGive(mutex);

    // Signal to consumer tasks that a slot in the buffer has been filled
    xSemaphoreGive(sem_filled);
  }

  // Delete self task
  vTaskDelete(NULL);
}

// Consumer: continuously read from shared buffer
void consumer(void *parameters) {

  int val;

  // Read from buffer
  while (1) {

    // Wait for at least one slot in buffer to be filled
    xSemaphoreTake(sem_filled, portMAX_DELAY);

    // Lock critical section with a mutex
    xSemaphoreTake(mutex, portMAX_DELAY);
    val = buf[tail];
    tail = (tail + 1) % BUF_SIZE;
    Serial.println(val);
    xSemaphoreGive(mutex);

    // Signal to producer thread that a slot in the buffer is free
    xSemaphoreGive(sem_empty);
  }
}

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

void setup() {

  char task_name[12];

  // 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 Semaphore Solution---");

  // Create mutexes and semaphores before starting tasks
  bin_sem = xSemaphoreCreateBinary();
  mutex = xSemaphoreCreateMutex();
  sem_empty = xSemaphoreCreateCounting(BUF_SIZE, BUF_SIZE);
  sem_filled = xSemaphoreCreateCounting(BUF_SIZE, 0);

  // Start producer tasks (wait for each to read argument)
  for (int i = 0; i < num_prod_tasks; i++) {
    sprintf(task_name, "Producer %i", i);
    xTaskCreatePinnedToCore(producer,
                            task_name,
```

```
                            1024,
                            (void *)&i,
                            1,
                            NULL,
                            app_cpu);
    xSemaphoreTake(bin_sem, portMAX_DELAY);
  }

  // Start consumer tasks
  for (int i = 0; i < num_cons_tasks; i++) {
    sprintf(task_name, "Consumer %i", i);
    xTaskCreatePinnedToCore(consumer,
                            task_name,
                            1024,
                            NULL,
                            1,
                            NULL,
                            app_cpu);
  }

  // Notify that all tasks have been created (lock Serial with mutex)
  xSemaphoreTake(mutex, portMAX_DELAY);
  Serial.println("All tasks created");
  xSemaphoreGive(mutex);
}

void loop() {

  // Do nothing but allow yielding to lower-priority tasks
  vTaskDelay(1000 / portTICK_PERIOD_MS);
}
```

Explanation

To effectively protect the circular buffer, we need 1 mutex and 2 semaphores, so we handles to them as global variables.

Copy Code

```
static SemaphoreHandle_t mutex;       // Lock access to buffer and Serial
static SemaphoreHandle_t sem_empty;   // Counts number of empty slots in buf
static SemaphoreHandle_t sem_filled;  // Counts number of filled slots in buf
```

In the producer threads, we add the 2 semaphores around the critical section. The first semaphore waits for an empty slot in the buffer. The second semaphore is incremented after the critical section and signals that a slot in the buffer has been filled. The mutex simply protects the critical section so producer tasks do not overwrite each other.

Copy Code

```
// Fill shared buffer with task number
for (int i = 0; i < num_writes; i++) {

  // Wait for empty slot in buffer to be available
  xSemaphoreTake(sem_empty, portMAX_DELAY);

  // Lock critical section with a mutex
  xSemaphoreTake(mutex, portMAX_DELAY);
  buf[head] = num;
  head = (head + 1) % BUF_SIZE;
  xSemaphoreGive(mutex);

  // Signal to consumer tasks that a slot in the buffer has been filled
  xSemaphoreGive(sem_filled);
}
```

We do the opposite in the consumer thread. We first wait for one of the slots in the buffer to be filled before executing the critical section. After reading (consuming) a value, we increment the semaphore that counts empty slots. We also protect the buffer and Serial object with the mutex.

Copy Code

```
// Read from buffer
while (1) {

  // Wait for at least one slot in buffer to be filled
  xSemaphoreTake(sem_filled, portMAX_DELAY);

  // Lock critical section with a mutex
  xSemaphoreTake(mutex, portMAX_DELAY);
  val = buf[tail];
  tail = (tail + 1) % BUF_SIZE;
  Serial.println(val);
  xSemaphoreGive(mutex);

  // Signal to producer thread that a slot in the buffer is free
  xSemaphoreGive(sem_empty);
}
```

In setup(), we create the mutex and semaphores.

Copy Code

```
mutex = xSemaphoreCreateMutex();
sem_empty = xSemaphoreCreateCounting(BUF_SIZE, BUF_SIZE);
sem_filled = xSemaphoreCreateCounting(BUF_SIZE, 0);
```

The mutex is initialized to 1 by default. We set the maximum value of both semaphores to 5 (the size of the buffer). One semaphore (sem_empty) is used to count the number of empty slots in the buffer, and we initialize it to 5, as the buffer starts completely empty. The other semaphore (sem_filled) counts the number of filled slots, so we initialize it to 0.

You can set the semaphore maximum values to less than the buffer size (5) but not greater than the buffer size. Try it to see how it affects execution!

I also added mutex protection around the Serial.println() line after the tasks have been created. You don't need to do this, but it can be useful to see how to protect the Serial object as a shared resource.

Copy Code

```
// Notify that all tasks have been created (lock Serial with mutex)
xSemaphoreTake(mutex, portMAX_DELAY);
Serial.println("All tasks created");
xSemaphoreGive(mutex);
```

Bonus Solution!

If using a mutex and two semaphores to protect some data being passed between threads seems overly complicated, you're right! Semaphores have their uses, but you'll often find that you can pass data more easily using a queue. As a result, here's a much better way to accomplish the above challenge using just one queue.

Copy Code

```
/**
 * FreeRTOS Queue Alternate Solution
 *
 * Demonstrate how it's often easier to use queues instead of counting
 * semaphores to pass information between tasks.
 *
 * Date: January 24, 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
static const uint8_t queue_len = 10;  // Size of queue
static const int num_prod_tasks = 5;  // Number of producer tasks
static const int num_cons_tasks = 2;  // Number of consumer tasks
static const int num_writes = 3;      // Num times each producer writes to buf

// Globals
static SemaphoreHandle_t bin_sem;     // Waits for parameter to be read
static SemaphoreHandle_t mutex;       // Lock access to Serial resource
static QueueHandle_t msg_queue;       // Send data from producer to consumer

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

// Producer: write a given number of times to shared buffer
void producer(void *parameters) {

  // Copy the parameters into a local variable
  int num = *(int *)parameters;

  // Release the binary semaphore
  xSemaphoreGive(bin_sem);

  // Fill queue with task number (wait max time if queue is full)
  for (int i = 0; i < num_writes; i++) {
    xQueueSend(msg_queue, (void *)&num, portMAX_DELAY);
  }

  // Delete self task
  vTaskDelete(NULL);
}

// Consumer: continuously read from shared buffer
void consumer(void *parameters) {

  int val;

  // Read from buffer
  while (1) {

    // Read from queue (wait max time if queue is empty)
    xQueueReceive(msg_queue, (void *)&val, portMAX_DELAY);

    // Lock Serial resource with a mutex
    xSemaphoreTake(mutex, portMAX_DELAY);
    Serial.println(val);
    xSemaphoreGive(mutex);
  }
}

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

void setup() {

  char task_name[12];

  // 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 Semaphore Solution---");

  // Create mutexes and semaphores before starting tasks
  bin_sem = xSemaphoreCreateBinary();
  mutex = xSemaphoreCreateMutex();
```

```
  // Create queue
  msg_queue = xQueueCreate(queue_len, sizeof(int));

  // Start producer tasks (wait for each to read argument)
  for (int i = 0; i < num_prod_tasks; i++) {
    sprintf(task_name, "Producer %i", i);
    xTaskCreatePinnedToCore(producer,
                            task_name,
                            1024,
                            (void *)&i,
                            1,
                            NULL,
                            app_cpu);
    xSemaphoreTake(bin_sem, portMAX_DELAY);
  }

  // Start consumer tasks
  for (int i = 0; i < num_cons_tasks; i++) {
    sprintf(task_name, "Consumer %i", i);
    xTaskCreatePinnedToCore(consumer,
                            task_name,
                            1024,
                            NULL,
                            1,
                            NULL,
                            app_cpu);
  }

  // Notify that all tasks have been created (lock Serial with mutex)
  xSemaphoreTake(mutex, portMAX_DELAY);
  Serial.println("All tasks created");
  xSemaphoreGive(mutex);
}

void loop() {

  // Do nothing but allow yielding to lower-priority tasks
  vTaskDelay(1000 / portTICK_PERIOD_MS);
}
```

Recommended Reading

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

Note that recent versions of FreeRTOS include a new type of inter-task communication called Task Notifications. These work similar to semaphores but must be sent to only one task at a time. They are supposedly more efficient than mutexes/semaphores and are worth checking out if you're working with FreeRTOS.

If you would like to dig into these topics further, I recommend checking out the following excellent articles:

- Difference between "lock," "mutex," and "semaphore:" https://stackoverflow.com/questions/2332765/lock-mutex-semaphore-whats-the-difference
- Mutexes and Semaphores Demystified: https://barrgroup.com/embedded-systems/how-to/rtos-mutex-semaphore
- FreeRTOS Semaphores (and Mutexes) API reference: https://www.freertos.org/a00113.html
- What is a Real-Time Operating System Part 1(RTOS)?
- Inthttps://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-4-memory-management/6d4dfcaa1ff84f57a2098da8e6401d9croduction 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 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 11 (Priority Inversion)
- Introduction to RTOS - Solution to Part 12 (Multicore Systems)

Key Parts and Components

1 Items

Mfr Part # 3405

HUZZAH32 ESP32 FEATHER LOOSE HDR

Adafruit Industries LLC

$19.95 Details

Add all Digi-Key Parts to Cart

TechForum

Have questions or comments? Continue the conversation on TechForum, DigiKey's online community and technical resource.

Visit TechForum

Arduino | 3D Printing | Raspberry Pi

Project Details

Platforms

Arduino

Development

C++

Tags

Arduino

RTOS

License

Attribution

Get Involved

Like

Save

1-800-344-4539 218-681-6674 sales@digikey.com 218-681-3380

Local Support: 701 Brooks Avenue South, Thief River Falls, MN 56701 USA