CSE344
Systems Programming Course
Final Project Report

Oğuzhan Agkuş
161044003

Deadline: June 28^{th}, 2020

1 Problem Definition

The project implements a server that accepting requests from clients. The clients want a path between two nodes of a graph that hosted by server. The server gets these requests and return the path to client. The avoid repetitive requests the serves have a cache structure. Firstly, it checks the cache if the request was reached before. If the request in the cache, it directly return from cache. Otherwise it perform BFS (Breadth First Search) on the graph at store the found path in cache.

The server is a daemon process. It means it has only one instance and has no terminal access. It can be stopped gracefully by sending SIGINT signal with kill command.

The server have a dynamic pool of threads to accept more request. Main thread accepts coming connections and forward them to threads for handling incoming requests. If there is not enough threads, pool will be extended by another thread called re-sizer thread. In this situation the the threads must access same resources. It occurs some synchronization problems. The details are below.

2 Solution Approach

The main threads accepts incoming connections and create a file descriptor to communicate. Then add this file descriptor to a global queue. The threads get a file descriptor from the queue and handle it. This is first synchronization problem which accessing the request queue.

In a short time many requests can arrive but there may be no threads available. In this situation main thread should wait by checking the server load. Some threads increase this load (accepting request), some are decrease (after replying), so the main thread make sure that it is the only one that access the load value. This is another synchronization problem.

The last problem is accessing the cache. In a time, only one thread can access the cache. There are two types of threads for cache, readers and writer. The program designed according to writer priority. (I tried to implement reader priority and equal priority each of them. It runs but I am not sure they will not cause another problem.)

The cache has not complex design. It is an array of paths. The searching is performed linearly. If the size is reach to capacity, it resizes itself by x2.

The graph implementation is based on adjacency list. Each node stores a list of adjacent nodes.

The resizer thread, extend the pool by 25% until the maximum limit which is given by user. The thread waits for a condition variable. It represents that the server needed more threads. Any thread that increase the server load send signal to this thread. If resizing needed, it resizes. And send a signal to main thread, if it is waiting for available threads.

The server has no access to terminal. So, all messages will be written to a log file.

The client program is basic with respect to the server. It only sends requests and wait for reply.

3 Server Code Flow

3.1 Input Checking

```
1
2   if (argc != 11)
3     error_exit("Less/more arguments!\nUsage: sudo ./server -i input
      .txt -o log.txt -p 8080 -s 4 -x 24", -1);
4
5   while ((opt = getopt(argc, argv, "i:o:p:s:x:")) != -1) {
6     switch (opt) {
7       case 'i':
8         input_path = optarg; break;
9       case 'o':
10        log_path = optarg; break;
11      case 'p':
12        port = atoi(optarg); break;
13      case 's':
14        start = atoi(optarg); break;
15      case 'x':
16        maximum = atoi(optarg); break;
17      default:
18        error_exit("Invalid options!\nUsage: sudo ./server -i input
      .txt -o log.txt -p 8080 -s 4 -x 24", -1);
19    }
20  }
21
22  if (port < 0 || port > 65535)
23  error_exit("Port number should be in [0, 65535] interval.", -1);
24
25  if (start < 2)
26    error_exit("Number of threads in the pool at startup should be
      at least 2!", -1);
27
28  if (start > maximum)
29    error_exit("Number of threads in the pool at startup should be
      smaller than maximum number of the threads!", -1);
30
31  if ((log_fd = open(log_path, O_WRONLY | O_CREAT, 0600)) < 0)
32    error_exit("Failed while opening/creating log file", errno);
33
34  close(log_fd);
```

3.2 Becoming Daemon Process

```
int become_daemon() {
  int lock_fd, temp_fd, max_fd;

  switch (fork()) {
    case -1: return -1;
    case  0: break;
    default: exit(EXIT_SUCCESS);
  }

  if (setsid() == -1)
    return -1;

  switch (fork()) {
    case -1: return -1;
    case  0: break;
    default: exit(EXIT_SUCCESS);
  }

  max_fd = sysconf(_SC_OPEN_MAX);
  if (max_fd == -1)
      max_fd = 8192;

  for (temp_fd = 0; temp_fd < max_fd; temp_fd++)
    close(temp_fd);

  umask(0);
  // chdir("/");
  lock_fd = create_lock();

  return lock_fd;
}
```

Creating a file in home directory an setting a write lock on it blocks multiple instantiation. when another instance of the program run, it checks the file and detect the lock and exits.

```c
int create_lock() {
  int fd, count;
  char buffer[16];
  char *file_directory = "/.config/server.pid";
  char *home_directory = getenv("HOME");
  char *lock_directory = malloc(strlen(home_directory) + strlen(
    file_directory) + 1);

  strncpy(lock_directory, home_directory, strlen(home_directory) +
    1);
  strncat(lock_directory, file_directory, strlen(file_directory) +
    1);

  fd = open(lock_directory, O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  if (fd == -1)
    return -1;

  free(lock_directory);

  struct flock lock;
  lock.l_type = F_WRLCK;
  lock.l_whence = SEEK_SET;
  lock.l_start = 0;
  lock.l_len = 0;

  if (fcntl(fd, F_SETLK, &lock) == -1) {
    close(fd);
    return -1;
  }

  if (ftruncate(fd, 0) == -1) {
    close(fd);
    return -1;
  }

  count = sprintf(buffer, "%d", getpid());
  write(fd, buffer, count);

  return fd;
}
```

3.3 Initializing Graph

```
1
2  graph_t *read_graph(char *input_path) {
3    int fd, flag, index, ignore, from, to;
4    char read_buffer, buffer[16];
5    graph_t *graph;
6
7    flag = 0;
8    index = 0;
9    ignore = 0;
10   memset(buffer, 0, 16);
11
12   if ((fd = open(input_path, O_RDONLY)) == -1)
13     exit(1);
14
15   graph = create_graph(10);
16
17   if (graph == NULL)
18     return NULL;
19
20   while (read(fd, &read_buffer, 1) == 1) {
21     if (ignore == 1) {
22       if (read_buffer == '\n')  // Ignore until \n
23         ignore = 0;
24       continue;
25     }
26
27     if (read_buffer == '#') {   // If there is a # ignore until \n
28       ignore = 1;
29       continue;
30     }
31
32     if (read_buffer == ' ' || read_buffer == '\t' || read_buffer ==
        '\n') {
33       if (flag == 0) {
34         from = atoi(buffer);
35         flag = 1;
36       }
37       else {
38         to = atoi(buffer);
39         flag = 0;
40
41         if (add_edge(graph, from, to) == -1) {
42           delete_graph(graph);
43           return NULL;
44         }
45       }
46       memset(buffer, 0, 16);
47       index = 0;
48     }
49     else
50       buffer[index++] = read_buffer;
51   }
52   close(fd);
53   return graph;
54 }
```

3.4 Initializing Mutexes and Conditional Variables

```
result = pthread_mutex_init(&load_mutex, NULL);
if (result != 0)
error_handler("pthread_mutex_init() error", result);

result = pthread_mutex_init(&queue_mutex, NULL);
if (result != 0)
  error_handler("pthread_mutex_init() error", result);

result = pthread_mutex_init(&cache_mutex, NULL);
if (result != 0)
  error_handler("pthread_mutex_init() error", result);

result = pthread_cond_init(&waiting_connection, NULL);
if (result != 0)
error_handler("pthread_cond_init() error", result);

result = pthread_cond_init(&available_thread, NULL);
if (result != 0)
error_handler("pthread_cond_init() error", result);

result = pthread_cond_init(&resize_condition, NULL);
if (result != 0)
error_handler("pthread_cond_init() error", result);

result = pthread_cond_init(&ready_to_write, NULL);
if (result != 0)
  error_handler("pthread_cond_init() error", result);

result = pthread_cond_init(&ready_to_read, NULL);
if (result != 0)
  error_handler("pthread_cond_init() error", result);
```

3.5 Preparing Socket

```
optval = 1;
address_len = sizeof(address);
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(port);

if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
  error_handler("socket() failed", errno);

if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT
  , &optval, sizeof(optval)) == -1)
  error_handler("setsockopt() failed", errno);

if (bind(server_fd, (struct sockaddr *) &address, sizeof(address)
  ) == -1)
  error_handler("bind() failed", errno);

if (listen(server_fd, 5) == -1)
  error_handler("listen() failed", errno);

```

3.6 Accepting Requests

```
  while (1) {
    if ((temp_fd = accept(server_fd, (struct sockaddr *) &address,
    (socklen_t *) &address_len)) == -1)
      error_handler("accept() failed", errno);

    pthread_mutex_lock(&load_mutex);
    while (working_count == thread_count) {
      write_log(log_fd, "No thread is available! Waiting for one.")
    ;
      pthread_cond_signal(&resize_condition);
      pthread_cond_wait(&available_thread, &load_mutex);
    }
    pthread_mutex_unlock(&load_mutex);

    pthread_mutex_lock(&queue_mutex);
    enqueue(request_queue, temp_fd);
    pthread_cond_signal(&waiting_connection);
    pthread_mutex_unlock(&queue_mutex);
  }

```

3.7 Error Handling

```
void error_handler(char *message, int error) {
  char buffer[256];

  if (error == -1)
    sprintf(buffer, "Error occurred: %s", message);
  else
    sprintf(buffer, "Error occurred: %s: %s", message, strerror(
    error));

  write_log(log_fd, buffer);
  write_log(log_fd, "Exited with failure.");

  exit_handler();
  exit(EXIT_FAILURE);
}
```

3.8 Interrupt Handling

```
signal(SIGINT, signal_handler);

void signal_handler(int signal) {
  char buffer[256];

  sprintf(buffer, "%s signal received.", sys_siglist[signal]);
  write_log(log_fd, buffer);

  exit_handler();
  exit(EXIT_SUCCESS);
}
```

3.9 Handlers

```
void exit_handler() {
  thread_handler();

  free(thread_arguments);
  free(thread_pool);
  delete_queue(request_queue);
  delete_cache(cache);
  delete_graph(graph);

  write_log(log_fd, "All heap blocks were freed.");
  write_log(log_fd, "Server shutting down.");

  close(server_fd);
  close(log_fd);
  close(lock_fd);
  delete_lock();
}

void thread_handler() {
  int i;

  exit_flag = 1;

  write_log(log_fd, "Waiting for ongoing threads to complete.");

  if (thread_pool != NULL) {
    pthread_cond_broadcast(&waiting_connection);

    for (i = 0; i < thread_count; i++)
      pthread_join(thread_pool[i], NULL);
  }

  pthread_cond_signal(&resize_condition);
  pthread_join(resizer_thread, NULL);

  write_log(log_fd, "All threads have terminated.");
}
```

3.10 Logging

```c
int get_timestamp(struct timeval timestamp, char *buffer) {
  struct tm time_struct;
  int year = 0, month = 0, day = 0, hour = 0, minute = 0, second =
    0, milisecond = 0;

  memset(&time_struct, 0, sizeof (struct tm));
  localtime_r(&timestamp.tv_sec, &time_struct);

  year = time_struct.tm_year + 1900;
  month = time_struct.tm_mon + 1;
  day = time_struct.tm_mday;
  hour = time_struct.tm_hour;
  minute = time_struct.tm_min;
  second = time_struct.tm_sec;
  milisecond = timestamp.tv_usec / 1000;

  return sprintf(buffer, "%02d-%02d-%04d %02d:%02d:%02d.%03d", day,
     month, year, hour, minute, second, milisecond);
}

```

```c
void write_log(int fd, char *message) {
  int count;
  char buffer[4096];
  struct timeval temp_time;

  gettimeofday(&temp_time, NULL);
  get_timestamp(temp_time, buffer);

  count = sprintf(buffer, "%s\t| %s\n", buffer, message);

  write(fd, buffer, count);
}

```

4 Client Code

```
if (send(client_fd, (void *) &request, sizeof(request_t), 0) == -1)
    error_exit("send() failed", errno);

  gettimeofday(&start_time, NULL);

  if (read(client_fd, (void *) &distance, sizeof(int)) == -1)
    error_exit("read() failed", errno);

  if (distance == 0) {
    gettimeofday(&end_time, NULL);
  }
  else {
    path = malloc((distance + 1) * sizeof(unsigned int));

    if (read(client_fd, (void *) path, (distance + 1) * sizeof(
    unsigned int)) == -1) {
      cleanup();
      error_exit("read() failed", errno);
    }

    gettimeofday(&end_time, NULL);
  }

  report(path, distance, get_duration(start_time, end_time));
```

5 Compiling and Running

The source code includes a makefile. It compiles all cources with needed parameters like -pthread. Also it compiles with -Wall and -Wextra for warnings.In my system (Ubuntu 18.04.4, Kernel version: 5.3.0) it is compiling without any error or warning.

Valgrind test are successful in many cases there are no memeory leak and error.

6 Output

Client outputs will written on terminal. Server outputs written will written on log file. Any important execution will reported by a timestamp.

In my tests, the requests are very fast handled so the program did not need to resize the pool. I extend the execution time by sleep for 2 or 3 seconds. Then it is easy to see that the pool resizing.