System Programming Final Report

Yusuf Patoglu

June 9, 2021

1 Problem Definition

This project aims to create a **server** that can serve multiple **clients**.The server will act as a SQL Server. It will listen the requests and handle them in most efficient way. Server's synchronisation problem will be solved by **POSIX mutexes** and **condition variables** only.

2 Server's Program flow

- ***Parsing command-line arguments:*** The program starts by parsing command-line arguments with *getopt* function. If the parameters are not valid then it returns an error message and terminates the program.

```
void
parse_args(int argc, char**argv, int *port_no, char
    *path_to_log_file, int *pool_size , char *dataset_path)
```

- ***Preventing multiple instantiations :*** The server can have only one instance at a time. Therefore we need to block multiple instantiations. To accomplish this, I created an *abstract unix socket.* Since sockets are implemented in kernel level, in case of any crush or normal termination of a server, the socket file descriptor will be released automatically. If the socket file descriptor is still in use, on another attempt bind will throws an *existing address* error.
  I also implemented another version of this preventing mechanism with file locks. But since it's forbidden in the project, I didn't use it. It's still available in the code.

```
struct sockaddr_un sun;
//https://unix.stackexchange.com/a/219687
s = socket(AF_UNIX, SOCK_STREAM, 0);
if (s < 0) {
    perror("socket");
    exit(1);
}
memset(&sun, 0, sizeof(sun));
sun.sun_family = AF_UNIX;
strcpy(sun.sun_path + 1, "anyNameYouWant");
if (bind(s, (struct sockaddr *) &sun, sizeof(sun)))
{
     fprintf(stderr,"Daemon server can only executed once.
         If the other daemon terminates the socket will be
         released.\n");
     exit(EXIT_FAILURE);
}
```

- ***Daemon process opens a log file :*** Since we've closed all of the file descriptors we can't do i/o from anywhere. Therefore we open a log file to print the logs.

```
for (fd = 0; fd < maxfd; fd++)
{
    if(fd != abstract_fd)
    {
        close(fd);
    }
}
```

- ***Setting a signal handler for SIGINT:*** In case of receiving the CTRL + C (SIGINT) signal the server process should clean up explicitly and also wait for working threads, then exits gracefully. I'll be talking about this on *main thread: synchronizer* section.

```
struct sigaction sa;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = handler;
if (sigaction(SIGINT, &sa, NULL) == -1)
{
    fprintf(log_file, "Sigaction returned an error.
        Exiting the program\n");
    exit(EXIT_FAILURE);
}
```

- ***Main thread creates a circular queue:*** Main thread pre-allocates a queue for incoming requests. Basically this will be a job queue. It's implemented using a dynamic array.

- ***Initialisation of database and the selection of the data structure:*** In order to keep the database entries, **3D char array** is used. The reason to use array is so simple. Because in our database operations we don't add or remove any columns. So whole operations will be completed in O(n) time except **SELECT DISTINCT** command. It's executed in a quadratic time complexity. Better solution would be using a hashmap for O(1) time complexity for all of the operations. But it's hard to implement in short time and many things could go wrong if not implemented correctly.

- ***Main thread creates the thread pool:*** Main threads are spawned before accepting any connections. Therefore we have to make sure that those threads should wait until the main thread arrives. Therefore I used a barrier to accomplish this.

```
robust_pthread_mutex_lock(&mutex);
    arrived++;
    if(arrived < pool_size)
    {
        pthread_cond_wait(&rendezvous, &mutex);
    }
    else
    {
        fprintf(log_file, "A pool of %d threads has been
            created.\n", pool_size);
        pthread_cond_broadcast(&rendezvous);
    }
    robust_pthread_mutex_unlock(&mutex);
```

- ***main thread: synchronizer:*** Before accepting any connections, the server should follow the mandatory steps: bind and listen. After completing these operations main thread is ready to accept connections.
  Main thread will accept connections in a while loop. Since accept system call blocks until a request is arrived the main thread will only work if there is a request. When main thread got a request it enqueues the client's file descriptor. But this is a common data structure between all threads. Therefore it has to be locked. This is lock is not just for protecting the critical area. It also used with a condition variable to signal one of the threads like: "Hello random thread, there is a job for you. Go ahead and execute it please."

```
        robust_pthread_mutex_lock(&mutex);
        enqueue_client(&clients, client_fd);
        pthread_cond_signal(&request_condition_variable);
        robust_pthread_mutex_unlock(&mutex);
```

Also there is a better solution for a main thread with this queue. If all of the threads are busy, then instead of refusing to accept any new requests, main thread will enqueue the incoming threads. So as soon as any threads become available, they will immediately dequeue the job and execute. If there are no remaning jobs they will sleep.

```
/*  DEQUEUE THE JOB*/
        robust_pthread_mutex_lock(&mutex);
        while(empty(clients))
        {
            current_sleeping_threads++;
            fprintf(log_file,"Thread #%lu: waiting for
                connection.\n", pthread_self());
            pthread_cond_wait(&request_condition_variable,
                &mutex);
            current_sleeping_threads--;
            if(current_sleeping_threads == 0)
            {
                fprintf(log_file,"No thread is available! (Main
                    thread is still be able to queue jobs.)\n");
            }
        }
        client_fd = dequeue_client(clients);
        if(client_fd == -1)
        {
            robust_pthread_mutex_lock(&mutex_terminate);
            threads_executed++;
            pthread_cond_signal(&terminate_cond);
            robust_pthread_mutex_unlock(&mutex_terminate);
            robust_pthread_mutex_unlock(&mutex);
            return NULL;
        }
        else
        {
            fprintf(log_file,"A connection has been delegated
                to thread id #%lu\n",pthread_self());
        }
        ll++;
        robust_pthread_mutex_unlock(&mutex);
```

When the termination signal is received main thread is waits for other threads to finish it with a condition variable. To make sure that each one

of the threads finished their jobs, main thread will send a negative client id for each thread. So when they got a negative value, those threads will understand, they should exit.

- ***pool threads:*** Pool threads will dequeue the incoming descriptors and handle them. They read the query from these descriptors, parse the query, modify-read the database and send back to specified client like this:

```
while((read = robust_read(client_fd, pack, PACKLEN)) > 0)
    {

        //TODO: FIX THIS. NUMBER IS NOT NEEDED.
        fprintf(log_file,"Thread #%lu: received query
            %s\n", pthread_self(), pack);

        if(command_classifier(pack) == 1)
        {
            robust_pthread_mutex_lock(&m);
            while((AW + WW) > 0)
            {
                WR++;
                pthread_cond_wait(&okToRead, &m);
                WR--;
            }
            AR++;
            robust_pthread_mutex_unlock(&m);
            //ACCESS DB HERE.
           // sleep(1);
            int record_count = SELECT(client_fd, pack);
            fprintf(log_file,"Thread #%lu: query completed,
                %d records have been returned.\n",
                pthread_self(), record_count);


            robust_pthread_mutex_lock(&m);
            AR--;
            if(AR == 0 && WW > 0)
            {
                pthread_cond_signal(&okToWrite);
            }
            robust_pthread_mutex_unlock(&m);
        }
        else
        {
            robust_pthread_mutex_lock(&m);
            while((AW + AR) > 0)
            {
                WW++;
                pthread_cond_wait(&okToWrite, &m);
```

```
            WW--;
        }
        AW++;
        robust_pthread_mutex_unlock(&m);
        //ACCESS DB HERE.
        //sleep(1);
        UPDATE(client_fd, pack);
        robust_pthread_mutex_lock(&m);
        AW--;
        if(WW > 0)
        {
            pthread_cond_signal(&okToWrite);
        }
        else if(WR > 0)
        {
            pthread_cond_broadcast(&okToRead);
        }
        robust_pthread_mutex_unlock(&m);
    }
}
```

As it can bee seen the reader-writer paradigm is used which priorities the writers.

3 Client's Program flow

- ***Parsing command-line arguments:*** The program starts by parsing command-line arguments with *getopt* function. If the parameters are not valid then it returns an error message and terminates the program.

  ```
  void
  parse_args_c(int argc, char**argv, int *client_id, char
      *ip_addr, int *port_no, char *query_file)
  ```

- ***Client creates a SIGINT handler:*** The client shouldn't be interrupted by SIGINT. Otherwise it could broke the connection between client and server and we'd crash with SIGPIPE signal in server's side. Therefore a SIGINT handler is created. This is not a perfect solution but it shows the awareness of a problem.

- ***Client connects and send queries:***Client connects to server and send queries. Then each client will receive the results of their queries.

References

[1] Little Book of Semaphores

[2] Advanced Programming in the Unix Environment

[3] Advanced Linux Programming

[4] https://linux.die.net/man/