Gebze Technical University
Computer Engineering

System Programming

2021 Spring

FINAL PROJECT REPORT

Dilara KARAKAŞ

171044010

1. Introduction

1.1. Main Design

Problem consists of 3 main part. Servant, client, and server. Each of the part communicates with each other using sockets.

1.1.1.Servant

Main design of the program contains multiple servant programs. Since servant and server uses the socket for communication, each servant process finds a unique port number for communicating. Each of the servant process covers different city ranges and this ranges provided each servant using command line argument. With this unique range values, each servant finds an appropriate port for itself.

```
connectPort=33500+portIndex;
resetStructAndIncreasePort(addr,&connectPort);
/*Bind created socket.*/
portTry=bind(conn_fd, (struct sockaddr*) addr, sizeof((*addr)));
while(portTry == EADDRINUSE){
    resetStructAndIncreasePort(addr,&connectPort);
    portTry=bind(conn_fd, (struct sockaddr*) addr, sizeof((*addr)));
}
```

Figure 1. socketManager.c file / findAvailablePortForServantSocketAndListen function

In this code block, servant sums higher port number with its lower city index. In this way it reaches unique port number from other servants. After this process, each servant looks for appropriate port for binding.

When servant find proper port, first thing it does is send port to the server so server can send request to servant anytime. For this servant uses the port number provided from the command line argument.

The other duty of servant processes is storing the dataset. For this each of the servant has a range of data. Each range represents index of alphabetical order of city files. For obtaining names of folders that dataset contains, servant programs uses readdir method and stores all folder names under one data structure.

```c
char** getDirectoryList(char* path,int *dirListSize){
    int maxDirCapacity=256;
    int currentDirCount=0,count=0;
    char **dirList = (char **)malloc(sizeof(char *)*maxDirCapacity);
    DIR *dir;
    struct dirent *directory;
    dir = opendir(path);
    if (dir){
        while ((directory = readdir(dir)) != NULL){
            if(!(strcmp(directory->d_name,".")==EQUAL || strcmp(directory->d_name,"..")==EQUAL)){
                addDirectory(&dirList,&maxDirCapacity,&currentDirCount,directory->d_name);
            }
        }
        closedir(dir);
    }
    (*dirListSize)=currentDirCount;
    return dirList;
}
```

Figure 2. directoryManager.c

After this process servant sorts all the filenames with alphabetical order and takes the subset of folder names which is in between provided upper and lower parameters.

```c
char ** filterDatasetWithGivenRange(int upperLimit,int lowerLimit,int *currentDirCount,char ***dirList){
    char **mainDirList = (char **)malloc(sizeof(char *)*(upperLimit-lowerLimit+1));
    int index=0;
    sortStringListAlphabetically((*currentDirCount),(*dirList));


    /*Filter dataset with given range*/
    for(int i=0;i<(*currentDirCount);i++){
        if((i+1)>=lowerLimit && (i+1)<=upperLimit){
            mainDirList[index++]=(*dirList)[i];
        }else{
            free((*dirList)[i]);
        }
    }
    free((*dirList));
    (*currentDirCount)=index;
    return mainDirList;
}
```

Figure 3. directoryManager.c

In this way servant processes obtains their responsible cities. With obtaining available port and responsible folder names, servant has all the necessary informations for informing server.

Before informing the server, servant reads all the files and stores in a special data structure.

The most basic node value in this dataset is estate component.

```c
struct estate{
    int transactionId;
    char *city;
    char *streetName;
    int surface;
    int price;
    struct estate *next;
};
```

Figure 4. commonStructs.h

This struct contains most basic information about estate values. Since estates holded inside a linked list structure it has a next node.

```c
struct estateLL{
    struct estate *front;
    struct estate *rear;
    int size;
};
```

Figure 5. commonStructs.h

In the request which come from client it uses 3 thing as a parameter. First is type other is date range and last and optional one is city. Since city is an optional variable, we store this value in estate struct.

In order to provide easy indexing for servants, program creates another linked list with date indexed and stores estateLL variables.

```c
struct dateEstate{
    struct date d;
    struct estateLL *estates;
    struct dateEstate *next;
    struct dateEstate *back;
};

struct dateEstateLL{
    struct dateEstate *front;
    struct dateEstate *rear;
    int size;
};
```

Figure 6. commonStructs.h

Servant always stores this linkedlist in a sorted order so that it can take the subset of date range values. In this way, servant can easily search the dataset using this date indexed linkedlist data structure. Also since one of the main parameter is Type value, program uses another linked list for storing type indexed values.

```c
struct typeEstate{
    char* type;
    struct dateEstateLL *dateEstates;
    struct typeEstate *next;
};

struct typeEstateLL{
    struct typeEstate *front;
    struct typeEstate *rear;
    int size;
};
```

Figure 7. commonStructs.h

So the data structure uses the following hierarchy;

Type Indexed Linkedlist

Date Indexed Linkedlist

Estate informations.

With this data structure if city parameter does not provided, it directly reaches the related subset in the servant. The downside of this design is if city parameter is provided it must search all the subsets for all servants.

After obtaining all necessary informations the servant processess communicates with server and tells founded available port, city list that it responsible for and ip. After that it directly connects to the port that it founded using lower city index, and waits for servers request. When server sends request it parses the request, search for data structure and sends it back to server process.

```c
if(error=connect(serverfd,(struct sockaddr*)(&addr),sizeof(addr))==ERROR){
    close(serverfd);
    sprintf(line,"connect error:%s,%d!\n",strerror(errno),errno);
    print_error(line);
    return ERROR;
}

sprintf(line,"s %d %d %d %s ",servantPort,lowerRange,upperRange,ip);

for(i=0;i<manDirListSize;i++){
    strcat(line,manDirList[i]);
    strcat(line," ");
}
error=write(serverfd,line,strlen(line));
if(error==ERROR){
    sprintf(line,"Error while write operation.!");
    print_error(line);
}
```

Figure 7. socketManager.c

1.1.2.Server

The server process uses a thread pool for handling the requests. When the request come to the server (from client or from servant) process it directly send it to thread pool and waits for new request. When server first initialized it creates the thread pool and opens the socket connection using command line arguments.

```
/*create thread pool*/
threads=(pthread_t**)malloc(sizeof(pthread_t*)*nThreads);
for(i=0;i<nThreads;i++){
    threads[i]=(pthread_t*)malloc(sizeof(pthread_t)*1);
    error=pthread_create(threads[i],NULL,requestHandlerThread,NULL);
    if(error!=0){
        sprintf(line,"Thread couldn't create!.\n");
        printError(line);
        exit(EXIT_FAILURE);
    }
}
```

Figure 8. server.c , server creates thread pool

For providing request to thread pool, program uses a queue structure. It uses a struct to store in the queue which has file descriptor for client. For notifying one of the threads in the pool server uses a conditional variable. When new request comes it just signals to the variable.

When new request arrive program checks if it is a client request or dervant request. Client requests send their request with “c…..” format and servants sends as “s……..” format. So only looking for first letter of request is enough to eliminates requests.

```
if(buffer=='s'){ /*servant information*/
    servants[currentServantCount] = readServant(&(r->fd));
    sprintf(line,"Servant x present at port %d handling cities %s-%s\n",
        servants[currentServantCount]->port,
        servants[currentServantCount]->cityNames[0],
        servants[currentServantCount]->cityNames[servants[currentServantCount]->cityNamesSize-1]);
    printTerminal(line);

    currentServantCount++;
    if(currentServantCount >= servantCacheSize){
        increaseServantCacheSize(&servantCacheSize,&servants);
    }
}
if(buffer=='c'){ /*client request*/
    handleClientRequest(&(r->fd));
}
```

Figure 9. server.c , eliminating requests

After eliminated the income requests, if request is coming from servant it stores necessary informations in a data structe for future client requests and print log to the screen.

If request is a client request, then it parses the request and sends it to directly servant using the information that servant is sended in the start of the program.

```
thrLine[0]='\0';
sprintf(thrLine,"Request arrived \"transactionCount %s %d-%d-%d %d-%d-%d %s\" \n",
    r->type,r->d1.day,r->d1.month,r->d1.year,r->d2.day,r->d2.month,r->d2.year,r->city);
printTerminal(thrLine);

res=getRequestResultFromServant(r->type,r->d1,r->d2,r->city,servants,currentServantCount);

thrLine[0]='\0';
sprintf(thrLine,"Response received: %d, forwarded to client...\n",res);
printTerminal(thrLine);

thrLine[0]='\0';
sprintf(thrLine,"%d*",res);

error=write((*conn_fd),thrLine,strlen(thrLine));
if(error==ERROR){
    sprintf(thrLine,"Error while write operation.!");
    printError(thrLine);
}
```

Figure 10. server.c , requests result from servants and send result to client

```
struct servantStruct{
    int port;
    char *ip;
    int lowerRange;
    int upperRange;
    char **cityNames;
    int cityNamesSize;
    int cityNamesCapacity;
    int servantId;
}servant;
```

Figure 11. the struct that stores servant information's

1.1.3.Client

In the client part, program first creates the thread for sending requests. The client reads requests from a request file. It counts how many requests there are during this read process.

Then, it creates threads as many as requests and sends each request to a thread. However, there is a situation like this: Every thread requested from us must send requests to the server at the same time. So every created thread should wait for each other. Even when the number of requests is created, each one continues from where it left off (removes it when making a request to the server).

```
error=pthread_mutex_lock(&thread_m);
if(error!=0){
    sprintf(threadLine,"Failed while locking mutex.\n");
    printError(threadLine);
    exit(EXIT_FAILURE);
}
++arrived;
if(arrived < threads->count){
    error=pthread_cond_wait(&thCondVar,&thread_m);
    if(error!=0){
        sprintf (threadLine, "Failed while waiting conditional variable.\n");
        printError(threadLine);
        exit(EXIT_FAILURE);
    }
}else{
    error=pthread_cond_broadcast(&thCondVar);
    if(error!=0){
        sprintf(threadLine, "Failed while broadcast conditional variable.\n");
        printError(threadLine);
        exit(EXIT_FAILURE);
    }
}

error=pthread_mutex_unlock(&thread_m);
if(error!=0){
    sprintf(threadLine,"Failed while unlocking mutex.\n");
    printError(threadLine);
    exit(EXIT_FAILURE);
}
```

Figure 12. synchronization barrier problem

How did I solve this problem? I created an incoming arrived variable. I increased it by one each time the thread came. At the same time, I created a mutex and a critical section so that there is no interference with the variable. I checked whether the number of incoming threads with the condition variable is equal to the number of threads created. If my arrived variable is less than the number of threads that should be created, threads waited. When the number was equal, I sent a signal to the waiting threads and each thread sent its request to the server.

2. Test Result