Introduction The purpose of this paper is to familiarize you with threading and sockets.

In this work you will change the distributed travelMonitor tool you made in Exercise 2 to use threads, while the communication with the parent process and monitors will be over sockets. The general idea and functionality of the work is similar to Exercise 2, ie here too a central (parent) process will accept requests from citizens who want to travel to other countries, will check if they have received the appropriate vaccination, and will approve if allowed the entry into a country by a traveler. Specifically, you will implement the travelMonitorClient application which will create a series of monitor processes that, together with the application, will answer user questions. In very general terms (details below), in relation to the 2nd exercise the differences are that the communication of the parent process with the monitors is done through sockets instead of pipes, and that the files with the data of the countries are read by a number of threads.

TravelMonitorClient (50%) and monitorServer (50%)

The travelMonitorClient application will be used as follows:

```
./travelMonitorClient –m numMonitors -b socketBufferSize -c cyclicBufferSize -s sizeOfBloom -i input_dir -t
   numThreads
```

where:

The numMonitors parameter is the number of monitorServer processes that the application.
- The socketBufferSize parameter: is the size of the buffer in bytes for reading over sockets.

The cyclicBufferSize parameter: is the size of the buffer in entries (eg 10 file names) for monitorServer circular buffer (see below)
The sizeOfBloom parameter specifies the size of bloom filters in * bytes *. Indicative size of the bloom filter for exercise data will be of the order of 100Kbytes. This parameter is of the same logic as the corresponding parameter of the second work.
- The numThreads parameter specifies the number of threads in each monitorServer process (details below) - The input_dir parameter: is a directory that contains subdirectories with the files to be processed by monitorServer processes. Each subdirectory will have the name of a country and will contain one or more files. For example, input_dir could contain subdirectories China / Italy / and France / which have the following files:

-

/input_dir/China/China-1.txt /input_dir/China/China-2.txt /input_dir/China/China-3.txt
...
/input_dir/Italy/Italy-1.txt /input_dir/Italy/Italy-2.txt
...

```
/input_dir/France/France-1.txt /input_dir/
France/France-2.txt /input_dir/France/
France-3.txt
...
```

Each file contains a series of citizen records where each line describes a citizen's vaccination status for a specific virus. For example, if the contents of the /input_dir/France/France-1.txt .txt file are:

```
889 John Papadopoulos France 52 COVID-19 YES 27-12-2020
889 John Papadopoulos France 52 H1N1 NO
776 Maria Tortellini France 36 SARS-1 NO
125 Jon Dupont France 76 H1N1 YES 30-10-2020
```

means that in France we have a citizen (John Papadopoulos) who has been vaccinated for COVID-19 on 27-12-2020 but not for H1N1, Maria Tortellini who has not yet been vaccinated for SARS-1, and
Jon Dupont who was vaccinated for H1N1. In this exercise you can assume that there will be no lines that will create inconsistency. If you want to keep the same functionality as in Exercise 2 (that is, you throw away the contradicting records) you can. Specifically, a record is an ASCII line of text the same as task 2.

To get started, the travelMonitorClient application should fork numMonitors child processes. Each child process will call the exec with an executable file a program called monitorServer that you will write and which will take as arguments:

```
monitorServer -p port -t numThreads -b socketBufferSize -c cyclicBufferSize -s sizeOfBloom path1 path2 ... pathn
```

where
- port is the port where he will listen for communication with the parent process. Both the parent process (ie travelMonitorClient) and the monitorServers will run on the same IP of the machine they started which they will have to find automatically **(be careful** to find the IP, not use 127.0.0.1). - t numThreads is the number of threads that will create the original thread for editing input files.
- The socketBufferSize parameter: is the size of the buffer in bytes for reading over sockets. - The cyclicBufferSize parameter: is the size of the buffer in entries for the circular buffer of monitorServer. It is the size of a **circular** buffer to be shared between threads created by the monitorServer process. Represents the number of paths that can be stored in it (eg 10, means 10 file names). The sizeOfBloom parameter specifies the size of bloom filters in * bytes *. - path1… pathn are the paths from the countries that this monitorServer undertakes to serve. TravelMonitorClient will distribute subdirectories evenly (in round-robin alphabetically as in Exercise 2) with the countries in input_dir on monitorServer

processes.

When the application (parent process) finishes the initialization actions, it will wait for a series of bloom filters from the monitorServer processes (as in Exercise 2) via the open sockets and when it receives all the information, it will be ready to accept input (commands) from the user from the keyboard (below for commands).

The initial thread in each monitorServer process will create a buffer where the data will be the **file names inside the** path1 ... pathn paths. The initial thread will then start numThreads threads and listen to the port for connections from the travelMonitorClient to receive queries. Each of the numThreads threads will remove a file from the buffer, read the corresponding file, and update the shared data structures used to answer the queries promoted by the parent (travelMonitorClient) process. These structures are the same as Exercise 2, ie one structure will be the bloom filter that will be used to quickly check whether a citizen (with a citizenID ID) has been vaccinated for this virus. You can assume that there will be no inconsistencies in the input files. Each of the numThreads threads wakes up when there is at least one path in the buffer. If the number of files is larger than the size of the buffer, then the parent thread should put, when there is space in the buffer, the rest of the files, so that everything can be read.

Each monitorServer process, after it has finished reading the input files, will send through a socket to the parent process a bloom filter for each virus that will represent all the vaccinated citizens of the countries managed by the monitorSever process. When the monitorServer process finishes reading its files and has sent all the Bloom filters to the parent, it notifies the parent process through the socket that it is ready to accept requests.

In this exercise you do not need to implement any functionality for the signals, ie you do not need the corresponding code for the signals of Exercise 2. When a monitorServer process is about to end (via / exit - see below), then it prints to a file named log_file.xxx (where xxx is its process ID), the name of the countries (subdirectories) it manages, the total number of requests it received to enter the countries it manages, and the total number of requests approved and rejected. Just like in the 2nd exercise.

If the parent process receives the / exit command, it will send a corresponding command over the socket to the monitorServers, and print it to a file named log_file.xxxx where xxx is its process ID, the name of all countries (subdirectories)) who participated in the application with data, the total number of requests received to enter the countries, and the total number of requests approved and rejected. Just like in the 2nd exercise.

The user will be able to give the following commands to the travelMonitorClient application:

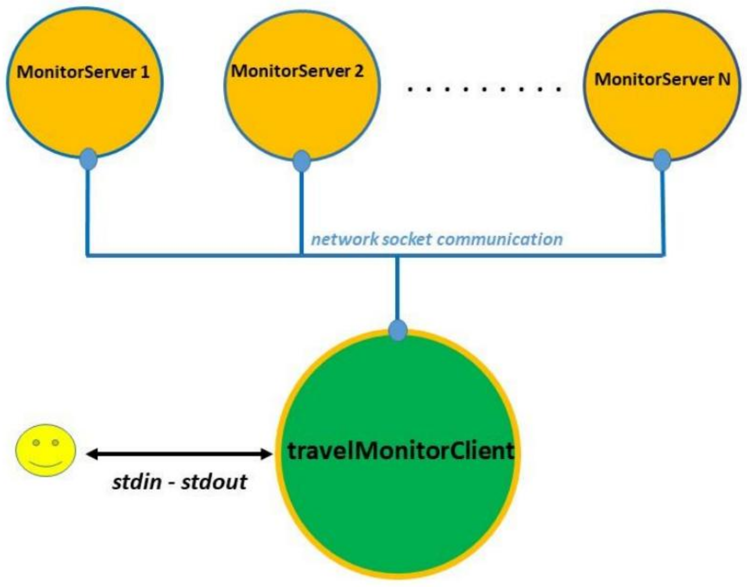
- / travelRequest citizenID date countryFrom countryTo virusName The function is exactly the same as the 2nd exercise. Communication is done over a socket.

- / travelStats virusName date1 date2 [country]
  The operation is exactly the same as the 2nd exercise. Communication is done over a socket.

Add / addVaccinationRecords country With this request the user has placed in input_dir / country one or more files for editing. The parent (travelMonitorClient) process sends a notification via socket to the monitorServer process that manages the country that there are input files to read in the directory. The monitorServer process reads whatever new file it finds, updates the data structures, and sends back to the parent process, via socket, the updated bloom filters that represent all the vaccinated citizens. More specifically, the initial (listening) thread puts in the circular buffer whatever new file it finds for reading and editing from the numThreads threads which will update the data structures. Once the processing of the new input files is completed, a thread will send back to the parent process the updated bloom filters representing the total number of citizens vaccinated in the countries managed by the monitorServer process.

• / searchVaccinationStatus citizenID The
function is exactly the same as exercise 2. Communication is done over a socket.

Exit / exit
Exit the application. The parent process sends a corresponding exit command (your choice exactly how it will be) to the monitorServers, via socket and prints in a file named log_file.xxxx where xxx is its process ID, the name of all countries (subdirectories) who participated in the application with data, the total number of requests received to enter the countries, and the total number of requests approved and rejected. Before it terminates, it will properly release all the free memory. The format is the same as in the 2nd exercise.



Communication between the parent process and each MonitorServer takes place via sockets (see image). How many sockets and what communication protocol you will have between travelMonitorClient and monitorServers is your design choice.

Whatever design choices you make, you should describe them in a README file that you will submit with your code.

Remarks -
This task requires a lot of thought and good planning in terms of allocated resources, especially in the case of threads that can access the same structures. Common variables shared between multiple threads should be protected using mutexes. It is emphasized that busy-waiting is not an acceptable solution for staying access to a common buffer between the threads of your programs. - The exercise does not describe all the details and structures for the simple reason that the design options are exclusively yours (make sure of course that you describe them in detail in README). If you have different options for a point in the exercise, think about the pros and cons, document them in README, choose what you think is right and logical, and describe why you chose it in README.

Deliverable
- A brief and comprehensive explanation of the choices you have made in designing your program. 1-2 pages of ASCII text are enough. Include the explanation and instructions for compiling and running your program in a README file along with the code you will submit.

- The code you will submit should be yours. It is forbidden to use code that has not been written by you **(this includes code from the Internet!) .** - All your work (source code, Makefile and README) in a tar.gz file named OnomaEponymoProject3.tar.gz. Be careful to submit only code, Makefile, README and not binaries.

- It would be good to have a .tar backup of your exercise just as it was submitted to an easily accessible machine (department server, private github repository, private cloud). - The correct submission of a correct tar.gz that contains the code of your exercise and whatever files are needed is your sole responsibility. **Licenses tar / tar.gz or tar / tar.gz that are wrong and are not extracted are not graded.**

Procedural -
For additional announcements, follow the course forum at piazza.com. The full address is https://piazza.com/uoa.gr/spring2021/k24/home. Attendance at the Piazza Forum is a must. - Your program should be written in C (or C ++). If you use C ++ you can not use the ready-made structures of the Standard Template Library (STL). In any case, your program should run on the Linux workstations of the Department.

- Your program should compile the executable (travelMonitorClient) and (monitorServer) have the same names for the parameters (-m, -b, -s, -i, -p) exactly as described in the pronunciation.

- Your password should consist of at least two (and preferably more) different files. Using separate compilation is imperative and your password should have a Makefile.

- Make sure you follow good software engineering practices when implementing the exercise. The organization, readability and existence of comments in the code are part of your rating.

- The submission will be done via eclass.

Other important remarks - The tasks are individual. - Anyone who submits / presents code that has not been written by him / her is reset to lesson.

- Although you are expected to discuss with friends and colleagues how you will try to solve the problem, copying code (of any kind) is not allowed. Anyone involved in copying code simply gets zero in the lesson. This applies to those involved regardless of who gave / received, etc. We emphasize that you should take the appropriate steps to ensure that your password is protected and not stored somewhere accessible to another user (eg, the excuse " I had put it in a github repo and he probably took it from there ", it is not accepted.)

- Programming exercises can be given with a delay of up to 3 days and with a penalty of 5% for every day of delay. Except for these 3 days, no exercises can be submitted.