GEBZE TECHNICAL UNIVERSITY, DEPARTMENT OF COMPUTER ENGINEERING

CSE344 – System Programming – Final Report

Ozan Şelte - 161044061

June 28, 2020

1 GENERAL STRUCTURES

1.1 Graph Structure

```
struct Graph {
        uint32_t count, cap, edgeCount;
        uint32_t *indices;
        struct Node *nodes;
        pthread_mutex_t m;
        pthread_cond_t okHigh, okLow;
        uint16_t aL, aH, wL, wH;
};
```

Since the number of nodes in the file and which numbers of these nodes is not known, the numbers read from the file are stored by matching the positions in the program. The index 5 in the file can be 10 in the graph and this operation has O(n) complexity. The matching operations after loading the graph will have O(1) complexity.

The graph stores the nodes as a vector. According to the example above, **indices[5]** equals to 10 so **nodes[10]** is the node that wanted. The graph has **count** nodes, **edgeCount** edges and can store maximum **cap** nodes. **cap** doubles itself when needed. The mutex, cond variables and 16 bit integers are using for Readers - Writers paradigm to manage caches.

1.2 Node Structure

```
struct Node {
    uint32_t count, cap;
    uint32_t *edges;
    uint32_t cacheCount;
    uint32_t *cache;
    uint32_t **paths;
};
```

A node stores its edges as an integer vector which contains indexes in the graph. It has also **cap** and **count** to manage the edges vector. Every node contains its own cache structure inside. When a new path is calculated, the **cacheCount** increments, **cache** and **paths** resizes and then destination file index written to the **cache**. **paths** stores every calculated path as array of file indexed nodes. So if a X -> Y path is known, A(graph index) and B(graph index) calculated. Then it can found in cache **nodes[A].cache[Q] = B**. After that, **nodes[A].paths[Q] = {5, X, Z, W, K, Y}** can be used. The first element is the length of the path.

1.3 Queue Structure

```
#define GET_BIT(a,n) (((a)[(n)/8] >> (7 - ((n)%8))) & 1)
#define SET_BIT(a,n) ((a)[(n)/8] |= 1 << (7 - ((n)%8)))
#define CLR_BIT(a,n) ((a)[(n)/8] &= ~(1 << (7 - ((n)%8))))
struct Queue {
    struct Queue *next;
    uint32_t *path;
};
```

Every queue shows its next element and stores a path from source to itself. For BFS, visited array could occupy too much space so a binary map used for this feature. Every node in the graph using one bit of space. When returning, every node index in the path is converting from graph index to file index.

2 Connection and Sockets

2.1 Server

```
#define EMPTY_SOCKET (-2)
#define PRIO_READER (0)
#define PRIO_WRITER (1)
#define PRIO_NONE (2)
#define LOCK_PATH "/tmp/cse344_final_lock.pid"
struct ServerConfig {
    char inputPath[LINE_LEN], logPath[LINE_LEN];
    uint16_t port;
    int logFD, inFD, lockFD;
    uint16_t initCap;
    uint16_t maxCap;
    uint16_t currentCap;
    uint16_t count;
    uint8_t priority;
    int sockfd;
    int *sockets;
    pthread_t pooler;
    pthread_t *threads;
    pthread_mutex_t m;
    pthread_cond_t e, f, r;
};
```

When a connection occurs, server find an empty thread, gave the new socket to it and signals a condition vairable for thread pooler. If there is not an empty thread it waits a condition variable. Threads and sockets are storing as vectors.
Waits: ServerConfig.e(empty); Signals ServerConfig.f(full)

2.2 Thread Pooler

When a signal comes from condition variable it checks the pool size. If new threads should be created, it resizes thread and socket vectors, assign new sockets EMPTY_SOCKET, creates worker threads and signals the condition variable for server thread. Then, sends a signal to all worker threads to run with new sockets.
Waits: ServerConfig.f(full); Signals: ServerConfig.r(reloaded) and ServerConfig.e, if resized

2.3 Worker Thread

It checks the socket position in memory when a signal comes. If it is empty, it waits another. If not, it gets two integers from socket. Do its own jobs and send the result to the client.
Waits: ServerConfig.r(reloaded); Signals: ServerConfig.e(empty)

2.4 Client

It just send two integers to the server as source and destination, then waits the path length, zero means there is no path, anything else is the count of the node indexes. After that, client receives the node indexes and prints them.

3 General Problems

3.1 SIGPIPE

When a client closes in a calculation, server receives SIGPIPE and program ends.

Solution SIGPIPE ignored.

3.2 SIGINT

When SIGINT comes, server should wait the working threads and then close itself. There should not any unfreed memory after the closing.

Solution SIGINT is blocked until the graph is loaded. For every thread but main SIGINT is blocked so the signal goes only the main thread. After that when a SIGINT came, the signal handler changes a sig_atomic_t variable and the main thread goes exit function. It signals all the condition variables and other threads check the exit variable. When all of them joined to the main, it frees, destroys, and exits.

3.3 Lister Backlog Queue Limit

If the waiters count aboves the limit, there could be errors.

Solution **SOMAXCONN** used. It is the maximum count according to the machine.

#	Main Thread	Pooler Thread	Worker Thread
Mutex Lock	ServerConfig.m	ServerConfig.m	ServerConfig.m
Wait Cond	ServerConfig.e	ServerConfig.f	ServerConfig.r
Signal Cond	ServerConfig.f	ServerConfig.r	ServerConfig.e
Mutex Unlock	ServerConfig.m	ServerConfig.m	ServerConfig.m

3.4 Socket Send Limit

Send function has a size limit, but path length can be too long.

SOLUTION **SEND_LIMIT** defined in **helper.h**. Sender thread send the path part by part which has length of **SEND_LIMIT**. The client receives also path by part.

3.5 Log File, PID File and Listener Socket

When becoming daemon all the files are closed. But there can be errors before the conversion so we cannot open the files after that. Socket can give error on binding.

SOLUTION Both files opening before the convertion, program checks the rights and closes them. After conversion, opens them again.

4 SYNCHRONIZATION PROBLEMS

4.1 Pool Resizing

It is a problem itself the timing of resizing. When it is done? How the pooler knows it is a good time to increase threads count? This working time, other working threads should not stop.

SOLUTION Producer - Consumer paradigm has used. Every thread is signalling other type of threads. With this order, every thread works accordingly. When exiting, **ServerConfig.f** and **ServerConfig.r** signalling so exiting done easily.

4.2 Exiting

After the moment that SIGINT came, all the resources should be closed, freed etc.

SOLUTION Every thread must be closed but only one of them gets the signal. Main thread is the chosen one.

```
/* Not exactly like this */
pthread_mutex_lock(&conf.m);
pthread_cond_broadcast(&conf.f);
pthread_cond_broadcast(&conf.r);
pthread_mutex_unlock(&conf.m);
close(conf.sockfd);
freeWorkers();
freeGraph();
serverLog("All threads have terminated, server shutting down.\n");
close(conf.logFD);
close(conf.lockFD);
exit(EXIT_SUCCESS);
```

4.3 Waiting Accept

If a signal cames when waiting and **accept**, the program not quits, blocks until a connection came. If we do not use **SA_RESTART** the other functions become more difficult to handle.

SOLUTION **close** function is a signal handler safe function. When a signal came, program opens a new socket, connects it to the listener socket, and accept passes. If exit variable is true, it exits the program.

4.4 Cache Prioritization and Bonus Part

Readers - Writers paradigm should be used but for **-r** parameter it can be dynamic. This means that sometimes, readers should have the priority, but if we reverse the Readers - Writers, more than one Writer can go into the critical section.

SOLUTION Four functions written. The higher prioritize cache function call **highPriority** functions and the other one calls **lowPriority** functions. In unlock functions, independent of the prioritized one, if writers will be awakened, they call signal for condition variables; if readers, they call broadcast.

```
#define PRIO_READER (0)
#define PRIO_WRITER (1)
#define PRIO_NONE (2)
#define READER (0)
#define WRITER (1)
void highPriorityLock();
void highPriorityUnlock(int caller);
void lowPriorityLock();
void lowPriorityUnlock(int caller);
```

5 NOTES

DEBUG FLAG In **helper.h** there is a define line named **F_DEBUG**, if it is uncommented, server runs on terminal, does not become a daemon.

TRANSFER SIZE Data transfer parts size could change from **SEND_LIMIT** in **helper.h**.