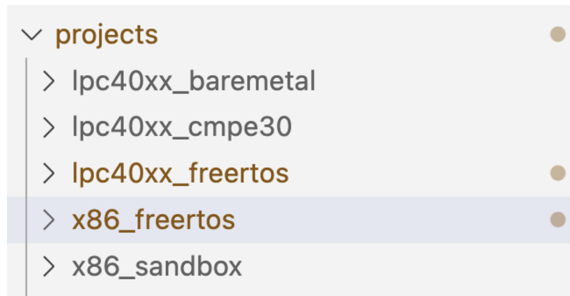
Assignments

- Multitasking: Hands-on
- Thread Stack
- FreeRTOS port
- Queues

Multitasking: Hands-on

In this assignment, we will experiment with RTOS tasks and see the multitasking in action. You may use the x86_freertos project folder to experiment with your code since this folder is already setup for the FreeRTOS POSIX simulator.



We will walk you through the different phases of the assignment. Your assignment should be turned in using a Gitlab Merge Request after the entire assignment is complete. You do not need to submit separate code per part.

Part 1: Setup Skeleton

Please setup the skeleton of your code using the reference code below. Requirements for this part are:

1. Create a task with an infinite loop that invokes `print_function("++++\n")`
2. Sleep the task for 100 ticks
3. Create the task with priority 2

```
void  eat_cpu() {
  for (int i = 0; i < 1 * 1000 * 1000; i++) {
    ;
  }
}
void print_function(const char *string) {
  for (int i = 0; i < strlen(string); i++) {
    putchar(string[i]);
```

```
    eat_cpu();
  }
}
int main(void) {
  // TODO: For you:
  xTaskCreate(printer_task, "name", 1000, NULL, 3, NULL);
  puts("Starting FreeRTOS Scheduler ..... \r\n");
  vTaskStartScheduler();
  return 0;}
```

At this point, you should build your code and test it before moving on.

Part 2: Multiple tasks

For this part, let us create two tasks that print a specific pattern. You can leverage from the "task parameter" that you can pass to a task like so:

```
void printer_task(void *p) {
  const char *what_to_print = (const char*) p;
  while(1) {
    print_function(what_to_print);
  }
}
int main(void) {
  xTaskCreate(printer_task, "name", 1000, "++++\n", 3, NULL);  ...
```

Requirements:

1. Retain existing that that prints `++++\n`
2. Create a new task that prints
3. `----\n`
4. Create both tasks with priority 2

Build and run the code and note down observations.

Part 3: Round-robin scheduler

So far what we should have observed is that when a task gives up its CPU using vTaskDelay() or another function that "blocks", the time allocation is given to other tasks that may be equal or lower priority. If we deliberately design the code such that tasks do not sleep, then the round-robin scheduler will kick-in. This round-robin scheduler requires "preemptive scheduler" option which is typically enabled by default by all RTOSes.

```
void task_1(void *p) {
  const char *what_to_print = "----\n";
  while(1) {
    print_function(what_to_print);
    eat_cpu();
  }
}
void task_2(void *p) {
  const char *what_to_print = "++++\n";
  while(1) {
    print_function(what_to_print);
    eat_cpu();
  }
}
int main(void) {
  //xTaskCreate(cpu_utilization_print_task, "cpu", 1, NULL, PRIORITY_LOW, NULL);
  xTaskCreate(task_1, "name", 1000, "++++\n", 3, NULL);
  xTaskCreate(task_2, "name", 1000, "****\n", 3, NULL);

  puts("Starting FreeRTOS Scheduler ..... \r\n");
  vTaskStartScheduler();
  return 0;
}
```

Assignment Submission

At this point you should have performed a number of experiments to get to know the RTOS scheduler. Note that all RTOSes will behave the same way. The scheduling policy is always A) High priority first, and then B) Round-robin scheduling between equal priority tasks.

Please create a README.MD file and add that to your MR to explain the following:

1. Change the priority of the `++++` task to 2, and the priority of the `----` task to 1
   - Have the `++++` task invoke vTaskDelay(100)
   - What is the output of the code?

2. Change the priorities of both tasks to 1
   - What is the output of the code?
   - What is different from the previous part?

3. Remove the vTaskDelay(100) with eat_cpu()
   - Use equal priorities for both tasks
   - What is the output of the code?

4. With equal priorities, make a change at FreeRTOSConfig.h and set `#define configUSE_PREEMPTION 0`
   - Ensure that you modify the FreeRTOS file at your project folder, such as x86_freertos
   - What is the output of the code?
   - Explain how the pre-emption option is behaving

Thread Stack

FreeRTOS port

Objective:

Understanding the FreeRTOS Portable Layer.

The objective of this assignment is to understand the FreeRTOS portable layer and how it allows FreeRTOS to run on various hardware architectures. By the end of this assignment, students should be able to explain the role of the portable layer and demonstrate their understanding by analyzing an existing port, specifically the POSIX port.

Tasks:

1. **Research and Study**:
    - Read the FreeRTOS documentation on the portable layer.
    - Study the implementation of the portable layer for an existing architecture, such as ARM Cortex-M (found in `FreeRTOS/Source/portable/GCC/ARM_CM4F`).

2. **Explain the Portable Layer**:
    - Write a detailed report (2-3 pages) explaining the role of the FreeRTOS portable layer. Include the following points:
        - What is the portable layer in FreeRTOS?
        - Why is the portable layer necessary?
        - Key components of the portable layer (e.g., context switching, stack management, and ISR handling).
        - How the portable layer interfaces with the core FreeRTOS kernel.
        - **Use diagrams**

3. **Explain a specific Port**:
    - Choose an existing port (e.g., ARM Cortex-M4) and analyze its implementation.
        - Choose a CPU architecture you are familiar with; you can even pick POSIX

- Identify and study the following components:
    - Context switch implementation.
    - Stack initialization for tasks.
    - Interrupt handling and how FreeRTOS interacts with the hardware.
    - Any architecture-specific optimizations.

4. **Submit Your Work**:
    - Submit the following:
        - The detailed report on the FreeRTOS portable layer.
        - The analysis of the existing port.
        - The explanation of the port you selected.

Grading Criteria:

- **Report Quality**
    - Clarity and completeness of the explanation.
    - Depth of understanding demonstrated.
    - Accuracy and detail in identifying key components of the existing port.
- **POSIX Port Explanation**
    - Correctness and completeness of the explanation of the POSIX port.
    - Clarity and depth of the analysis.

Additional Resources:

- FreeRTOS Documentation
- FreeRTOS Porting Guide
- FreeRTOS Kernel Source Code
- FreeRTOS POSIX Port Documentation

Queues

In this assignment, we will learn about RTOS queues. We will setup three tasks:

1. Light sensor task
2. Temperature sensor task
3. File writer task

This is a classic "Many-to-One" design pattern in RTOS. In this scenario, the **File Writer** acts as a centralized consumer, while the sensors act as producers. This prevents "resource contention" where two tasks try to write to the same file or serial port at the same time.

The Assignment Objective

1. **Create a Queue** capable of holding a custom `struct`.
2. **Task 1 & 2 (Producers):** Periodically read a (mock) sensor and "Send" the data to the queue.
3. **Task 3 (Consumer):** Block (sleep) until data arrives in the queue, then "Receive" and print it.

Template

Step 1: Define the Data Structure

Since the File Writer needs to know *which* sensor sent the data, we use a structure. You could be a bit more fancy and use a "union" for the data structure, but we leave this as an optional exercise for you.

```
typedef enum {
    SOURCE_LIGHT,
```

```
    SOURCE_TEMP
} SensorSource_t;
typedef struct {
    SensorSource_t source;    int32_t
 value;    uint32_t
 timestamp;
} SensorData_t;
// Global handle for the queue
QueueHandle_t xSensorQueue;
```

Step 2: The Task Templates

Producer Template (Light & Temp Tasks)

Both sensor tasks will look very similar. You should implement logic to "mock" a sensor reading (e.g., a random number or a counter).

```
void vSensorTask(void *pvParameters)
{
    SensorSource_t mySource = (SensorSource_t)pvParameters;
    SensorData_t xMessage;
    for
 (;;) {        // 1. MOCK: Generate a fake sensor value
        xMessage.source = mySource;        xMessage.value = (mySource == SOURCE_LIGHT) ? 450
 : 22
;
        xMessage.timestamp = xTaskGetTickCount();
        // 2. SEND: Post to the queue
        // Use a 0ms block time if you don't want to wait if the queue is full
        if (xQueueSend(xSensorQueue, &xMessage, 0
) != pdPASS) {            // Handle queue full error here
```

```
        }
        vTaskDelay(pdMS_TO_TICKS(1000)); // Poll every 1 second
    }
}
```

Consumer Template (File Writer Task)

This task should be the most efficient. It stays in a **Blocked** state until the queue has data, meaning it consumes 0% CPU while waiting.

```
void vFileWriterTask(void *pvParameters)
{
    SensorData_t xReceivedData;
    for
 (;;) {        // 3. RECEIVE: Wait indefinitely (portMAX_DELAY) for data
        if
 (xQueueReceive(xSensorQueue, &xReceivedData, portMAX_DELAY) == pdPASS) {
                        // 4. LOG: Check source and "write" to console/file
            const char* label = (xReceivedData.source == SOURCE_LIGHT) ? "LIGHT" : "TEMP"
;            printf("[%u] %s Reading: %d\n"
, xReceivedData.timestamp, label, xReceivedData.value);
        }
    }
}
```

Step 3: Main Initialization

The most important part of the assignment is ensuring the queue is created **before** the tasks start.

```
int main(void)
```

```
{    // Initialize the queue: (Length of 10 items, size of our struct)
    xSensorQueue = xQueueCreate(10, sizeof
(SensorData_t));
    if (xSensorQueue != NULL
) {        // Create Producers
        xTaskCreate(vSensorTask, "Light", 1000, (void*)SOURCE_LIGHT, 1, NULL
);        xTaskCreate(vSensorTask, "Temp", 1000, (void*)SOURCE_TEMP, 1, NULL
);
        // Create Consumer (Give it a slightly higher priority if data is high-volume)
        xTaskCreate(vFileWriterTask, "Writer", 1000, NULL, 2, NULL
);
        vTaskStartScheduler();
    }
    for (;;); // Should never reach here
}
```

Assignment Challenges:

- **Buffer Overflow:** What happens if the `vSensorTask` delays for 100ms but the `vFileWriterTask` takes 500ms to process? (Students should observe the queue filling up).
- **Priority Inversion:** Change the priorities so the producers are higher than the consumer. Observe how the queue behaves.