CSE344 – System Programming
Final Project – Report

Berk Pekgöz
171044041

Solving The Problem & Design:

Server Part

On server process first executes these steps in order:

- Taking measures against double instantiation
- Parsing command line arguments(with getopt)
- Becoming daemon
- Opening log file as writing
- Printing executing parameters
- Settings for SIGINT handling
- Opening dataset file for reading
- Creating Database and Filling Database
- Creating pool threads
- Initializing socket queue
- Socket, bind and listen

Taking measures against double instantiation:
To prevent double instantiation, I used named semaphore with O_CREAT | O_EXCL flags. With this way, if another server program try to create semaphore, returns error. Name of named semaphore is defined at top the .c file.

Becoming daemon:
To make the server daemon, *same steps with the example which is shown in lecture is coded*. There is a small difference that only one flag can be given to the function and that flag controls *umask()* call.

Settings for SIGINT handling:
To achieve interrupt signal handling, first a handler function is set with *sigaction()*. Then, SIGINT is masked for threads with *pthread_sigmask()* function. After the pool threads are created, mask is recovered for only main thread. With this way, only main thread can catch interrup signal and synchronization of the program is making it possible to achieve handling signal and terminating.

Creating Database and Filling Database:
After the dataset file opened for reading, program reads first row of the file and gets number of columns. Then read rest of the file line by line and gets number of columns. After these reads program allocates space for database with number of columns and number of rows. Then, read file again and fills allocated database with entries. Size of entry string(char array) is defined at top of the .c file. Database structures are defined at beginning of the code.

Socket, bind and listen:
To connect to socket, these functions are called in order: *socket(), setsockopt(), bind()* and *listen()*. Also *htons(port)* function is called.

After the server initialization part and start barrier...

Main thread cycle:
This cycle is a while loop. First accepts a connection from socket and puts that socket to the socket queue (This queue structure is same with queue structure in HW# 4). After that there is a synchronization which is very similar to producer consumer problem and main thread is the producer side of the problem. This synchronization is achieved with mutexes and condition variables.

Waits with "*empty*" condition variable until one of the pool threads signals that condition variable. And checks for busythreads and term_flag for this condition variable.

After getting lock of that problem increments *busythreads* and *socketcount*. These are shared data between main thread and pool threads and provide synchronization.

Sends signal to "full" condition variable of pool threads to make them unblocked and check.

If there are no available thread to assign connection, thread will be blocked with condition variable "*empty*" .

Pool threads:
Pool threads are executes a while loop. In that loop these steps are executed in order: wait until job assigned(this is the consumer part of the problem which is mentioned in main thread cycle part), get a socket file descriptor from socket queue (this queue is protected with mutex), parse query, get reader/writer lock, execute query and finally release reader/writer lock. A thread continues reading from same socket until receive "end" from that socket.

Consumer part of producer/consumer problem is achieved with "full" condition variable. And that condition variable checks "socketcount" and "term_flag" .

After completing job with current connection, decreases "busythreads" and signals to main thread's "empty" condition variable.

Queries with DISTINCT keyword does not work as asked, they behave as normal SELECT query.

Server side of connection:
To communicate with clients, there is a communication protocol. To send data from server to client, first server sends a integer. If that integer is zero or less, it means server will send a message with BUFFER_LEN. If that integer is greater than zero, it means server will send table information and that integer is the column size. After that server sends another integer which means row size of the table which will be sent. Then server starts sending column $*$ row times entries with ENTRY_LEN. After that wait for another query or "end" message.

Terminating:
Program handles SIGINT with int_handler() function. Only main thread can catch signal(explained above). After handling, term_flag becomes true and threads can not continue their loops. After exiting loops, pool threads returns to main thread. When flag becomes TRUE, main thread's loop can not continue too and exits loop. After exiting loop, main thread joins all pool threads and free all resources then exit.

Client Part

Client has simple work.

- First, parses command line arguments with getopt().
- Open the file which contains queries.
- Open socket and connect.
  Uses inet_addr() function to convert ip address.
- Read lines from file until EOF.
- If line start with client's own id, then sends that query to the server using connected socket and waits for server's response.
- Receives column size and row size.
- Creates temporary table with given sizes. Size of entry string(char array) is defined at top of the .c file. Database structures are defined at beginning of the code.
- Fill that table with entries.(Exactly row $*$ column entries will be received).
- Print that table
- Continue reading from file.
- If EOF reached, send "end" message with BUFFER_LEN to server for informing.
- Exit program.

Notices:

- In client output, print of entries formatted with % 20.20s to be nicely formatted.
- There is timestamp on each row for both client and server.
- Dataset file must be in csv structure.
- Input files are not modified.

- Server is a daemon process and code of become_daemon() is very similar to the code shown in lecture.
- Server process has no controlling terminal.
- Length of table entries defined at top of .c file as ENTRY_LEN. You can change that value.
- Length of buffers defined at top of .c file as BUFFER_LEN. You can change that value.
- All kinds of synchronization between thread achieved with mutexes and condition variables.
- Only one instance of server can be executing at the same time.
- Main thread clean up after all threads.
- In case of an arbitrary error, exit by printing to stderr a nicely formatted informative message and free resources.
- In case of SIGINT the program (that means all threads), return all resources to the system and exit with an information message.
- If the required command line arguments are missing/invalid, program print usage information and exit.
- All requirements mentioned in PDF except “DISTINCT” keyword have been achieved.
- No memory leak with valgrind.