FINAL ASSIGNMENT

Anaïs Cabrol

anais.cabrol@ipsa.fr

April 2024

Table of contents

Introduction

In this project, we will Design an RTOS with different task to learn how to create and use a Real Time Operating System. The different task will be :

• Periodic Task 1: print “Working” or something else which says everything is working as normal

• Periodic Task 2: Convert a fixed Fahrenheit temperature value to degree Celsius

• Periodic Task 3: Define any two long int big numbers and multiply them, print the result

• Periodic Task 4: Binary search a list of 50 elements (fix the list and element to search)

Explanations

Initialisation

We start by importing the libraries we are going to use:

```
#include <stdio.h>
#include <windows.h>
```

The first is the basic library to import in order to have the inputs and outputs of our coe

And the second will be used to use the Windows tools (because I work on Windows and not Linux).

Then we define the variables that we're going to use afterwards :

```

#define NUM_TASKS 4

#define TASK1_PERIOD_MS 1000
#define TASK2_PERIOD_MS 2000
#define TASK3_PERIOD_MS 3000
#define TASK4_PERIOD_MS 4000

#define LIST_SIZE 50
#define SEARCH_ELEMENT 25
```

Here we define the period for each task.

Then declare the mutex

```
HANDLE hMutex;
```

This will be used to declare synchronised access to shared resources between tasks.

Then comes this little piece of code:

```
typedef struct {
    void (*task)(void);
    unsigned long period;
    unsigned long last_run;
} Task;
```

We'll use it to look at the tasks and wait until it's finished before moving on to the next one.

Then we declare the functions for each task, which we'll define next:

```
void task1(void);
void task2(void);
void task3(void);
void task4(void);
```

And we also define how we're going to use them (using their addresses) :

```
Task tasks[NUM_TASKS] = {
    {&task1, TASK1_PERIOD_MS, 0},
    {&task2, TASK2_PERIOD_MS, 0},
    {&task3, TASK3_PERIOD_MS, 0},
    {&task4, TASK4_PERIOD_MS, 0}
};
```

Task 1

Our first task is written by a simple command that displays a text message.

```
void task1(void) {
    printf("Task 1: it seems that everything is working as normal\n");
}
```

This code allows us to print a message saying that it is working.

If the message is displayed, then the code is working and the RTOS is working.

Task 2

For our second task, I started by creating the code as follows :

```
void task2(void) {
    float fahrenheit = 98.6;
    float celsius = (fahrenheit - 32) * 5 / 9; // conversion (32 °F − 32) × 5/9
= 0 °C
    printf("Task 2: %.2f Fahrenheit = %.2f Celsius\n", fahrenheit, celsius);
}
```

Here the temperature is predefined and can only be changed by accessing the code.

I therefore wanted the RTOS execution to be a little more 'interactive' and so I made a new code:

```
// Task 2: Convert a fixed Fahrenheit temperature value to degree Celsius
void task2(void) {
    float fahrenheit;

    printf("Task 2 : Enter the temperature in Fahrenheit: ");
    scanf("%f", &fahrenheit);
    float celsius = (fahrenheit - 32) * 5 / 9; // conversion (32 °F − 32) × 5/9
= 0 °C
    printf(" %.2f Fahrenheit = %.2f Celsius\n", fahrenheit, celsius);
}
```

This allows the user to choose a temperature in Fahrenheit, and then convert from Fahrenheit to Celsius.

To do this, we define a float. We then print a message asking the user to choose a temperature value that we will assign to this float.

The temperature entered by the user is then converted into degrees Celsius using the conversion calculation. The task then displays the chosen temperature in degrees Fahrenheit and its equivalent in degrees Celsius.

So in the first case, the value in fahrenheit is fixed and in the second case it is chosen.

Task 3

As for task 2, for task 3 we can already predefine the two numbers to be multiplied with this code snippet:

```
// Task 3: Define any two long int big numbers and multiply them, print the
result
void task3(void) {
    WaitForSingleObject(hMutex, INFINITE);

    long int num1 = 1234567890;
    long int num2 = 987654321;
    long int result = num1 * num2;

    printf("Task 3: %ld * %ld = %ld\n", num1, num2, result);

    ReleaseMutex(hMutex);
}
```

But if we want the user to choose the 2 numbers to be multiplied, we can change the code as follows:

```
void task3(void) {
    WaitForSingleObject(hMutex, INFINITE);

    long long num1;
    long long num2;
    printf("Task 3  :  Enter a first long number: ");
    scanf("%lld", &num1);
    printf("   Now enter a second long number whih will mutliply the first one
: ");
    scanf("%lld", &num2);

    long long result = num1 * num2;

     printf("   Result : %.2lf * %.2lf = %lld\n", (double)num1, (double)num2,
result);

    ReleaseMutex(hMutex);
}
```

In this way we ask the user for a first and then a second number. The user enters them and we take care of associating them with the number.

Using long int, you can enter numbers ranging from

Long int: Signed long integer type, capable of representing at least the numbers [-2 147 483 647; +2 147 483 647].

long long: Long, signed integer type, capable of representing at least the numbers [-9 223 372 036 854 775 807; +9 223 372 036 854 775 807].

This second flaot is used because the result of two large numbers will be very large. The result must therefore be larger to contain all the digits of the multiplication.

Task 4

This code implements a binary search in a list of 50 items to find a specific item. Here's what it does:

```
void task4(void) {
    WaitForSingleObject(hMutex, INFINITE);

    int list[LIST_SIZE];
    for (int i = 0; i < LIST_SIZE; i++) {
        list[i] = i;
    }

    int first = 0, last = LIST_SIZE - 1, middle;

    while (first <= last) {
        middle = (first + last) / 2;
        if (list[middle] == SEARCH_ELEMENT) {
            printf("Task 4  :  Element %d found at index %d\n", SEARCH_ELEMENT,
middle);
```

```
            break;
        } else if (list[middle] < SEARCH_ELEMENT) {
            first = middle + 1;
        } else {
            last = middle - 1;
        }
    }

    ReleaseMutex(hMutex);
}
```

First we initialise a list of integers with 50 elements (ranging from 0 to 49).

We define two variables, **first** and **last**, which represent the extreme indices of the sub-list currently being examined.

To search for the specified element, we use a **while** loop to find the element in the list using binary search. In each iteration of the loop, the average index of the current sub-list is calculated.

If the element is found at the **middle** index, a message is displayed saying that the element has been found at this index. If it is, we stop the search and exit the loop.

If the element with the **middle** index is lower than the element being searched for, it updates **first** so that the search continues in the right-hand part of the sub-list (using indices higher than **middle**). On the other hand, if the element with the **middle** index is greater than the element being searched for, it updates last so that the search continues in the left-hand part of the sub-list (using indices lower than **middle**).

This loop continues until **first** exceeds **last**, which means that the entire list has been searched without finding the element. Once the search is complete, the mutex is released to indicate that the shared resource is once again available to other threads or processes

Scheduler

```
DWORD WINAPI scheduler(LPVOID lpParam) {
    while (1) {
        for (int i = 0; i < NUM_TASKS; i++) {
            if ((unsigned long)(GetTickCount() - tasks[i].last_run) >=
tasks[i].period) {
                tasks[i].task();
                tasks[i].last_run = GetTickCount();
            }
        }
        Sleep(1);
    }
    return 0;
}
```

We create a **scheduler** function, which is an infinite loop that runs through the tasks.

It checks whether a task should run according to its period and its last execution time.

If a task is due to run, it calls the associated function and updates the last execution time.

Main

```
int main() {
    HANDLE scheduler_thread;

    hMutex = CreateMutex(NULL, FALSE, NULL);
    if (hMutex == NULL) {
        printf("CreateMutex error: %d\n", GetLastError());
        return 1;
    }

    scheduler_thread = CreateThread(NULL, 0, scheduler, NULL, 0, NULL);
    if (scheduler_thread == NULL) {
        printf("CreateThread error: %d\n", GetLastError());
        return 1;
    }

    WaitForSingleObject(scheduler_thread, INFINITE);

    CloseHandle(hMutex);

    return 0;
}
```

The main function creates the mutex, then starts the scheduler thread. It then waits indefinitely for the scheduler thread to finish. Once the thread has finished, it releases the resources (the mutex) and terminates.

The program creates a thread for the scheduler, which manages the execution of tasks. Each task runs periodically according to its defined period. Tasks 2, 3 and 4 wait for the mutex to be acquired before accessing shared resources, and then release it.