Computer Architectures & Operating Systems

Project – HacIOSsim

Academic Year 2023/2024

Group No. 09
s323386 – Antonio Capece
s328833 – Federico Cagnazzo
s331031 – Giovanni de Maria
s331618 – Andrea Carcagnì

Politecnico di Torino
1859

Project Goals

Acquire proficiency in utilizing an embedded operating system on **QEMU**

Customize the chosen operating system (**FreeRTOS**) to implement a **new feature**

Evaluate on significant **workloads**

FreeRTOS
Real-Time OS for Microcontrollers

FreeRTOS is an open-source real-time operating system **kernel for embedded systems**, widely used in microcontrollers and small microprocessors.
It provides a **set of software tools and libraries** that facilitate the development of embedded systems with real-time constraints.

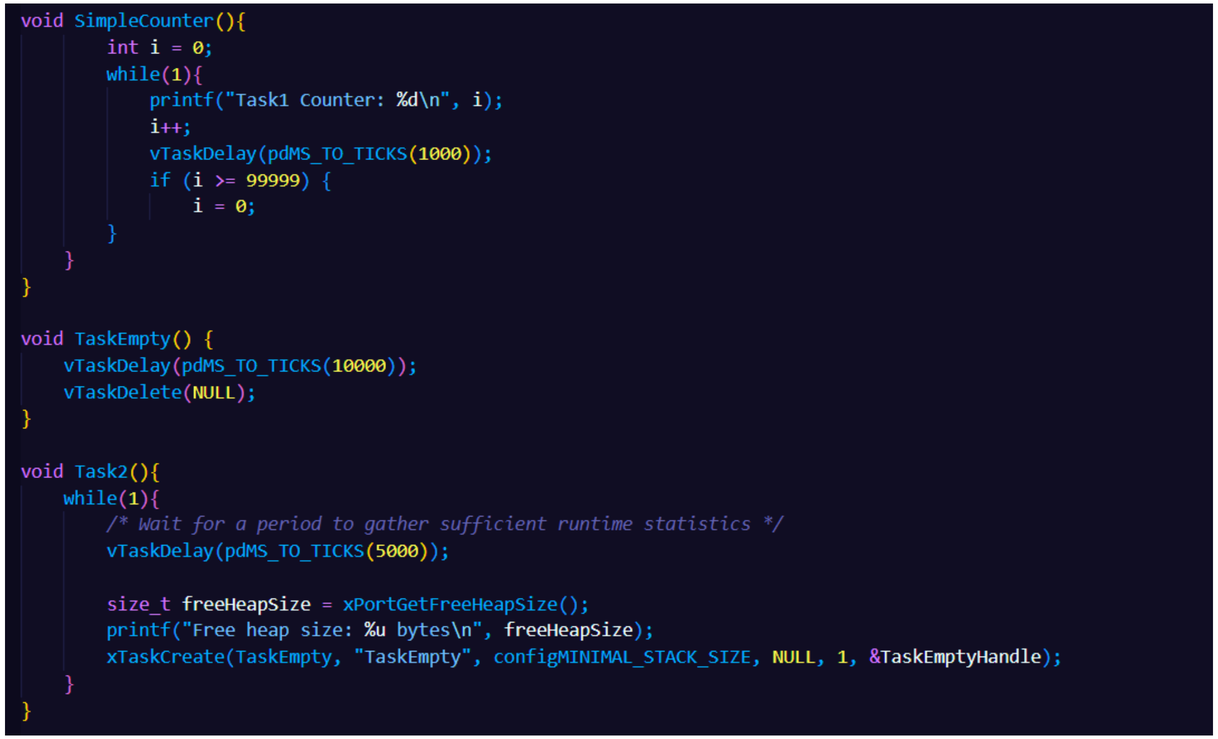
Distributed **freely** under the **MIT open-source license**, FreeRTOS includes a kernel.

FreeRTOS is built with an emphasis on reliability and ease of use.

demoStats.c

This program is an example to **show** how much **space remains unallocated** in the **heap**.

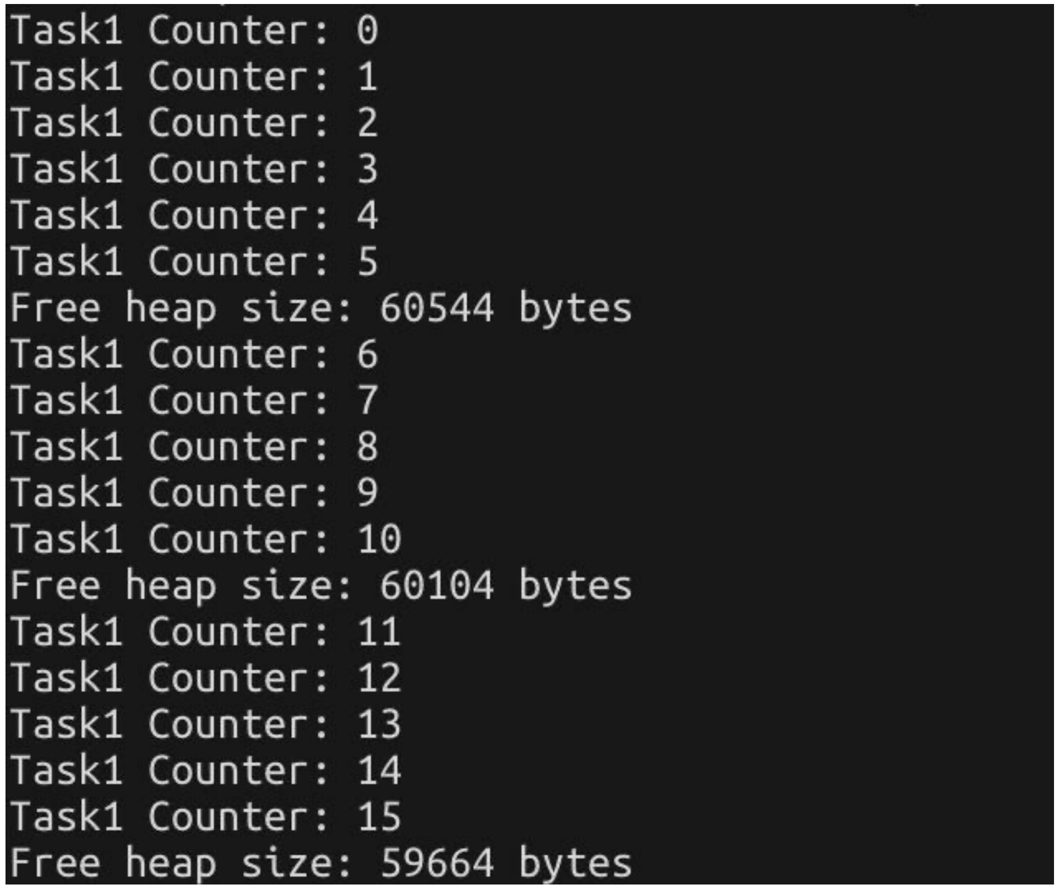
It creates **two tasks**, run **concurrently**. The former is a **simple counter**, the latter creates each time a new empty task and after call the **xPortGetFreeHeapSize()** function (after a delay), which returns the total amount of **heap** that remains **unallocated**.



```c
void SimpleCounter(){
        int i = 0;
        while(1){
            printf("Task1 Counter: %d\n", i);
            i++;
            vTaskDelay(pdMS_TO_TICKS(1000));
            if (i >= 99999) {
                i = 0;
            }
        }
}

void TaskEmpty() {
    vTaskDelay(pdMS_TO_TICKS(10000));
    vTaskDelete(NULL);
}

void Task2(){
    while(1){
        /* Wait for a period to gather sufficient runtime statistics */
        vTaskDelay(pdMS_TO_TICKS(5000));

        size_t freeHeapSize = xPortGetFreeHeapSize();
        printf("Free heap size: %u bytes\n", freeHeapSize);
        xTaskCreate(TaskEmpty, "TaskEmpty", configMINIMAL_STACK_SIZE, NULL, 1, &TaskEmptyHandle);
    }
}
```

demoStats.c - Output

Running the demo, we can see how the **heap unallocated is less** each time a new **task is created** and is running.
When a **task terminates**, the heap occupied is **automatically deallocated**.

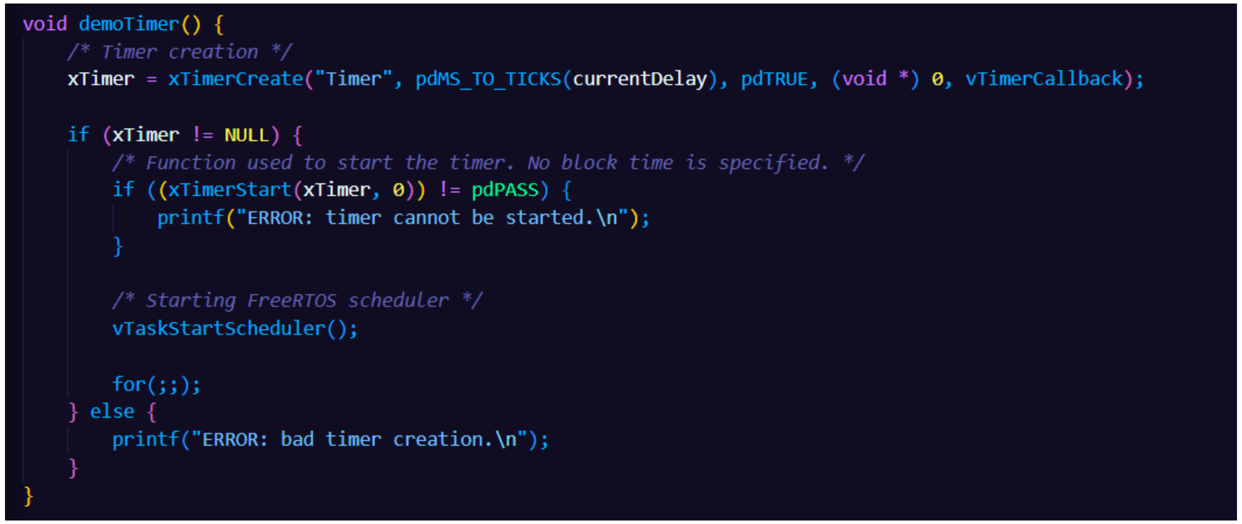


```
Task1 Counter: 0
Task1 Counter: 1
Task1 Counter: 2
Task1 Counter: 3
Task1 Counter: 4
Task1 Counter: 5
Free heap size: 60544 bytes
Task1 Counter: 6
Task1 Counter: 7
Task1 Counter: 8
Task1 Counter: 9
Task1 Counter: 10
Free heap size: 60104 bytes
Task1 Counter: 11
Task1 Counter: 12
Task1 Counter: 13
Task1 Counter: 14
Task1 Counter: 15
Free heap size: 59664 bytes
```

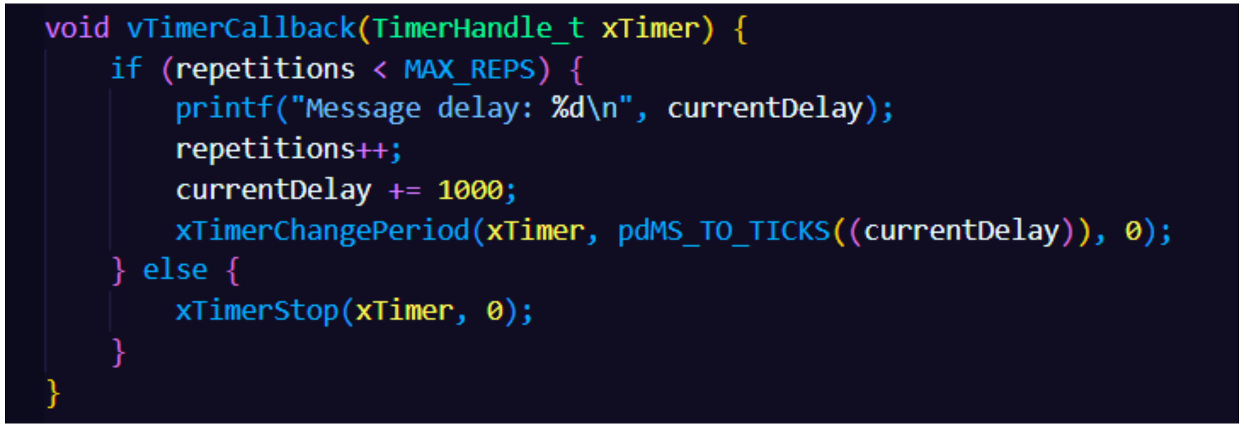
demoTimer.c

The program is an example used to make simpler to understand a basic function of FreeRTOS as **timers**.

A message is printed on the console after an **increasing timeout**.
After each print, the delay is increased of 1 second.



```c
void demoTimer() {
    /* Timer creation */
    xTimer = xTimerCreate("Timer", pdMS_TO_TICKS(currentDelay), pdTRUE, (void *) 0, vTimerCallback);

    if (xTimer != NULL) {
        /* Function used to start the timer. No block time is specified. */
        if ((xTimerStart(xTimer, 0)) != pdPASS) {
            printf("ERROR: timer cannot be started.\n");
        }

        /* Starting FreeRTOS scheduler */
        vTaskStartScheduler();

        for(;;);
    } else {
        printf("ERROR: bad timer creation.\n");
    }
}
```

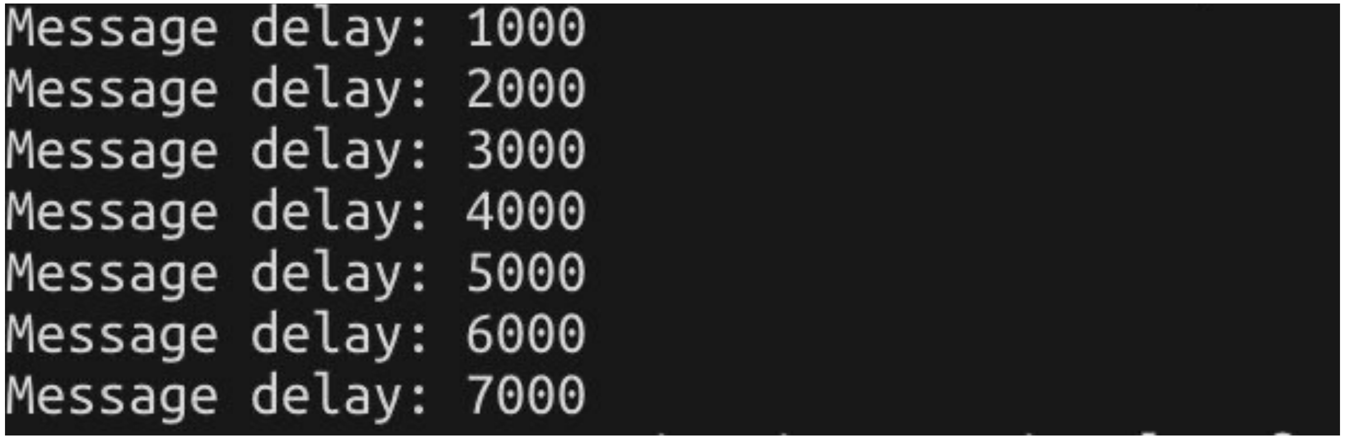


```c
void vTimerCallback(TimerHandle_t xTimer) {
    if (repetitions < MAX_REPS) {
        printf("Message delay: %d\n", currentDelay);
        repetitions++;
        currentDelay += 1000;
        xTimerChangePeriod(xTimer, pdMS_TO_TICKS((currentDelay)), 0);
    } else {
        xTimerStop(xTimer, 0);
    }
}
```

demoTimer.c - Output

The following is the **obtained output**, after running the demo.

We can notice the **incremental delay** printed in the **output message**.

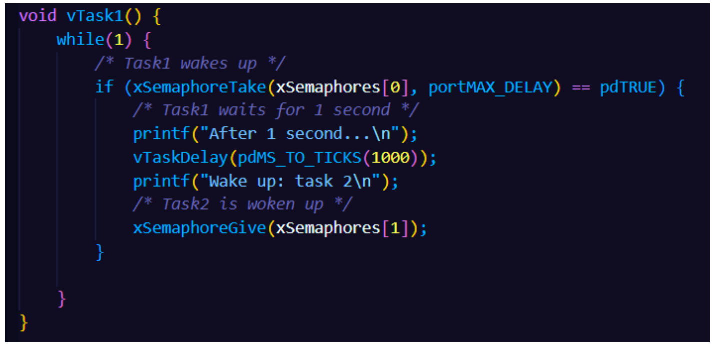


```
Message delay: 1000
Message delay: 2000
Message delay: 3000
Message delay: 4000
Message delay: 5000
Message delay: 6000
Message delay: 7000
```

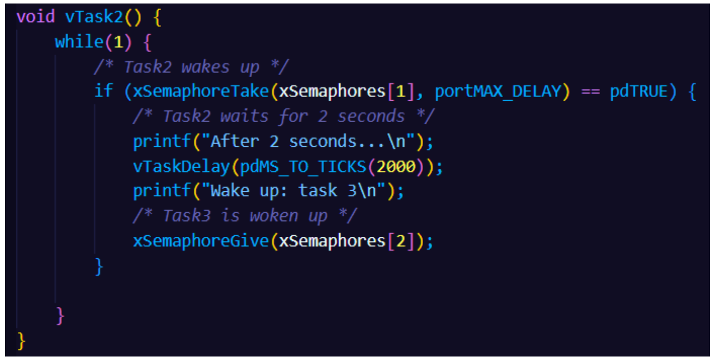
demoSemaphores.c

The program implements **three mutex semaphores** and three tasks which access a **critical section**, in which a message is printed and are used respectively **timers delays of 1, 2, and 3 seconds**.

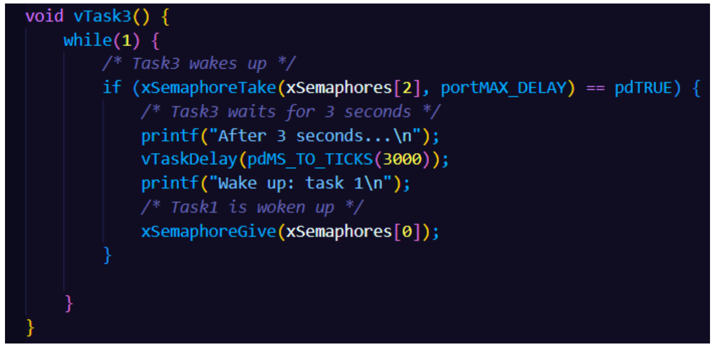
When a **task accesses** the critical section, the other **two remained blocked**.
When it finishes, it unblocks the following task.
This is performed **cyclically**.



```c
void vTask1() {
    while(1) {
        /* Task1 wakes up */
        if (xSemaphoreTake(xSemaphores[0], portMAX_DELAY) == pdTRUE) {
            /* Task1 waits for 1 second */
            printf("After 1 second...\n");
            vTaskDelay(pdMS_TO_TICKS(1000));
            printf("Wake up: task 2\n");
            /* Task2 is woken up */
            xSemaphoreGive(xSemaphores[1]);
        }

    }
}
```



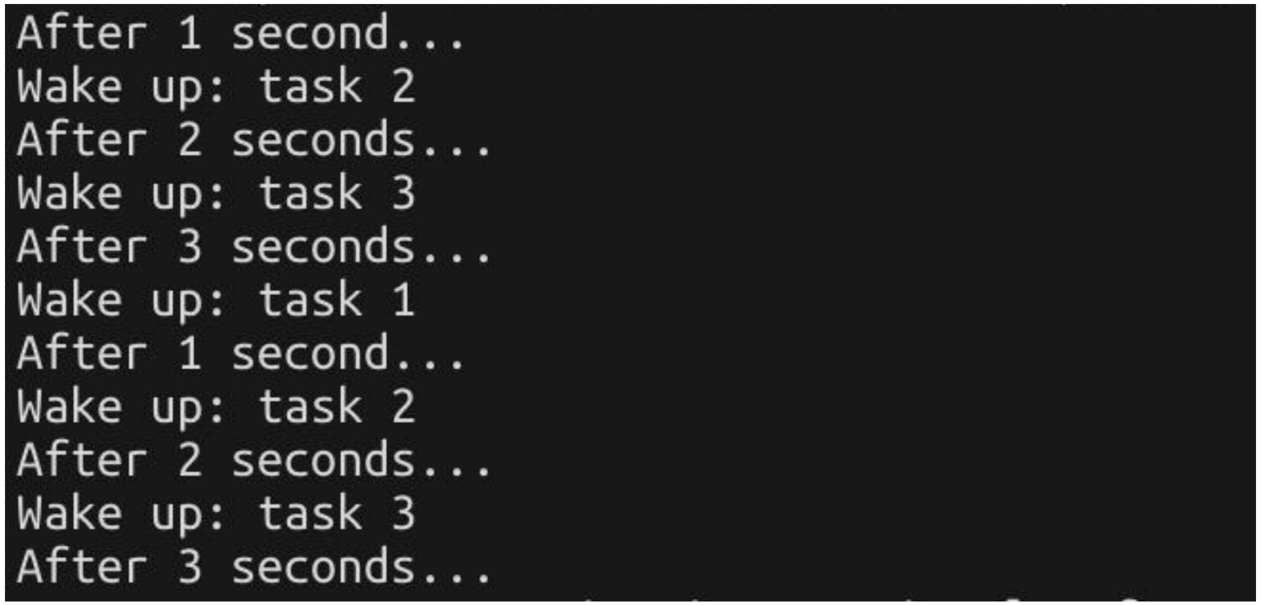
```c
void vTask2() {
    while(1) {
        /* Task2 wakes up */
        if (xSemaphoreTake(xSemaphores[1], portMAX_DELAY) == pdTRUE) {
            /* Task2 waits for 2 seconds */
            printf("After 2 seconds...\n");
            vTaskDelay(pdMS_TO_TICKS(2000));
            printf("Wake up: task 3\n");
            /* Task3 is woken up */
            xSemaphoreGive(xSemaphores[2]);
        }

    }
}
```



```c
void vTask3() {
    while(1) {
        /* Task3 wakes up */
        if (xSemaphoreTake(xSemaphores[2], portMAX_DELAY) == pdTRUE) {
            /* Task3 waits for 3 seconds */
            printf("After 3 seconds...\n");
            vTaskDelay(pdMS_TO_TICKS(3000));
            printf("Wake up: task 1\n");
            /* Task1 is woken up */
            xSemaphoreGive(xSemaphores[0]);
        }

    }
}
```

demoSemaphores.c - Output

This is the **obtained output** after running the demo. It is shown the **circular running** of the three tasks, which **access** the **critical section** in **mutual exclusion** and, after performing operation in this section, the **following task is woke up**.

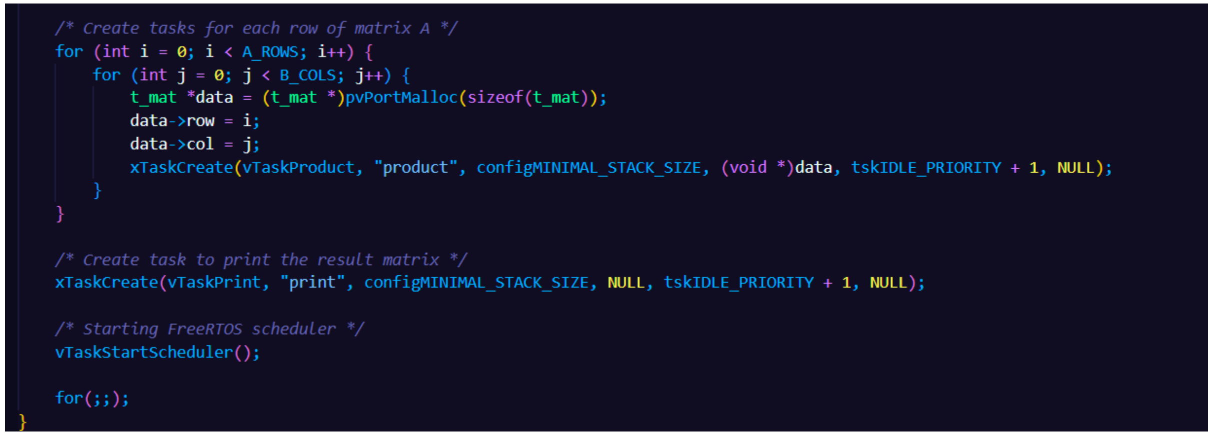


```
After 1 second...
Wake up: task 2
After 2 seconds...
Wake up: task 3
After 3 seconds...
Wake up: task 1
After 1 second...
Wake up: task 2
After 2 seconds...
Wake up: task 3
After 3 seconds...
```

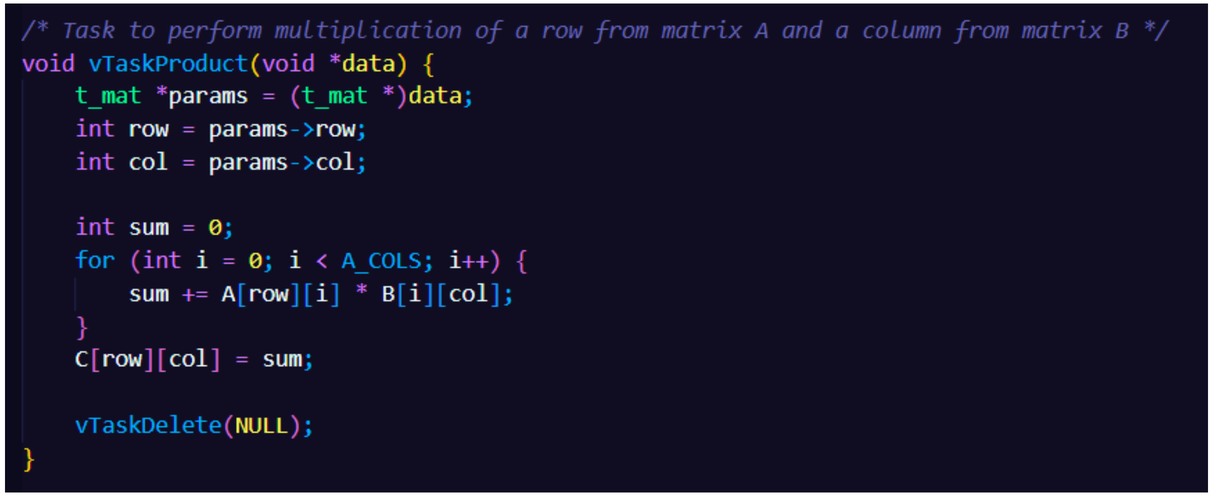
demoMatrix.c

The program implements a **multiplication** between **two matrices**, A and B.
It creates **N tasks**, one **for each result** that need to be computed and assigned to the result matrix C.

These tasks are **run in parallel** and only after their computation, another task is created to **print the matrix C**, containing the result of the multiplication.



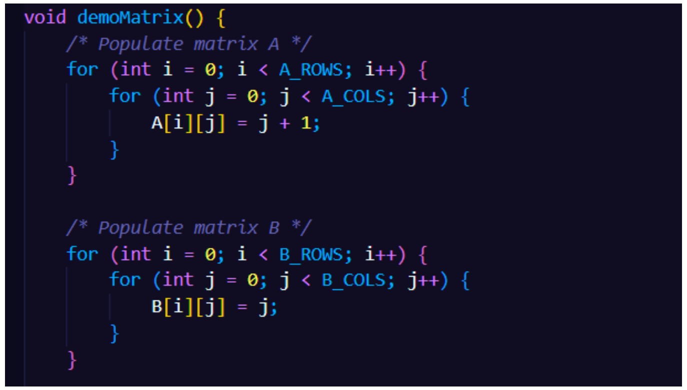
```c
    /* Create tasks for each row of matrix A */
    for (int i = 0; i < A_ROWS; i++) {
        for (int j = 0; j < B_COLS; j++) {
            t_mat *data = (t_mat *)pvPortMalloc(sizeof(t_mat));
            data->row = i;
            data->col = j;
            xTaskCreate(vTaskProduct, "product", configMINIMAL_STACK_SIZE, (void *)data, tskIDLE_PRIORITY + 1, NULL);
        }
    }

    /* Create task to print the result matrix */
    xTaskCreate(vTaskPrint, "print", configMINIMAL_STACK_SIZE, NULL, tskIDLE_PRIORITY + 1, NULL);

    /* Starting FreeRTOS scheduler */
    vTaskStartScheduler();

    for(;;);
}
```



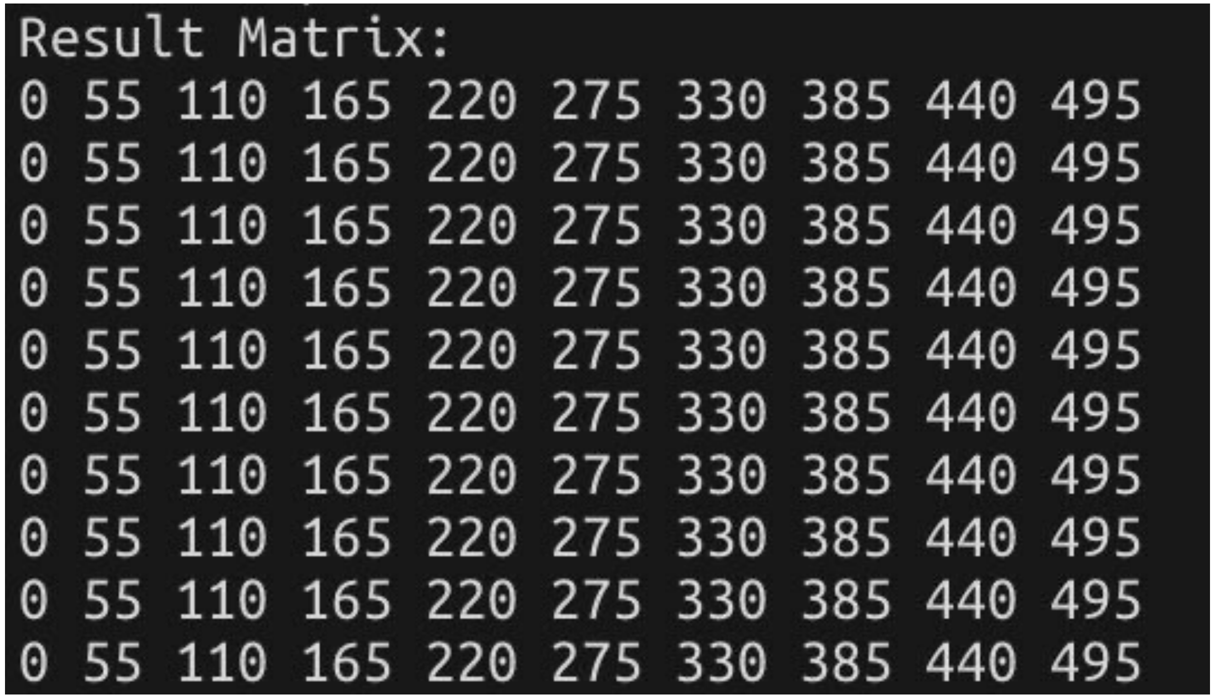
```c
/* Task to perform multiplication of a row from matrix A and a column from matrix B */
void vTaskProduct(void *data) {
    t_mat *params = (t_mat *)data;
    int row = params->row;
    int col = params->col;

    int sum = 0;
    for (int i = 0; i < A_COLS; i++) {
        sum += A[row][i] * B[i][col];
    }
    C[row][col] = sum;

    vTaskDelete(NULL);
}
```

demoMatrix.c - Output

Before starting the multiplication, we **populate** the two **matrices** in the following way.
The **result matrix** is obtained **very fast** since we can **parallelize** the **operations**.



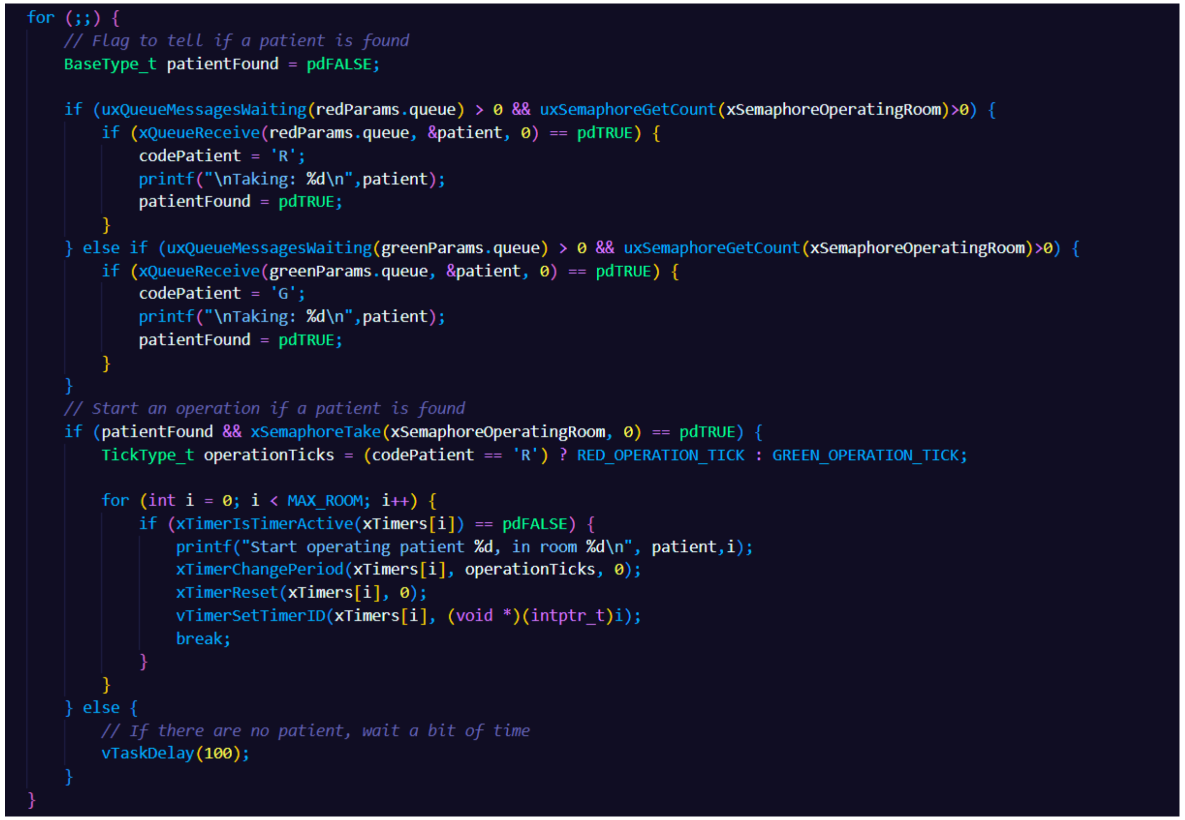
```
void demoMatrix() {
    /* Populate matrix A */
    for (int i = 0; i < A_ROWS; i++) {
        for (int j = 0; j < A_COLS; j++) {
            A[i][j] = j + 1;
        }
    }

    /* Populate matrix B */
    for (int i = 0; i < B_ROWS; i++) {
        for (int j = 0; j < B_COLS; j++) {
            B[i][j] = j;
        }
    }
```



```
Result Matrix:
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
0 55 110 165 220 275 330 385 440 495
```

demoHospital.c

This demo simulates the management of a **hospital** system.
The hospital has a **limited** number of operating **rooms**, with only **one patient** being operated **on at a time** in each room.
Patients are operated on based on a **priority** system determined by **color codes**, which **define** the **priority** and the **duration** time of the operation.



```c
for (;;) {
    // Flag to tell if a patient is found
    BaseType_t patientFound = pdFALSE;

    if (uxQueueMessagesWaiting(redParams.queue) > 0 && uxSemaphoreGetCount(xSemaphoreOperatingRoom)>0) {
        if (xQueueReceive(redParams.queue, &patient, 0) == pdTRUE) {
            codePatient = 'R';
            printf("\nTaking: %d\n",patient);
            patientFound = pdTRUE;
        }
    } else if (uxQueueMessagesWaiting(greenParams.queue) > 0 && uxSemaphoreGetCount(xSemaphoreOperatingRoom)>0) {
        if (xQueueReceive(greenParams.queue, &patient, 0) == pdTRUE) {
            codePatient = 'G';
            printf("\nTaking: %d\n",patient);
            patientFound = pdTRUE;
        }
    }
    // Start an operation if a patient is found
    if (patientFound && xSemaphoreTake(xSemaphoreOperatingRoom, 0) == pdTRUE) {
        TickType_t operationTicks = (codePatient == 'R') ? RED_OPERATION_TICK : GREEN_OPERATION_TICK;

        for (int i = 0; i < MAX_ROOM; i++) {
            if (xTimerIsTimerActive(xTimers[i]) == pdFALSE) {
                printf("Start operating patient %d, in room %d\n", patient,i);
                xTimerChangePeriod(xTimers[i], operationTicks, 0);
                xTimerReset(xTimers[i], 0);
                vTimerSetTimerID(xTimers[i], (void *)(intptr_t)i);
                break;
            }
        }
    } else {
        // If there are no patient, wait a bit of time
        vTaskDelay(100);
    }
}
```

demoHospital.c

Queues: used to maintain the **list of patients waiting** for surgery. There are **separate queues for code** red and green patients. Tasks **fill these queues with patient arrival times**.

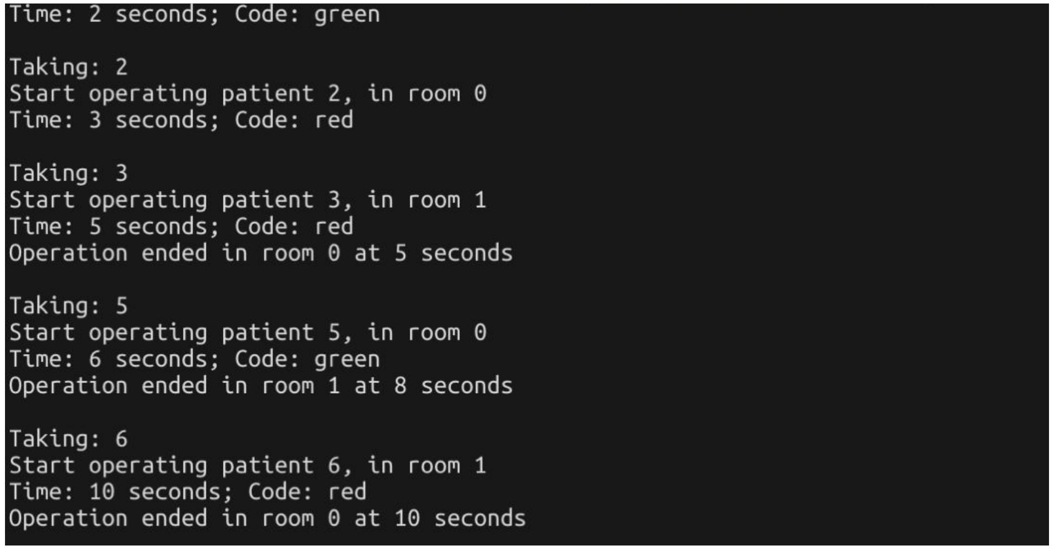
Semaphore: managing **concurrent access** to operating rooms, when a room is available, the semaphore is taken (*xSemaphoreTake*) and then released (*xSemaphoreGive*) when the operation is completed.

Multitasking: *fillQueue* and *operatingRoomTask* manage patient arrival and operation in operating rooms, working in **parallel**, allowing **realistic** simulation of the hospital environment.

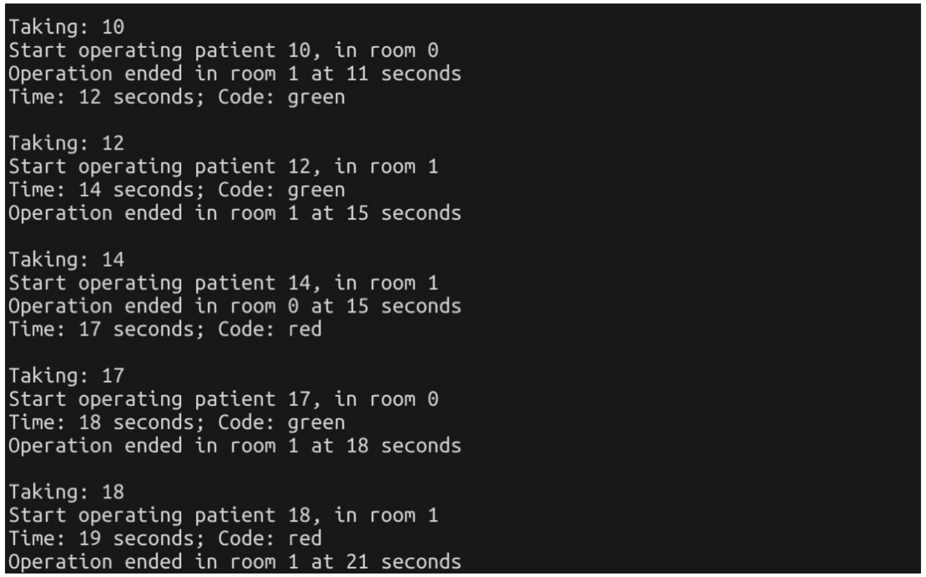
Timer: they manage the **duration** of operations in each operating room.

demoHospital.c - Output

This is the **resulting scheduling** of the incoming patients, based on their **priority** and on the **current condition** of the **surgery room** (free or occupied).



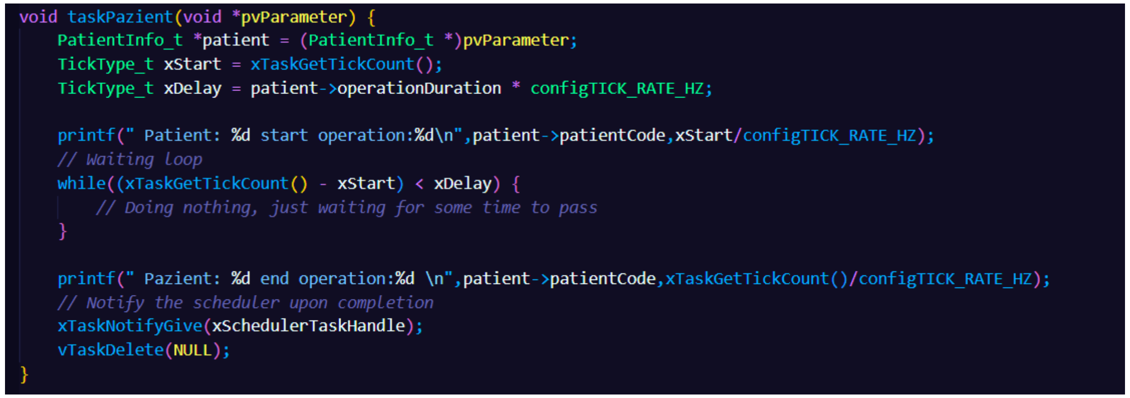
```
Time: 2 seconds; Code: green

Taking: 2
Start operating patient 2, in room 0
Time: 3 seconds; Code: red

Taking: 3
Start operating patient 3, in room 1
Time: 5 seconds; Code: red
Operation ended in room 0 at 5 seconds

Taking: 5
Start operating patient 5, in room 0
Time: 6 seconds; Code: green
Operation ended in room 1 at 8 seconds

Taking: 6
Start operating patient 6, in room 1
Time: 10 seconds; Code: red
Operation ended in room 0 at 10 seconds
```



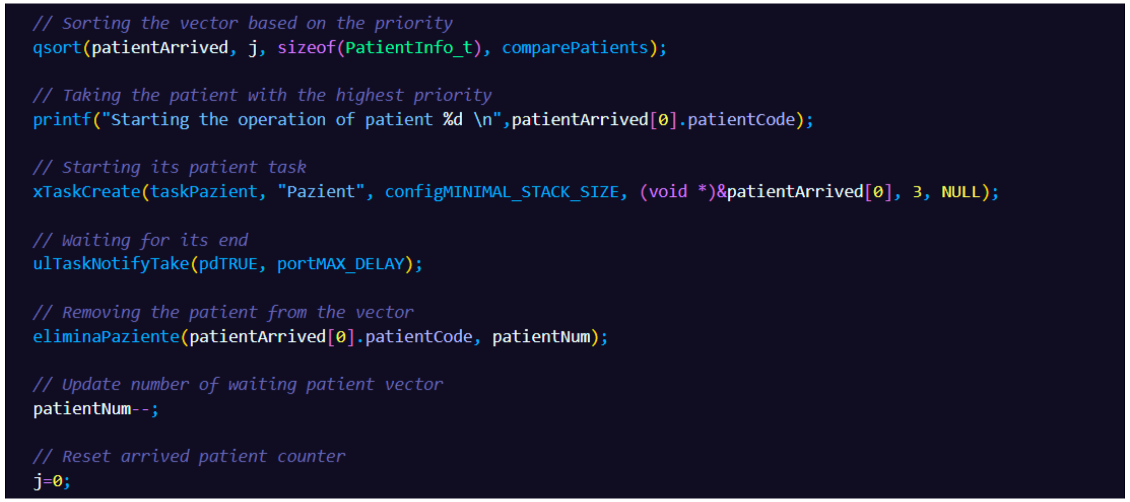
```
Taking: 10
Start operating patient 10, in room 0
Operation ended in room 1 at 11 seconds
Time: 12 seconds; Code: green

Taking: 12
Start operating patient 12, in room 1
Time: 14 seconds; Code: green
Operation ended in room 1 at 15 seconds

Taking: 14
Start operating patient 14, in room 1
Operation ended in room 0 at 15 seconds
Time: 17 seconds; Code: red

Taking: 17
Start operating patient 17, in room 0
Time: 18 seconds; Code: green
Operation ended in room 1 at 18 seconds

Taking: 18
Start operating patient 18, in room 1
Time: 19 seconds; Code: red
Operation ended in room 1 at 21 seconds
```

demoHospital2.c

This demo represents an **improvement** over the previous hospital management system, with **new functionalities** and **optimizations** to better manage patient operations based on priorities.
The **additional functionalities** include **managing** patient **priorities**, **removing patients** from the list in case of death, and **enhancing operation** management using **task notifications**.



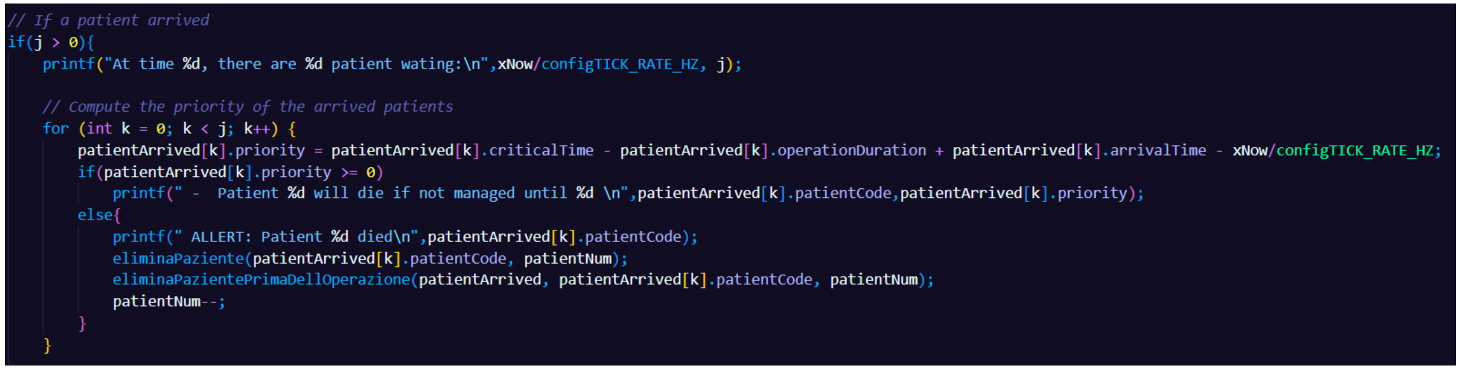
```
void taskPazient(void *pvParameter) {
    PatientInfo_t *patient = (PatientInfo_t *)pvParameter;
    TickType_t xStart = xTaskGetTickCount();
    TickType_t xDelay = patient->operationDuration * configTICK_RATE_HZ;

    printf(" Patient: %d start operation:%d\n",patient->patientCode,xStart/configTICK_RATE_HZ);
    // Waiting loop
    while((xTaskGetTickCount() - xStart) < xDelay) {
        // Doing nothing, just waiting for some time to pass
    }

    printf(" Pazient: %d end operation:%d \n",patient->patientCode,xTaskGetTickCount()/configTICK_RATE_HZ);
    // Notify the scheduler upon completion
    xTaskNotifyGive(xSchedulerTaskHandle);
    vTaskDelete(NULL);
}
```



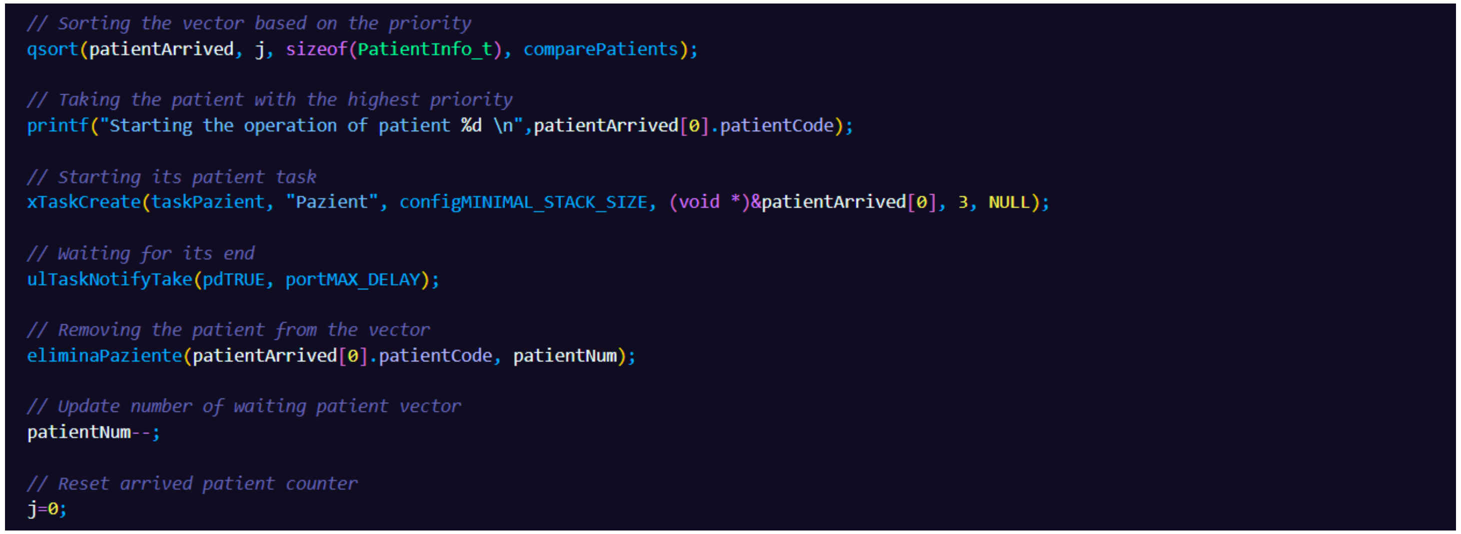
```
// Sorting the vector based on the priority
qsort(patientArrived, j, sizeof(PatientInfo_t), comparePatients);

// Taking the patient with the highest priority
printf("Starting the operation of patient %d \n",patientArrived[0].patientCode);

// Starting its patient task
xTaskCreate(taskPazient, "Pazient", configMINIMAL_STACK_SIZE, (void *)&patientArrived[0], 3, NULL);

// Waiting for its end
ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

// Removing the patient from the vector
eliminaPaziente(patientArrived[0].patientCode, patientNum);

// Update number of waiting patient vector
patientNum--;

// Reset arrived patient counter
j=0;
```

demoHospital2.c

This is how the **patient arrival**, the patient **enqueueing** and surgery operations **starting** is managed by the hospital. The task scheduler calculates priorities based on **critical time**, **operation duration**, and **arrival** time, allowing for a more efficient and **realistic** management of operations.



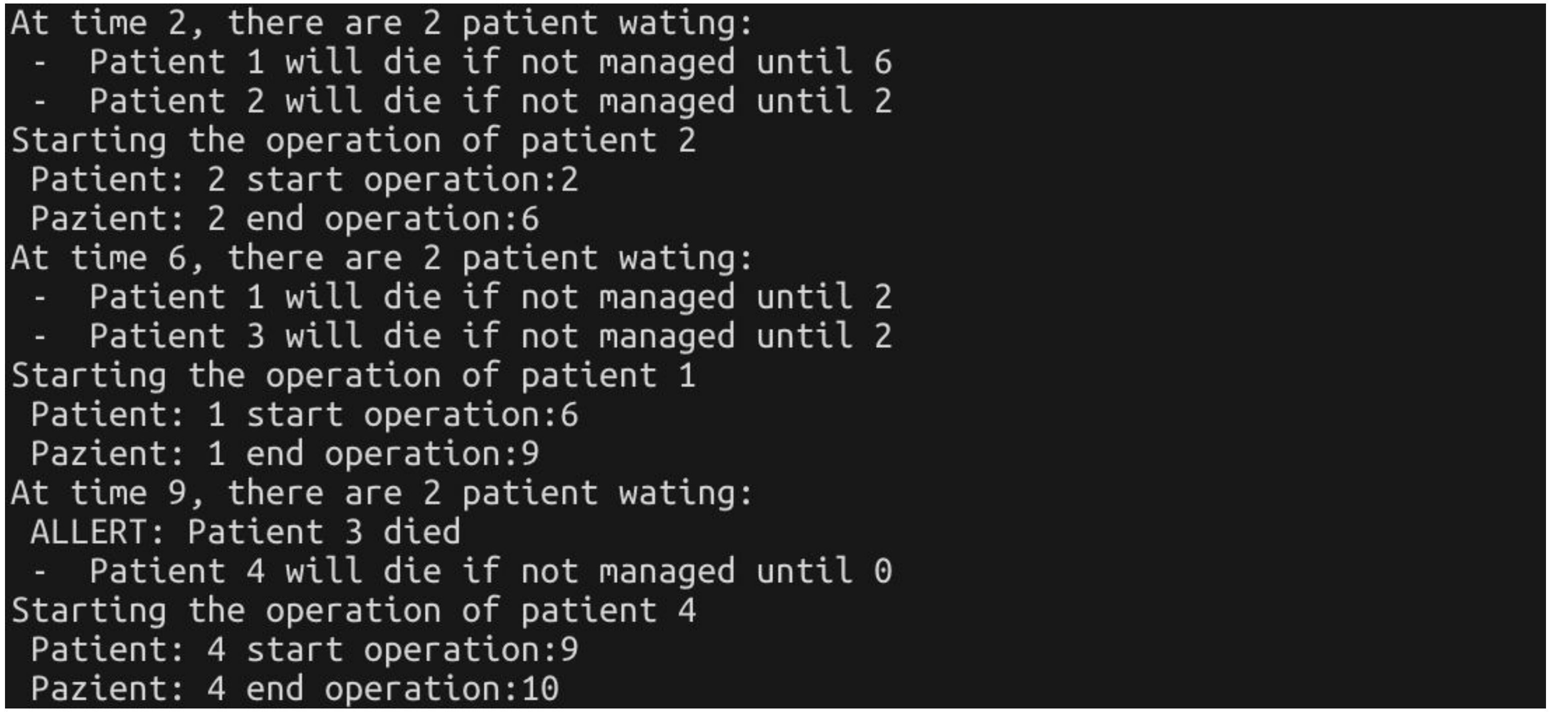
```c
// If a patient arrived
if(j > 0){
    printf("At time %d, there are %d patient wating:\n",xNow/configTICK_RATE_HZ, j);

    // Compute the priority of the arrived patients
    for (int k = 0; k < j; k++) {
        patientArrived[k].priority = patientArrived[k].criticalTime - patientArrived[k].operationDuration + patientArrived[k].arrivalTime - xNow/configTICK_RATE_HZ;
        if(patientArrived[k].priority >= 0)
            printf(" -  Patient %d will die if not managed until %d \n",patientArrived[k].patientCode,patientArrived[k].priority);
        else{
            printf(" ALLERT: Patient %d died\n",patientArrived[k].patientCode);
            eliminaPaziente(patientArrived[k].patientCode, patientNum);
            eliminaPazientePrimaDellOperazione(patientArrived, patientArrived[k].patientCode, patientNum);
            patientNum--;
        }
    }
```



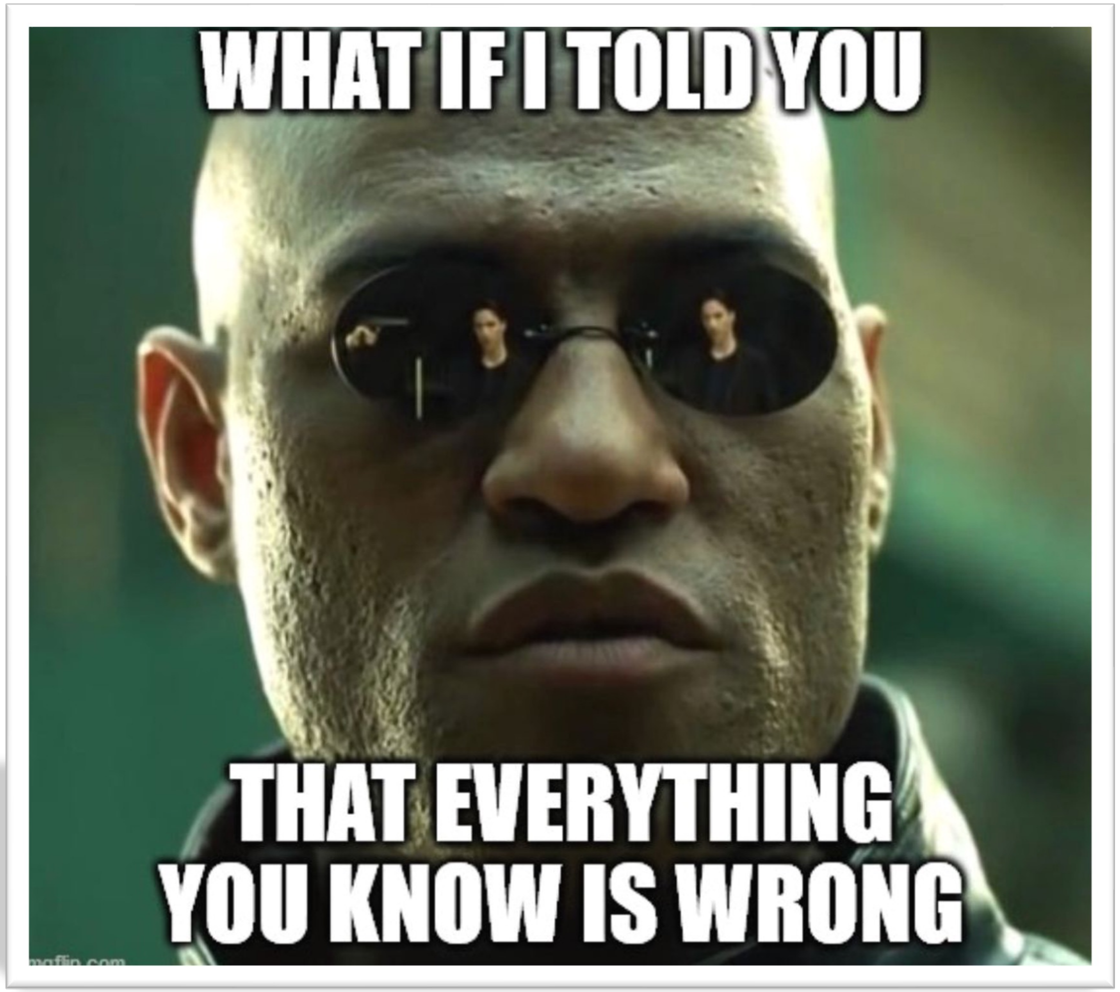
```c
// Sorting the vector based on the priority
qsort(patientArrived, j, sizeof(PatientInfo_t), comparePatients);

// Taking the patient with the highest priority
printf("Starting the operation of patient %d \n",patientArrived[0].patientCode);

// Starting its patient task
xTaskCreate(taskPazient, "Pazient", configMINIMAL_STACK_SIZE, (void *)&patientArrived[0], 3, NULL);

// Waiting for its end
ulTaskNotifyTake(pdTRUE, portMAX_DELAY);

// Removing the patient from the vector
eliminaPaziente(patientArrived[0].patientCode, patientNum);

// Update number of waiting patient vector
patientNum--;

// Reset arrived patient counter
j=0;
```

demoHospital2.c - Output

This is the **results obtained** running the demo.
As we can see, the **patient** with **higher priority** (nearest deadline) is **chosen** to start the surgery operation.

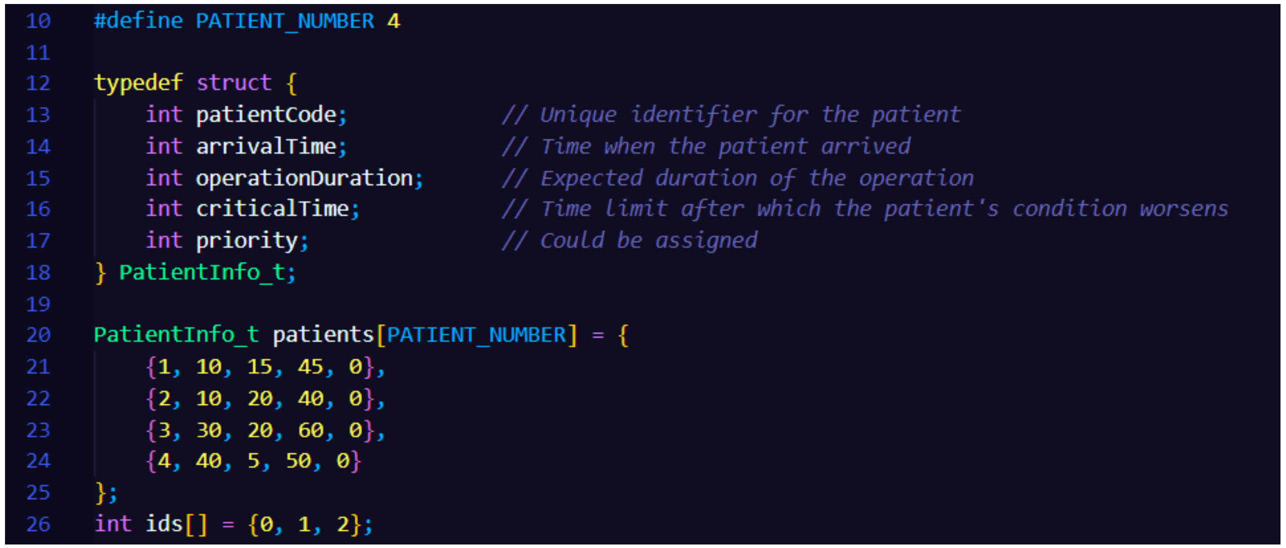


```
At time 2, there are 2 patient wating:
 -   Patient 1 will die if not managed until 6
 -   Patient 2 will die if not managed until 2
Starting the operation of patient 2
 Patient: 2 start operation:2
 Pazient: 2 end operation:6
At time 6, there are 2 patient wating:
 -   Patient 1 will die if not managed until 2
 -   Patient 3 will die if not managed until 2
Starting the operation of patient 1
 Patient: 1 start operation:6
 Pazient: 1 end operation:9
At time 9, there are 2 patient wating:
 ALLERT: Patient 3 died
 -   Patient 4 will die if not managed until 0
Starting the operation of patient 4
 Patient: 4 start operation:9
 Pazient: 4 end operation:10
```


WHAT IF I TOLD YOU
THAT EVERYTHING YOU KNOW IS WRONG
imgflip.com

Scheduler

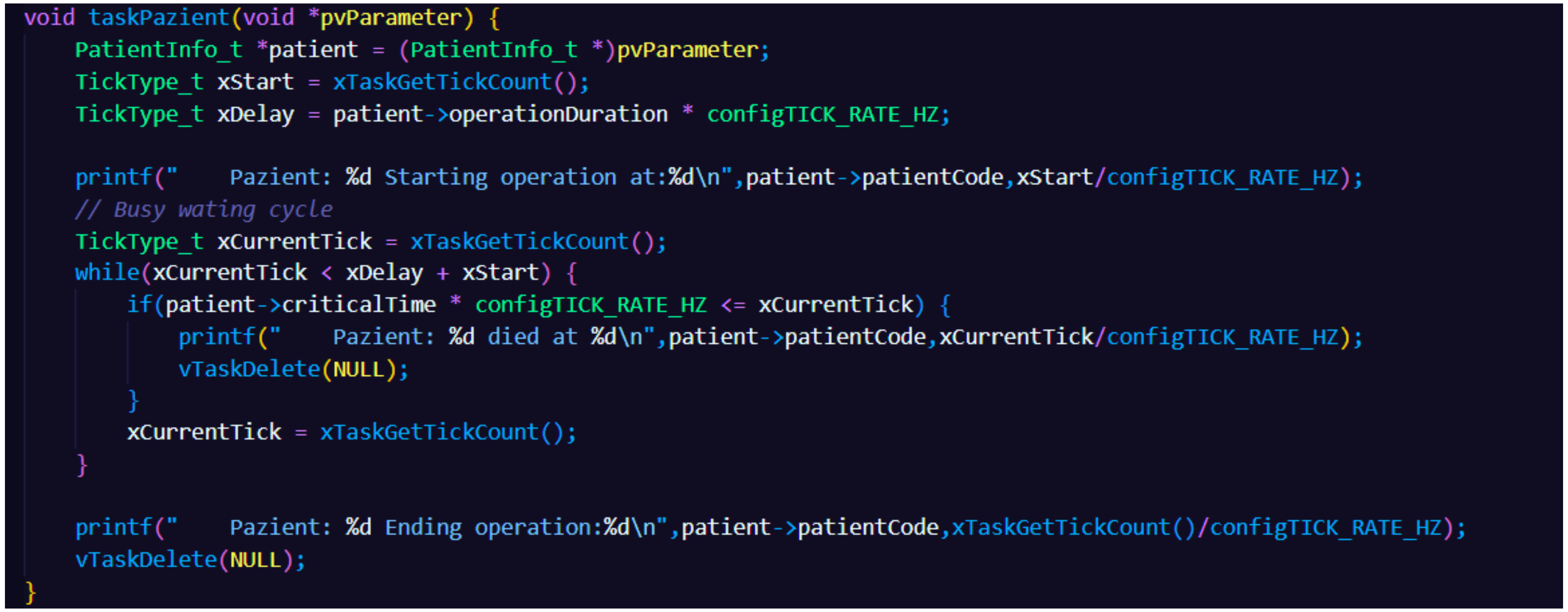
This is the **new** implemented **feature**, which aims to **better schedule** the hospital incoming **patients**, to show an actual implementation of an hospital.
Each **patient** (task) is characterized by an **ID**, **arrival time**, **operation duration** (expected), a **critical time** after which patient's condition worsens, and a **priority**.



```c
#define PATIENT_NUMBER 4

typedef struct {
    int patientCode;            // Unique identifier for the patient
    int arrivalTime;            // Time when the patient arrived
    int operationDuration;      // Expected duration of the operation
    int criticalTime;           // Time limit after which the patient's condition worsens
    int priority;               // Could be assigned
} PatientInfo_t;

PatientInfo_t patients[PATIENT_NUMBER] = {
    {1, 10, 15, 45, 0},
    {2, 10, 20, 40, 0},
    {3, 30, 20, 60, 0},
    {4, 40, 5, 50, 0}
};
int ids[] = {0, 1, 2};
```

Scheduler

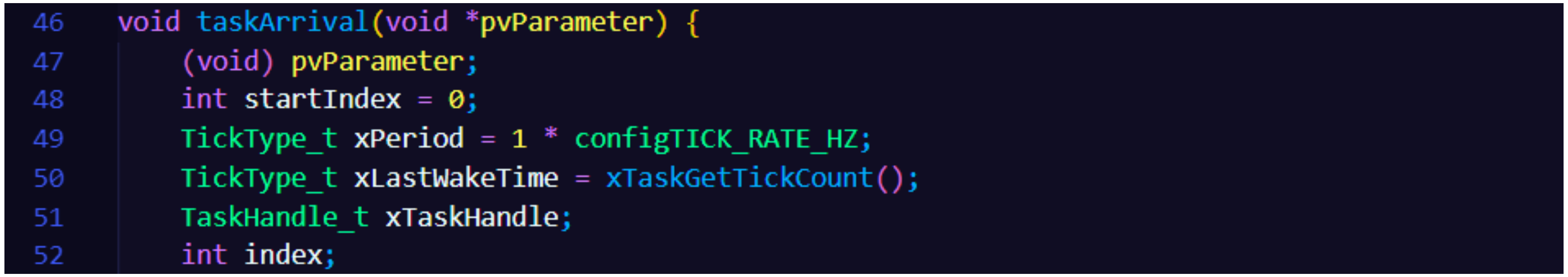
Each **aperiodic task** (patient) which **arrive** at a time and based on its **critical time**, the patient can **die** or **start** the **surgery operation** and occupies the surgery room.
When the **operation ends**, the patient **leave the room**.



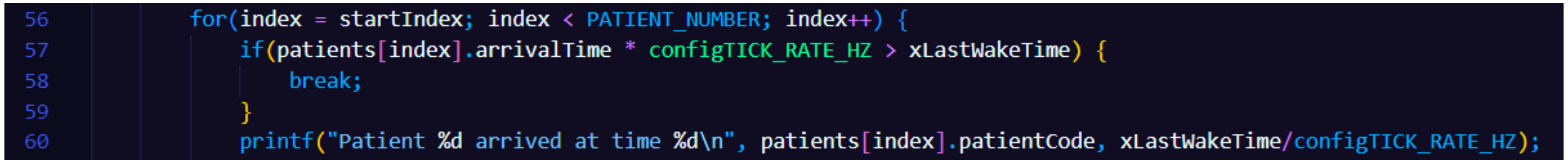
```c
void taskPazient(void *pvParameter) {
    PatientInfo_t *patient = (PatientInfo_t *)pvParameter;
    TickType_t xStart = xTaskGetTickCount();
    TickType_t xDelay = patient->operationDuration * configTICK_RATE_HZ;

    printf("    Pazient: %d Starting operation at:%d\n",patient->patientCode,xStart/configTICK_RATE_HZ);
    // Busy wating cycle
    TickType_t xCurrentTick = xTaskGetTickCount();
    while(xCurrentTick < xDelay + xStart) {
        if(patient->criticalTime * configTICK_RATE_HZ <= xCurrentTick) {
            printf("    Pazient: %d died at %d\n",patient->patientCode,xCurrentTick/configTICK_RATE_HZ);
            vTaskDelete(NULL);
        }
        xCurrentTick = xTaskGetTickCount();
    }

    printf("    Pazient: %d Ending operation:%d\n",patient->patientCode,xTaskGetTickCount()/configTICK_RATE_HZ);
    vTaskDelete(NULL);
}
```

Scheduler

To **simulate the arrivals** of the patients, it was implemented a **periodic task,** which is executed every second and creates the **patient task**.



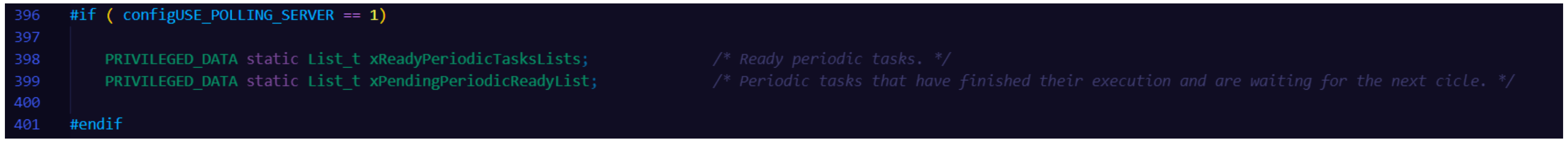
```c
void taskArrival(void *pvParameter) {
    (void) pvParameter;
    int startIndex = 0;
    TickType_t xPeriod = 1 * configTICK_RATE_HZ;
    TickType_t xLastWakeTime = xTaskGetTickCount();
    TaskHandle_t xTaskHandle;
    int index;
```



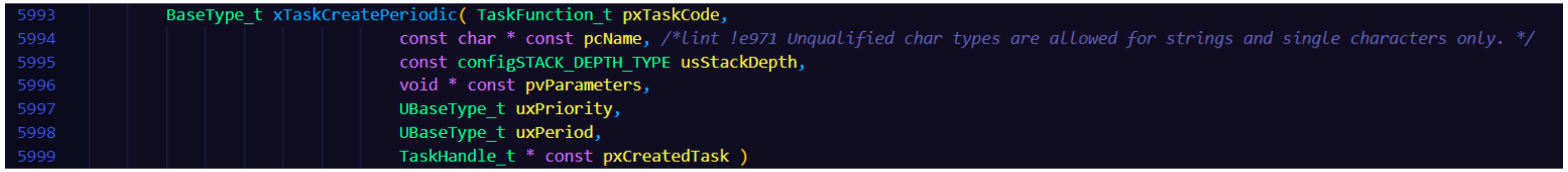
```c
        for(index = startIndex; index < PATIENT_NUMBER; index++) {
            if(patients[index].arrivalTime * configTICK_RATE_HZ > xLastWakeTime) {
                break;
            }
            printf("Patient %d arrived at time %d\n", patients[index].patientCode, xLastWakeTime/configTICK_RATE_HZ);
```

Scheduler

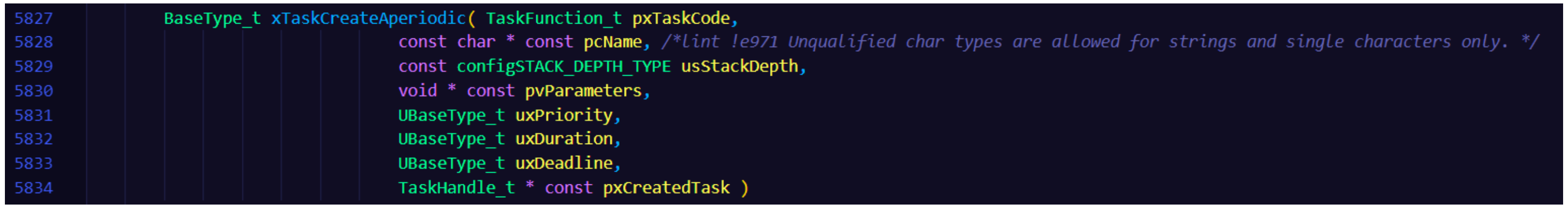
The **scheduler** simulates a sort of **polling server**, which **iteratively checks** if the **surgery room** is **free** and if there is a **patient ready** in the queue.



```
#if ( configUSE_POLLING_SERVER == 1)

    PRIVILEGED_DATA static List_t xReadyPeriodicTasksLists;             /* Ready periodic tasks. */
    PRIVILEGED_DATA static List_t xPendingPeriodicReadyList;            /* Periodic tasks that have finished their execution and are waiting for the next cicle. */

#endif
```



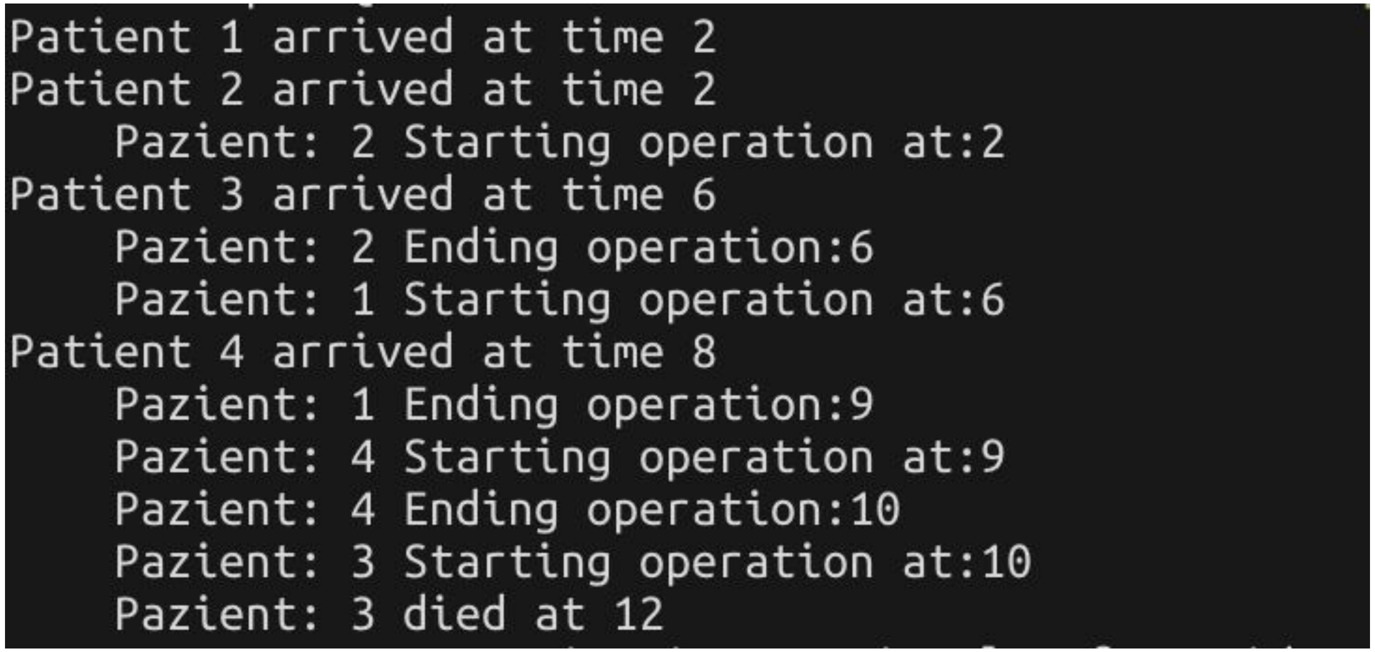
```
        BaseType_t xTaskCreatePeriodic( TaskFunction_t pxTaskCode,
                                        const char * const pcName, /*lint !e971 Unqualified char types are allowed for strings and single characters only. */
                                        const configSTACK_DEPTH_TYPE usStackDepth,
                                        void * const pvParameters,
                                        UBaseType_t uxPriority,
                                        UBaseType_t uxPeriod,
                                        TaskHandle_t * const pxCreatedTask )
```



```
        BaseType_t xTaskCreateAperiodic( TaskFunction_t pxTaskCode,
                                        const char * const pcName, /*lint !e971 Unqualified char types are allowed for strings and single characters only. */
                                        const configSTACK_DEPTH_TYPE usStackDepth,
                                        void * const pvParameters,
                                        UBaseType_t uxPriority,
                                        UBaseType_t uxDuration,
                                        UBaseType_t uxDeadline,
                                        TaskHandle_t * const pxCreatedTask )
```

Scheduler - Output

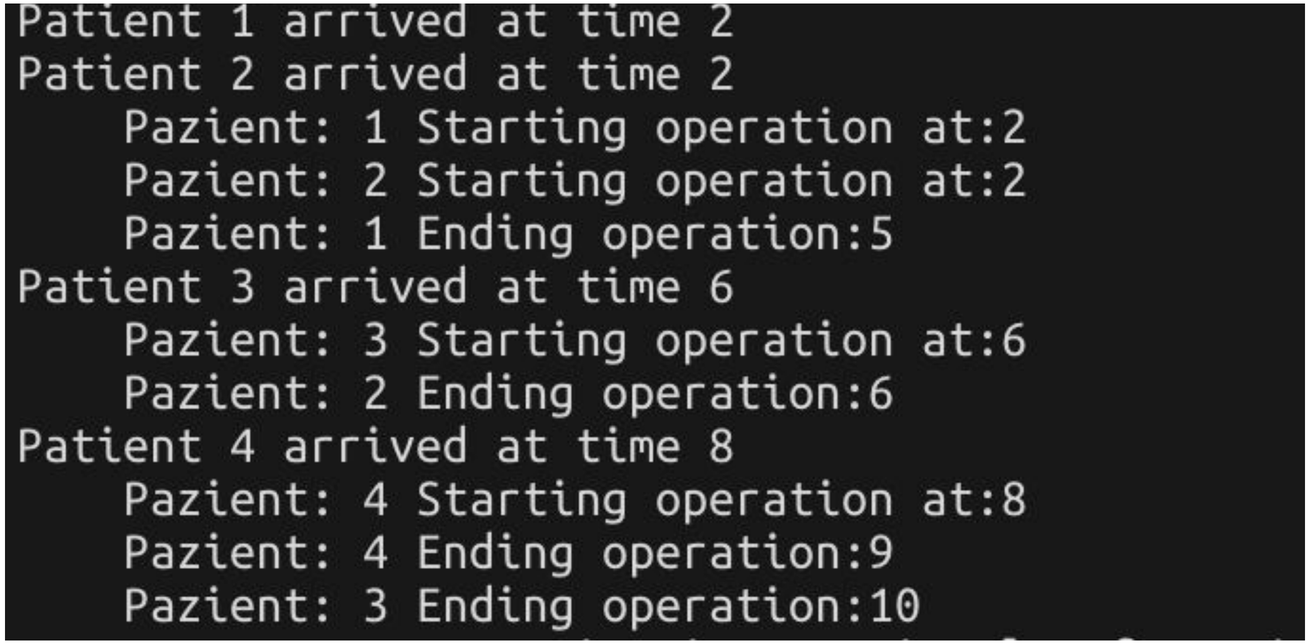
The following **output** shows the **obtained results** running the demoScheduler with **our scheduler** implemented **without preemption**.



```
Patient 1 arrived at time 2
Patient 2 arrived at time 2
    Pazient: 2 Starting operation at:2
Patient 3 arrived at time 6
    Pazient: 2 Ending operation:6
    Pazient: 1 Starting operation at:6
Patient 4 arrived at time 8
    Pazient: 1 Ending operation:9
    Pazient: 4 Starting operation at:9
    Pazient: 4 Ending operation:10
    Pazient: 3 Starting operation at:10
    Pazient: 3 died at 12
```

Scheduler - Output

The following **output** shows the **obtained results** running the demoScheduler with **FreeRTOS scheduler** implemented **with preemption**.



```
Patient 1 arrived at time 2
Patient 2 arrived at time 2
    Pazient: 1 Starting operation at:2
    Pazient: 2 Starting operation at:2
    Pazient: 1 Ending operation:5
Patient 3 arrived at time 6
    Pazient: 3 Starting operation at:6
    Pazient: 2 Ending operation:6
Patient 4 arrived at time 8
    Pazient: 4 Starting operation at:8
    Pazient: 4 Ending operation:9
    Pazient: 3 Ending operation:10
```

Results

FreeRTOS Scheduler With Preemption						
Patient	**Arrival Time**	**Starting Operation**	**Duration**	**Deadline**	**Ending Operation**	**Died (yes/no)**
P1	2	2	3	9	5	no
P2	2	2	4	8	6	no
P3	6	6	4	12	10	no
P4	8	8	1	10	9	no

Our Scheduler Without Preemption						
Patient	**Arrival Time**	**Starting Operation**	**Duration**	**Deadline**	**Ending Operation**	**Died (yes/no)**
P1	2	6	3	9	9	no
P2	2	2	4	8	6	no
P3	6	10	4	12	-	yes :(
P4	8	9	1	10	10	no

THANK YOU!