FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA
DEPARTAMENTO DE INFORMÁTICA

Concurrent Programming with Pthreads
(Lab-04 – October 2019)

1. Introduction and Lab objective

A *thread* (or *light process*) represents an independent flow of control within the context of a single process that can be scheduled by the Operating System. All threads residing in a particular process share its context, e.g. the threads may communicate using the process' global variables, and they may use its resources as long as the process exists. However, each thread owns a replica of the minimum set of necessary resources that allows it to be independently scheduled by the Operating System. This set defines the thread's state and includes a private set of the CPU registers (PC/Program Counter, SP/Stack Pointer, general purpose registers, etc.), scheduling properties (such as policy or priority), thread specific data (e.g a thread's stack frame), etc.

POSIX Threads is a standard API defining a set of operations for thread management, namely thread creation and control (priority definition, thread suspension and deletion) and thread synchronization. Implementations of the standard are known as POSIX Threads or Pthreads.

The objective of this lab is the utilization of the Pthreads library for concurrent and parallel programming in the C language.

2. Using the Pthreads API

Use the Pthreads API's library calls for thread management (*pthread_create, pthread_join*) to program a parallel version of the sequential program below. Compare and discuss the execution results of both programs and to propose a solution to correct the problem.
The program generates a big array of integers with random values between 0 and 3; after this it will count the number of elements equal to 3 that exist in the array.

2.1. Sequential version

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

#define SIZE 256*1024*1024
int *array;
int length, count;
int count3s(){
      int i;
```

```
      count = 0;

      for(i=0; i < length; i++){
        if(array[i] == 3){
               count++;
        }
      }
      return count;
}

int main( int argc, char *argv[]){
      int i, r;

      array= (int *)malloc(SIZE*sizeof(int));
      length = SIZE;
      srand(0);
      for(i=0; i < length; i++){
        array[i] = rand() % 4;
      }
      r = count3s();
      printf("Count of 3s = %d\n", r);
      return 0;
}
```

The number sequence generated by the function *rand( )* is always the same because it uses the *default seed 0*. Execute the program above several times and verify that the result is always the same.

2.2. Parallel Version

The program is now modified to use the Pthreads API for counting 3s using multiple threads; each thread counts the number of array elements equal to 3 in a distinct part of the array (why ?).

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

#define SIZE 256*1024*1024
int *array;
int length, count;

pthread_t *ids;
int length_per_thread;

// TODO: The code executed by each thread

// TODO: The code with the creation of multiple threads
void count3s(int n){

      /* code to fill in */

}

int main( int argc, char *argv[]){
      int i;

      if ( argc != 2 ) {
```

```
        printf("Usage: %s <number of threads>\n", argv[0]); return 1;
    }
    int n = atoi(argv[1]);
    ids = (pthread_t *)malloc(n*sizeof(pthread_t));

    array= (int *)malloc(SIZE*sizeof(int));
    length = SIZE;
    srand(0);
    for(i=0; i < length; i++){
      array[i] = rand() % 4;
    }
    length_per_thread = SIZE / n;
    printf("Using %d threads; length_per_thread =  %d\n", n, length_per_thread);

    count3s(n);

    printf("Count of 3s = %d\n", count);

    return 0;
}
```

Try the previous program with only one thread and verify that the result is the same as the sequential version. When compiling in Linux add as link option the threads library, i.e. "gcc –o prog prog.c –lpthread". Execute now the program with more than one thread and verify that the result is wrong and it is different in each run. Explain why.

2.3. Correct Parallel Execution

a) Modify the code in the previous step to correct the problem using only the *pthread_create*, *pthread_join* functions.

b) Modify the code in the previous step to correct the problem using *mutexes*.

3. Find_and_fill Operation

The goal of this exercise is to implement a *find_and_fill()* operation that finds the maximum value of a vector of integers and replaces all positions in the vector with that value. For instance, considering a vector with the following elements

```
       V= [25,11,5,7,1,0]
the vector's final result is
       V= [25,25,25,25,25,25]
```

3.1. Sequential version

The outline of the operation is:

```
void find_and_fill( int vec[], int length){

    int max;
    // find/calculate the maximum value
    for(int i=0; i < length; i++){
        max = ...
    }
    //
    // fill the vector with the maximum value
```

```
      for(int i=0; i < length; i++)
           vec[i] = max;
}
```

3.2. Parallel version

Assuming that the vector may be very large, your program has to implement a parallel version of the *find_and_fill()* operation using the Pthreads library to optimize its execution. For this,

- **a)** your program creates a set of threads to divide the work so that each thread processes part of the vector to calculate its local maximum;
- **b)** all threads communicate to calculate the global maximum (i.e. the major element in the whole vector);
- **c)** all threads have to synchronize among them using a *barrier* to guarantee that the vector is totally evaluated;
- **d)** each thread fills its part of the vector with the global maximum value;
- **e)** Your program waits that all threads finish executing and terminates.

The outline/pseudo-code of the parallel operation is:

```
void find_and_fill_parallel( int vec[], int length, int nworkers){
      int max;

      // 1- launch the nworkers
      // Each worker calculates the local maximum value of the vector
      int first_element = …;
      int local_length = …;
       int local_max;
      for(int i = first_element; i < (first_element+local_length); i++){
           local_max = ...

      }

      // 2- each thread contributes to calculate the global maximum
      if( local_max > global_max ) global_max = local_max;

      // 3- each thread blocks in a barrier until all threads arrive at this point in their
      // execution to guarantee that the whole vector has been evaluated
      barrier();

      // 4- each thread fills its part of the vector with the maximum value
      for(int i = first_element; i < (first_element+local_length); i++){
           vec[i] = max;

      }
      // 5- the main thread waits for all threads to terminate, may print (part of) the
      // vector's value for debug, and terminates.
}
```

3.3. Barrier Operation

`Your code has to implement the barrier() operation using two methods`
a) semaphores;

b) condition variables.

4. Reading List

- Part II Concurrency of the FSO recommended book http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf
- POSIX Threads programming tutorial: https://computing.llnl.gov/tutorials/pthreads/
- On-line manual pages for the pthreads library