Fundamentos de los Sistemas Operativos

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

fSO

Lab session 5

Thread creation and performace evaluation

Content

1. Objetives 2
2. Threads creation 2
2.1 Exercise 1: working with pthread_join and pthread_exit 4
3. Sequentiality vs. concurrency 4
3.1 Exersice 2: adding the rows sequentially "SequentialAdd.c" 7
3.2 Exersice 3: adding rows concurrently "ThreadsAdd.c" 7
3.3 Exersice 4: comparing execution times with command “time” 8
3.4 Exersice 5: optimizing for the number of available cores 8
4. Working with periodic threads 9
4.1 Exercise 6: Animation with threads. 9
4.2 Exercise 7: Threads that create other threads. 10
4.3 Exercise 8. All threads are created at the same brotherhood level. 10
4.4. Exercise 9. Completing the animation. 10
Annex. Comments about matrix_basic.c 12

1. Objetives

The main aim of this lab session is to **acquire experience using the functions of the POSIX standard for thread creation and waiting**. Working with a scenario with concurrent operations, in particular we will see an example with concurrent operations on an array of data to assess the improvement which, in terms of execution time, is obtained by the use of threads in a multicore processor.

2. Threads creation

The code in Figure-1 is the basic skeleton of an operation implemented with threads.

```
/**
* Sample program "Hello World" with pthreads.
* To compile type:
*   gcc hello.c -lpthread -o Hello
*/
#include <stdio.h>
#include <pthread.h>
#include <string.h>

void * Print (void * ptr) {
  char * men;
  men =(char*) ptr;
  // EXERCISE 1.b
  write (1, men, strlen (men));
}

int main() {

  pthread_attr_t attrib;
  pthread_t thread1, thread2;

  pthread_attr_init (& attrib);

  pthread_create (& thread1, & attrib, Print, "Hello");
  pthread_create (& thread2, & attrib, Print, "World \n");

  // EXERCISE 1.a
  pthread_join (thread1, NULL);
  pthread_join (thread2, NULL);

}
```

Figure-1: Basic skeleton of a thread based program.

Create a file "hello.c" that contains this code, compile it and run it from the command line.

```
$ gcc hello.c -lpthread -o hello
$ ./hello
```

As shown in Figure-1 code, the novelties introduced by the management of threads go hand in hand with the necessary functions to initialize ant to finish them properly. We have only made use of the more basic or essential ones.

- Types **`pthread_t`** and **`pthread_attr_t`** from the header file `pthread.h`.

```
#include <pthread.h >
pthread_t  th;
 pthread_attr_t attr;
```

- **`pthread_attr_init`** is responsible for assigning default values to the elements of the thread attributes structure. WARNING! If the attributes are not initialized, the thread cannot be created.

```
#include <pthread.h >

 int pthread_attr_init(pthread_attr_t *attr)
```

- **pthread_create** creates a thread.

```
#include <pthread.h >

 int pthread_create(pthread_t  *thread,const pthread_attr_t *attr,
                    void *(*start_routine)(void *), void *arg);
```

Pthread_create parameters:

thread: It is the first parameter of this function, *thread,* will contain the ID of the thread
attr: The argument *attr* specifies attributes for the thread. Can take the NULL value, in which case indicates values by default: "*the created thread is joinable (not detached) and have default (non-real - time) scheduling policy*'.
start_routine: the behavior of the thread to be created is defined by the function that is passed as the third parameter *start_routine* and it receive as an argument the pointer *arg*.

Pthread_create () function return value:
Returns 0 if the function runs successfully. In case of error, the function returns a nonzero value.

- ***pthread_join*** suspends the thread that calls to it until the thread specified as a parameter ends. This behavior is necessary because when the main thread "ends" destroys the process and, therefore, requires the abrupt completion of all threads that have been created.

```
#include <pthread.h >

 int pthread_join(pthread_t thread, void **exit_status,);
```

Pthread_join parameters:

thread: parameter that identifies the thread to wait for.
exit_status: contains the value that the finished thread communicates to the thread that invokes pthread_join (notice that is a pointer to pointer, because the parameter passed by reference is a pointer to void).

- ***pthread_exit*** allows a thread to end its execution. The last ending thread in a process sets the process to end. Parameter `exit_status` allows communicating a termination value to another thread waiting for its end, through pthread_join().

```
#include <pthread.h >

 int pthread_exit(void *exit_status);
```

2.1 Exercise 1: working with pthread_join and pthread_exit

Check the behavior of call pthread_join () making the following changes in program "hello.c" shown above.

Questions Exercise 1:

Remove (or comment) pthread_join calls on the main thread.

- What happens? Why does it happen?

Replace pthread_join calls by a single pthread_exit(0) call, close to the program point marked as // Exercise 1.a

- Does the program complete it execution correctly? Why?

Remove (or comment) all pthread_join or pthread_exit calls (close to comment // Exercise 1.a) and put in that point a 1 second delay (using usleep(…))

```
#include <unistd.h>
  void usleep(unsigned long usec);  // usec in microseconds
```

- What happens with the proposed changes?

Now put a 2 seconds delay close to comment `// Exercise 1.b`

- What happens now? Why?

3. Sequentiality vs. concurrency

We are going to check the difference between sequential execution of a set of actions and parallel execution of the same actions using threads, both on a multicore processor.
Let us consider a set of NUMROWS vectors of dimension DIMROW, so DIMROW is much greater than NUMROWS. The program will do AddRow() operation on each vector or row. This operation has a high computational cost since DIMROW is very big. In the example, the AddRow() operation will add the value of a function applied to all the elements in a vector and it will store the result in the field `addition` of the corresponding row. The main

program will do NUMROWS calls to function AddRow() function in a sequential way. The data structures and the calling scheme that does the main thread are shown in Figure 2.

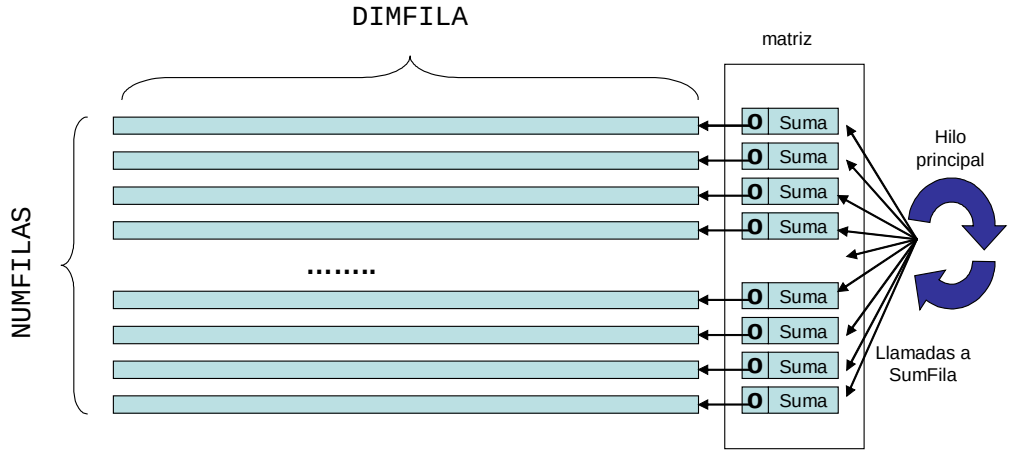

Figura2. NUMROWS vectors of DIMROW size

The code for this example is shown in Figure-3, where it appears the definition of "`struct row`" and the Declaration of the global variable `matrix`. In the main program, after the loop of sequential calls to AddRow(), all the partial results are added and the total addition is shown on the screen, that must be in this case be DIMROW * NUMROWS since all vector elements are 1.

```c
// Program SequentialAdd.c
// To compile do:
//     gcc SequitialAdd.c  -o SeqAdd -lm

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

#define DIMROW 1000000
#define NUMROWS 20
typedef struct row {
        int vector[DIMROW];
        long addition;
} row;
struct row matrix[NUMROWS];

void *AddRow( void *ptr ) {
    int k;
    row *fi;
    fi = (row *)ptr;

    fi->addition=0;
    for(k=0;k<DIMROW;k++) {
        fi->addition += exp((k*(fi->vector[k])+
          (k+1)*(fi->vector[k]))/(fi->vector[k]+2*k))/2;
    }
 }

int main() {
    int i,j;
    long total_addition=0;
    pthread_t  threads[NUMROWS];
    pthread_attr_t atrib;

    // Vector elements are initialized to 1
    for(i=0;i<NUMROWS;i++) {
        for(j=0;j<DIMROW;j++) {
            matrix[i].vector[j]=1;
        }
    }
    // Thread attributes initialization
    pthread_attr_init( &atrib );

    // EXERCISE 2.a
    for(i=0;i<NUMROWS;i++) {
        AddRow(&matrix[i]);
    }
    // EXERCISE 2.b

    for(i=0;i<NUMROWS;i++) {
        total_addition += matrix[i].addition;
    }
    printf("Total addition is: %ld \n", total_addition);

}
```

Figure-3: SequentialAdd.*c* program. In ordet to increase the execution time for each iteration the following expression is applied to every element before adding:

“exp((k*(fi->vector[k])+(k+1)*(fi->vector[k]))/(fi->vector[k]+2*k))/2”

3.1 Exersice 2: adding the rows sequentially "SequentialAdd.c"

Compile SequentialAdd.c and run it, verify that the result is correct

```
$ gcc SequentialAdd.c -o SeqAdd -lm
$ ./SeqAdd
 Total addition is: 20000000
```

3.2 Exersice 3: adding rows concurrently "ThreadsAdd.c"

Modify the proposed code to make concurrent calls to the AddRowa() function. Figure 4 shows a diagram of the resulting distribution of threads.

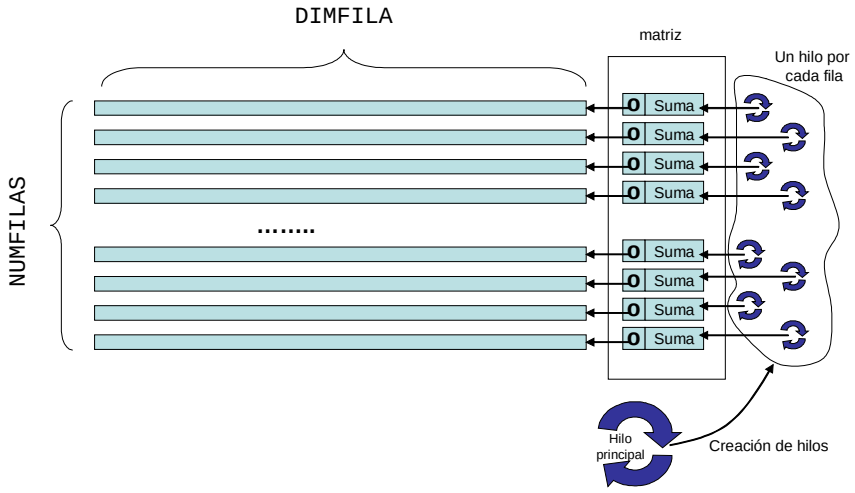

Figura 4: Threading scheme to be implement in ThreadAdd.c

For this exercise, follow these steps:

- Copy SequentialAdd.c to ThreadsAdd.c and do the required changes on ThreadsAdd.c
- Change the lines of code that appear among the comments `//EXERCISE2.a` and `//EXERCISE2.b` in shuch a way that:
  - There should be a loop of `pthread_create` calls to create the concurrent activities.
  - As the total addition cannot be computed until all threads finish, insert a loop of `pthread_join` calls
- Compile and run the code. Verify that the result of the "total addition" is the same as in the sequential case.

```
$ gcc ThreadsAdd.c -o ThAdd -lm -lpthread
$ ./ThAdd
Total addition is: 20000000
```

3.3 Exersice 4: comparing execution times with command "time"

To check the execution times of the two versions of the program (SeqAdd and ThAdd) run the *shell* command `time`. Command *time* runs the command that is passed to it as a parameter and, after the end of the running command, *time* reports on the **actual execution time**, the **time** that the process has been **executing instructions in "user mode"** and the **time** that it has been in **"system mode"**. For more information see the manual pages for `time`.

```
$ man time
$ time ./SeqAdd
$ time  ./ThAdd
```

- Check the execution times of SeqAdd and ThAdd and fill the table

Adding rows	SeqAdd	ThAdd
Real execution time		
User mode time		
System mode time		

Exercise 4 question:

- Note the differences and similarities of the results between sequential and concurrent execution. Try to justify the observed behavior.

3.4 Exersice 5: optimizing for the number of available cores

We intend now to optimize the concurrent code in the example so that the overhead associated with the creation and termination of threads be minimized, in order to take full advantage of processor cores available to achieve maximum execution speed. So, the number of threads at every moment must match the number of cores in the processor, and also the load associated with each thread should be approximately the same.

In order to know the number of cores in the processor you can execute the command *top* and press key 1 to see every core workload. You can also look at file /proc/cpuinfo doing any of the following commands:

```
$ grep processor /proc/cpuinfo | wc -l
$ cat /proc/cpuinfo | grep "cpu cores"
```

To do the proposed exercise follow these steps:

- Copy `ThreadAdd.c` into `ThreadAdds2.c`, do the changes on `ThreadAdd2.c`
- Change the lines of code that appear among the comments `//EXERCISE 2.a` and `//EXERCISE 2.b` in such a way that:
  - There must be a loop of `pthread_create` calls to create as many threads as cores the processor has.
  - Each created thread has to process a proportional part of `AddRows` calls. You should consider that the total number of calls to `SumaFila` may not be an exact multiple of the number of cores, so some threads may have to do one more call rather than others.
  - `pthread_join` calling loop must have as many iterations as cores the processor has.

- Compile and run the code. Check that the result of the "total addition" is the same as in the sequential case (SeqAdd).
- Check the execution times of ThAdd and ThAdd2 and fill in the following table:

Adding rows with maximum performance	ThAdd	ThAdd2
Real execution time		
User mode time		
System mode time		

Exercise 5 question:

- Look at the differences and similarities of the results for this optimized version of concurrently adding rows and the one that uses one thread per row.

4. Working with periodic threads

Many times threads are used to do tasks that have to be repeated periodically, i.e. every time interval. A simple way to do it is performing an sleep() call inside a loop. Following this idea, try to complete a program that performs an animation that remember the well-known "digital rain" effect where a series of random characters appear regularly as descending columns.

You can download from PoliformaT the starting code "matrix_basic.c" that is shown in Figure 5. It is based on creating a **drawing thread** for every column in the display. Every drawing thread will complete the column that is assigned to it, writing its characters taken from a shared two-dimensional array named “m”. At the same time, another (single) **refresh thread** will write periodically the full content of “m” in the screen to create the animation.

4.1 Exercise 6: Animation with threads. Write a program named “matrix_draw.c” starting from program “matrix_basic.c”, adding in its “main” function the required code to create a drawing thread for every display column, relying on the constant COLUMNS and the global variables already defined. Use the “DrawCol” function as the drawing threads body. This function will receive as an argument (BY VALUE) the column index assigned the thread, from 0 to COLUMNS-1. You have to create also a screen refresh thread using the “Refresh” function. Be sure that the main thread waits for all the drawing threads created, so the program will end when all the columns are drawn (you will notice that some columns are not drawn, this happens on purpose).

Exercise 6 question:

- The main thread must wait for the *Refresh* thread?

4.2 Exercise 7: Threads that create other threads. Save the program "matrix_draw.c" as "matrix_erase.c" as the working code file for this exercise. Analyse the "DrawCol" function code and extended in such a way that when every drawing thread reaches half of its rows loop it will create an erase thread. The drawing thread will pass to its erase thread its column index. The erase thread will be created using the function "EraseCol" that removes the characters from the column from the top, while the drawing thread is completing the column. Every drawing thread must wait to its erase thread in the proper moment. In this way the program execution will end when the drawing and erasing of all column is completed, so the display end up empty.

Exercise 7 question:

- Is it required that the main function waits for the completion of all the erase threads of achieve the demanded behavior? Why?

4.3 Exercise 8. All threads are created at the same brotherhood level. Open a new terminal while the program from exercise 7 is running. Verify with command "`ps -laT`" that all created threads are brothers, there is no parent-child relationship between them. Independently of which thread is the creator one.

4.4. Exercise 9. Completing the animation. Save "matrix_erase.c" as "matrix_complete.c" as the working file for this exercise. Modify the "DrawCol" function in such a way that the sequence Draw-Erase will be repeated forever in order to achieve the famous "digital rain" from the Matrix movie.

Exercise 9 question:

- The drawing threads do not end now. So, remove from the main function the waiting code for those threads. What code do you need now to avoid the animation abrupt ending?

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

#define COLUMNS 80
#define ROWS 25

char m[ROWS][COLUMNS];
long delay[COLUMNS];
int row_b[COLUMNS];

pthread_attr_t attrib;
pthread_t  draw_thread[COLUMNS];
pthread_t  erase_thread[COLUMNS];
pthread_t  refresh_thread;


void *EraseCol(void *ptr) {
  int row, col=(int)(long)ptr;

  for (row=0; row<ROWS; row++) {
    m[row][col]= ' '; // Write space
    usleep(delay[col]); // Wait before the following erase
  }
}

void *DrawCol(void *ptr) {
  int row, col=(int)(long)ptr;

  delay[col]= 50000+rand()%450000; // Random delay: 0,05s to 0,5s
  if (rand()%10 > 4) { // Sometimes do not draw column
    usleep(delay[col]*ROWS); // Wait without drawing
  } else {
    for (row=0; row<ROWS; row++) {
      row_b[col] = row;
      m[row][col] = 32+rand() % 94; // Write random char
      usleep(delay[col]); // Wait before next char
    }
  }
}

void *Refresh (void *ptr) {
  int row, col;
  char order[20];

  while(1) {
    write(1,"\033[1;1f\033[1;40;32m",16); // Back to left-up corner, Green text
    for (row=0; row<ROWS; row++) {
      write(1,m[row],COLUMNS); write(1,"\n",1); // Refresh row
    }
    write(1,"\033[1;37m",7); // White text
    for (col=0; col<COLUMNS; col++) {
      sprintf(order,"\033[%d;%df%c",row_b[col]+1,col+1,m[row_b[col]][col]);
      // Rewrite in white the last character in column col
      if (row_b[col]<ROWS-1) write(1,order,strlen(order));
    }
    usleep(100000); // Wait 0,1s before refreshing again
  }
}
```

```
int main()
{
  int col;
  memset (m,' ', ROWS*COLUMNS); // Erase matrix m
  write(1,"\033[2J\033[?25l",10); // Clean screen and hide cursor

  pthread_attr_init(&attrib);

  // Create a drawing thread for every column

  // Create a screen refresh thread

  // Wait for drawing threads ending

  write(1,"\033[0m\033[?25h\r",11); // Reset usual text and cursor
}
```

Figure 5: “matrix_basic.c” code

Annex. Comments about matrix_basic.c

Help about creating threads DrawCol y EraseCol

Thread functions DrawCol y EraseCol expect to receive the working column index by value. This requires to do a type cast of the column loop variable `col` in the last parameter of pthread_create() call, that has to be include on function main:

```
(void*)(long)col
```

This double cast convert the value of variable “col” to the pointer type tha the function requires. The intermediate conversion `long` avoids the generation of a warning by the compiler about the size mismatch between the initial and the final types.

About scape sequences on the terminal

The program uses the ANSI X3.64 standard scape codes to clearing the screen, moving the cursor and changing text color.