Operating Systems [2023-2024]

Assignment 05 – Threads and synchronization I

Introduction

Threads are a lightweight version of processes and many operating systems are able to manage processes that have more than one thread. Threads share with the process (that they belong to) the address space, the global variables, file descriptors, child processes, the signs, the handlers of signals and accounting information. On the other hand, there is information that is private to each thread, such as the program counter, the records of execution, state information and stack.

Operations on threads such as creation, destruction and switching threads are faster than the same operations on processes because threads share the resources from the parent process what does not require "hard" replication of addressing space or extensive context switches.

Objectives

Students concluding this work successfully should be able to:

- Create, destroy and execute multiple threads.
- Use POSIX semaphores and mutexes to synchronize threads

Support Material

- K. A. Robbins, S. Robbins, “Unix Systems Programming: Communication, Concurrency, and Threads”, Prentice Hall:
    - Chapter 12 – POSIX Threads
    - Chapter 13 – Thread Synchronization
    - Chapter 14 – Critical Sections and Semaphores
- “Programming in C and Unix”:
    - Threads: Basic Theory and Libraries
    - Further Threads Programming: Synchronization

Exercises

Note: Only some of the exercises provided in this assignment will be done during the practical classes. The extra exercises should be done by the student as homework and any questions about them should be clarified with the teacher.

Note 2:

- Compile a file named '`prog.c`' with

```
     'gcc prog.c -pthread -Wall -o prog'
OR   'gcc prog.c -lpthread -D_REENTRANT -Wall -o prog'
```

- Use semaphores that can be used with threads!

1. Process tasks

A process creates N threads to process N tasks. When a thread starts executing, it first checks the number of the next available task. Then, to acknowledge that it will execute that specific task, it saves its worker id in an array, at the position corresponding to the number of the task. After, the thread executes the task and leaves. In the end, the process prints a list with each task and the thread that processed it.

Read the code (file *process_tasks.c*) supplied with this assignment and modify it to solve the existing problems. Run the code more than one time to compare the results.

a) Verify that, most of the times, the process terminates before the threads finish processing the tasks. Correct the program by waiting for all threads to finish their tasks.
b) In the end, when checking the list of tasks, many tasks appear not to have been acknowledged! What happened? To solve the problem, protect the critical region of the process with a POSIX mutex.

Note: In this exercise the function *sched_yield()* is used to force the change of thread. See more on this command using *man*.

2. Store

A store receives a maximum of N_REQUESTS requests from its clients. Until the maximum number of requests is met, the clients keep sending new orders. When the orders limit is reached, the store manager calculates the volume of sales made. To simulate the scenario, each client will be represented by a thread. The requests will be saved in a common buffer.

Read the code (file *store.c*) supplied with this assignment and complete the missing sections according to the following requirements:

- The number of clients is equal to N_CLIENTS and each is simulated by a thread;
- Each client is always generating new requests (at intervals of 10 µs), which are saved in a common request buffer;
- The buffer of requests can hold N_REQUESTS;
- When there are no more free positions in the buffer, the client threads die, cleaning all used resources;

- Each client requests a random number of products (1-100);
- Each request holds information about the quantity of products and the identification of the client that requested them;
- Due to technical reasons the maximum number of threads that can write in the buffer at the same time is limited to MAX_THREADS;
- After all client threads die, the main thread prints in the screen the total number of products sold.

The executable of the program is given for reference (both 32 and 64 bits versions are provided).

3. Stock market v2

Adapt the Assignment 04-exercise 2 – "Stock market" to support threads instead of processes. Analyse the results of the *ps* command while running the two programs. The executable of the Assignment 04-exercise 2 is given for comparison.

4. Matrix multiplication

Complete the file `matrix.c` that implements a multi-threaded multiplication of matrices, as A[M][K]xB[K][N]=C[M][N]. There are MxN+2 threads, the main thread, a "printing" thread and MxN threads responsible for the calculation of each element of matrix C.

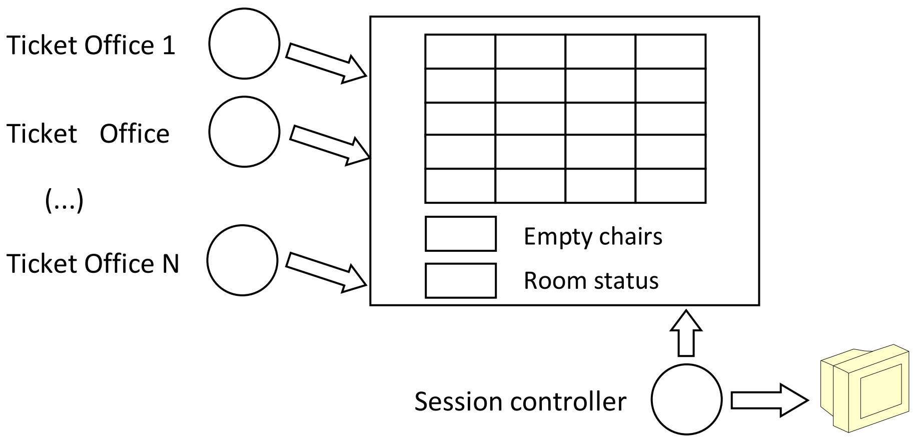
Some notes:

- void *worker(void* coord) – receives a pointer with the coordinates of an element in matrix C, sleeps between 1 to 10 seconds and then calculates it by multiplying the correct row from matrix A and the correct column from matrix B;
- void *show_matrix() – executed by the "printing" thread, it prints the status of matrix C in each second, along a period of 10 seconds; uses the function print_matrix() to do the actual printing;
- void print_matrix() - prints matrices in screen while having exclusive access to matrix C; use a mutex to control the access;
- In main() create the threads and send each the coordinate to calculate. Before leaving wait for all threads and then print the final matrix C.

The executable of the program is given for reference.

5. Ticket office

Create a ticket office simulator that allows the reservation of places in a cinema using Threads. The user should define the number of rows, chairs per rows and ticket offices.



To simulate the reservation of places in the cinema each ticket office will be represented by a thread. After registering a reservation, each ticket office should wait for a random time before trying to perform a new reservation (range between 1 and 2 seconds). The choice of which seat to reserve must also be random. Seats are identified by the pair row/seat number. If the selected seat is available, it is marked as reserved, if not, a new seat must be randomly selected, and its availability checked before a free seat is found. If after 3 attempts the thread is unable to find an available seat, it rests for 1 second before starting the search again.

Each 10 seconds a *session controller* thread prints on the screen the seats of the room, differentiating the occupied ones from the unoccupied, and after 60 seconds closes the reservations to start the film.

6. Horse track

Implement a very simple horse track stakes house simulator. The application will consist of two processes: BOOKIE and CLIENT. The second process is multi-threaded, and there are two types of threads: the main and bettors. These two processes (threads included) communicate through shared memory.

The program should ask the user for the number of horses per race, the bet value, number of bettors, number of races to be held, the maximum bet value and the bookie initial cash.

The BOOKIE is responsible for performing the races (e.g. choose a random winner among all horses). For each race the BOOKIE notifies the threads BETTOR (CLIENT process) that can start making bets. After all bets are done, BOOKIE closes the bets and starts the race. After the race, when the winning horse is selected, the BOOKIE checks for winning bets and updates the cash of the BOOKIE and bettors. After performing all the races, the BOOKIE ends, and presents in the screen the results of all races and bets, each bettor’s winnings and losses, and the house (BOOKIE) credit, winnings and losses.

The threads represent BETTORs. Bettors are required to bet on every race simulation a random value between 0 and the maximum bet value. The stakes are just for the winning horse (six times the bet value), second (four times the bet value) and third (two times the bet value). After betting on all races, BETTOR threads should end.