Final Report

System Programming

June 8, 2021

Muhammed Okumuş

151044017

Contents

1 Functions

Under this section I will mention some of the important functions utilized in the program flow.

```c
// Worker thread function
void *worker_thread(void *data);
// Misc functions
void print_usage(void);
void print_inputs(void);
char *timestamp(void);
void exit_on_invalid_input(void);
// Database initilization functions
DataBase *db_init(void);
int lines(const char *path);
// Database debugging functions
void db_print(int start, int end, int field_indices[MAX_FIELDS], char keys[MAX_FIELDS][MAX_LINE], int key_count);
// SQL parsing functions
int process_cmd(char cmd[MAX_REQUEST], int socket_fd);
// SQL helper functions
int get_field_index(char *field_name, char delim);
void db_print_row(int n, int socket_fd, int field_indices[MAX_FIELDS], char keys[MAX_FIELDS][MAX_LINE], int key_count);
void db_print_fields(int key_count, int field_indices[MAX_FIELDS], char keys[MAX_FIELDS][MAX_LINE], int socket_fd);
// SQL qeury handlers
int gather_output_star(int key_count, int field_indices[MAX_FIELDS],
char keys[MAX_FIELDS][MAX_LINE], int socket_fd);
int gather_output_some(int key_count, int field_indices[MAX_FIELDS], char keys[MAX_FIELDS][MAX_LINE], int socket_fd);
int gather_output_distinct(int key_count, int field_indices[MAX_FIELDS], char keys[MAX_FIELDS][MAX_LINE], int socket_fd);
int gather_output_update(int key_count, int field_indices[MAX_FIELDS], char keys[MAX_FIELDS][MAX_LINE], int socket_fd, int index_where, char *where);
int gather(int key_count, int field_indices[MAX_FIELDS], char keys[MAX_FIELDS][MAX_LINE], int socket_fd, int index_where, char *where, int mode);
// Hashing & Searching functions
int jenkins_one_at_a_time_hash(char *key);
int linear_search(int *arr, int n, int x);
// Job queue access functions
void add_request(int client_socket);
Job get_next_request(void);

// Signal handler
void sigint_handler(int sig_no);
```

Worker thread function is assigned to each worker thread that handles connections. It handles all the reads and writes between the server and the assigned client. Its data is simple struct that holds an integer to represent the thread id. It can function as a both reader and writer according to clients request. Syncronization used by threads and reader/writer paradigm is explained in detail in later sections.

Database initialization function handles the file reading and tokenization of the strings. It's designed to handle comma seperated files with double or single quotes and long nested comma seperated strings. Files are read in to a three dimensional character array. Each field of each row is allocated to have just enough space to hold each string and a termination character(backslash 0). Design choice behind this is explained in detail in later sections.

Gather functions are called after a command is parsed, they receive their parameters from the process command function and perform the query by linearly searching and comparing each row according to the given command. There are four different gather functions and they are encapsulated a wrapper called 'gather' to improve portability and readability.

Hash and linear search functions are used for 'SELECT DISTINCT' queries. Instead of comparing strings which is costly, each row is hashed and checked for duplicity. Hash string is builded by combining each field of the query and then the string hashed by using the 'jenkins one at a time' hashing algorithm.

Add request and get next request functions are used to access a job queue structure represent by a linked list that consists of nodes. Each node holds a clients socket descriptor and next nodes information. This structure must be protected by a mutex to prevent race condtions and that is handled by the main and the worker threads.

2 Data Structures

Data structure used to hold the data table is a three dimensional char array. This has some advantages and disadvantages. These can be listed as the following.
Advantages:

- Simple to implement.
- Takes about the same space as the csv file.
- O(1) on access operations, O(N) in search operations.
- Easily expandable with realloc since it's all pointers.

Disadvantages:

- Search operation is slow compared to hash tables(Avarage case).
- No way to prevent duplicate insertions without traversing the whole array.

To hold the client request a queue structure is used. It supports O(1) add operation and O(1) pop operation.

```c
typedef struct{
  int client_socket;
} Job;

struct node{
  struct node *next;
  Job job;
};

typedef struct{
  struct node *head;
  struct node *tail;
  int n;
  int requests_handled;
} RequestList;
```

By using a job queue main thread can keep queuing jobs even tough if all threads are busy. When a job is added to the queue by the main thread, it signal a condtion variable to wake up atleast one thread to let them know there are jobs in the queue, if all threads are busy, when one becomes available it skips the wait on the condition variable by checking the job queue size.

```c
// Inside worker loop
pthread_mutex_lock(&mutex_job); //mutex lock
while (RL.n == 0)
{
  //print_log("Thread #%d: waiting for job.", td->id);
  pthread_cond_wait(&cond_job, &mutex_job); //wait for the condition
  if (exit_requested)
  {
    print_log("Thread #%d: terminating.", td->id);
    pthread_mutex_unlock(&mutex_job);
    return NULL;
  }
}

print_log("A connection has been delegated to thread id #%d", td->id);
Job curr_job;
curr_job = get_next_request();
//print_log("Thread #%d: received query '%s'", td->id, curr_job.command);
pthread_mutex_unlock(&mutex_job);

// Process command
// ...
```

3 Server-Client Structure and Signal Handling

Server-Client structure is very simple thanks the the job queue. The server simply waits for a connection in a while loop that checks for atomic variable each loop which is modified by a signal handler that handles SIGINT. When a connection is accepted, client socket descriptor that is received from the accept function is added to the queue by add queue function. This fucntion call is protected by a mutex. After the call is made, a condtion variable is signaled to let thread(s) know there is a new job in the queue. If SIGINT is received, server loop falls through to a broadcast call to same condition variable used for jobs. Since each thread checks the same atomic variable after the it wakes up from a condtion and at start of its each loop, it also falls through to the termination part of the code.

```
// MAIN THREAD
// ...
// Handle connections while no SIGINT
while (!exit_requested){
addr_len = sizeof(addr_client);
client_socket = accept(server_socket, (struct sockaddr *)&addr_client, &addr_len);
if (client_socket > 0){
      // Add request to jobs
      pthread_mutex_lock(&mutex_job);
      add_request(client_socket);
      pthread_mutex_unlock(&mutex_job);
      pthread_cond_signal(&cond_job); // Signal new job
    }
}
  print_log("Termination signal received, waiting for ongoing threads to complete.");
  exit_requested = 1;
  pthread_cond_broadcast(&cond_job);
// ...
// WORKER THREAD
  while (!exit_requested){
    pthread_mutex_lock(&mutex_job); //mutex lock
    while (RL.n == 0) {
      //print_log("Thread #%d: waiting for job.", td->id);
      pthread_cond_wait(&cond_job, &mutex_job); //wait for the condition
      if (exit_requested){
        print_log("Thread #%d: terminating.", td->id);
        pthread_mutex_unlock(&mutex_job);
        return NULL;
      }
    }
    print_log("A connection has been delegated to thread id #%d", td->id);
    Job curr_job;
    curr_job = get_next_request();
    //print_log("Thread #%d: received query ‘%s‘", td->id, curr_job.command);
    pthread_mutex_unlock(&mutex_job);
    // Do processing ...
}
```

4 Reader-Writer Paradigm

This part is directly implemented by the guidence of the lecture slayts. One minor difference is that a thread can switch between a writer and reader. This is decided by parsing the command received by the client. Whole command is not parsed to decide, command is only check if it has a 'SELECT' or and 'UPDATE' statement and thread proceeds accordingly.

```
while (read(curr_job.client_socket, buffer, MAX_REQUEST) > 0) {
  //...
    if (opt[0] == 'U'){
      // Writer
      pthread_mutex_lock(&mutex_lock);
      while (_AW + _AR > 0){ // If any readers or writers, wait
        _WW++;  // Waiting Writer
        pthread_cond_wait(&okToWrite, &mutex_lock);
        _WW--;
      }
      _AW++;    // Active writer
      pthread_mutex_unlock(&mutex_lock);
      n_queries = process_cmd(buffer, curr_job.client_socket);  // Access DB
      pthread_mutex_lock(&mutex_lock);
      _AW--;
      if (_WW > 0)  // Give write priority
        pthread_cond_signal(&okToWrite);
      else if (_WR > 0)
        pthread_cond_broadcast(&okToRead);
      pthread_mutex_unlock(&mutex_lock);
    }
    else{
      //Reader
      pthread_mutex_lock(&mutex_lock);
      while (_AW + _WW > 0){ // If any writers wait
        _WR++;  // Waiting reader
        pthread_cond_wait(&okToRead, &mutex_lock);
        _WR--;
      }
      _AR++;  // Active reader
      pthread_mutex_unlock(&mutex_lock);
      n_queries = process_cmd(buffer, curr_job.client_socket); // Access DB
      pthread_mutex_lock(&mutex_lock);
      _AR--;
      if (_AR == 0 && _WW > 0)
        pthread_cond_signal(&okToWrite);
      pthread_mutex_unlock(&mutex_lock);
    }
    //...
  }
```

5 Single Instance Server

Since we do not care about opereting system portability we can use special tools from the Linux tool kit. To achieve single instance check, the program utilizes the abstract sockets. By giving a name to socket via sun path variable and checking for the bind function call return value, we can make sure the program runs only a single instance. This concept is directly acquired from our text book.

```c
// From the book
// https://man7.org/tlpi/code/online/dist/sockets/us_abstract_bind.c.html
char *str;
int sockfd;
struct sockaddr_un addr;
memset(&addr, 0, sizeof(struct sockaddr_un)); /* Clear address structure */
addr.sun_family = AF_UNIX;                     /* UNIX domain address */
str = "xyz";                                   /* Abstract name is "\0xyz" */
strncpy(&addr.sun_path[1], str, strlen(str));
sockfd = socket(AF_UNIX, SOCK_STREAM, 0);
if (sockfd == -1)
  errExit("socket");
if (bind(sockfd, (struct sockaddr *)&addr, sizeof(sa_family_t) + strlen(str) + 1) == -1)
{
  printf("A server instance is already running.\n");
  system("ps -C server -o \"pid ppid pgid sid tty stat command\"");
  exit(EXIT_FAILURE);
}
```

6 Daemonization

Daemonization is initiated after the single instance check. Program forks and the parent process is terminated. Child process parses the original programs arguments and opens the log file. Then all open file descriptos are closed except the abstract socket and the log file the program going to write to. stdout and stderr are redirected to the log file by a dup2 call. Program doesn't redirect stdin since it's doesn't take input from the terminal.

```c
int pid = fork();
if (pid == -1){
    printf("Fork failed\n");
    return -1;
}
// Set process ID in order to cut connection with the terminal
// Will fail in parent, succeed in the child
if (setsid() == -1)
    return -1;
// Exit parent, we daemon now
else if (pid != 0)
    exit(EXIT_SUCCESS);
// ... parse args
int fd_log = open(_O, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
// Close open file descriptors after the parent exits
for (int i = 0; i < FD_MAX; i++){
    if (!(i == fd_log || i == sockfd))
      close(i);
}
// Open log file and redirect stdout and stderr

if (fd_log == -1)
    errExit("Failed to open log file");
// Close open file descriptors after the parent exits
dup2(fd_log, 1); //stdout > fd
dup2(fd_log, 2); //stderr > fd
// ...
```

7 Extras And Notes

In addition to required SQL commands **'SELECT * WHERE column1='key1', column2='key';'** command is implemented. Below commands are tested in the **'new-dwellings-consented-by-statistical-area-2-april-2021-csv'** file which is around 50MB.

```
1 SELECT * FROM TABLE;
2 SELECT * FROM TABLE WHERE territorial_authority='Far North District'
2 SELECT SA2_code, SA2_name FROM TABLE;
1 UPDATE TABLE SET total_dwelling_units='1453', WHERE month='1990-04-01';
3 SELECT DISTINCT SA2_name FROM TABLE;
4 SELECT DISTINCT territorial_authority FROM TABLE;
5 SELECT DISTINCT month FROM TABLE;
1 UPDATE TABLE SET SA2_name='Waikiki', WHERE SA2_name='Whakapaku';
```

Server running script

```
./server -p 9090 -o Extras/logs.txt -l 4 -d Extras/table1.csv
```

Daemon testing script

```
ps -C server -o "pid ppid pgid sid tty stat command"
```

Client request script

```
./client -i 1 -a 127.0.0.1 -p 9090 -o Extras/query1.txt > Outputs/c1.txt &
./client -i 2 -a 127.0.0.1 -p 9090 -o Extras/query1.txt > Outputs/c2.txt &
./client -i 3 -a 127.0.0.1 -p 9090 -o Extras/query1.txt > Outputs/c3.txt &
./client -i 4 -a 127.0.0.1 -p 9090 -o Extras/query1.txt > Outputs/c4.txt &
./client -i 5 -a 127.0.0.1 -p 9090 -o Extras/query1.txt > Outputs/c5.txt
```