Göktuğ Ali Akın
161044018

CSE 344
SYSTEM PROGRAMMING

FINAL PROJECT REPORT

Problem definition

Multi-threaded client server : A multi-threaded server and a client. The server will load a graph from a text file, and use a dynamic pool of threads to handle incoming connections. The client(s) will connect to the server and request a path between two arbitrary nodes, and the server will provide this service.

Loading graph from file

- create_graph_from_file();
- read_file();

These functions used to load the graph into memory before the starting the server. Implementation can be found in utils.h and utils.c source folders.

Dynamic thread pool

In order to run the server more faster, dynamic thread pool implemented in this project. By this way, clients gets service more faster instead using one thread to serve.

Before the starting, accepting the connections, server process creates a number of threads that will handle the client's requests. Firstly, these threads are sleeping, waiting by condition variable.

When a connection accepted by main thread of server process, main thread push the **file descriptor** that was returned by the **accept()** method into shared job queue between worker threads. And then, signal the condition variable to wake up the one thread to handle the connection. After pushing the new job into queue, any thread can get this file descriptor and handle the client's request.

In order to represent the job queue (it holds file descriptors taken by accept() method), Linked List representation of queue data structure used.

To able to thread pool behave like dynamic, I used linked list data structure. Threads (pthread_t objects) stored in the global linked list. By this way, it is easy to manipulate the data structure. If the load of the server is reaches the %75, the resizer thread activated and adds %25 more thread into the thread pool.

Resizer thread function in the source code : **resizer_function(void*)**

Every time threads get job from job queue, they increment the busy_thread counter and signal the resizer thread the represent the number of busy threads in the server. By this way, resizer thread can check the how much the server is busy. All synchronization problems are solved by mutex and condition variables.

Synchronization problems

Producer - Consumer problem between main thread and worker threads

There is a simple producer consumer problem between main thread and the worker threads in the server process. Main thread, which listens the connections, creates (produces) jobs for the worker threads into the shared queue. And worker threads gets these jobs from the queue and handle them. So let's look the actors of the synchronization problem one by one ;

Producer : Main thread
Consumer : Worker threads
Product : file descriptor, returned by the **accept()** -> represents the request.

By this way, worker threads gets file descriptors from the shared job queue, reads the request as string from this file descriptor by **read()** system call, and writes the result into this file descriptor as string by **write()** system call.

Readers / Writers problem between worker threads

Worker threads are reading and writing the cache data structure. But, writing is more important than the reading, because after the write operation, the database (cache) will be more actual than the older. In order to give more priority and protect the shared data structure, I used 4 variables and two condition variables.

Implemented data structures (includes cache)

```
typedef struct LinkedList
{
        node_t* head;
        node_t* tail;
        int size;
}LinkedList;
```

→ Storing the cache data structure, adjacency list in the graph, storing the all threads (dynamic thread pool), queue for BFS and job queue between threads. All data structures implemented by linked list.
In order to access / add the end of the list, tail node is stored in the struct to do operations in **O(1)** time.

```
typedef struct Graph
{
        LinkedList** adj;
        int vertex;
        int edge;
}Graph;
```

→ Graph is implemented by adjacency list method. This implementation uses memory better.

```
typedef struct path_t
{
        int from;
        int to;
        char* str;
}path_t;
```

→ Cache data structure, contains path_t objects in the linked list in order to store the old requests taken by the clients. This linked list, does not contain duplicate entries.

Becoming a daemon process

Avoid multiple instances of program running in the machine

In order to achieve this goal, I used named semaphore. Server process will creates named semaphore by this system call ;

sem_open(SERVER_SEMAPHORE_NAME,O_CREAT | O_EXCL,0644,0);

By giving the **O_CREAT | O_EXCL** mode flags, we can guarantee that, only one server process can run at the same time. If other user tries to run server process again, **sem_open** system call returns **EEXIST** error number that indicates that another server process creates the named semaphore before, thanks to **O_EXCL** flag.

Server process will close and unlink the named semaphore before the terminating by this system calls.

- **sem_close()**
- **sem_unlink()**

By this way, another time, new server process can be executed.

Closing inherited file descriptors and detaching from terminal.

In order to make the process daemon, I do some operations. These are ;

- Fork() and terminate the parent process.
- Become childs process the leader of the session by setsid() call.
- fork() again and terminate the parent process. Child (server) continues.
- Change the current working directory by chdir("/") call.
- Close all inherited file descriptors except input file and log file.
- Re-open the 0,1 and 2 with opening the /dev/null device.

Implementation can be found in the server.c source code, **startDaemon()** function.

Handling CTRL-C (SIGINT) terminating signal in server

If server process gets SIGINT signal, first it waits all threads to complete their already given jobs. The threads that are busy, will send response to the clients. They are not popping new jobs from job queue, just complete their already given tasks. After the completing the jobs, worker threads check the volatile atomic variable **cancel_request** before accepting new job**.** If it is true, worker threads exit.

cancel_request = 1;
pthread_cond_broadcast(&cond_queue_empty); // signal worker threasd
pthread_cond_signal(&cond_resize_need); // signal resizer thread

If there are any threads that waiting on a condition, it will wake up the threads.

Project folder includes ;

- **server.c** : source code of the server process.
- **client.c** : source code of the client process.
- **tester.c** : source code of the tester program. This is not a part of homework. It makes easy to test the server process by creating large number of clients using fork + execve.