CSE344 – System Programming Final Project Report Processes, Sockets, Threads, IPC, Synchronization

151044085-EMRE KAVAK

June 2020

1 Daemon Process

When program start, created locked file for protect double instance and after that called fork() syscall and create child process and main process exited. Main thread checked and started daemon process. After that,closed inherited fd and argument check started.

```c
/////////////// child process area
  if(getpid() == syscall(SYS_gettid)){    // MAIN THREAD START

      int opt,s,x,indx=0;
      char *pathToFile;
      char* portNum;

      if(argc!=11){                        // argument count check
          fprintf(stderr, "%s. Entered count: %d\n","Argument count should be 11",argc );
          usage();
          exit(EXIT_FAILURE);
      }
      while((opt = getopt (argc, argv,"i:p:o:s:x:")) != -1){      // argument chech
          switch(opt){
              case 'i':
                  pathToFile=optarg;
                  break;
              case 'p':
                  portNum=optarg;
                  indx=0;
                  for (; indx<strlen(portNum); indx++){
                      if (!isdigit(portNum[indx])){
                          printf("\n -p %s argument not integer. It should be integer.\n\n",portNum);
                          usage();
                          exit(EXIT_FAILURE);
                      }
```

Figure 1: daemon process

After argument check, created log file and argument information written into log file.

```c
int close_fd=0;
close(close_fd);
struct timeval  tm1, tm2;

mode_t mode= S_IRUSR | S_IWUSR;
logFileFd=open(pathToLogFile,O_CREAT|O_WRONLY|O_TRUNC,mode);

char *msg=" Executing with parameters:\n";
writeLogFile(msg);

char p1[512];
sprintf(p1," -i %s\n",pathToFile);
writeLogFile(p1);


char p2[512];
sprintf(p2," -p %s\n",portNum);
writeLogFile(p2);

char p3[512];
sprintf(p3," -o %s\n",pathToLogFile);
writeLogFile(p3);

char p4[512];
sprintf(p4," -s %d\n",s);
writeLogFile(p4);

char p5[512];
sprintf(p5," -x %d\n",x);
writeLogFile(p5);
```

Figure 2: started log file write

2 Graph Data Structure

I used adjacency List for keep graph into memory. Fort keep graph, I used linked list structure into adjacency list. List size equal unique start node count. When I read graph file, I calculated unique start node count to efficient memory usage. Each unique edges added this adjacency list when file read.

```
struct node {                    // adj list elements load
    long id;
    struct node* next;
};

struct adjList {                 // keep graphs in linked list array
    struct node *head;
};

struct Graph {                   // graph loaded this struct
    long int size;
    long int addedNodeCount;
    struct adjList* array;
};
```

Figure 3: Graph Data Structure

CreateGraph function create dynamic graph according to unique graph size.Array represent Adjaceny list array. All heap filled with NULL because of check empty or not.

```
void createGraph(long int gSize) {
    graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->size =gSize;
    graph->array = (struct adjList*) malloc(gSize * sizeof(struct adjList));
    long int i;
    for (i = 0; i < gSize; ++i)
        graph->array[i].head = NULL;
    graph->addedNodeCount=0;
}
```

Figure 4: create graph function

3 DataBase Structure

DataBase keep calculated paths. It use data struct objects and keeped these objects in an array. Data struct keep start, destination node and result of this nodes calculation. DataBase struct keep data struct nodes array. For search in databse, I used hash code function.

```
struct data{                     // keep database datas
    long int start;
    long int destination;
    char *result;
};

struct dataBase{                 // data base keep calculated paths
    long int addedCount;
    long int capacity;
    struct data* paths;
};
```

Figure 5: DataBase structure

DataBase created with this function. Capacitiy is edges count. All start and destination variables filled with -1 for represent empty index. Data base use getHashCode function for fast search calculated paths in database. This function simply get 2 start node sum get produce hash code according to capacity of database.

```c
void createDataBase(int capacity){
    db=(struct dataBase*)malloc(sizeof(struct dataBase));
    db->paths=(struct data*)malloc(sizeof(struct data)*capacity);

    int i=0;
    for(;i<capacity;i++){           // -1 means empty
        db->paths[i].start=-1;
        db->paths[i].destination=-1;
    }
    db->capacity=capacity;
    db->addedCount=0;
}
```

Figure 6: DataBase create

```c
long int getHashCode(long int value){       // get start+endnode sum and calculate hash code
    long int index=(value)%(db->capacity);
    return index;
}
```

Figure 7: Hash code function

When threads calculate paths, they call this function and this function add results into database with use hash code. If produced hash code index full, function search until found empty place in database. Also, if database full, there is function to extand database size with capacity*2.

```c
int addDataBase(long int s, long int d,char res[]){     // add two nodes path into db wi
    long int index=getHashCode(s+d);
    if(db->paths[index].start==-1 && db->paths[index].destination==-1){         // if in
        db->paths[index].start=s;
        db->paths[index].destination=d;
        db->paths[index].result=(char*)malloc(sizeof(char)*(strlen(res)));      // add r
        sprintf(db->paths[index].result,"%s",res);
        db->addedCount++;
    }else{
        index++;

        int check=0;
        while(db->paths[index].start!=-1 && db->paths[index].destination!=-1){  // false
            if(index==db->capacity){
                index=index%db->capacity;
                check++;
                if(check==2) break;
            }
            index++;
        }
        if(check<2){
            db->paths[index].start=s;
            db->paths[index].destination=d;
            db->paths[index].result=(char*)malloc(sizeof(char)*(strlen(res)));
            sprintf(db->paths[index].result,"%s",res);
            db->addedCount++;
        }
    }
    if(db->addedCount==db->capacity){
        expandDb(2*db->capacity);
    }
    return -1;          // this means, db full;
}
```

Figure 8: database add element function

For get database entry, first call getDbEntryINdex function for check if this entry exists or not with using hash code function. Start and end nodes sum send to hash code function. And returned index checked to -1 or not. -1 means this index empty.

```c
long int getDbEntryIndex(long int n1, long int n2){     // -1 there is no entry, otherwi
    long int index=getHashCode(n1+n2);
    if(db->paths[index].start==-1 && db->paths[index].destination==-1){
        return -1;
    }else{
        if(db->paths[index].start==n1 && db->paths[index].destination==n2)
            return index;
        else{
            index++;
            int check=0;
            while(db->paths[index].start!=n1 && db->paths[index].destination!=n2){  // f
                index++;
                if(index==db->capacity-1){
                    index=index%db->capacity;
                    check++;
                    if(check==2) return 0;          // this means, db full;
                }
            }
            if(index<db->capacity) return index;
            else return -1;
        }
    }
    return -1;
}
```

Figure 9: getting databse entry index

4 BFS Search

This function, first check if requested start node in unique start nodes or not. If there is not, send not path found message to client. If there is, start bfs search. First get start node and add visited array. After that, get start node directed nodes and add these nodes into queue. When finished start nodes adjacents, start pop these nodes from queue and get these nodes adjacents and add visited nodes array this nodes and so on until found destination node. When search paths, I used 2d array for visited nodes paths and when destionation node founds, I just get these paths and according to destination node parent node, I found path from these paths array.

```c
char* bfs(struct Graph *g, long int startNode, long int endNode){       // calculate paths with use bfs search

    int foundeIndex=0, noPath=0,founded=0, okStart=0,okEnd=0,okToFirst=0;
    long int item,i, path_index=0, visited_index=0, lookedEdgeCount=0;
    if(endNode<=maxNodeId && startNode>=minNodeId){

        for(i=0; i<uniqSCount;i++){                // first check start node is there unique start nodes arr
            if(uniqSArr[i]==startNode){
                okStart=1;
                foundeIndex=i;
                i=uniqSCount;
            }
        }
        if(okStart==1){                            // if stat nodes found, do bfs search

            struct node* temp=g->array[foundeIndex].head;

            item=temp->id;
            enqueue(que,item);
            item=dequeue(que);                     // pop start node and add visited

            visited[visited_index]=item;
            visited_index++;

            paths[path_index][0]=item;
            temp=temp->next;
            paths[path_index][1]=temp->id;
            lookedEdgeCount++;
            while(temp){                           // get all connected graph
```

Figure 10: BFS search function

5 Thread Pool

After graph load finished, I first create resizing thread. After that, I created thread pool according to given s size. Maximum thread size and instant thread size keeped in variables and these variables used by threads function and main thread. Thread pool threads use tasksArr array for check there is a request for them or not.

```c
char loMsg2[128];
sprintf(loMsg2, " Graph loaded in %.1f seconds with %ld nodes and %ld edges\n",totalTime,maxNodeId+1,edgesCount);
writeLogFile(loMsg2);

maxThreadCount=x;
instantThreadCount=s;
if(pthread_create(&resizingThread, NULL, resizingThreadFunc,(void*)1)!=0){      // initiliaze helper thread check
        writeLogFile("Error in thread creation");
}
threadPool=(pthread_t*)malloc(sizeof(pthread_t)*instantThreadCount);    /* Theread pool created in there*/
createTasksArr(maxThreadCount);

char msg3[40];
sprintf(msg3," A pool of %d threads has been created\n",instantThreadCount);
writeLogFile(msg3);

for(i=0;i<s;i++){
    if(pthread_create(&threadPool[i], NULL, threadFunc,(void*)i)!=0){   // initiliaze threads
        writeLogFile("Error in thread creation");
    }
}
```

Figure 11: Thread Pool

When threads enter thread function, they check these struct array with their id and if full is 1, it means there is a job fort hem. Sockedfd refers client socket fd for send answer the result them.Each thread used these struct array. Each thread have their own mutex and condition variables.

```c
struct threadTask{
    int id;
    int status;
    int socketFd;
    int full;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
};
```

Figure 12: ThreadTask struct

6 Socket Structure

After dynamic thread pool created, I created socket with socket syscall. I used AF INET type because of we use IPV4 structure. Server will call with 127.0.0.1 ip adres and this IP adress constant in server.c file. After created socket, entered port number and ip adress assigned and used bind for bind ip adress and socket. After that, socket listening started and will accept in infinite loop by main thread. Also signals listen in this line end.

```c
}
int socketFd = 0;
struct sockaddr_in serv_addr;

char resBuff[1025];

memset(&serv_addr, '0', sizeof(serv_addr));
memset(resBuff, '0', sizeof(resBuff));

if((socketFd = socket(AF_INET, SOCK_STREAM, 0))<0){     // socket initialize
    writeLogFile("socket error!");
    exit(EXIT_FAILURE);
}
int PORT=atoi(portNum);

serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(IP);
serv_addr.sin_port = htons(PORT);

if((bind(socketFd, (struct sockaddr*)&serv_addr, sizeof(serv_addr))) < 0){  // bi
    writeLogFile("bind error");
    exit(EXIT_FAILURE);
}

if((listen(socketFd, x)) < 0){          // listen socket
    writeLogFile("listen error");
    exit(EXIT_FAILURE);
}
signal(SIGINT, sighandler);         // listen SIGCTRL

while(1){                           // main thread manage accepts in infinite
    socklen_t len=sizeof(serv_addr);
```

Figure 13: socket structure

7 Main Thread

Main thread manage server. First accept request. If accept success, check avaible thread from tasks array and if found an thread, assign socket fd send send condition signal for these thread. After send, worker thread count increase and load calculate. If load percentage big or equal percentage 75, resizing thread condition signal send. If worker thread count equal to instant thread count, main thread wait condition signal from threads. Threads also check these equalization. If SIGINT signal have sent, gotSIGINTSig static atamic variable checked. If signal have sent, break and finish listening socket.

```c
    signal(SIGINT, sighandler);         // listen SIGCTRL

    while(1){                            // main thread manage accepts in infinite loop
        socklen_t len=sizeof(serv_addr);

        int accptfd=accept(socketFd,(struct sockaddr*)&serv_addr, &len);    // accept connects
        pthread_mutex_lock(&main_mutex);    // main thread mutex
        int foundFlag=0;
        for(i=0; i<instantThreadCount;i++){ // after connect came, find not bust thread and assign job and send sign
            if(tasksArr[i].status==0){
                tasksArr[i].status=1;
                tasksArr[i].socketFd=accptfd;
                tasksArr[i].full=1;
                pthread_cond_signal(&tasksArr[i].cond); // send signal to thread
                instantWorkerCount++;                   // increase worker size
                char msgToFile[75]=" A connection has been delegated to thread id #";
                loadPer=((double)instantWorkerCount*100.0)/(double)instantThreadCount;      // calculate load percen

                if(loadPer>=75.0) pthread_cond_signal(&cond_resize);    // if load bif or equatl %75, send signal re

                char tempL[30];
                sprintf(tempL,"%ld, system load %.1f%%\n",i,loadPer);
                strcat(msgToFile,tempL);
                writeLogFile(msgToFile);
                i=instantThreadCount;
                foundFlag=1;
            }
        }
        pthread_mutex_unlock(&main_mutex);
        if(foundFlag==0){                    // if not thread avaible, wait until one thread avaible threads send this co
            pthread_mutex_lock(&full_mutex);
            writeLogFile(" No thread is available! Waiting for one.\n");
            while(instantWorkerCount!=instantThreadCount-1){
                pthread_cond_wait(&cond_full,&main_mutex);
            }
        }
        if(gotSIGCHLDSig==1) break;
    }
}
```

Figure 14: Main Thread

8 Thread Pool Function

All thread pool threads use this function. First wait condition signal from main thread when job assing and after that read request with socket fd. After that first call reader function for check database entry. If there is no entry in database, call writer function and send result to client and add results into database. If SIGINT send, exit and return 0.

```c
void* threadFunc(void* id){         // thread pool use this function
    int t_id=(int)id, i=0,bytes=0;
    char info[40];
    sprintf(info," Thread #%d: waiting for connection\n",t_id);
    while(1){
        writeLogFile(info);

        if(gotSIGINTSig==1) {
            break;
        }
        char *st,*des;
        char buff[1025];
        pthread_mutex_lock(&tasksArr[t_id].mutex);

        if(tasksArr[t_id].full==0){     // if thread not has a job, wait cond signal
            while(tasksArr[t_id].full==0)
                pthread_cond_wait(&tasksArr[t_id].cond, &tasksArr[t_id].mutex);
                tasksArr[t_id].status=1;
        }

        if(gotSIGINTSig==0){
            if((bytes=recv(tasksArr[t_id].socketFd, buff, 1025, 0)) < 0){   // if jop came, read request
                writeLogFile("Receive failed!\n");
            }
            buff[bytes]='\0';

            char* token = strtok(buff, "-");
            i=0;
            while(token != NULL){
                 if(i==0) st=token;
                 if(i==1) des=token;
                 i++;
                 token = strtok(NULL, "-");
            }
            long int start=atoi(st);    // convert nodes into long int
            long int end=atoi(des);

            char ans[100];
            sprintf(ans," Thread #%d: searching database for a path from node %ld to node %ld\n",t_id,start,end);
            writeLogFile(ans);

            /// reader start
            long int res=reader(t_id,start,end);    // first read db and send client if found
```

Figure 15: Thread Pool Function 1

```c
            /// reader start
            long int res=reader(t_id,start,end);    // first read db and send client if found
            // writer start
            if(res==-1){                        // -1 means not node found, so thread fill calculate bfs and
                sprintf(ans," Thread #%d: no path in database, calculating %ld->%ld\n",t_id,start,end);
                writeLogFile(ans);
                writer(t_id,start,end);
            }
            close(tasksArr[t_id].socketFd);         // close assigned socket fd
            pthread_mutex_lock(&main_mutex);
            tasksArr[t_id].full=0;
            tasksArr[t_id].status=0;
            instantWorkerCount--;                   // dcreace worker count
            pthread_mutex_unlock(&main_mutex);
            if(instantWorkerCount==instantThreadCount) pthread_cond_signal(&cond_full); // send signal if all th
            pthread_mutex_unlock(&tasksArr[t_id].mutex);
            close(tasksArr[t_id].socketFd);
        }

    }
    return 0;
}
```

Figure 16: Thread Pool Function 2

9 Resizing Thread

These thread, wait resize mutex signal from main thread and if load percentage equal or big percentage 80, do realloc call and get new size from memory and assign this threadPool variable. Use mutex and condition variables for senkronization with main thread. If SIGINT signal have sent, check gotSIGINTSig variable and break to join in signal handler function

```c
void* resizingThreadFunc(void* id){            // wait signal from main thread and increase thread count

    while(instantThreadCount<maxThreadCount){
        pthread_mutex_lock(&resize_mutex);
        pthread_cond_wait(&cond_resize,&resize_mutex);

        pthread_mutex_lock(&main_mutex);

        loadPer=((double)instantWorkerCount*100.0)/(double)instantThreadCount;

        if(loadPer>=75.0){
            int passCount=instantThreadCount;
            instantThreadCount+=(instantThreadCount*25)/100;
            if(instantThreadCount<=maxThreadCount){
                char msg[50];
                sprintf(msg," System load %.1f, pool extended to %d threads\n",loadPer,instantThreadCount);
                writeLogFile(msg);
                if(instantThreadCount>passCount){       // create new threads

                    pthread_t* temp=(pthread_t *)realloc(threadPool,(instantThreadCount)*sizeof(pthread_t));    // extand th

                    threadPool=temp;        // assign new sized thread pool
                    int i=passCount;
                    for(;i<instantThreadCount;i++){
                        if(pthread_create(&threadPool[i], NULL, threadFunc,(void*)i)!=0){   // initiliaze threads
                            writeLogFile("Error in thread creation!\n");
                    }
                }
            }else instantThreadCount=passCount;
        }
        pthread_mutex_unlock(&main_mutex);
        pthread_mutex_unlock(&resize_mutex);

        }
        if(gotSIGINTSig==1) {
            break;
        }
    }
    return 0;
}
```

Figure 17: Resizing Thread function

10 Reader Function

Theread pool thread use this function when client send request. First lock database mutex. After that, check AW(active writer)+WW(Wait writer) count and wait okToRead signal from writer function. After that check database for entry. If entry exist, send client. If not exist, return res (-1 means not exist). If (AR == 0 and WW bigger than 0) send to writer oktowrite signal (priority writer) and unlock database mutex.

```c
long int reader(int t_id,long int start,long int end){
    pthread_mutex_lock(&db_mutex);

    while ((AW + WW) > 0) { // if any writers, wait
        WR++; // waiting reader
        pthread_cond_wait(&okToRead,&db_mutex);
        WR--;
    }
    AR++;
    pthread_mutex_unlock(&db_mutex);
    long int res=getDbEntryIndex(start,end);    // reader first get index fro

    if(res>=0){
        char* path=getPath(res);                // get path if exist
        char* ans=(char*)malloc(sizeof(char)*strlen(path));
        sprintf(ans," Thread #%d: path found in database: %s\n",t_id,path);
        writeLogFile(ans);
        write(tasksArr[t_id].socketFd, path, strlen(path));
    }

    pthread_mutex_lock(&db_mutex);
    AR--;
    if (AR == 0 && WW > 0)
        pthread_cond_signal(&okToWrite);        // prioriy writers
    pthread_mutex_unlock(&db_mutex);
    return res;
}
```

Figure 18: Reader function

11 Writer Function

If path no exist in database, thread function call this function and with use bfs function get result, send client and add path into database. This function first lock database mutex and check AW + AR and wait signal if bigger than 0 this equation. After that, use BFS function and get result. Send result to client and add calculated path into database. Finally check if Wait writer exists and if, send oktowrite signal. If not, send signal to reader thread.

```c
void writer(int t_id,long int start,long int end){  // writer thread first get bfs result,

    pthread_mutex_lock(&db_mutex);

    while ((AW + AR) > 0) { // if any readers or writers, wait
        WW++;               // waiting writer
        pthread_cond_wait(&okToWrite, &db_mutex);
        WW--;
    }
    AW++; // active writer
    pthread_mutex_unlock(&db_mutex);

    // writer start

    char* resultBfs=bfs(graph, start, end);
    char* ans=(char*)malloc(sizeof(char)*strlen(resultBfs));

    sprintf(ans," Thread #%d: path calculated: %s\n",t_id,resultBfs);
    writeLogFile(ans);
    write(tasksArr[t_id].socketFd, resultBfs, strlen(resultBfs));

    sprintf(ans," Thread #%d: responding to client and adding path to database\n",t_id);
    writeLogFile(ans);
    addDataBase(start,end,resultBfs);

    AW--;
    if (WW > 0)     // give priority to other writers
        pthread_cond_signal(&okToWrite);
    else if (WR > 0)
        pthread_cond_broadcast(&okToRead);
    pthread_mutex_unlock(&db_mutex);

}
```

Figure 19: Writer Function

12 Signal Handler

Signal Handler function first assign 1 to atomic variable gotSIGINTSig becauce of other threads understand SIGINT have sent. After that, if threads finished their job and wait condition signal, send threads condition signals and wait all thread pool threads join. Again send signal to resizing thread and wait for join. After all threads join, deallocated all variables and write to log file.

```c
void sighandler(int sig){                                   // threads use this handler for hand S
    int j=0,i=0;
    switch (sig){
        case SIGINT:
            gotSIGINTSig = 1;
            writeLogFile(" Termination signal received, waiting for ongoing threads to complete.\n");
            for(j=0;j<instantThreadCount;j++){
                if(tasksArr[j].full==0) {
                    tasksArr[j].full=1;
                    pthread_cond_signal(&tasksArr[j].cond);
                }
                void * ptr = NULL;
                i=pthread_join(threadPool[j],&ptr);                 // after canceled, wait for terminate
                if(i!=0)
                    writeLogFile(" Error p_thread Join");

            }
            free(threadPool);
            pthread_cond_signal(&cond_resize);
            void * ptr = NULL;
            i=pthread_join(resizingThread,&ptr);                 // after canceled, wait for terminate
            if(i!=0)
                writeLogFile(" Error resizimg thread_join\n");
            free(bfsRes);           // free all variables
            free(visited);
            for(i = 0; i < pathsSize; i++)
                free(paths[i]);
            free(paths);
            free(db->paths);
            free(db);
            for(i=0; i<graph->size;i++){
                if(graph->array[i].head!=NULL){
                    struct node* tmp;
                       while (graph->array[i].head!=NULL){
                           tmp = graph->array[i].head;
                           graph->array[i].head = graph->array[i].head->next;
                           free(tmp);
                       }
                       free(graph->array[i].head);
                }
                free(graph->array[i].head);
            }
```

Figure 20: Signal Handler Function1

```c
void sighandler(int sig){                                                // threads use this handler for hand S
    int j=0,i=0;
    switch (sig){
        case SIGINT:
            gotSIGINTSig = 1;
            writeLogFile(" Termination signal received, waiting for ongoing threads to complete.\n");
            for(j=0;j<instantThreadCount;j++){
                if(tasksArr[j].full==0) {
                    tasksArr[j].full=1;
                    pthread_cond_signal(&tasksArr[j].cond);
                }
                void * ptr = NULL;
                i=pthread_join(threadPool[j],&ptr);                     // after canceled, wait for terminate
                if(i!=0)
                    writeLogFile(" Error p_thread Join");

            }
            free(threadPool);
            pthread_cond_signal(&cond_resize);
            void * ptr = NULL;
            i=pthread_join(resizingThread,&ptr);                        // after canceled, wait for terminate
            if(i!=0)
                writeLogFile(" Error resizimg thread_join\n");
            free(bfsRes);           // free all variables
            free(visited);
            for(i = 0; i < pathsSize; i++)
                free(paths[i]);
            free(paths);
            free(db->paths);
            free(db);
            for(i=0; i<graph->size;i++){
                if(graph->array[i].head!=NULL){
                    struct node* tmp;
                    while (graph->array[i].head!=NULL){
                        tmp = graph->array[i].head;
                        graph->array[i].head = graph->array[i].head->next;
                        free(tmp);
                    }
                    free(graph->array[i].head);
                }
                free(graph->array[i].head);
            }
```

Figure 21: Signal Handler Function2

13 Input Check Server Examples

```
cse312@ubuntu:~/Desktop/homeworks/system/final$ ./server -i /home/cse312/Desktop/homeworks/system/final/p2p-Gnutella08.txt -p 5000 -o pathToLogFile -s 4 -x
Argument count should be 11. Entered count: 10
Program finished
usage: ./server -i pathToFile -p PORT -o pathToLogFile -s 4 -x 24
-pathToFile: Containing a directed unweighted graph from the Stanford Large Network Dataset Collection ( https://snap.stanford.edu/data/ )
-PORT: This is the port number the server will use for incoming connections.
-pathToLogFile: Relative or absolute path of the log file to which the server daemon will write all of its output (normal output & errors)
-s : this is the number of threads in the pool at startup (at least 2)
-x : this is the maximum allowed number of threads, the pool must not grow beyond this number.
cse312@ubuntu:~/Desktop/homeworks/system/final$
```

Figure 22: input check server1

```
cse312@ubuntu:~/Desktop/homeworks/system/final$ ./server -i /home/cse312/Desktop/homeworks/system/final/p2p-Gnutella08.txt -p 5000 -o pathToLogFile -s 4 -x fsdf

 -x fsdf argument not integer. It should be integer.

Program finished
usage: ./server -i pathToFile -p PORT -o pathToLogFile -s 4 -x 24
-pathToFile: Containing a directed unweighted graph from the Stanford Large Network Dataset Collection ( https://snap.stanford.edu/data/ )
-PORT: This is the port number the server will use for incoming connections.
-pathToLogFile: Relative or absolute path of the log file to which the server daemon will write all of its output (normal output & errors)
-s : this is the number of threads in the pool at startup (at least 2)
-x : this is the maximum allowed number of threads, the pool must not grow beyond this number.
cse312@ubuntu:~/Desktop/homeworks/system/final$
```

Figure 23: input check server2

```
        3611 kworker/u16:1
cse312@ubuntu:~/Desktop/homeworks/system/final$ ./server -i /home/cse312/Desktop/homeworks/system/final/p2p-Gnutella08.txt -p 5000 -o pathToLogFile -s 4 -x 24
Server already running...
cse312@ubuntu:~/Desktop/homeworks/system/final$
```

Figure 24: double instance example

14 Client

```
cse312@ubuntu:~/Downloads/151044085_EMRE_KAVAK_FINAL_PROJECT$ ./client -a 127.0.0.1 -p 5000 -s 768 -d
Argument count should be 9. Entered count: 8
usage:
You should enter like this arguments: ./client -a 127.0.0.1 -p PORT -s 768 -d 979
-a: IP address of the machine running the server
-p: port number at which the server waits for connections
-s: source node of the requested path
-d: destination node of the requested path
cse312@ubuntu:~/Downloads/151044085_EMRE_KAVAK_FINAL_PROJECT$
```

Figure 25: client wrong count argument

```
cse312@ubuntu:~/Downloads/151044085_EMRE_KAVAK_FINAL_PROJECT$ ./client -a 127.0.0.1 -p 5000 -s 768a -d 12313

 -s 768a argument not unsigned integer. It should be integer. Program finished
usage:
You should enter like this arguments: ./client -a 127.0.0.1 -p PORT -s 768 -d 979
-a: IP address of the machine running the server
-p: port number at which the server waits for connections
-s: source node of the requested path
-d: destination node of the requested path
cse312@ubuntu:~/Downloads/151044085_EMRE_KAVAK_FINAL_PROJECT$
```

Figure 26: client wrong type argument

Client first check arguments with getopt function. After that, create socket with given ip adress and port number. Connect with server, send requested path and wait answer from client. If answer have sent from server, print answer and exit.

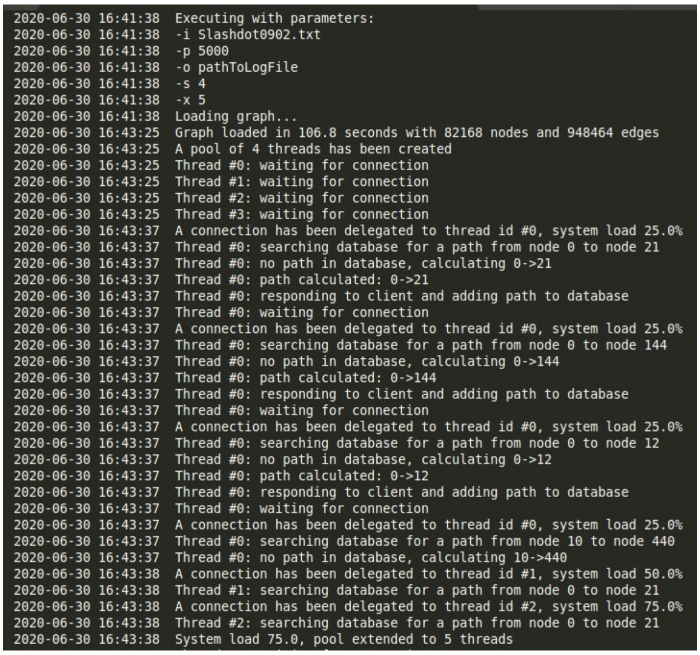
```c
    int sockfd = 0, readedByte = 0;
    char responseBuff[1024];
    struct sockaddr_in serv_addr;

    memset(responseBuff, '0',sizeof(responseBuff));
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Error : Could not create socket \n");
        return 1;
    }
    memset(&serv_addr, '0', sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(port);
    serv_addr.sin_addr.s_addr = inet_addr(ipAdress);
    if(inet_pton(AF_INET, ipAdress, &serv_addr.sin_addr)<=0)
    {
        printf("\n inet_pton error occured\n");
        return 1;
    }
    printf("Client (%d) connecting to %s:%d\n",getpid(),ipAdress,port );
    int connectfd=0;

    if((connectfd=connect(sockfd,(struct sockaddr*)&serv_addr, sizeof(serv_addr))) < 0){
       printf("\n Error : Connect Failed \n");
       return 1;
    }

    struct timeval  tm1, tm2;
    gettimeofday(&tm1, NULL);

    printf("Client (%d) connected and requesting a path from node %s to %s\n",getpid(),s,d);
    strcat(s,"-");
    strcat(s,d);
    write(sockfd, s, strlen(s));

    readedByte = read(sockfd, responseBuff, sizeof(responseBuff)-1);
    responseBuff[readedByte] = 0;
    if(readedByte < 0){
        printf("\n Read error! Program finished. \n");
        exit(EXIT_FAILURE);
    }
    gettimeofday(&tm2, NULL);
    double totalTime=(double) (tm2.tv_usec - tm1.tv_usec) / 1000000 +(double) (tm2.tv_sec - tm1.tv_sec);
    printf("Server's response to (%d): %s, arrived in %.1fseconds.\n\n",getpid(),responseBuff,totalTime);
    close(connectfd);
    return 0;
```

Figure 27: client code

```
Client (3567) connecting to 127.0.0.1:5000
Client (3567) connected and requesting a path from node 10 to 440
Server's response to (3567):  path not possible from node 10 to 440, arrived in
0.0seconds.

Client (3568) connecting to 127.0.0.1:5000
Client (3568) connected and requesting a path from node 768 to 979
Server's response to (3568):  path not possible from node 768 to 979, arrived in
 0.0seconds.

Client (3569) connecting to 127.0.0.1:5000
Client (3569) connected and requesting a path from node 0 to 143
Server's response to (3569): 0->5->127->177->143, arrived in 0.0seconds.

Client (3570) connecting to 127.0.0.1:5000
Client (3570) connected and requesting a path from node 7 to 1689
Server's response to (3570): 7->145->390->391->427->695->2596->1689, arrived in
0.0seconds.

Client (3571) connecting to 127.0.0.1:5000
Client (3571) connected and requesting a path from node 8 to 128
Server's response to (3571): 8->665->5->128, arrived in 0.0seconds.

Client (3572) connecting to 127.0.0.1:5000
Client (3572) connected and requesting a path from node 10 to 4564646
Server's response to (3572):  path not possible from node 10 to 4564646, arrived
 in 0.0seconds.

Client (3573) connecting to 127.0.0.1:5000
Client (3573) connected and requesting a path from node 0 to 252
Server's response to (3573): 0->9->252, arrived in 0.0seconds.

cse312@ubuntu:~/Desktop/homeworks/system/final$ ./test2.sh
```

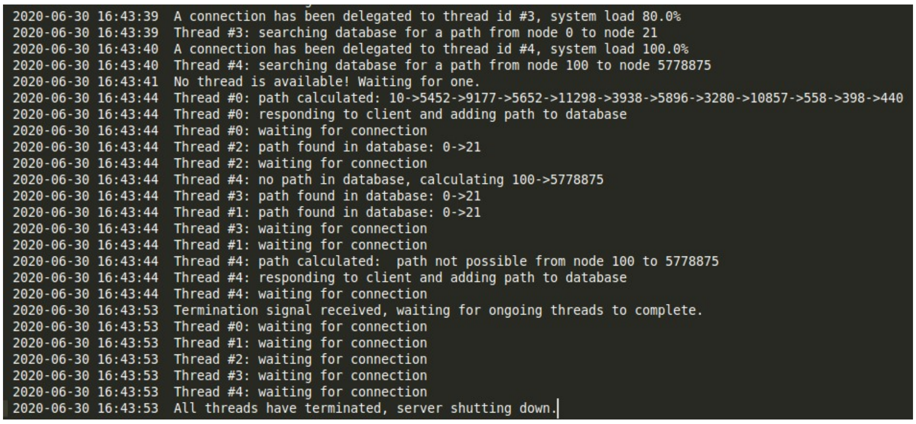
Figure 28: client code output example

15 Log File Example



```
2020-06-30 16:41:38  Executing with parameters:
2020-06-30 16:41:38  -i Slashdot0902.txt
2020-06-30 16:41:38  -p 5000
2020-06-30 16:41:38  -o pathToLogFile
2020-06-30 16:41:38  -s 4
2020-06-30 16:41:38  -x 5
2020-06-30 16:41:38  Loading graph...
2020-06-30 16:43:25  Graph loaded in 106.8 seconds with 82168 nodes and 948464 edges
2020-06-30 16:43:25  A pool of 4 threads has been created
2020-06-30 16:43:25  Thread #0: waiting for connection
2020-06-30 16:43:25  Thread #1: waiting for connection
2020-06-30 16:43:25  Thread #2: waiting for connection
2020-06-30 16:43:25  Thread #3: waiting for connection
2020-06-30 16:43:37  A connection has been delegated to thread id #0, system load 25.0%
2020-06-30 16:43:37  Thread #0: searching database for a path from node 0 to node 21
2020-06-30 16:43:37  Thread #0: no path in database, calculating 0->21
2020-06-30 16:43:37  Thread #0: path calculated: 0->21
2020-06-30 16:43:37  Thread #0: responding to client and adding path to database
2020-06-30 16:43:37  Thread #0: waiting for connection
2020-06-30 16:43:37  A connection has been delegated to thread id #0, system load 25.0%
2020-06-30 16:43:37  Thread #0: searching database for a path from node 0 to node 144
2020-06-30 16:43:37  Thread #0: no path in database, calculating 0->144
2020-06-30 16:43:37  Thread #0: path calculated: 0->144
2020-06-30 16:43:37  Thread #0: responding to client and adding path to database
2020-06-30 16:43:37  Thread #0: waiting for connection
2020-06-30 16:43:37  A connection has been delegated to thread id #0, system load 25.0%
2020-06-30 16:43:37  Thread #0: searching database for a path from node 0 to node 12
2020-06-30 16:43:37  Thread #0: no path in database, calculating 0->12
2020-06-30 16:43:37  Thread #0: path calculated: 0->12
2020-06-30 16:43:37  Thread #0: responding to client and adding path to database
2020-06-30 16:43:37  Thread #0: waiting for connection
2020-06-30 16:43:37  A connection has been delegated to thread id #0, system load 25.0%
2020-06-30 16:43:37  Thread #0: searching database for a path from node 10 to node 440
2020-06-30 16:43:37  Thread #0: no path in database, calculating 10->440
2020-06-30 16:43:38  A connection has been delegated to thread id #1, system load 50.0%
2020-06-30 16:43:38  Thread #1: searching database for a path from node 0 to node 21
2020-06-30 16:43:38  A connection has been delegated to thread id #2, system load 75.0%
2020-06-30 16:43:38  Thread #2: searching database for a path from node 0 to node 21
2020-06-30 16:43:38  System load 75.0, pool extended to 5 threads
```

Figure 29: Log File Example example 1



```
2020-06-30 16:43:39  A connection has been delegated to thread id #3, system load 80.0%
2020-06-30 16:43:39  Thread #3: searching database for a path from node 0 to node 21
2020-06-30 16:43:40  A connection has been delegated to thread id #4, system load 100.0%
2020-06-30 16:43:40  Thread #4: searching database for a path from node 100 to node 5778875
2020-06-30 16:43:41  No thread is available! Waiting for one.
2020-06-30 16:43:44  Thread #0: path calculated: 10->5452->9177->5652->11298->3938->5896->3280->10857->558->398->440
2020-06-30 16:43:44  Thread #0: responding to client and adding path to database
2020-06-30 16:43:44  Thread #0: waiting for connection
2020-06-30 16:43:44  Thread #2: path found in database: 0->21
2020-06-30 16:43:44  Thread #2: waiting for connection
2020-06-30 16:43:44  Thread #4: no path in database, calculating 100->5778875
2020-06-30 16:43:44  Thread #3: path found in database: 0->21
2020-06-30 16:43:44  Thread #1: path found in database: 0->21
2020-06-30 16:43:44  Thread #3: waiting for connection
2020-06-30 16:43:44  Thread #1: waiting for connection
2020-06-30 16:43:44  Thread #4: path calculated:  path not possible from node 100 to 5778875
2020-06-30 16:43:44  Thread #4: responding to client and adding path to database
2020-06-30 16:43:44  Thread #4: waiting for connection
2020-06-30 16:43:53  Termination signal received, waiting for ongoing threads to complete.
2020-06-30 16:43:53  Thread #0: waiting for connection
2020-06-30 16:43:53  Thread #1: waiting for connection
2020-06-30 16:43:53  Thread #2: waiting for connection
2020-06-30 16:43:53  Thread #3: waiting for connection
2020-06-30 16:43:53  Thread #4: waiting for connection
2020-06-30 16:43:53  All threads have terminated, server shutting down.
```

Figure 30: Log File Example example2