Multithreading with Posix Threads

Laboratory number 4

SID-LAKHDAR Riyane

11/04/2015

1 A first program...

1.1 Question.1

In the main function, we create several threads in a same process to execute separate tasks. To be able to identify each thread, the system needs to label each thread with a unique identifier among the process threads. This unique identifier is called the TID: Thread ID.
The **tids** variable is an array that stores the identifier of each created thread.

In this program, the global variables (shared by all the threads) are allocated and freed by the main thread (the one which creates all the others). This global variables are also initialized by the main thread.

But each thread has its local variables which are not accessible to other threads as there are declared in them specific function of the thread. This variables are declared initialized and managed by this specific threads in the local environment of the specific function of each thread.

1.2 Question.2

A thread is the smallest sequence of programmed instructions executed independently. Thus to create a thread we first need to gather the sequence of instructions to be executed by the thread in a single function.

The creation of the thread and the association between the new thread and its specific funtion is done by the function **pthread create**. This function also run the thread and retrieves him its parameters.

In an other hand, a thread needs to be identified amoung the other threads of the same process. Thus the function **pthread create** will generate this id and returne it in the first parameter(ouput parameter).

1.3 Question.3

When a runing thread call the function **usleep** the schedular removes the CPU access from the thread. Thus the thread can not carry on his execution for a given time (input of the function). This thread state is called **blocked** or **waiting**.

1.4 Question.4

In the match program, each created thread (excluding the main thread) terminates once it has printed its text. Once it terminates, it enters in a dead **state**. At the end of the match program, the main thread waits for all its subthread to terminate. At the end of each subthread, the main thread removes it (function pthread_join). The main thread terminates when all its sub threads have been removed.
If the main thread did not remove all its created subthread, the process would have done it once the main thread ends. Indeed, when the main thread terminates, the hole process is removed by the system including the memory space allocated by the subthreads.

2 Parameter Passing

The expected program has been written in the file **match_parameters.c**. It can be:

- Compiled by running the command **make match_parameter**
- tested by running the command **./match_parameters team1 team2 nbr_song_team1 nbr_song_team2**

3 Return Value

3.1 Question .5.

The expected program has been written in the file **sumArray.c**. It can be:

- Compiled by running the command **make sumArray_sequential**
- tested by running the command **./sumArray_sequential**

3.2 Question .6.

The expected program has been written in the file **sumArray.c**. It can be:

- Compiled by running the command **make sumArray_multiThread**
- tested by running the command **./sumArray_multiThread**

3.3 Question .7.

Using the previous programs we have computed the execution time of our program on the same input array size. The results are :

- **Sequential:** 107 ms
- **1 thread:** 345 ms
- **2 thread:** 426 ms
- **4 thread:** 608 ms
- **6 thread:** 731 ms
- **8 thread:** 11251 ms

4 Variable Sharing between Threads

The expected program has been written in the file **sumArray_sharedVariable.c**. It can be:

- Compiled by running the command **make sumArray_sharedVariable**
- tested by running the command **./sumArray_sharedVariable**

This program allowed us to compute the following time results:

- **1 thread:** 391 ms
- **2 thread:** 468 ms
- **4 thread:** 554 ms
- **6 thread:** 895 ms
- **8 thread:** 1191 ms

This results are verry close to the time result computed without synchronization. This similarity may be explained by the fact that all the threads are executing the same task in the same time (roughly).
However, using this synchronization would be verry useful in the case where the threads would have verry diffrent time execution as the main thread would not loose time waiting for the first threads while other threads may have already returned them results.