Operating Systems Design 2023-24 Statement of the 1st part of the LEIC-A/LEIC-T/LETI project

The aim of this project is to develop the IST "Event Management System" (IST-EMS), an event management system that allows you to create, book and check the availability of tickets for events such as concerts and theater shows.

IST-EMS explores parallelization techniques based on multiple processes and multiple tasks in order to speed up request processing. By developing IST-EMS students will also learn how to implement scalable synchronization mechanisms between tasks as well as communication mechanisms between processes (FIFOs and signals). IST-EMS will also interact with the file system, thus offering the possibility of learning how to use the POSIX file system programming interfaces.

Base code

The base code provided offers a sequential implementation that accepts the following commands:

1. **CREATE <event_id> <num_rows> <num_columns>**
    - This command is used to create a new event with a room where '*event_id*' is a unique identifier for the event, '*num_rows*' the number of rows and '*num_columns*' the number of columns in the room.
    - This event is represented by a matrix in which each position encodes the state of the place:
        - **0 indicates an empty seat;**
        - **res_id > 0 indicates a reserved seat with the reservation identifier res_id**.
    - Usage syntax: **CREATE 1 10 20**
        - Create an event with identifier *1* with a room of *10* rows and *20* columns.
2. **RESERVE <event_id> [(<x1>,<y1>) (<x2>,<y2>) ...]**
    - Allows you to reserve one or more seats in an existing event room. '*event_id*' identifies the event and each pair of coordinates *(x,y)* specifies a seat to reserve.
    - Each reservation is identified by a strictly positive integer identifier **(res_id > 0)**.
    - Usage syntax: **RESERVE 1 [(1,1) (1,2) (1,3)]**
        - Reserve seats *(1,1), (1,2), (1,3)* at event *1.*
3. **SHOW <event_id>**

- Prints the current status of all the places in an event. Available seats are marked with '*0*' and reserved seats are marked with the identifier of the reservation that booked them.
- Usage syntax: **SHOW 1**
  - Displays the current status of the seats for event *1.*

4. **LIST**
   - This command lists all the events created by your identifier.
   - Usage syntax: **LIST**
5. **WAIT <delay_ms> [thread_id]**
   - It introduces a delay in the execution of commands, which is useful for testing the system's behavior under load conditions.
   - The [thread_id] parameter is only introduced in exercise 3, and until then, it must add a delay to the only existing task.
   - Usage syntax: **WAIT 2000**
     - Adds a delay of the next command by 2000 milliseconds (2 seconds).
6. **BARRIER**
   - Only applicable from exercise 3 onwards, but the parsing of the command already exists in the base code.
7. **HELP**
   - It provides information on the commands available and how to use them.

Comments on Input:
Lines beginning with the **'#'** character are considered comments and are ignored by the command processor (useful for testing).
- Example: '*# This is a comment and will be ignored'.*

1st part of the project

The first part of the project consists of 3 exercises.

Exercise 1: Interaction with the file system

The base code only receives requests via the terminal (*std-input*). In this exercise we want to change the base code so that it can process batch requests from files.

To do this, IST-EMS must now receive as an argument on the command line the path to a "*JOBS*" directory, where the command files are stored.

IST-EMS must obtain the list of files with a ".jobs" extension contained in the "JOB" directory. These files contain sequences of commands that respect the same syntax accepted by the base code.

IST-EMS processes all the commands in each of the ".jobs" files, creating a corresponding output file with the same name and ".out" extension that reports the status of each event.

File access and manipulation should be done through the POSIX interface based on file descriptors, and not using the *stdio.h* library and the *FILE stream* abstraction.

Example output from the test file */jobs/test.jobs:*

```
1 0 2
0 1 0
0 0 0
```

Exercise 2. Parallelization using multiple processes

After completing Exercise 1, students should extend the code they have created so that each ".job" file is processed by a child process in parallel.

The program must ensure that the maximum number of child processes active in parallel is limited by a constant, **MAX_PROC**, which must be passed in by command line at program startup.

To ensure the correctness of this solution, the ".jobs" files must contain requests for different events, i.e. two ".jobs" files cannot contain requests for the same event. For simplicity's sake, the students don't need to ensure or check that this condition is respected (they can assume that it will always be respected in the tests carried out in the assessment phase).

The parent process will have to wait for each child process to finish and print it out via the *std-output* the corresponding termination status.

Exercise 3. Parallelization using multiple tasks

This exercise aims to take advantage of the possibility of parallelizing the processing of each .job file using multiple tasks.

The number of tasks to be used for processing each ".job" file, **MAX_THREADS,** must be specified by command line at program start-up. Synchronization solutions for accessing the state of events that maximize the degree of parallelism achievable by the system will be valued. However, the synchronization solution developed must guarantee that any operation is carried out in a way that "atomic" (i.e. "all or nothing"). For example, it should be avoided that when executing a "SHOW" operation for an event, partially executed reservations can be observed, i.e. reservations for which only a subset of all the desired places have been allocated.

It is also intended to extend the set of commands accepted by the system with these two additional commands:

- **WAIT <delay_ms> [thread_id]**
  This command injects a delay of the duration specified by the first parameter into all tasks before processing the next command, if the optional parameter *thread_id* is not used. If this parameter is used, the delay is injected only into the task with the "thread_id" identifier.

  Examples of use:
  - **WAIT 2000**
    - All tasks must wait 2 seconds before executing the next command.
  - **WAIT 3000 5**
    - The task with *thread_id* = 5, i.e. the 5th task to be activated, waits 3 seconds before executing the next command.

- **BARRIER**
  Forces all tasks to wait for the previous commands to finish.
  BARRIER before resuming execution of the following commands.

  To implement this functionality, the tasks, on encountering the **BARRIER** command, should return from the function executed by *pthread_create returning* an *ad hoc* return value (e.g., the value 1) in order to indicate that they have encountered the **BARRIER command** and have not finished processing the command file (in which case the tasks should return a different return value, e.g., 0).

  The *main* task, i.e. the task that starts the "worker" tasks using *pthread_create()* should observe the return value returned by the worker tasks using *pthread_join* and, if it detects that the **BARRIER** command has been found, it starts a new round of parallel processing that should resume after the **BARRIER** command.

  Examples of use:
  - **BARRIER**
    - All tasks must reach this point before proceeding with their next commands.

This exercise should ideally be carried out from the code obtained after solving exercise 2. In this case, the degree of parallelism achievable will be **MAX_PROC * MAX_THREADS**. However, no penalties will be applied if this exercise is solved from the solution of exercise 1.

Submission and evaluation

Submission is made through Fénix **until 15/12/2023 at 23h59**.

Students must submit a file in *zip* format with the source code and the *Makefile*. The submitted file should not include other files (such as binaries). In addition, the *make clean* command should clean up all files resulting from the compilation of the project.

We recommend that students ensure that the project compiles/runs correctly on the *sigma* cluster. When evaluating submitted projects, if there is any doubt about the functioning of the submitted code, the teachers will use the sigma cluster to make the final validation.

The use of other environments for the development/testing of the project (e.g. macOS, Windows/WSL) is permitted, but the faculty will not provide technical support for queries relating specifically to these environments.

Assessment will be carried out according to the assessment method described on the course website.

Students cannot share code or solutions with other groups. The code submitted must be the result of each group's original work. Submitting code with a high degree of similarity to other groups or made using entities external to the group will lead to the groups involved failing and the situation being reported to the LEIC coordinator and the IST Pedagogical Council.