Concurrent Programming in Windows - Practical Exercises

1. Introduction

Modern applications are mostly multi-threaded. A *browser* may consist of a *thread* that displays *images* or text and another *thread* that reads data from the network. There are also applications that perform several similar tasks. For example, a web server may have multiple clients accessing it simultaneously. If the web server is only made up of one *thread,* it will only be able to serve one client at a time. The total waiting time for a customer can therefore be very long. One solution would be to define a set of *threads,* associated with the same server program, to execute the different tasks.

This tutorial explores the use of *threads* in the Windows Operating System, more precisely in the Win32 subsystem, as well as, in general, the basic concepts of concurrent programming.

2. Threads Windows (Win32)

The Windows Operating System defines a process model that differs from the traditional model by separating the concepts of process and *thread.* A *thread* in Windows defines an independent thread of execution control within the same process. The Windows process object defines an address space in which multiple *threads* of control that share all resources run.

The simplest situation is that there is only one *thread* (known as the "parent" *thread*) executing in the address space of a process. This is the situation we encounter whenever we execute a program. However, the programmer can create more *threads* on his own initiative (known as "child" *threads*) within the same program. The Windows operating system offers the `CreateThread` function of the Win32 API for developing concurrent *(multi-threaded) programs.*

Since a *thread* represents a line of execution control in the context of a shared address space, the programmer will have to provide, at the time of *thread creation,* information related to the *thread* 's execution environment, such as the address of the function/procedure that the *thread* will execute.

The life cycle of a *thread* begins with the creation of the respective object *(thread),* continues with the execution of the function/procedure indicated when the *thread was created,* and will end when the execution of the same function/procedure ends.

Consider the following example.

Example 1

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

DWORD WINAPI print_message(LPVOID ptr)
{
	char *message;
	message = (char *) ptr;
	printf("%s", message);
	return 0;
}

int main(int argc, char* argv[])
{
	DWORD threadID1, threadID2;
	HANDLE threadH1, threadH2;

	char *message1 = "hello \n";
	char *message2 = "world";

	threadH1 = CreateThread(NULL, 0, print_message, message1, 0, &threadID1);
	threadH2 = CreateThread(NULL, 0, print_message, message2, 0, &threadID2);

	/* (a) */
	WaitForSingleObject(threadH1, INFINITE);

	/* (b) */
	WaitForSingleObject(threadH2, INFINITE);

	CloseHandle(threadH1);
	CloseHandle(threadH2);

	return 0;
}
```

This program creates a system consisting of three *threads* (including the “parent” *thread*) that print the message “hello world" in the *standard output,* with each substring of the message being printed by a *thread.* Each of the two “child” *threads* will execute the `print_message` function, each receiving, as a parameter, the substring to be printed. The program creates the first *thread* and passes it the string “hello" as an argument; the second *thread* is created with the argument “world".

Task 1

Consult the documentation to study the functions of the Win32 `CreateThread` API, `WaitForSingleObject` and `CloseHandle`.

Task 2

Compile and run the Example 1 program.

Task 3

Put the system to work with some variants and explain the results:

- Comment out (delete) line below (a) and (b);
- Comment out (delete) line below (a);
- Comment out (delete) line below (b).

3. Concurrent applications on Windows (Win32)

The scenario that serves as the basis for this practical exercise is that of a boxes deposit. A producer goes to the warehouse to store the boxes he is producing. A consumer goes to the warehouse to pick up those same boxes. The program presented below corresponds to the `producer` code, including the `capacity` variable that simulates the deposit capacity and the `items` variable that simulates the current quantity of boxes, and allowing the and placement of boxes through the `consumer` and `producer` functions respectively. The producer and `consumer functions` take an integer argument that represents the period of time between production and consumption respectively.

Program 1

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

int capacity = 5;
int items = 0;

DWORD WINAPI producer(LPVOID T)
{
    int *period;
    period = (int *) T;

    while (1) {
       if (items < capacity) {
           items ++;
           printf("\n Stored box: They became %d boxes", items);
           Sleep(*period);
       }
    }
    return 0;
}

DWORD WINAPI consumer(LPVOID T)
```

```
{
    int *period;
    period = (int *) T;

    while(1) {
       if (items > 0) {
           items --;
           printf("\n Box removed: %d boxes left", items);
           Sleep(*period);
       }
    }
    return 0;
}

int main(int argc, char* argv[])
{
    // start producer
    // start consumer
    // wait for the end of threads
    return 0;
}
```

Task 4

Create a thread that simulates the producer, that is, it simulates the placement of boxes in the warehouse. This thread should execute the `producer` function.

Task 5

Create a thread that simulates the consumer, which simulates the consumption of boxes from the warehouse. This thread must execute the `consumer` function.

Task 6

Get the system up and running and try some variations, such as:

- Add to the functions `consumer` and `producer` messages that allow identifying what each consumer/producer is doing at each moment, and in particular if it is blocked waiting for there to be boxes or for there to be space in the warehouse respectively.
- Change the number of consumers or producers and the time periods between productions and consumptions.

Task 7

Using appropriate consumption and production time periods try to force the occurrence of race-conditions.

The threads associated with a process execute in the context of the same address space and share the same resources. The operating system must provide synchronization mechanisms so that the various independent threads coordinate their execution. The synchronization problems we study are the mutual exclusion problem and the interthread coordination problem.

4.1 Mutual Exclusion

Task 8

Analyze the code of Program 1 and identify potential race-conditions problems.

The Windows (Win32) operating system offers, among others, the Mutex synchronization object. The Mutex object exists to solve the mutual exclusion problem. A Mutex object is created by the `CreateMutex` function, and once created, it is shared among all threads running in the context of the same process. The state of a Mutex can be nonsignaled or signaled, depending on whether it is in possession of a thread or not, respectively. A Mutex can be requested by a thread at the time of its creation (see arguments of the `CreateMutex` function) or if the thread invokes the `WaitForSingleObject` function on the respective Mutex (the result of invoking this function must be checked, if a timeout is used). To release the mutex, the thread invokes the `ReleaseMutex` function.

Task 9

Consult the bibliography to study the `CreateMutex`, `WaitForSingleObject` and `ReleaseMutex` functions.

Task 10

Using the Mutex synchronization object, change the implementation of the box deposit system (Program 1) to prevent race conditions from occurring.

4.2 Thread coordination

In addition to avoiding race-conditions , it is also important to allow coordination between threads. The Windows (Win32) operating system offers, among others, the Semaphore synchronization object. The Semaphore synchronization object can also be used for mutual exclusion problems.

A Semaphore object is created by the `CreateSemaphore` function, and once created, it is shared by all threads running in the context of the same process. The Semaphore object has associated with an initial integer value between 0 and `lMaximumCount` (see the `CreateSemaphore` function). The state of a Semaphore can be nonsignaled or signaled, if its value is 0 or between 1 and `lMaxinumCount` respectively. The initial value of the Semaphore object is directly manipulated by the `WaitForSingleObject` and `ReleaseSemaphore` functions. The `WaitForSingleObject` function decrements the value of the Semaphore object by one whenever it unblocks a thread. The `ReleaseSemaphore` function increments the value of the Semaphore object by `lReleaseCount` units (see `ReleaseSemaphore` function).

Task 11

Identify the reasons why using the Mutex synchronization mechanism may not be sufficient to ensure proper functioning of this system.

Task 12

Consult the bibliography to study the `CreateSemaphore`, `WaitForSingleObject` and `ReleaseSemaphore` functions.

Task 13

Using the methods above, change the system implementation to allow for proper coordination between threads.