Politecnico di Torino

Computer Architecture and Operating System

HaclOSsim ~ Group 39

A.A. 2023/2024

Riccardo Bruno, s323563
Luigi Bartolomeo, s324471
Federico Civitareale, s323440

Goal of the project

- Acquire proficiency in utilizing real time OS FreeRTOS exploiting the QEMU simulator.
- Create a tutorial detailing the installation and usage procedures
- Develop practical examples demonstrating the functionality of the OS
- Customize FreeRTOS operating system to implement new feature
- Evaluate the result obtained

TOOLS

- **FreeRTOS**

Stands for Free Real Time Operating System designed for embedded system, as microcontrollers and small processors. It provides a kernel (open source) that is very efficient in task management and scheduling capabilities.

- **QEMU**

Qemu is an open source system emulator, it virtualizes a model of a machine (CPU, memory, etc.) to run a guest OS (in this case FreeRTOS). The qemu machine used for this project is the mps2-an385

- **GDB**

Gdb is a tool for debugging programs. It is used when running the qemu emulator with program executed by FreeRTOS

QEMU: Installation & Usage

To install QEMU (Ubuntu/Debian):

```
$ apt-get install qemu-system
```

To launch QEMU with a compiled kernel file (RTOSDemo.out):

```
$ qemu-system-arm -s -S -M mps2-an385 -cpu cortex-m3 -monitor none -nographic -serial stdio -kernel
/path/to/RTOSDemo.out
```

- -s: shorthand for -gdb tcp::1234
- -S: starts debugger in paused state
- -M: select machine ('-machine help' for list)
- -cpu: select CPU ('-cpu help' for list)
- -monitor: redirect the monitor to char device 'dev'
- -serial: redirect the serial port to char device 'dev'
- -nographic: disable graphical output and redirect serial I/Os to console

ARM Toolchain & GDB

To install GCC from ARM Toolchain (Ubuntu/Debian):

```
$ apt-get install gcc-arm-none-eabi
```

To install GDB (Ubuntu/Debian):

```
$ apt-get install gdb-arm-none-eabi
```

To launch GDB:

```
$ arm-none-eabi-gdb /path/to/RTOSDemo.out -ex "target remote :1234”
```

➢ -ex: executes the standard input, which opens a TCP connection with QEMU system

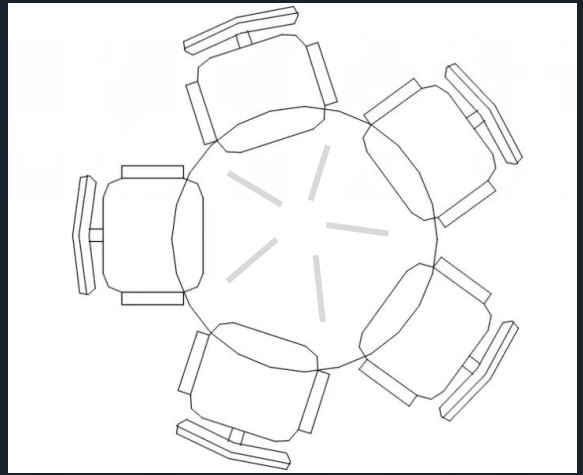
Demo developed

5 DINING PHILOSOPHERS PROBLEM

The Dining Philosophers Problem is a classic synchronization problem in operating systems that illustrates the challenges of resource sharing and deadlock prevention.

There are five philosophers seated around a circular table. Each philosopher has a plate of food, and there is one fork between each pair of philosophers. Each philosopher needs two forks (one on the left and one on the right) to eat.

NO DEADLOCK: even-numbered philosophers first pick up their right fork, then the left one and vice versa for odd philosophers



demoDinner.c

Forks semaphores are initialized and Philosophers Tasks are launched.

```c
void demoDinner() {
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        forks[i] = xSemaphoreCreateBinary();
        xSemaphoreGive(forks[i]);
    }
    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        static int philosopherId[NUM_PHILOSOPHERS];
        philosopherId[i] = i;
        xTaskCreate(philosopherTask, "Philosopher", configMINIMAL_STACK_SIZE * 4, &philosopherId[i], 1, NULL);
    }
    vTaskStartScheduler();
    for (;;);
}
```

The goal of philosopher's task is to pick up the left and right forks if they are available, thanks to the Fork semaphore. When he has both forks, eating is simulated by a delay, and then he releases the forks.

```c
void philosopherTask(void *param) {
    int id = *(int *)param;
    int leftFork = id;
    int rightFork = (id + 1) % NUM_PHILOSOPHERS;
    int firstFork = (id % 2 == 0) ? leftFork : rightFork;
    int secondFork = (id % 2 == 0) ? rightFork : leftFork;
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(1000));
        xSemaphoreTake(forks[firstFork], portMAX_DELAY);
        xSemaphoreTake(forks[secondFork], portMAX_DELAY);
        vTaskDelay(pdMS_TO_TICKS(2000));
        xSemaphoreGive(forks[firstFork]);
        xSemaphoreGive(forks[secondFork]);
    }
}
```

demoDinner.c OUTPUT

This is an excerpt from the demo. It illustrates how the philosophers pick up a fork only when it is available and eat only when they have both forks. If a philosopher has only one fork, they will wait until the other fork is released before picking it up.

```
[ Philosopher 3 ]   picked up fork 3 (SX)
[ Philosopher 1 ]   picked up fork 2 (DX)
[ Philosopher 1 ]   is eating
[ Philosopher 4 ]   finished eating & put down forks
[ Philosopher 5 ]   picked up fork 5 (SX)
[ Philosopher 3 ]   picked up fork 4 (DX)
[ Philosopher 3 ]   is eating
[ Philosopher 2 ]   is thinking
[ Philosopher 4 ]   is thinking
[ Philosopher 1 ]   finished eating & put down forks
[ Philosopher 5 ]   picked up fork 1 (DX)
[ Philosopher 5 ]   is eating
[ Philosopher 3 ]   finished eating & put down forks
[ Philosopher 2 ]   picked up fork 3 (DX)
[ Philosopher 2 ]   picked up fork 2 (SX)
[ Philosopher 2 ]   is eating
[ Philosopher 1 ]   is thinking
[ Philosopher 3 ]   is thinking
```

Demo developed

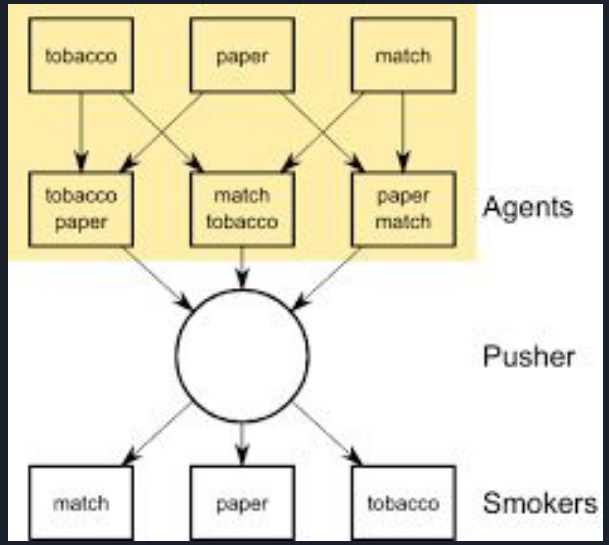
AGENT & SMOKERS PROBLEM

The **Smokers Problem** is a classical synchronization problem in operating systems. It involves **three smokers** and **one agent**, highlighting issues of **resource sharing and process synchronization**.

There are 3 smokers, each having one ingredient: tobacco, papers and matches. There is 1 agent who randomly places two ingredients on the table. Only the smokers who has all three ingredients can smoke.

4 semaphores are implemented in order to prevent race condition, deadlock and starvation:

```
SemaphoreHandle_t xSemaphoreTobaccoSmoker;
SemaphoreHandle_t xSemaphorePaperSmoker;
SemaphoreHandle_t xSemaphoreMatchesSmoker;
SemaphoreHandle_t xSemaphoreSmokerDone;
```



demoSmokers.c

The agent randomly selects two of the three ingredients (tobacco, paper, matches) and places them on the table. The agent notifies the smoker with the missing ingredient using the appropriate semaphore.

```
void taskAgent(void *pvParameter) {
    while (1) {
        int choice = custom_rand() % 3;
        printf("\033[1;92m[+]\033[0m  Agent places ");
        switch (choice) {
            case TOBACCO:
                printf("\033[92mPAPER\033[0m & \033[92mMATCHES\033[0m\n");
                xSemaphoreGive(xSemaphoreTobaccoSmoker);
                break;
            case PAPER:
                printf("\033[92mTOBACCO\033[0m & \033[92mMATCHES\033[0m\n")
                xSemaphoreGive(xSemaphorePaperSmoker);
                break;
            case MATCHES:
                printf("\033[92mTOBACCO\033[0m & \033[92mPAPER\033[0m\n");
                xSemaphoreGive(xSemaphoreMatchesSmoker);
                break;
        }
        xSemaphoreTake(xSemaphoreSmokerDone, portMAX_DELAY);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

demoSmokers.c

Each smoker waits for the agent to provide the required ingredients through the relative semaphore, smokes, and then notifies the agent that they have finished smoking.

```
void taskSmoker(void *pvParameter) {
    int smokerType = *(int *)pvParameter;
    while (1) {
        switch (smokerType) {
            case TOBACCO:
                xSemaphoreTake(xSemaphoreTobaccoSmoker, portMAX_DELAY);
                break;
            case PAPER:
                xSemaphoreTake(xSemaphorePaperSmoker, portMAX_DELAY);
                break;
            case MATCHES:
                xSemaphoreTake(xSemaphoreMatchesSmoker, portMAX_DELAY);
                break;
        }
        printf("\033[1;90m[~]\033[0m  Smoker with ");
        switch (smokerType) {
            case TOBACCO: printf("TOBACCO "); break;
            case PAPER: printf("PAPER "); break;
            case MATCHES: printf("MATCHES "); break;
        }
        printf("\033[1;90msmokes\033[0m\n");
        printf("\033[1;90m[~]\033[0m\t    \033[1;90m. . .\033[0m\n");
        vTaskDelay(pdMS_TO_TICKS(2000));
        printf("\033[1;92m[*]\033[0m  Smoker \033[92mfinished\033[0m smoking\n\n");
        xSemaphoreGive(xSemaphoreSmokerDone);
    }
}
```

demoSmokers.c

This function sets up the necessary semaphores and tasks to simulate the smokers problem. It creates binary semaphores for each smoker type and a semaphore to signal when a smoker is done. It then creates the agent task and three smoker tasks, each with a different type of smoker.

```c
void demoSmokers() {

    xSemaphoreTobaccoSmoker = xSemaphoreCreateBinary();
    xSemaphorePaperSmoker = xSemaphoreCreateBinary();
    xSemaphoreMatchesSmoker = xSemaphoreCreateBinary();
    xSemaphoreSmokerDone = xSemaphoreCreateBinary();
    xTaskCreate(taskAgent, "Agent", configMINIMAL_STACK_SIZE, NULL, 2, NULL);
    static int smoker1 = TOBACCO, smoker2 = PAPER, smoker3 = MATCHES;
    xTaskCreate(taskSmoker, "SmokerTobacco", configMINIMAL_STACK_SIZE, &smoker1, 1, NULL);
    xTaskCreate(taskSmoker, "SmokerPaper", configMINIMAL_STACK_SIZE, &smoker2, 1, NULL);
    xTaskCreate(taskSmoker, "SmokerMatches", configMINIMAL_STACK_SIZE, &smoker3, 1, NULL);
    vTaskStartScheduler();
    for (;;);
}
```

demoSmokers.c OUTPUT

This is the output of the demo. As we can see, every time the agent places 2 ingredients the smoker with the 3rd ingredient enters, pick up the ingredients and smoke.
After finishing his cigarette, he leaves, and the agent places two new ingredients on the table.

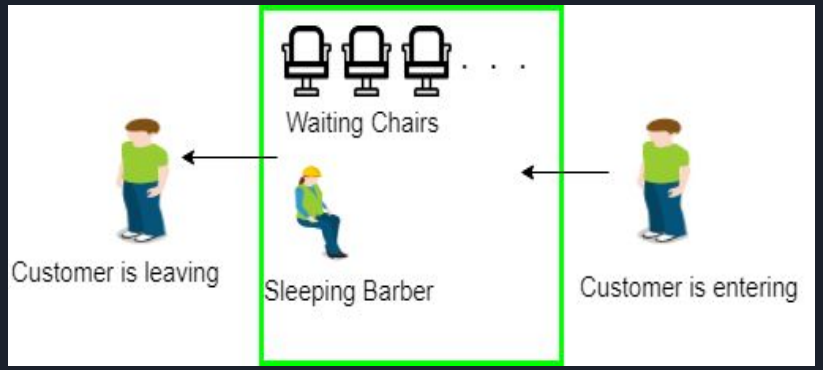
```
[+]  Agent places TOBACCO & PAPER
[~]  Smoker with MATCHES smokes
[~]          . . .
[*]  Smoker finished smoking

[+]  Agent places TOBACCO & MATCHES
[~]  Smoker with PAPER smokes
[~]          . . .
[*]  Smoker finished smoking

[+]  Agent places PAPER & MATCHES
[~]  Smoker with TOBACCO smokes
[~]          . . .
[*]  Smoker finished smoking

[+]  Agent places TOBACCO & MATCHES
[~]  Smoker with PAPER smokes
[~]          . . .
[*]  Smoker finished smoking

[+]  Agent places TOBACCO & MATCHES
[~]  Smoker with PAPER smokes
[~]          . . .
[*]  Smoker finished smoking
```

Demo developed

SLEEPING BARBER PROBLEM

The Sleeping Barber Problem is a classic synchronization problem in operating systems, illustrating issues with process synchronization.

There is a barber who always wants to sleep when no customer is in the shop. N customer arrives at different time but there are only S seats (3 seats). If all chair are occupied, the customer leaves the shop.



To solve this problem, semaphores and queues are implemented:

```
QueueHandle_t xWaitingRoomQueue;
SemaphoreHandle_t xBarberChair;
```

demoBarber.c

The barber either sleeps or cuts hair.
When cutting hair the chair is released
No more customers: barber goes to sleep

```c
void vBarberTask( void *pvParameters )
{
    int customerID;
    BaseType_t xStatus;
    ( void ) pvParameters;
    for ( ;; ) {
        xStatus = xQueueReceive(xWaitingRoomQueue, &customerID, portMAX_DELAY);
        if (xStatus == pdPASS) {
            printf("\033[95m [  BARBER  ]\033[0m\t\033[1;90mWakes up\033[0m\n");
            xSemaphoreTake(xBarberChair, portMAX_DELAY);
            printf("\033[95m [  BARBER  ]\033[0m\t\033[93mCutting\033[0m customer %d's hair\n", customerID);
            vTaskDelay(pdMS_TO_TICKS(HAIR_CUT_DURATION_MS));
            printf("\033[95m [  BARBER  ]\033[0m\t\033[92mFinished haircut for customer %d\033[0m\n", customerID);
            xSemaphoreGive(xBarberChair);
            if (uxQueueMessagesWaiting(xWaitingRoomQueue) == 0) {
                printf("\033[95m [  BARBER  ]\033[0m\t\033[1;90mGoes to sleep\033[0m\n");
            }
        } else {
            printf("\033[93m[!]\033[0m\t\033[41mError receiving customers in waiting room queue\033[0m\n");
        }
    }
}
```

The customer arrives:

1. checks for a free chair in the waiting room
2. sits if there is room, else leaves

```c
void vCustomerTask( void *pvParameters )
{
    int customerID = (int)pvParameters;
    BaseType_t xStatus;
    printf("\033[95m[ CUSTOMER %d ]\033[0m\t\033[92mArrived\033[0m at the barber shop\n", customerID);
    if (uxQueueMessagesWaiting(xWaitingRoomQueue) < NUM_SEATS) {

        printf("\033[95m[ CUSTOMER %d ]\033[0m\tEntering the \033[93mwaiting room\033[0m\n", customerID);
        xStatus = xQueueSendToBack(xWaitingRoomQueue, &customerID, 0);
        if (xStatus != pdPASS) {
            printf("\033[95m[ CUSTOMER %d ]\033[0m\tLeaves: \033[91mwaiting room is full\033[0m\n", customerID);
        }
    } else {
        printf("\033[95m[ CUSTOMER %d ]\033[0m\tLeaves: \033[91mwaiting room is full\033[0m\n", customerID);
    }
    vTaskDelete(NULL);
}
```

demoBarber.c

xWaitingRoomQueue
xBarberChair semaphores.

- 1 vBarberTask
- 1 vCustomerGeneratorTask
  - N vCustomerTask

vBarberTask has highest priority

```c
int demoBarber( void )
{
    BaseType_t xReturned;
    xWaitingRoomQueue = xQueueCreate(NUM_SEATS, sizeof(int));
    xBarberChair = xSemaphoreCreateBinary();
    if (xBarberChair == NULL || xWaitingRoomQueue == NULL) {
        printf( "\033[93m[!]\033[0m\t\033[41mError creating barber semaphore or waiting room queue\033[0m\n" );
        return 1;
    }
    xSemaphoreGive(xBarberChair);
    xReturned = xTaskCreate(
        vBarberTask,                    /* Function that generates barber task. */
        "Barber",                       /* Text name for the task. */
        configMINIMAL_STACK_SIZE,       /* Stack size in words, not bytes. */
        NULL,                           /* Parameter passed into the task. */
        tskIDLE_PRIORITY + 2,           /* Priority at which the task is created. */
        NULL );                         /* Used to pass out the created task's handle. */

    if (xReturned != pdPASS) {
        printf("\033[93m[!]\033[0m\t\033[41mError creating barber task\033[0m\n");
        return 1;
    }
    xReturned = xTaskCreate(
                    vCustomerGeneratorTask,     /* Function that generates customer tasks. */
                    "CustomerGenerator",        /* Text name for the task. */
                    configMINIMAL_STACK_SIZE,   /* Stack size in words, not bytes. */
                    NULL,                       /* Parameter passed into the task. */
                    tskIDLE_PRIORITY + 1,       /* Priority at which the task is created. */
                    NULL );                     /* Used to pass out the created task's handle. */
```

demoBarber.c OUTPUT

The output shows that customers arrive at different times, and the barber performs haircuts in their arrival time order

In this instance, all customers were able to enter the waiting room and receive their haircuts successfully.

```
[*] BARBER SHOP IS NOW OPEN [*]
      - 3 Seats
      - 5 Customers
[ CUSTOMER 1 ]  Arrived at the barber shop
[ CUSTOMER 1 ]  Entering the waiting room
[  BARBER  ]    Wakes up
[  BARBER  ]    Cutting customer 1's hair
[ CUSTOMER 2 ]  Arrived at the barber shop
[ CUSTOMER 2 ]  Entering the waiting room
[ CUSTOMER 3 ]  Arrived at the barber shop
[ CUSTOMER 3 ]  Entering the waiting room
[ CUSTOMER 4 ]  Arrived at the barber shop
[ CUSTOMER 4 ]  Entering the waiting room
[  BARBER  ]    Finished haircut for customer 1
[  BARBER  ]    Wakes up
[  BARBER  ]    Cutting customer 2's hair
[ CUSTOMER 5 ]  Arrived at the barber shop
[ CUSTOMER 5 ]  Entering the waiting room
[  BARBER  ]    Finished haircut for customer 2
[  BARBER  ]    Wakes up
[  BARBER  ]    Cutting customer 3's hair
[  BARBER  ]    Finished haircut for customer 3
[  BARBER  ]    Wakes up
[  BARBER  ]    Cutting customer 4's hair
[  BARBER  ]    Finished haircut for customer 4
[  BARBER  ]    Wakes up
[  BARBER  ]    Cutting customer 5's hair
[  BARBER  ]    Finished haircut for customer 5
[  BARBER  ]    Goes to sleep
```

Times change and so the barber

- The barber now is caffeinated and workaholic
- He is designing the most efficient barber shop in town
- He is uncertain whether how to serve customers:
  - PREEMPTIVELY
  - NON-PREEMPTIVELY

SCHEDULER

In this new version of the barber shop have been added some customers feature, namely the arrival time, the service time (how long it takes for the haircut) and the expiration time (deadline).

Like in the previous scenario, there is just one barber, and the scheduler has been tested with 5 customers arriving and a waiting room of a maximum of 3 seats.

```c
#define NUM_CUSTOMERS 5
#define NUM_SEATS 3
#define HAIRCUT_TIME 2000
#define CUSTOMER_ARRIVAL 500

You, 5 hours ago | 1 author (You)
typedef struct {
    int id;                 // Customer ID
    int arrivalTime;        // Time when the customer arrived
    int expirationTime;     // Time after which the customer leaves
    int serviceTime;        // Haircut duration
    int remainingTime;      // Remaining time when preempted
} CustomerData_t;


QueueHandle_t xWaitingRoomQueue;
SemaphoreHandle_t xBarberChair;
TaskHandle_t xSchedulerTaskHandle;
int servedCustomers = 0, lostCustomers = 0, preemptedCustomers = 0;

CustomerData_t customers[NUM_CUSTOMERS] = {
    {1, 1, 10, 2, 0},
    {2, 3, 8, 5, 0},
    {3, 5, 12, 1, 0},
    {4, 1, 7, 1, 0},
    {5, 4, 15, 2, 0},
};
```

demoScheduler.c

This task continuously checks for customers in the waiting room queue and serves them based on their priority and deadlines.
The barber can preempt the current customer if a higher-priority customer arrives, depending on the configuration.

- If the barber is not busy, it picks the highest-priority customer from the queue.
- Checks if the customer's deadline has expired. If expired, the customer is marked as lost.
- If the customer was preempted before, it resumes from the remaining time.
- During the haircut, it periodically checks if the customer's deadline has expired.
- The task yields periodically to allow other tasks to run.

```c
void vBarberTask(void *pvParameters) {
    CustomerData_t currentCustomer;
    int barberBusy = 0;       You, 4 hours ago • Uncommitted changes

    for (;;) {
        if (!barberBusy) {
            // Pick the highest-priority customer from the queue
            if (xQueueReceive(xWaitingRoomQueue, &currentCustomer, portMAX_DELAY) == pdTRUE) {
                TickType_t currentTime = xTaskGetTickCount();

                // **Check if the customer's deadline has already expired**
                if (currentTime >= (currentCustomer.expirationTime * configTICK_RATE_HZ)) {
                    printf("\033[95m[ CUSTOMER %d ]\033[0m\tLeft: \033[91mEXPIRED\033[0m @ \033[1;90m[%ds] \033[0m\n",
                           currentCustomer.id, currentTime / configTICK_RATE_HZ);
                    lostCustomers++;
                    continue;
                }

                // If this customer was preempted before, resume from remaining time
                if (currentCustomer.remainingTime == 0) {
                    currentCustomer.remainingTime = currentCustomer.serviceTime * configTICK_RATE_HZ;
                }

                printf("\033[95m[ CUSTOMER %d ]\033[0m\t\033[92mCutting\033[0m hair     @ \033[1;90m[%ds] \033[0m\n",
                       currentCustomer.id, currentTime / configTICK_RATE_HZ);
                barberBusy = 1;
            }
        }

        if (barberBusy) {
            // Check if a more urgent customer has arrived
            CustomerData_t nextCustomer;
            if (xQueuePeek(xWaitingRoomQueue, &nextCustomer, 0) == pdTRUE) {

                if (nextCustomer.expirationTime < currentCustomer.expirationTime) {

                    // Conditional Preemption based on config flag
                    #if configUSE_HAIRCUT_PREEMPTION
                        printf("\033[95m[ CUSTOMER %d ]\033[0m\t\033[1;93mPreempted\033[0m customer %d @ \033[1;90m[%ds] \033[0m\n",
                               nextCustomer.id, currentCustomer.id, xTaskGetTickCount() / configTICK_RATE_HZ);
                        preemptedCustomers++;

                        // Store remaining time before preempting
                        currentCustomer.remainingTime -= (xTaskGetTickCount() - (currentCustomer.serviceTime * configTICK_RATE_HZ - currentCustomer.remainingTime));

                        // Put the unfinished customer back in the queue
                        xQueueSendToBack(xWaitingRoomQueue, &currentCustomer, 0);

                        // Switch to the new customer immediately
                        barberBusy = 0;
                        taskYIELD();
                        continue;
                    #endif
                }
            }

            // Check during the haircut if the deadline has expired
            if (xTaskGetTickCount() >= (currentCustomer.expirationTime * configTICK_RATE_HZ)) {
                printf("\033[95m[  BARBER  ]\033[0m\t\033[91mCustomer \033[1m%d\033[0m left\033[0m mid-haircut: \033[91mEXPIRED\033[0 @ \033[1;90m[%ds] \033[0m\n",
                       currentCustomer.id, xTaskGetTickCount() / configTICK_RATE_HZ);
                lostCustomers++;
                barberBusy = 0;
                continue;
            }

            // Continue haircut for remaining time
            vTaskDelay(pdMS_TO_TICKS(1000)); // Simulate 1 second of cutting
            currentCustomer.remainingTime -= configTICK_RATE_HZ;

            if (currentCustomer.remainingTime <= 0) {
                printf("\033[95m[ CUSTOMER %d ]\033[0m\t\033[1;42mFinished\033[0m haircut @ \033[1;90m[%ds] \033[0m\n",
                       currentCustomer.id, xTaskGetTickCount() / configTICK_RATE_HZ);
                servedCustomers++;
                barberBusy = 0;
            }
        }

        vTaskDelay(pdMS_TO_TICKS(50));
```

demoScheduler.c

The customer's priority is dynamically set based on their expiration time, and they attempt to enter the waiting room queue. If the queue is full, the customer is considered lost.

- Calculates the customer's priority based on their expiration time.
- Checks if there is space in the waiting room queue.

```
void vCustomerTask(void *pvParameters) {
    CustomerData_t *customer = (CustomerData_t *)pvParameters;

    UBaseType_t minPriority = tskIDLE_PRIORITY + 1;
    UBaseType_t maxPriority = configMAX_PRIORITIES - 1;

    // Scale expiration time to fit within FreeRTOS priority range
    UBaseType_t priority = minPriority + ((customer->expirationTime * (maxPriority - minPriority)) / 15);

    // Set dynamic priority
    vTaskPrioritySet(NULL, priority);

    printf("\033[95m[ CUSTOMER %d ]\033[0m\t\033[93mWaiting\033[0m in room  @ \033[1;90m[%ds], \033[1;90mEXPIRATION:\033[0m %d sec, \033
        customer->id, xTaskGetTickCount() / configTICK_RATE_HZ, customer->expirationTime, priority);

    if (uxQueueMessagesWaiting(xWaitingRoomQueue) < NUM_SEATS) {
        xQueueSendToBack(xWaitingRoomQueue, customer, 0);

        // Force a reschedule to allow higher-priority tasks to take over.
        taskYIELD();
    } else {
        printf("\033[95m[ CUSTOMER %d ]\033[0m\tLeft: \033[91mNo seats available\033[91m\n", customer->id);
        lostCustomers++;
    }

    vTaskDelete(NULL);
}
```

demoScheduler.c

The first function is called when a customer generation timer expires. It retrieves the customer index from the timer ID, calculates the priority for the customer task, and creates the customer task.

The second function sorts the customers based on their arrival and expiration times using the Earliest Deadline First (EDF) scheduling algorithm. It then creates and starts a timer for each customer, which will trigger the customer generation callback when the timer expires.

```c
void vCustomerGeneratorCallback(TimerHandle_t xTimer) {
    int customerIndex = (int)pvTimerGetTimerID(xTimer);
    UBaseType_t priority = tskIDLE_PRIORITY + 1 + (NUM_CUSTOMERS - customerIndex); // Assign higher priority to earlier customers
    xTaskCreate(vCustomerTask, "Customer", configMINIMAL_STACK_SIZE, &customers[customerIndex], priority, NULL);
}

void vCustomerGeneratorTask(void *pvParameters) {
    ( void ) pvParameters;
    // Sort customers first by arrival time, then by expiration time (EDF)
    for (int i = 0; i < NUM_CUSTOMERS - 1; i++) {
        for (int j = i + 1; j < NUM_CUSTOMERS; j++) {
            if (customers[i].arrivalTime > customers[j].arrivalTime ||
               (customers[i].arrivalTime == customers[j].arrivalTime && customers[i].expirationTime > customers[j].expirationTime)) {
                CustomerData_t temp = customers[i];
                customers[i] = customers[j];
                customers[j] = temp;
            }
        }
    }

    TimerHandle_t xTimers[NUM_CUSTOMERS];
    for (int i = 0; i < NUM_CUSTOMERS; i++) {
        xTimers[i] = xTimerCreate("CustomerTimer", pdMS_TO_TICKS(customers[i].arrivalTime * 1000), pdFALSE, (void *)i, vCustomerGeneratorCallback);
        xTimerStart(xTimers[i], 0);
    }
    vTaskDelete(NULL);
}
```

demoScheduler.C OUTPUT with Preemption

The outcomes show that every customer is satisfied because their deadlines are respected. When the 2nd customer, who has a shorter deadline, arrives at the shop, the barber is already cutting hair for the 1st customer. However, since preemption is enabled, the barber pauses work on the 1st customer and begins with the 2nd one to ensure he finishes on time. After completing the 2nd customer, the barber returns to the 1st and then reorders the remaining customers based on their deadlines, prioritizing the 3rd customer before the 5th.

```
[*] BARBER SCHEDULER DATA [*]
TOTAL Customers:        5
PREEMPTED Customers     2   (PREEMPTION: 1)
LOST Customers          0
SERVED Customers        5
```

```
[*] BARBER SCHEDULER [*]
SEATS     = 3
CUSTOMERS = 5
HAIRCUT PREEMPTION 1
CUSTOMERS DATA:
+----+--------------+-----------------+--------------+
| ID | Arrival Time | Expiration Time | Service Time |
|    |              |    (PRIORITY)   |              |
+----+--------------+-----------------+--------------+
| 1  | 1            | 10              | 2            |
| 2  | 3            | 8               | 5            |
| 3  | 5            | 12              | 1            |
| 4  | 1            | 7               | 1            |
| 5  | 4            | 15              | 2            |
+----+--------------+-----------------+--------------+
[*] BARBER SHOP IS OPEN [*]

[ CUSTOMER 4 ]  Waiting in room  @ [1s], EXPIRATION: 7 sec, PRIORITY: 4
[ CUSTOMER 4 ]  Cutting hair @ [1s]
[ CUSTOMER 1 ]  Waiting in room  @ [1s], EXPIRATION: 10 sec, PRIORITY: 5
[ CUSTOMER 4 ]  Finished haircut @ [2s]
[ CUSTOMER 1 ]  Cutting hair @ [2s]
[ CUSTOMER 2 ]  Waiting in room  @ [3s], EXPIRATION: 8 sec, PRIORITY: 4
[ CUSTOMER 2 ]  Preempted customer 1 @ [3s]
[ CUSTOMER 2 ]  Cutting hair @ [3s]
[ CUSTOMER 5 ]  Waiting in room  @ [4s], EXPIRATION: 15 sec, PRIORITY: 8
[ CUSTOMER 3 ]  Waiting in room  @ [5s], EXPIRATION: 12 sec, PRIORITY: 6
[ CUSTOMER 2 ]  Finished haircut @ [8s]
[ CUSTOMER 1 ]  Cutting hair @ [8s]
[ CUSTOMER 1 ]  Finished haircut @ [9s]
[ CUSTOMER 5 ]  Cutting hair @ [9s]
[ CUSTOMER 3 ]  Preempted customer 5 @ [9s]
[ CUSTOMER 3 ]  Cutting hair @ [9s]
[ CUSTOMER 3 ]  Finished haircut @ [10s]
[ CUSTOMER 5 ]  Cutting hair @ [10s]
[ CUSTOMER 5 ]  Finished haircut @ [11s]
```

demoScheduler.C OUTPUT without Preemption

In the scenario without preemption, an angry customer leaves because his deadline was not met. In contrast to the previous case, the 2nd customer waits for his haircut after the 1st one, which means his service begins at 4 seconds. Since his deadline is at 8 seconds and his haircut takes 5 seconds, the barber fails to complete his haircut on time, resulting in a lost client. Unlike in the preemption mode, the 3rd and 5th customers are not swapped, as the operating system cannot interrupt the 5th customer to attend to the 3rd one because of the arrival time (unlikely from the 4th and 1st, who arrive at the same time so it can decide who to serve first).

```
[*] BARBER SCHEDULER DATA [*]
TOTAL Customers:        5
PREEMPTED Customers     0   (PREEMPTION: 0)
LOST Customers          1
SERVED Customers        4
```

```
[*] BARBER SCHEDULER [*]
SEATS      = 3
CUSTOMERS = 5
HAIRCUT PREEMPTION 0
CUSTOMERS DATA:
+----+--------------+-----------------+--------------+
| ID | Arrival Time | Expiration Time | Service Time |
|    |              |    (PRIORITY)   |              |
+----+--------------+-----------------+--------------+
| 1  | 1            | 10              | 2            |
| 2  | 3            | 8               | 5            |
| 3  | 5            | 12              | 1            |
| 4  | 1            | 7               | 1            |
| 5  | 4            | 15              | 2            |
+----+--------------+-----------------+--------------+
[*] BARBER SHOP IS OPEN [*]

CUSTOMER 4 ]  Waiting in room  @ [1s], EXPIRATION: 7 sec, PRIORITY: 4
CUSTOMER 4 ]  Cutting hair @ [1s]
CUSTOMER 1 ]  Waiting in room  @ [1s], EXPIRATION: 10 sec, PRIORITY: 5
CUSTOMER 4 ]  Finished haircut @ [2s]
CUSTOMER 1 ]  Cutting hair @ [2s]
CUSTOMER 2 ]  Waiting in room  @ [3s], EXPIRATION: 8 sec, PRIORITY: 4
CUSTOMER 5 ]  Waiting in room  @ [4s], EXPIRATION: 15 sec, PRIORITY: 8
CUSTOMER 1 ]  Finished haircut @ [4s]
CUSTOMER 2 ]  Cutting hair @ [4s]
CUSTOMER 3 ]  Waiting in room  @ [5s], EXPIRATION: 12 sec, PRIORITY: 6
[  BARBER  ]  Customer 2 left mid-haircut: EXPIRED  [8s]
[ CUSTOMER 5 ]  Cutting hair @ [8s]
[ CUSTOMER 5 ]  Finished haircut @ [10s]
[ CUSTOMER 3 ]  Cutting hair @ [10s]
[ CUSTOMER 3 ]  Finished haircut @ [11s]
```

THANK YOU FOR YOUR ATTENTION