Introduction to RTOS - Solution to Part 10 (Deadlock and Starvation)

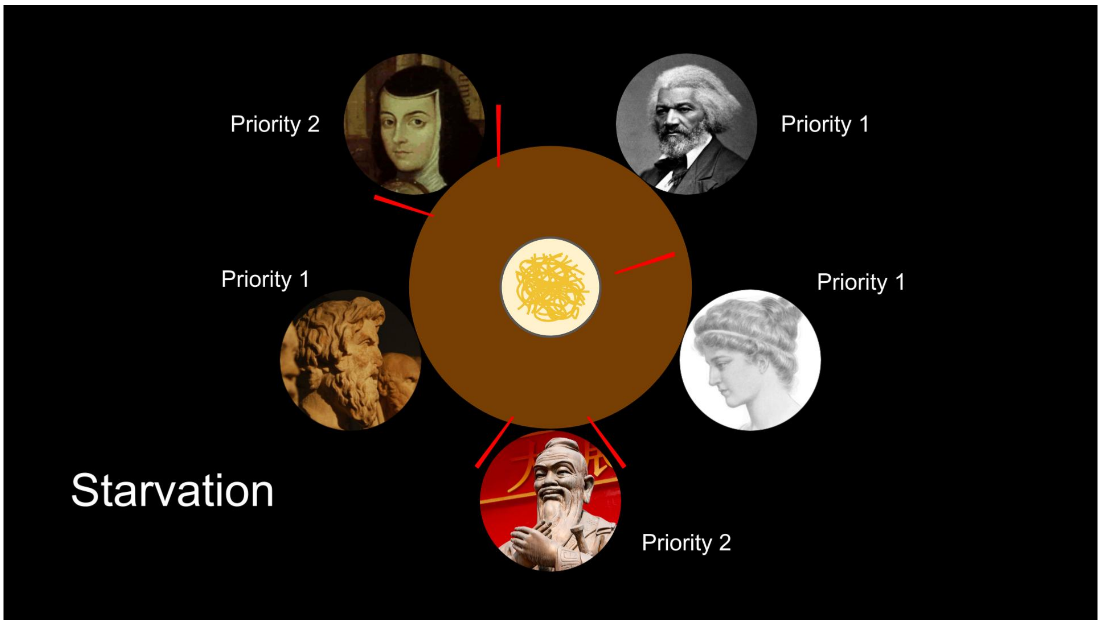
By ShawnHymel

Concepts

Imagine 5 philosophers sitting at a round table with a bowl of noodles in the middle of the table. A chopstick is placed in between each philosopher. A philosopher may only eat if they are holding two chopsticks. When they are done eating, they put the chopsticks down to let another philosopher eat.

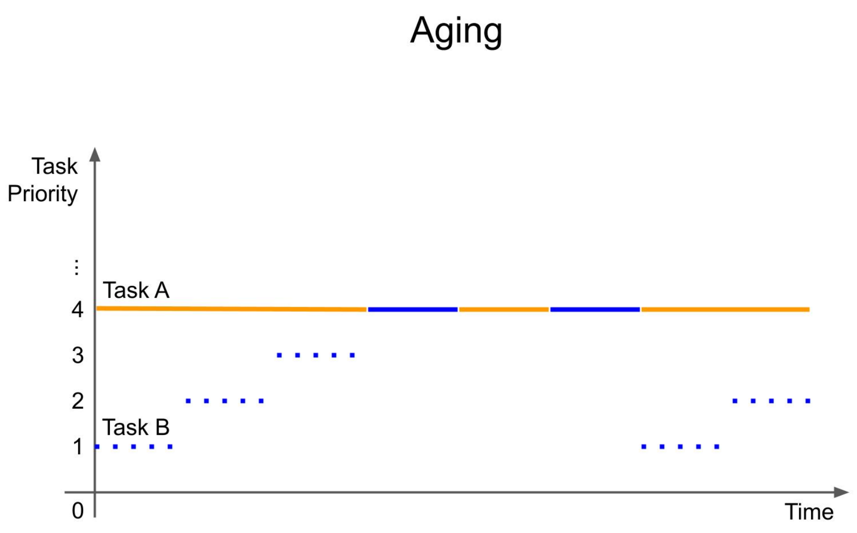
The idea is to come up with an algorithm to allow all the philosophers a chance to eat. This is a description of the classic Dining Philosophers problem posed by Edsger Dijkstra to his computer science students in 1965. It's an analogy to demonstrate starvation and deadlock in a multi-threaded system. The philosophers are like tasks (or threads) trying to perform some job with a shared resource (the bowl of noodles). The chopsticks are analogous to locks (semaphores or mutexes) that are needed prior to accessing the shared resource.

If a couple of the philosophers have higher priority than the rest and never yield the chops, then the rest of the philosophers (threads) are in danger of never eating. This is known as "starvation."



There are a few ways to combat starvation. The first is to simply make sure that your high priority tasks always yield some time to the processor (e.g. by waiting for a mutex/semaphore or through a delay function to put itself to sleep).

The second method can be built into the scheduler or rely on a higher priority task to monitor how long other tasks have been asleep. If a lower-priority task has been asleep (blocked) for some time, its priority level is gradually raised until it gets a chance to run. Once it has run for some time, its priority level is dropped back to its original level. This is known as "aging."

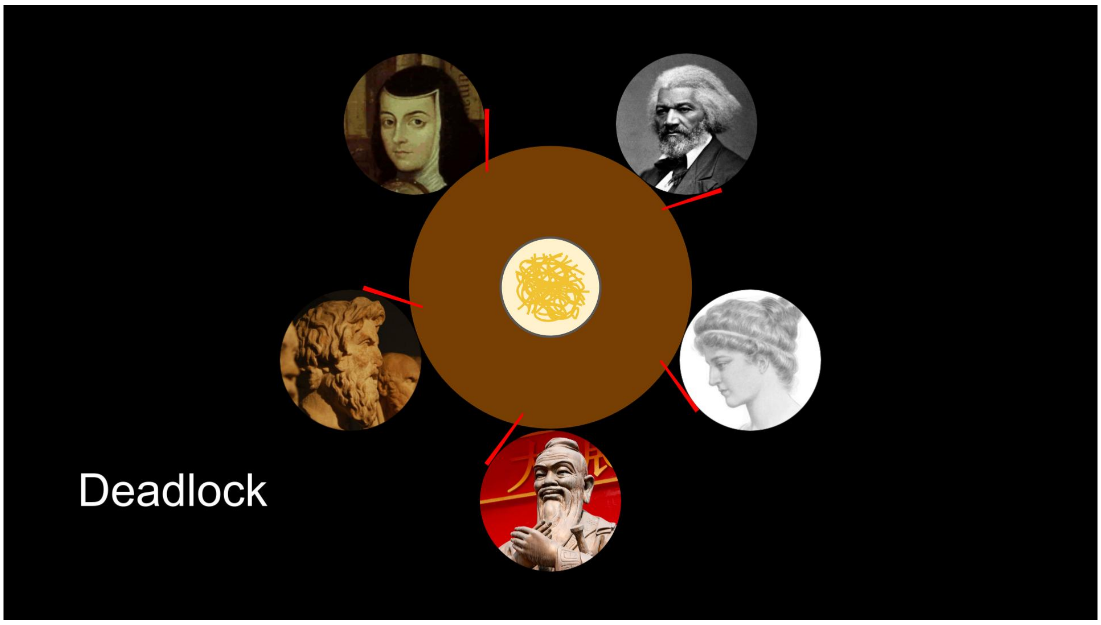


Even with aging, another problem may occur. Let's say that our algorithm for each philosopher is as follows:

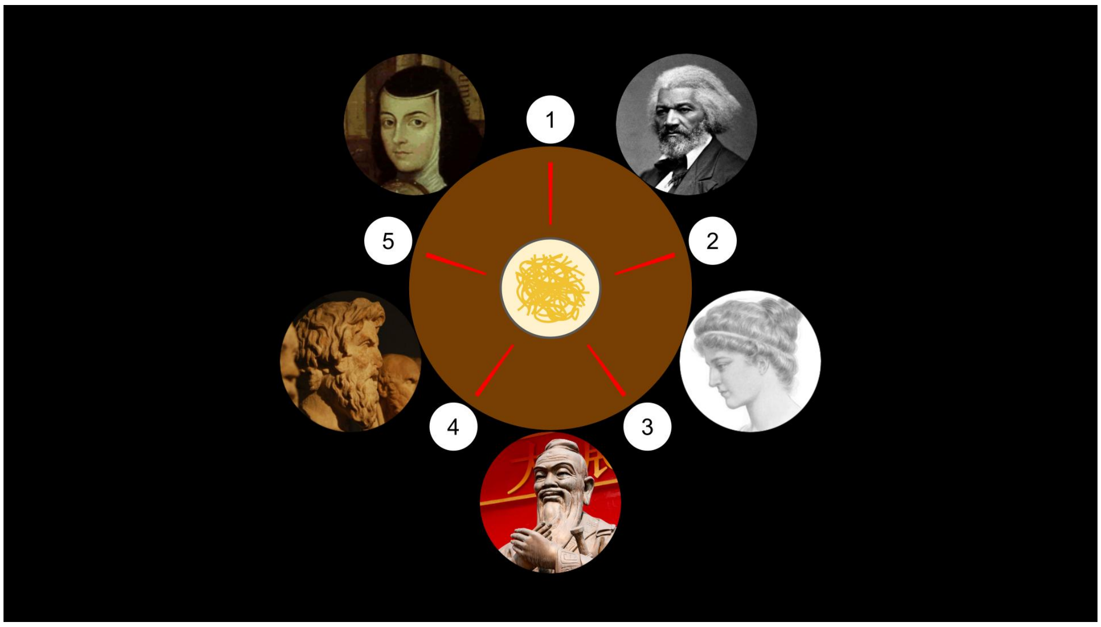
- Pick up left chopstick
- Pick up right chopstick
- Eat for a while
- Put down left chopstick
- Put down right chopstick

If each philosopher picks up their left chopstick and is immediately interrupted by another philosopher that picks up their left chopstick, we'll eventually get to a point where all the philosophers are holding their left chopstick.

However, none of the philosophers can pick up their right chopstick because the philosopher to their right is also waiting. We've just created a situation known as "deadlock." The system comes to a halt as all threads are circularly waiting for locks to be released.

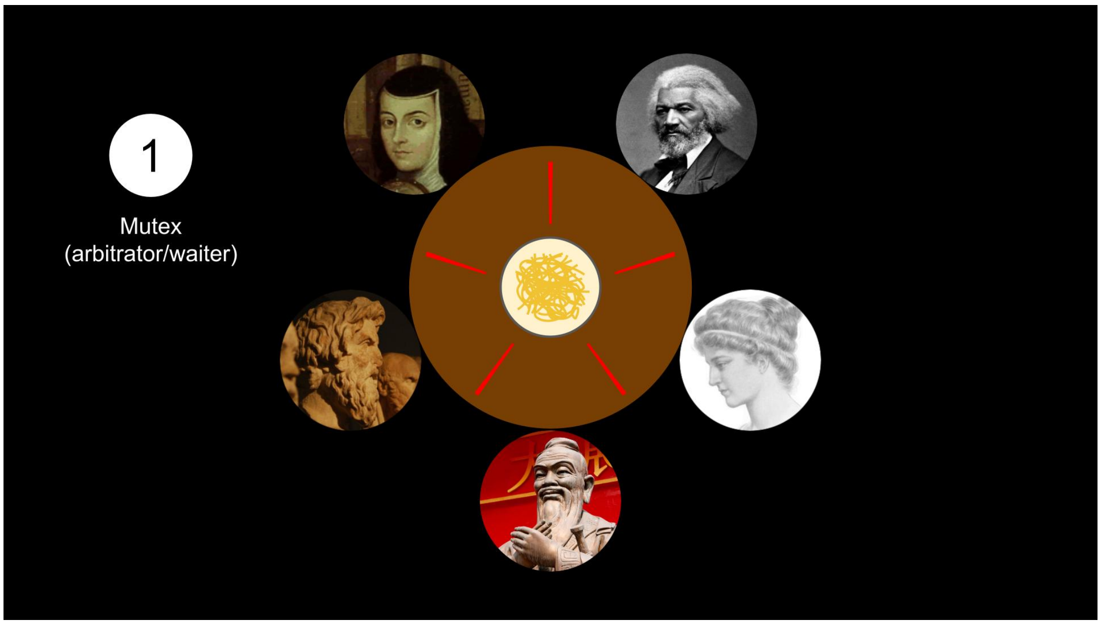


One way to solve this issue is to give a priority or hierarchy value to each chopstick. We modify the algorithm so that each philosopher must pick up the lowest (or highest--it doesn't matter so long as we're consistent) numbered chopstick first.



This will prevent deadlock from occurring, as the last philosopher must wait for the lowest-numbered chopstick to return before they pick up their other chopstick.

Another solution involves a "waiter" or "arbitrator." This abitrator essentially locks all the chopsticks with a global mutex.



Any time a philosopher wishes to pick up any chopstick, they must first request permission from this arbitrator. The mutex (arbitrator) decreases in value and lets the philosopher pick up both chopsticks to eat. Any other philosopher wishing to pick up any chopstick will be denied so long as the first philosopher has the mutex.

This works as a solution to prevent deadlock, but it removes any efficiency gains we might see by allowing multiple philosophers to eat at the same time (parallelism).

See this article to learn more about the dining philosophers problem.

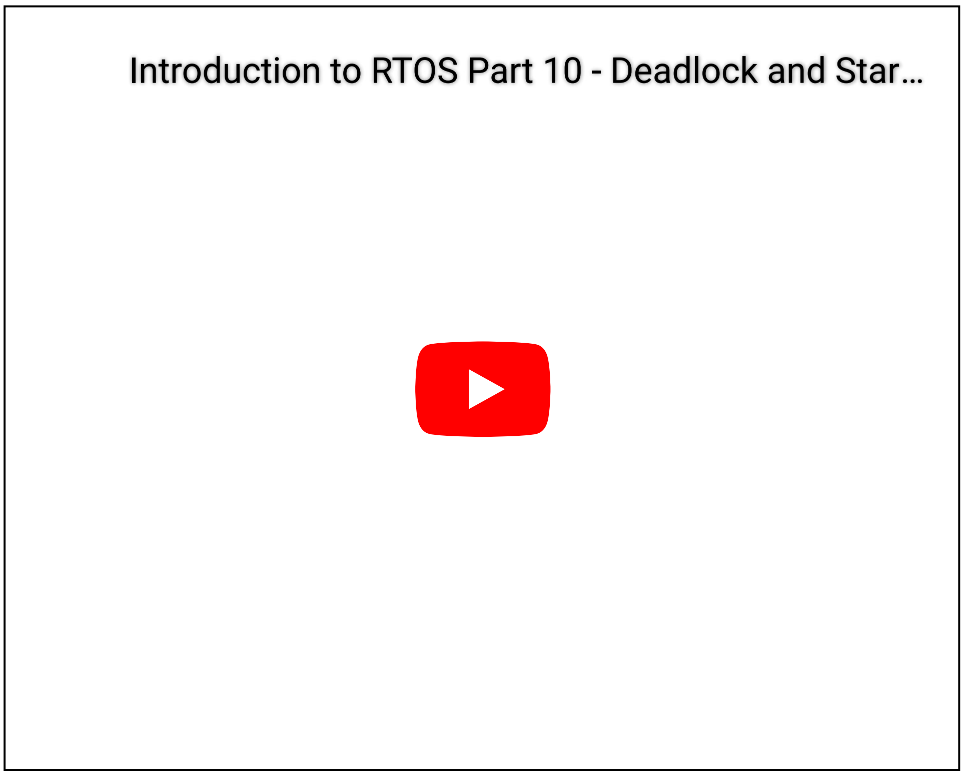
Required Hardware

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

Your challenge is to solve the dining philosophers problem using the demo code found here:

Copy Code

```
/**
 * ESP32 Dining Philosophers
 *
 * The classic "Dining Philosophers" problem in FreeRTOS form.
 *
 * Based on http://www.cs.virginia.edu/luther/COA2/S2019/pa05-dp.html
 *
 * Date: February 8, 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 { NUM_TASKS = 5 };           // Number of tasks (philosophers)
enum { TASK_STACK_SIZE = 2048 };  // Bytes in ESP32, words in vanilla FreeRTOS

// Globals
static SemaphoreHandle_t bin_sem;   // Wait for parameters to be read
static SemaphoreHandle_t done_sem;  // Notifies main task when done
static SemaphoreHandle_t chopstick[NUM_TASKS];

//*****************************************************************************
// Tasks
// The only task: eating
void eat(void *parameters) {

  int num;
  char buf[50];

  // Copy parameter and increment semaphore count
  num = *(int *)parameters;
  xSemaphoreGive(bin_sem);

  // Take left chopstick
  xSemaphoreTake(chopstick[num], portMAX_DELAY);
  sprintf(buf, "Philosopher %i took chopstick %i", num, num);
  Serial.println(buf);

  // Add some delay to force deadlock
  vTaskDelay(1 / portTICK_PERIOD_MS);

  // Take right chopstick
  xSemaphoreTake(chopstick[(num+1)%NUM_TASKS], portMAX_DELAY);
  sprintf(buf, "Philosopher %i took chopstick %i", num, (num+1)%NUM_TASKS);
  Serial.println(buf);

  // Do some eating
  sprintf(buf, "Philosopher %i is eating", num);
  Serial.println(buf);
  vTaskDelay(10 / portTICK_PERIOD_MS);

  // Put down right chopstick
  xSemaphoreGive(chopstick[(num+1)%NUM_TASKS]);
  sprintf(buf, "Philosopher %i returned chopstick %i", num, (num+1)%NUM_TASKS);
  Serial.println(buf);

  // Put down left chopstick
  xSemaphoreGive(chopstick[num]);
  sprintf(buf, "Philosopher %i returned chopstick %i", num, num);
  Serial.println(buf);

  // Notify main task and delete self
  xSemaphoreGive(done_sem);
  vTaskDelete(NULL);
}

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

void setup() {

  char task_name[20];

  // 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 Dining Philosophers Challenge---");

  // Create kernel objects before starting tasks
  bin_sem = xSemaphoreCreateBinary();
  done_sem = xSemaphoreCreateCounting(NUM_TASKS, 0);
  for (int i = 0; i < NUM_TASKS; i++) {
    chopstick[i] = xSemaphoreCreateMutex();
  }

  // Have the philosphers start eating
  for (int i = 0; i < NUM_TASKS; i++) {
    sprintf(task_name, "Philosopher %i", i);
    xTaskCreatePinnedToCore(eat,
                            task_name,
                            TASK_STACK_SIZE,
                            (void *)&i,
                            1,
                            NULL,
                            app_cpu);
    xSemaphoreTake(bin_sem, portMAX_DELAY);
  }


  // Wait until all the philosophers are done
  for (int i = 0; i < NUM_TASKS; i++) {
    xSemaphoreTake(done_sem, portMAX_DELAY);
  }

  // Say that we made it through without deadlock
  Serial.println("Done! No deadlock occurred!");
}

void loop() {
  // Do nothing in this task
}
```

When you run this program on your ESP32, you should see it immediately enter into a deadlock state. Implement both the “hierarchy” solution and the “arbitrator” solution to solve it.

Note that the "eat" task function does not loop forever. If deadlock can be avoided, all tasks will delete themselves, and you should see a "Done! No deadlock occurred!" message appear in the serial terminal.

Solution

Here is the hierarchy solution:

Copy Code

```
/**
 * ESP32 Dining Philosophers Solution: Hierarchy
 *
 * One possible solution to the Dining Philosophers problem.
 *
 * Based on http://www.cs.virginia.edu/luther/COA2/S2019/pa05-dp.html
 *
 * Date: February 8, 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 { NUM_TASKS = 5 };           // Number of tasks (philosophers)
enum { TASK_STACK_SIZE = 2048 };  // Bytes in ESP32, words in vanilla FreeRTOS

// Globals
static SemaphoreHandle_t bin_sem;   // Wait for parameters to be read
static SemaphoreHandle_t done_sem;  // Notifies main task when done
static SemaphoreHandle_t chopstick[NUM_TASKS];

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

// The only task: eating
void eat(void *parameters) {

  int num;
  char buf[50];
  int idx_1;
  int idx_2;

  // Copy parameter and increment semaphore count
  num = *(int *)parameters;
  xSemaphoreGive(bin_sem);

  // Assign priority: pick up lower-numbered chopstick first
  if (num < (num + 1) % NUM_TASKS) {
    idx_1 = num;
    idx_2 = (num + 1) % NUM_TASKS;
  } else {
    idx_1 = (num + 1) % NUM_TASKS;
    idx_2 = num;
  }

  // Take lower-numbered chopstick
  xSemaphoreTake(chopstick[idx_1], portMAX_DELAY);
  sprintf(buf, "Philosopher %i took chopstick %i", num, num);
  Serial.println(buf);

  // Add some delay to force deadlock
  vTaskDelay(1 / portTICK_PERIOD_MS);

  // Take higher-numbered chopstick
  xSemaphoreTake(chopstick[idx_2], portMAX_DELAY);
  sprintf(buf, "Philosopher %i took chopstick %i", num, (num+1)%NUM_TASKS);
  Serial.println(buf);

  // Do some eating
  sprintf(buf, "Philosopher %i is eating", num);
  Serial.println(buf);
  vTaskDelay(10 / portTICK_PERIOD_MS);

  // Put down higher-numbered chopstick
  xSemaphoreGive(chopstick[idx_2]);
  sprintf(buf, "Philosopher %i returned chopstick %i", num, (num+1)%NUM_TASKS);
  Serial.println(buf);

  // Put down lower-numbered chopstick
  xSemaphoreGive(chopstick[idx_1]);
  sprintf(buf, "Philosopher %i returned chopstick %i", num, num);
  Serial.println(buf);

  // Notify main task and delete self
  xSemaphoreGive(done_sem);
  vTaskDelete(NULL);
}

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

void setup() {

  char task_name[20];

  // 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 Dining Philosophers Hierarchy Solution---");

  // Create kernel objects before starting tasks
  bin_sem = xSemaphoreCreateBinary();
  done_sem = xSemaphoreCreateCounting(NUM_TASKS, 0);
  for (int i = 0; i < NUM_TASKS; i++) {
    chopstick[i] = xSemaphoreCreateMutex();
  }

  // Have the philosphers start eating
  for (int i = 0; i < NUM_TASKS; i++) {
    sprintf(task_name, "Philosopher %i", i);
    xTaskCreatePinnedToCore(eat,
                            task_name,
                            TASK_STACK_SIZE,
                            (void *)&i,
                            1,
                            NULL,
                            app_cpu);
    xSemaphoreTake(bin_sem, portMAX_DELAY);
  }


  // Wait until all the philosophers are done
  for (int i = 0; i < NUM_TASKS; i++) {
    xSemaphoreTake(done_sem, portMAX_DELAY);
  }

  // Say that we made it through without deadlock
  Serial.println("Done! No deadlock occurred!");
}

void loop() {
  // Do nothing in this task
}
```

Here is the arbitrator solution:

Copy Code

```
  /**
   * ESP32 Dining Philosophers Solution: Arbitrator/Waiter
   *
   * One possible solution to the Dining Philosophers problem.
   *
   * Based on http://www.cs.virginia.edu/luther/COA2/S2019/pa05-dp.html
   *
   * Date: February 8, 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 { NUM_TASKS = 5 };           // Number of tasks (philosophers)
  enum { TASK_STACK_SIZE = 2048 };  // Bytes in ESP32, words in vanilla FreeRTOS

  // Globals
  static SemaphoreHandle_t bin_sem;   // Wait for parameters to be read
  static SemaphoreHandle_t done_sem;  // Notifies main task when done
  static SemaphoreHandle_t chopstick[NUM_TASKS];
  static SemaphoreHandle_t arbitrator;  // Controls who eats when (e.g. waiter)

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

  // The only task: eating
  void eat(void *parameters) {

    int num;
    char buf[50];

    // Copy parameter and increment semaphore count
    num = *(int *)parameters;
    xSemaphoreGive(bin_sem);

    // We have to wait our turn to eat
    xSemaphoreTake(arbitrator, portMAX_DELAY);

    // Take left chopstick
    xSemaphoreTake(chopstick[num], portMAX_DELAY);
    sprintf(buf, "Philosopher %i took chopstick %i", num, num);
    Serial.println(buf);

    // Add some delay to force deadlock
    vTaskDelay(1 / portTICK_PERIOD_MS);

    // Take right chopstick
    xSemaphoreTake(chopstick[(num+1)%NUM_TASKS], portMAX_DELAY);
    sprintf(buf, "Philosopher %i took chopstick %i", num, (num+1)%NUM_TASKS);
    Serial.println(buf);

    // Do some eating
    sprintf(buf, "Philosopher %i is eating", num);
    Serial.println(buf);
    vTaskDelay(10 / portTICK_PERIOD_MS);

    // Put down right chopstick
    xSemaphoreGive(chopstick[(num+1)%NUM_TASKS]);
    sprintf(buf, "Philosopher %i returned chopstick %i", num, (num+1)%NUM_TASKS);
    Serial.println(buf);

    // Put down left chopstick
    xSemaphoreGive(chopstick[num]);
    sprintf(buf, "Philosopher %i returned chopstick %i", num, num);
    Serial.println(buf);

    // Tell the arbitrator (waiter) that we're done
    xSemaphoreGive(arbitrator);

    // Notify main task and delete self
    xSemaphoreGive(done_sem);
    vTaskDelete(NULL);
  }

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

  void setup() {

    char task_name[20];

    // 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 Dining Philosophers Arbitrator Solution---");

    // Create kernel objects before starting tasks
    bin_sem = xSemaphoreCreateBinary();
    done_sem = xSemaphoreCreateCounting(NUM_TASKS, 0);
    for (int i = 0; i < NUM_TASKS; i++) {
      chopstick[i] = xSemaphoreCreateMutex();
    }
    arbitrator = xSemaphoreCreateMutex();

    // Have the philosphers start eating
    for (int i = 0; i < NUM_TASKS; i++) {
      sprintf(task_name, "Philosopher %i", i);
      xTaskCreatePinnedToCore(eat,
                              task_name,
                              TASK_STACK_SIZE,
                              (void *)&i,
                              1,
                              NULL,
                              app_cpu);
      xSemaphoreTake(bin_sem, portMAX_DELAY);
    }


    // Wait until all the philosophers are done
    for (int i = 0; i < NUM_TASKS; i++) {
      xSemaphoreTake(done_sem, portMAX_DELAY);
    }
    // Say that we made it through without deadlock
    Serial.println("Done! No deadlock occurred!");
  }

  void loop() {
    // Do nothing in this task
  }
```

Explanation

In the hierarchy solution, we use the index of the “chopsticks” (array of mutexes) as the order number. We change the “eat” task to the following:

Copy Code

```
// The only task: eating
void eat(void *parameters) {

  int num;
  char buf[50];
  int idx_1;
  int idx_2;

  // Copy parameter and increment semaphore count
  num = *(int *)parameters;
  xSemaphoreGive(bin_sem);

  // Assign priority: pick up lower-numbered chopstick first
  if (num < (num + 1) % NUM_TASKS) {
    idx_1 = num;
    idx_2 = (num + 1) % NUM_TASKS;
  } else {
    idx_1 = (num + 1) % NUM_TASKS;
    idx_2 = num;
  }

  // Take lower-numbered chopstick
  xSemaphoreTake(chopstick[idx_1], portMAX_DELAY);
  sprintf(buf, "Philosopher %i took chopstick %i", num, num);
  Serial.println(buf);

  // Add some delay to force deadlock
  vTaskDelay(1 / portTICK_PERIOD_MS);

  // Take higher-numbered chopstick
  xSemaphoreTake(chopstick[idx_2], portMAX_DELAY);
  sprintf(buf, "Philosopher %i took chopstick %i", num, (num+1)%NUM_TASKS);
  Serial.println(buf);

  // Do some eating
  sprintf(buf, "Philosopher %i is eating", num);
  Serial.println(buf);
  vTaskDelay(10 / portTICK_PERIOD_MS);

  // Put down higher-numbered chopstick
  xSemaphoreGive(chopstick[idx_2]);
  sprintf(buf, "Philosopher %i returned chopstick %i", num, (num+1)%NUM_TASKS);
  Serial.println(buf);

  // Put down lower-numbered chopstick
  xSemaphoreGive(chopstick[idx_1]);
  sprintf(buf, "Philosopher %i returned chopstick %i", num, num);
  Serial.println(buf);

  // Notify main task and delete self
  xSemaphoreGive(done_sem);
  vTaskDelete(NULL);
}
```

All we do is change which chopstick is picked up first by figuring out which chopstick (left or right) is lowest-numbered. If that chopstick is in use, the task blocks itself (waiting for the chopstick mutex to be released). Notice that the order in which we give the mutexes back does not ultimately matter, but it's generally considered good practice to return them in reverse order (e.g. return higher-numbered chopstick first then the lower-numbered chopstick).

In the arbitrator solution, we simply add a global mutex that determines if each task can pick up its first chopstick. When that task is done "eating" and has returned both chopsticks, it returns the "arbitrator" mutex.

Copy Code

```
// The only task: eating
void eat(void *parameters) {

  int num;
  char buf[50];

  // Copy parameter and increment semaphore count
  num = *(int *)parameters;
  xSemaphoreGive(bin_sem);

  // We have to wait our turn to eat
  xSemaphoreTake(arbitrator, portMAX_DELAY);

  // Take left chopstick
  xSemaphoreTake(chopstick[num], portMAX_DELAY);
  sprintf(buf, "Philosopher %i took chopstick %i", num, num);
  Serial.println(buf);

  // Add some delay to force deadlock
  vTaskDelay(1 / portTICK_PERIOD_MS);

  // Take right chopstick
  xSemaphoreTake(chopstick[(num+1)%NUM_TASKS], portMAX_DELAY);
  sprintf(buf, "Philosopher %i took chopstick %i", num, (num+1)%NUM_TASKS);
  Serial.println(buf);

  // Do some eating
  sprintf(buf, "Philosopher %i is eating", num);
  Serial.println(buf);
  vTaskDelay(10 / portTICK_PERIOD_MS);

  // Put down right chopstick
  xSemaphoreGive(chopstick[(num+1)%NUM_TASKS]);
  sprintf(buf, "Philosopher %i returned chopstick %i", num, (num+1)%NUM_TASKS);
  Serial.println(buf);

  // Put down left chopstick
  xSemaphoreGive(chopstick[num]);
  sprintf(buf, "Philosopher %i returned chopstick %i", num, num);
  Serial.println(buf);

  // Tell the arbitrator (waiter) that we're done
  xSemaphoreGive(arbitrator);

  // Notify main task and delete self
  xSemaphoreGive(done_sem);
  vTaskDelete(NULL);
}
```

Recommended Reading

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

- The Dining Philosophers challenge was ported to Arduino from this University of Virginia course's lab: http://www.cs.virginia.edu/luther/COA2/S2019/pa05-dp.html
- You can try the Dining Philosophers challenge without an Arduino here (using C++): https://leetcode.com/problems/the-dining-philosophers/
- Another good article discussing deadlock on FreeRTOS: https://microcontrollerslab.com/freertos-recursive-mutex-avoid-deadlocks-examples-arduino/

- 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 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

C++

Tags

Arduino

RTOS

License

Attribution

Get Involved

Like

Save