Universitat Pompeu Fabra Barcelona

Seminar 4 - Threads

The main objective of this seminar is to provide a basic understanding of threads within an operating system, as well as all the tools available for their management, such as locks, monitors, semaphores, or barriers. In theory, you have covered concepts such as multithreading, race conditions, deadlocks, starvation, and the producer/consumer analogy. These concepts will be needed to understand the underlying behavior of these seminar exercises.

The following exercises need to be solved in C using the functions explained in class. For their details check in aula global. Remember that in this course we are not using the `pthread` implementation of barriers and semaphores, since they are not fully implemented in OSx. You can find the sources and headers of our own implementation in the aula global (in the “Barriers / semaphores” folder). Remember to compile also the source to generate the executable.

The list of exercises has two parts, the first to be done during the seminar class and the second for the students to deepen their understanding working at home.

Seminar exercises

1. Create `100` threads such that all wait 1 second (using the function `sleep` provided in unistd.h) and the main thread waits for them to finish at the end. Remember that you can use `pthread_join` to wait for them.
2. Set a global array `A[100]` to the square of its index, meaning `A[i] = i*i`, using a thread per cell.
    a. To do so, pass the address of the index to the thread function. What’s the result? What has happened?
    b. Now, pass a pointer to a copy of the index value. To do so, you can allocate the required space using `malloc`, and then copy the value of the index into the pointer. Remember to free the pointer. What’s the result? What has happened?
3. There is an array (A[10]) in which each position has a random value from 0 to 2. The values represent the number of particles that are in a specific moment in a specific position (cell). Implement a threaded “particle” move operation: `void * move(void arg)`, which arg is a struct containing to integers `i` and `j`, so that if in cell `i` there are more than 0 particles, the function moves one of the particles in `i` to the cell with index `j`.
    a. Create the number of threads required to move simultaneously a particle from each even cell to the next greater odd cell, meaning from cell 0 one particle moves to cell 1, from cell 2 one particle moves from cell 3, etc. Do you need any synchronization mechanism?
    b. Now assume 10 particles move, but decide randomly from which cell to which other cell they move. Hence no specific order needs to be preserved. Do you need any synchronization mechanism?

c. Now assume that a gust of wind blows from the cell with the lowest index, moving particles from all cells to the next higher index cell. From the ninth cell they can't move. Do you need any synchronization mechanism? How would you do it?

4. Assume you have to develop a simple resource management system to limit the amount of services querying into a database, which for this exercise will be 5. Hence, create 20 threads that will aim to access the database, but they will need to wait if there are already 5 threads sending a query to the database. To simulate the access to the database, you can use the following function:

```
#define NUM_RESOURCES 5
int nThreadsAccessing = 0;

int getFromDatabase() {
   nThreadsAccessing++;
   printf("Number of Threads accessing the DB = %d\n",
nThreadsAccessing);
   if (nThreadsAccessing > NUM_RESOURCES) _exit(1);
   usleep(rand()%10000);
   nThreadsAccessing--;
   return rand();
}
```

Use a semaphore to provide the proper control.

Extra exercises

5. [Challenging exercise] Assume exercise 4, but now consider that threads can have 3 rankings (1, 2, and 3), the thread ranking is specified at the beginning of the thread function. The value of the thread ranking (which is determined randomly) specifies the number of resources that are taken from the database, meaning that if a thread has a ranking 3, it uses the database as if there were 3 threads of ranking 1. Implement a resource management system with a mutex and a condition variable (monitors) to control the usage of the database. There is no need for any type of optimization pattern of the resource usage, a first-come first-served strategy is enough.
   To do so, create a `request_resources` function that prints the thread id, its rank and the resources available, specifying if resources were acquired or not. Similarly, create a `relase_resources` function that specifies the thread id, its rank, and the number of resources released, as well as the number of resources available.

6. Write a 5x5 matrix into a file, each cell containing a random integer from 0 to 9. Store it in the file as binary elements in a row major order, remember that this means that the first 5 elements are the first row, the next five elements are the 2nd row, and so on. Hence for access a random element (i, j) the needed index is NumCols*i + j.

To do so, implement the matrix writing with threads. Each thread will be in charge of a specific row, so create a random vector of 5 elements, print in the screen the row and the thread responsible for writing it. Consider that the thread index is used to determine the row of the matrix.

Notice that you will not need to synchronize the threads so that they write in order, but no more than 1 thread can access the file for writing simultaneously, so use a `lock`. Finally, consider adding a small delay at the beginning of the function to avoid sequential access (e.g. `usleep(1000*10);`).

Remember to use `lseek` to determine the position in which the row must be written.

Use a barrier to make them wait, once they are all finished, each thread will have to read the element of the diagonal of the stored matrix that corresponds to its index. Finally, the main program function prints the diagonal on the screen.

Discuss the synchronization elements required for each part of the exercise.

Extra challenge: Open the file for the matrix in the main, using global variables or passed in the arguments of the thread function.