Seminar 5 - Exam preparation

In this seminar, we will solve exercises from past exams (March and July 2024) to familiarize you with the types of questions typically asked. The session will focus on two key topics: threads and processes with file management.
We will work through four exercises:

- The first two will cover threads, reinforcing concepts such as synchronization and concurrency.
- The last two will focus on processes and file operations, providing practice with inter-process communication and file handling.

By reviewing these exercises, you will gain a better understanding of the exam format and strengthen your problem-solving skills.

Seminar exercises

1. We want to do a multithreaded computation of normalising a vector `u` to make it unitary, and storing it in `unorm`. The sequential code is given below as an example, and consists of a first step computing the sum of its squares, and a second step dividing each component of the vector by the square root of the norm. Notice that any step 2 must be done only after step 1 has been fully completed.

```
void normaliseVectorSequential(double *u, double *u_norm , int N) {
// Step 1 : compute u_norm_2 := || v || ^ 2
double u_norm_2 = 0 ;
for ( int i = 0 ; i < N ; ++i )
      u_norm_2 += vNext [ i ] * vNext [ i ] ;
// Step 2 : compute vNext = vNext / sqrt ( u_norm_2 )
for ( int i = 0 ; i < N ; ++i )
      u_norm [ i ] = u [ i ] / sqrt ( u_norm_2 )
}
```

   If you have problems with the sqrt, use -lm flag when you run the code in the terminal.

a) Create a threaded version of the code above, creating a thread for each element of the vector. In this exercise, you will create two sets of threads.
b) Implement a barrier, as seen in class. Remember that the barrier will block until all threads have reached it, and then release all waiting threads simultaneously. You can find the struct and functions of barriers that you need to implement in the cheatsheet.
c) Use a barrier implementation of the normalisation, where both steps are done in a single function. What are the advantages when compared to the solution of the first part of this problem?

2. The objective of this exercise is to implement a program that concurrently increases and decreases a counter using threads. It involves a thread increasing the counter a random number whenever it reaches 0, and three decreasing threads decreasing the counter, each one decreasing the counter by 1, 5 and 10 units at a time

respectively. A code for the increase part is already given below, and consists of a single thread that adds a random number to the counter reaches 0. Assume that there is a global variable `end` that will be set to one to mark that the program will end when the counter reaches 0. You do not need to worry about race conditions related to the end variable, and you can assume it will change due to an external event.

```
int counter = 0 ;
int end = 0 ;
int increment ( void * a ) {
while ( end == 0 ) {
if ( counter == 0 ){
counter += rand ( ) \%1000 ; // add a random number
}
}
```

a) Are there any race conditions in the increment thread, when the decrement threads are executing? If so, identify the variables involved, the critical region (in lines) and which synchronisation tool you would use.
b) Implement the decrease function, that will decrease the counter by 1, 5 and 10 units at a time, depending on the thread executing it. Avoid race conditions and starvation.
c) Implement the main program, that will create the 3 decreasing threads and the increment thread, and then waits until all threads finish.

3. Write a program whose main will create seven processes that will search for “gold” in a text file. The name of the file that each child will open will be passed through a pipe (one pipe for each child process). Then, the children will search for appearances of the substring “gold” in their file, and whenever they find it, they will send it to the main through another pipe, indicating the position and filename where it was found (this pipe shared among the different children). To pass information you can use the structure GoldPosition below. Then, the main will read from the pipe all instances of found gold, and write them in the standard in/output. Remember to use the adequate functions to compare and copy strings, and that strings need to have a ’\0’ at the end.
    a) Create a function checkStringAtPos that, given a file descriptor, a position and a string, returns 1 if the file contains that string at the given position, -1 if the string would fall outside of the range of the file, and 0 otherwise.
    b) Using the previous function, implement the described code for the main and the children. Remember to properly close all pipes.

```
typedef struct { // you can modify i f you want
char fileName [100] ;
int pos ;
} Gold Position;
```

4. In this exercise, you will implement a multi-process program that uses pipes for inter-process communication. The task is to read a series of numbers from a file, encoded in ASCII as a string, convert them to a binary-encoded integer, and write them in a second file. The code will be concurrent, using a father and a child

processes. The father will read numbers one by one from the input file, convert them to integers and pass them through a pipe to the son. The son will read from the pipe and write each integer to the output file. The two processes will end when the input file has been completely read, and all integers have been processed. The path of the input and output files is given as the first and second command-line arguments.

a) Write a function that reads a string from a source identified by a filedescriptor, until a ”\0” byte marking the end of the string is read. You cannot use `lseek`, and you cannot read ahead of the end of the string.
b) Implement the rest of the program, using the previous function. Make sure that you avoid deadlocks related to pipe communication.

Hint: remember that you can use `sscanf` or `atoi` to convert from string to integer and that you can use “`hexdump -e '1/4 "%d\n"' output.bin`” for displaying the content of a binary file in human-readable form.