June 6th, 2022

CSE344 – System Programming – Final project - v1

A - The dataset

The dataset will be in the form of a directory structure as below:

input_directory/
input_directory/ADANA/
input_directory/ADANA/05-05-2026
input_directory/ADANA/07-11-2001
input_directory/ADIYAMAN/
input_directory/ADIYAMAN/26-08-2071
input_directory/ADIYAMAN/06-03-2056
…
input_directory/ZONGULDAK/
input_directory/ZONGULDAK/29-05-2053
input_directory/ZONGULDAK/17-12-2051

In short it will contain a certain number of city sub-directories, and in each city-subdirectory it will contain a fixed number of ASCII files with a date as a filename in the format DD-MM-YYYY (all months assumed to be 30 days long). The dates will start from 01-01-2000.

Each date file's contents will start with a constant number of records as rows, and each record will represent a real-estate transaction that took place during that date at that city in the form of a space-separated line; e.g.:

```
661 TARLA DEMIRYOLU 1000 1500000
```

where

661 is the transaction id (a unique across dataset non-zero positive integer identifier)
TARLA is the type of real-estate (one of a fixed set of possible options, no spaces.)
DEMIRYOLU is the name of the street where the real estate takes place (no spaces).
1000 is the surface in square meters.
1500000 is its price in TL.

B - The application

There are 3 programs to be developed: servant, server and client (Figure 1). The servant processes will answer the requests coming from the server through sockets. The client will make requests to the server through sockets, and the server will respond to those requests via the information acquired from the servants.

e.g. the client will ask the server the question X, the server will check its notes about which servant(s) know(s) the answer, contact the servant(s), get the answer(s), and respond to the client.

B1 – The Servants

```
./servant -d directoryPath -c 10-19 -r IP -p PORT
```

where

directoryPath denotes the path of the root directory containing the dataset
10-19 denotes that it will handle the cities 10 to 19 in alphabetical order inside the dataset
IP is the IPv4 address of the server
PORT is the port number that the server is listening to (remember that the first 2000 ports are very often occupied by kernel related processes, so go -very- beyond 2000 in order not to cause any “port occupied” errors).

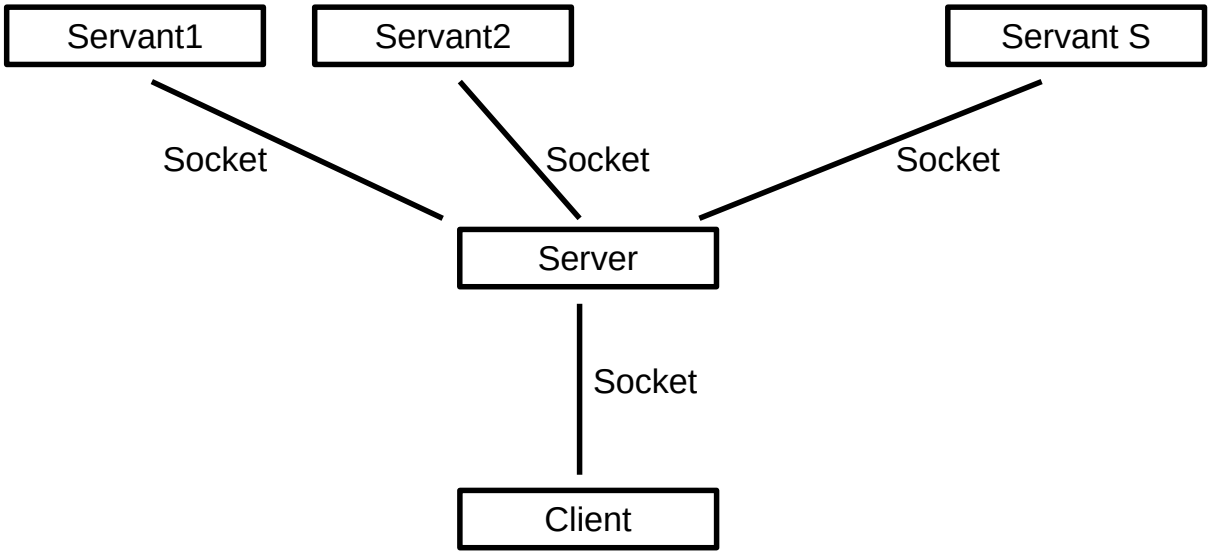

Figure 1 – Overall schema of the programs to be developed.

Each servant process will start by loading from the disk the part of the dataset for which it’s responsible and store it in a data structure (this data structure will NOT be shared across servants). The choice of data structure is up to you. Don’t use an array, make a sensible choice appropriate for a 3rd year engineering student.

Each servant process will then connect to the server process using the IP and PORT provided in the commandline, and let the server know (at least) 2 things:

a) for which subset of the dataset it is responsible for.
b) from which PORT number the server can connect to this servant afterwards. Careful, we want the server to be able to contact the servant processes later, in order to ask them questions about the dataset. For this, the server needs to know at which port it should connect to each servant. So, find a robust way to allocate (*tahsis etmek*) a unique port number to each servant and share it with the server process during this socket communication. Then you can close that connection.

Once that stage is done, each servant process will open their own port and listen for connections coming from the server. Each servant process will create a new thread to handle incoming connections from the server.

The servant processes will terminate when they receive a SIGINT.
The outputs will be written to STDOUT, and each row will be preceded by “Servant process_id”

Output examples:

```
Servant 6672: loaded dataset, cities ADANA-ANKARA
Servant 6672: listening at port 16001
...
Servant 6673: loaded dataset, cities ANTALYA-BİNGÖL
Servant 6673: listening at port 16002
...
Servant 6672: termination message received, handled 10 requests in total.
Servant 6673: termination message received, handled 2 requests in total.
...
```

B3 – The server

```
./server -p PORT -t numberOfThreads
```

where

PORT is the port number from which the server is listening for incoming connections.

NumberOfThreads (at least 5) is the number of connection handling threads to be created for your pool (only once at the beginning of the process), that will be used to handle the incoming connections. The number of threads of the server will be fixed.

The server will wait for connections on its port. The main thread will always forward incoming connections to a pool of threads and return to waiting new connections as soon as possible. Establish a queue to store the incoming connections and distribute them to non-busy pool threads through a monitor as explained in class.

However, the server can be contacted either by servant processes on its PORT, or by a client process. It is up to you to decide a mechanism to figure out who's at the other end of the connection.

When the server is contacted by a servant process, it will learn from the servant dataset information (e.g. for which part of the dataset that servant process is responsible for and how to contact it for requests...).

When the server is contacted by a client on the other hand, its job is to respond to the client's requests. The requests that can arrive from the client process are as follows:

```
transactionCount TYPE d1 d2 [CITY]
```

As a response the server must return the number of real-estate transactions that take place in the dataset between dates d1 and d2 (inclusive) of the type TYPE. If additionally the CITY parameter is provided, then the search must be limited for that city alone.

To accomplish this task, the server thread handling this connection will first decide which servant processes it must communicate with (based on the information they shared at the beginning with the server). Then the thread handling this connection will open up connection(s) to the servant(s) involved (one by one), collect the responses, and respond with the cumulative answer to the client. If there is no servant that can satisfy the request it will return an error to the client; e.g. if the client asks for transactions at TUNCELI, but there is no servant responsible for TUNCELI.

The server process will terminate when it receives a SIGINT (and will forward it to the servants). It will write its output to STDOUT and each server output row will be preceded by a human readable timestamp (i.e. current date and time):

Output example

```
Servant 6672 present at port 16001 handling cities ADANA-ANKARA
Servant 6673 present at port 16002 handling cities ANTALYA-BINGOL
...
Request arrived “transactionCount TARLA 01-01-2000 10-01-2021 ISTANBUL”
Contacting servant 6675
Response received: 1, forwarded to client
Request arrived “transactionCount TARLA 01-01-2000 10-01-2021”
Contacting ALL servants
Response received: 11, forwarded to client
...
SIGINT has been received. I handled a total of 30 requests. Goodbye.
```

B4 – The client

```
./client -r requestFile -q PORT -s IP
```

where

requestFile is the path of the file containing requests to be sent to the server
PORT is the port number of the server process to connect to
IP is the IP address of the server process.

The client process will start by reading all the requests in the request file. For every request it will create one thread and every thread will be resposible for connecting to the server with its own connection, sending the request and receiving its response, which the thread will print on STDOUT.

However, the threads must not send their requests to the server before all threads have been created. Yes, this is a synchronization barrier. We want all threads to make their connection requests to the server at approximately the same time. Let’s see if your server can handle the suuden load of N requests. Each thread that receives its response will print it and terminate. The client will terminate after the last thread terminates.

Output example

```
Client: I have loaded 30 requests and I’m creating 30 threads.
Client-Thread-0: Thread-0 has been created
Client-Thread-1: Thread-1 has been created
...
Client-Thread-29: Thread-29 has been created
Client-Thread-5: I am requesting “/transactionCount TARLA 01-11-2021 30-11-2021 SAMSUN”
...
Client-Thread-5: The server’s response to “/transactionCount TARLA 01-11-2021 30-11-2021 SAMSUN” is 1
...
Client-Thread-5: Terminating
Client: All threads have terminated, goodbye.
```

Code organization

It has come to my attention that your code organization is often...just awful. Organize your source file per module, (client, servant, server), have a separate file for utilities/data structures/networking, always separate headers from source.
I don’t want to see any code repetition.

I want to see documented functions and variables.
I want to see proper indentation.
I want to see function return values being checked.
I want to see proper name notation for functions (underscored or camel notation).

Restrictions/requirements
All paths can be relative or absolute
Don't add new command line arguments to the given ones, or modify existing ones.
Files don't have to have only the content you expect (they might have trailing empty rows)
Don't use the system() or popen() or signal() or getpid() functions.
No late submissions; don't leave submission to the last minute.
In case of a runtime error, exit by printing to **STDERR** an informative message about the source of error.
All sockets are to be stream sockets.
All mutexes are to be normal/fast mutexes.

Guidelines and tips.
If something is unspecified then it means it's up to you.
For questions, ask at the teams channel, private questions won't be answered.
Think hard and design well your application **before** starting to code.
I don't want to see many statements in your signal handlers. Remember what I taught you.

Evaluation
I've shared the execution script and dataset with you. The execution script at the time of the demo might have a different number of servants, however the number of cities will be divisible by the number of servants. The client process might be executed multiple times with the same or different request file. The dataset and the request files will be different at the time of the demo from those that have been shared with you.

Grading
1) Compilation error: -100 with a possibility of resolution **IF** it's due to architecture/distribution/C library version issues.
2) Compilation warning (with respect to the **-Wall** flag): -10 (regardless of the number of warnings)
3) Makefile without -Wall at the compilation step or makefile without "make clean" (cleaning all binary files): -25 You are at the 6. semester, yoru are expected to write a proper Makefile!
4) No report or an insufficient report (as a pdf or as a README text file): -20
5) The program crashes/deadlocks/freezes: -100
6) Poor synchronization (sleep/nanosleep, busy waiting, timed waits, trylocks): -100
7) The submission violates a restriction or doesn't satisfy a requirement: -100
8) The submission doesn't satisfy a specification: -X, will depend on the missing specification.
9) Presence of memory leak (regardless of amount – checked with valgrind) -30
10) Provide usage information if the parameters are missing/invalid/wrong: -15 otherwise
11) Submitting the right files is your responsibility. Empty or wrong submissions will not be graded.
12) If you don't cleanup after children processes/threads and/or leave zombies: -50
13) If your code is not well-organized and well-documented (directory organization into modules, header/source separation, function and variable documentation) then -10.
14) In case any 2 or more submissions contain snippets of identical/similar code, all involved parties will be graded with -100, regardless of who shared with whom or from which common source you copied it. Do your own coding.

What to submit:

- Your source files, your makefile and a report; place them all in a directory with your student number as its name, and zip the directory.
- Your report must contain: how you solved this problem, your design decisions, which requirements you achieved and which you have failed
- The report must be in English.
- Your makefile must only compile the program, not run it!
- Do not submit any object files.
- Your code will be compared against online sources; you are free to be inspired, but you are also expected to write your own code.
- Proven cases of plagiarism will be punished to the full extent.

Good luck