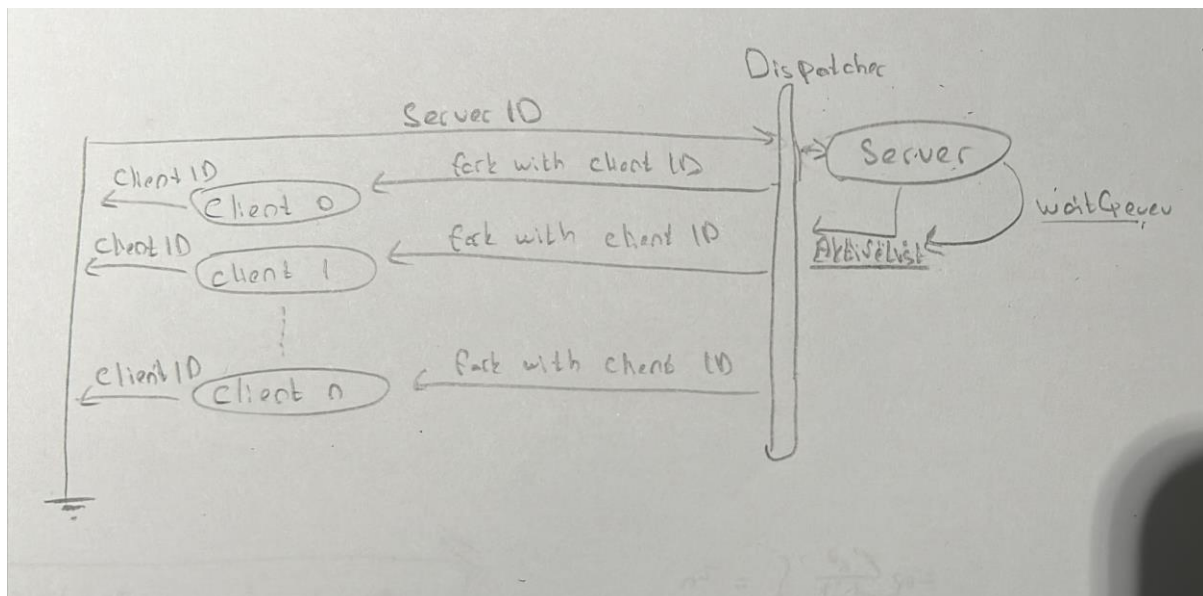


CSE 344 MIDTERM

Project description

This project aims to develop server-client relationships with using fork to serve a lot of clients and sheared memory to communicate clients.

Project Design



Firstly I design server side and client side with connect each other via Server PID. When I create a server in a terminal I take server PID and create a client with that. Server clients write own client PID to Fifo that named server PID. After that if server accept the client, server create a fork. This fork serve to specific client through client Fifo named client PID.

Therefore server add user activeUsers list. Of course this list has a limit that giving by user when server created. If activeUsers list is full, server add the client to waitQueue lists. (I use list structure instead of queue structure but its work same as queue.) I use list because client who stayed in the middle of queue can be kill itself. If it is does, we cannot erase from queue. So logic is queue but structure is list.

My implementation

There are some screen shot for server and clients for their communication.

Client.h; this header file show my design basically. So I share this screenshot.

```
Main > C client.h > Client > connectId
1  #ifndef COMMUNICATION
2  #define COMMUNICATION
3
4
5  #define bufferSize 1024
6
7
8  // Define request struct type
9  struct Request{
10     char requestType[bufferSize]; // request type is Connect or tryConnect
11     char content[bufferSize]; // request content is pid of the client
12 };
13
14 struct Response{
15     char content[bufferSize];
16 };
17
18 typedef struct Client{
19     int clientPid;
20     int clientFd;
21     int connectId; // 0 not connected, otherwise connected
22     struct Request request;
23     struct Response response;
24 }Client;
25
26 #endif
```

activeUser and waitQueue structure; User can be inside of the activeUser list or waitQueue. If conditions are available to adding new client to server, it add activeUser list. Otherwise client should be wait in waitQueue. There are headers of the functions to demonstrate struct of the design.

```
18
19 void initClientLists(); // Initialize the active clients and waiting queue
20 void addClientToList(Client **list, int *count, int *capacity, Client newClient); // Add a client to the list
21 void addClientToActive(Client newClient); // Add a client to the active clients list
22 void addClientToWaiting(Client newClient); // Add a client to the waiting queue
23 int removeClientFromList(Client *list, int *count, int pid); // Remove a client from the list
24 void activateClientFromQueue(); // Activate a client from the waiting queue
25
26 void manageClient(char* clientReturnFifo); // Manage the client
27 char* getLine(FILE *file, int lineNumber); // Get a specific line from a file
28
```

Here are a lot user connect same server simultaneously;

```

ahmet@ahmet-Lenovo-V15-IIL: ~/Desktop/200104004127_Mi...
ahmet@ahmet-Lenovo-V15-IIL:~/Desktop/200104004127_Midtern/Main$ ./clientExe Conn
ect 49301
>> Waiting for Que...Connection established:
>> Enter comment:

ahmet@ahmet-Lenovo-V15-IIL:~/Desktop/200104004127_Midtern/Main$ ./clientExe Conn
ect 49301
>> Waiting for Que...Connection established:
>> Enter comment:

ahmet@ahmet-Lenovo-V15-IIL:~/Desktop/200104004127_Mi...
ahmet@ahmet-Lenovo-V15-IIL:~/Desktop/200104004127_Midtern/Main$ ./serverExe her
2
>> Server Started PID 49301
>> Waiting for clients...
>> Client PID 49340 connected as "Client 1 "
>> Client PID 49349 connected as "Client 2 "
>> Client PID 49435 connected as "Client 3 "

```

This server side main loop; read server fifo and if detected clients try to add active users otherwise add wait queue.

```

138 while(1){
139     // open fifo for reading connection
140     int serverResponseFd = open(mainFifo, O_RDONLY);
141     // Read client PID
142     int clientPid;
143     if (read(serverResponseFd, &clientPid, sizeof(int)) < 0) {
144         perror("read");
145         close(serverResponseFd);
146         exit(EXIT_FAILURE);
147     } else foundClient = 1;
148     if(foundClient == 1 && !(activeCount >= activeCapacity)){
149         foundClient = 0;
150         // Fork a new process to handle client
151         int pid = fork();
152         if (pid < 0) {
153             perror("fork error \n");
154             exit(EXIT_FAILURE);
155         } else if (pid == 0) { // Child process
156             ++activeCount; // when created a new fork for a new client increase the client counter
157             char newClientPid[256]; // string of client pid
158             sprintf(newClientPid, "%d", clientPid); // Convert client PID to string
159             manageClient(newClientPid); // Open FIFO for writing with the spesific client
160             exit(EXIT_SUCCESS);
161         }
162     } else if(foundClient == 1 && activeCount >= activeCapacity){
163         // printf serverside to full clients
164         printf("Connection request PID %d... Que FULL\n", clientPid);
165         // add wait queue the client
166         Client newClient;
167         newClient.clientPid = clientPid;
168         // add wait queue the client
169         addClientToWaiting(newClient);
170     } else if(foundClient == 0 && (activeCount < activeCapacity) && waitingCount > 0){
171         // printf serverside to full clients
172         Client newClient = getLastElement(waitingQueue);
173         int pid = fork();
174         if (pid < 0) {
175             perror("fork error \n");
176             exit(EXIT_FAILURE);
177         } else if (pid == 0) { // Child process
178             ++activeCount; // when created a new fork for a new client increase the client counter
179             char newClientPid[256]; // string of client pid
180             sprintf(newClientPid, "%d", newClient.clientPid); // Convert client PID to string
181             manageClient(newClientPid); // Open FIFO for writing with the spesific client
182             exit(EXIT_SUCCESS);
183         }
184     }
}

```

This client main loop; after connecting server it wait accepted response. When client take this response it has 2 operation these sending request and taking response.

Client doesn't know anything about management of server side. It is just use server properties and server's allows.

```
122 char request[1024], response[1024];
123 while(1){
124     // Read request from user
125     memset(request, 0, sizeof(request)); // clear the tempMssg
126     printf(">> Enter comment: ");
127     fgets(request, sizeof(request), stdin);
128
129     requestFd = open(clientFifoName, O_WRONLY); // open client fifo for request from server
130     if(requestFd < 0){
131         perror("client open fifo failed");
132         return -1;
133     }
134     // Send request to server
135     if (write(requestFd, &request, sizeof(request)) == -1){
136         perror("Server requesting error.");
137         return -1;
138     }
139     close(requestFd);
140
141     // Read response from server
142     responseFd = open(clientFifoName, O_RDONLY); // open client fifo for reading response from server
143     if(responseFd < 0){
144         perror("client open fifo failed");
145         return -1;
146     }
147     // Read response from server for ensure connection is okay
148     while (read(responseFd, &response, sizeof(response)) > 0){
149         printf("%s\n", client.response.content);
150         memset(client.response.content, 0, sizeof(client.response.content)); // clear the tempMssg
151     }
152     close(responseFd);
153 }
154 unlink(clientFifoName);
```

Semaphores and shared memory; We use a lot of clients in the server. Therefore they are use server properties commonly. In addition for their management server fork should know their states(active or not). Therefore we use same memory places for some variables. (all process has own memory address spaces)

Of course, this situation cause a race condition. For solving them we combine shared memory with semaphores. I used activeUserCounter in server side. When user in there I increase this value, otherwise decrease. For doing that I create semaphore and below some screenshot to show that.

```
85
86
87 // shared memory Design with semaphoers////////
88 int shm_fd = shm_open(counterSharedMemory, O_CREAT | O_RDWR, 0666); // define shared memory file descriptor
89 if (shm_fd == -1) {
90     perror("shm_open");
91     return EXIT_FAILURE;
92 }
93 if (ftruncate(shm_fd, sizeof(int)) == -1) { // set size of shared memory
94     perror("ftruncate");
95     return EXIT_FAILURE;
96 }
97 sharedSequentialClientCounter = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0); // map shared memory
98 if (sharedSequentialClientCounter == MAP_FAILED) {
99     perror("mmap");
100     return EXIT_FAILURE;
101 }
102 *sharedSequentialClientCounter = 0; // initialize shared memory
103 // create semaphore
104 sem = sem_open(semaphoreName, O_CREAT, 0666, 1);
105 if (sem == SEM_FAILED) {
106     perror("sem_open");
107     return EXIT_FAILURE;
108 }
109 // shared memory Design with semaphoers////////
110
```

```

40
41 ////////////////////////////////////////////////// shared memory Design with semaphoers Constants/////////////////////////////////
42 int *sharedSequentialClientCounter; // shared memory for sequentialClientCounter
43
44 const char *semaphoreName = "mysemaphore"; // semaphore name
45 sem_t *sem; // define semaphore
46
47 const char *counterSharedMemory = "/clientCounter"; // shared memory name
48 ////////////////////////////////////////////////// shared memory Design with semaphoers Constants/////////////////////////////////
49
50 void handleInterruptSignal(int sig) {
51     printf("SIGTSTP received, unlinking FIFO...\n");
52
53     // Unlink the FIFO
54     if (unlink(mainFifo) == -1) { // if already unlinked, it will return -1
55         perror("Failed to unlink FIFO");
56     }
57     // kill clients and child process
58     for (int i = 0; i < activeCount; i++) {
59         kill(activeClients[i].clientPid, SIGKILL);
60     }
61     // release shared memory
62     munmap(sharedSequentialClientCounter, sizeof(int));
63     shm_unlink(counterSharedMemory);
64
65     // release semaphores
66     sem_close(sem);
67     sem_unlink(semaphoreName);
68
69     kill(getpid(), SIGKILL); // kill the server process
70 }
71

```

Here are definition of semaphores **with signal interrupt**; Signal interrupts have critical role because when client forcibly quit terminal server should know that.

When server send a kill request to **kill** server, server should kill whole child process (clients), otherwise they are hang. Also this is cause optimization problems. Client signal shown below, too.

```

15 void handleInterruptSignal(int sig) {
16     printf("SIGTSTP received, unlinking FIFO...\n");
17
18     // send quit signal to server for killing the client serverside
19     int requestFd = open(clientFifoName, O_WRONLY); // open client fifo for request from server
20     if(requestFd < 0){
21         perror("In signal client open fifo failed");
22         return;
23     }
24     // Send request to server
25     if (write(requestFd, "quit", sizeof("quit")) == -1){
26         perror("Server requesting error.");
27         return;
28     }
29     close(requestFd);
30
31     // Unlink the FIFO
32     if (unlink(clientFifoName) == -1) { // if already unlinked, it will return -1
33         perror("Failed to unlink FIFO");
34     }
35     printf("killed cleint");
36     kill(getpid(), SIGKILL); // kill the client process
37     exit(EXIT_SUCCESS);
38 }
39

```

Some utilities

Server have a lot of utilities like list directories, help about properties, upload/download, quit and kill server etc.

Here are example of **help** and **list** request code block and their terminal results;

Help Codeblock;

```

243
244 // - help; display the list of possible client request
245 if (strcmp(newReq[0], "help") == 0) {
246     // Help request
247     if (newReq[1] == NULL) {
248         snprintf(tempMssg, sizeof(tempMssg), "Available comments are : \nhelp, list, readF, writeT, upload, download, archServer, quit, killServer");
249         printf("burasi temp mesaj %s\n", tempMssg);
250     } else if (strcmp(newReq[1], "readF") == 0) {
251         printf("anans");
252         snprintf(tempMssg, sizeof(tempMssg), "readF <file> <line # \n> requests to display the # line of the <file>, if no line number is given \n the whole contents of the file is requested (and displayed on the client side)");
253     } else if (strcmp(newReq[1], "writeT") == 0) {
254         strcpy(tempMssg, "writeT <file> <line #> <string> \n : request to write the content of \"string\" to the #th line the <file>, if the line # is not given \n writes to the end of file. If the file does not exists in Servers directory creates and edits the \n file at the same time");
255     } else if (strcmp(newReq[1], "upload") == 0) {
256         strcpy(tempMssg, "upload <file> \n uploads the file from the current working directory of client to the Servers directory \n (beware of the cases no file in clients current working directory and file with the same \n name on Servers side)");
257     } else if (strcmp(newReq[1], "download") == 0) {
258         strcpy(tempMssg, "download <file> \n request to receive <file> from Servers directory to client side");
259     } else if (strcmp(newReq[1], "archServer") == 0) {
260         strcpy(tempMssg, "archServer <fileName>.tar \n Using fork, exec and tar utilities create a child process that will collect all the files currently \n available on the the Server side and store them in the <fileName>.tar archive");
261     } else if (newReq[1] == "killServer") {
262         strcpy(tempMssg, "killServer \n Sends a kill request to the Server");
263     } else if (newReq[1] == "quit") {
264         strcpy(tempMssg, "quit \n Send write request to Server side log file and quits");
265     }
266
267     int clientFd = open(clientReturnFifo, O_WRONLY);
268     if (clientFd < 0) {
269         perror("server open fifo failed");
270         exit(EXIT_FAILURE);
271     }
272     if (write(clientFd, tempMssg, sizeof(tempMssg)) < 0) {
273         perror("server write failed");
274         exit(EXIT_FAILURE);
275     }
276     close(clientFd);

```

List Codeblock;

```

276     close(clientFd);
277 } else if (strcmp(newReq[0], "list") == 0) {
278     // list all server directory elemtns
279     DIR *dir;
280     struct dirent *ent;
281     int clientFd = open(clientReturnFifo, O_WRONLY, 0666);
282     if (clientFd < 0) {
283         perror("server open fifo failed");
284         exit(EXIT_FAILURE);
285     }
286     // Open the current directory
287     if ((dir = opendir("./server")) != NULL) {
288
289         // Iterate over entries in the directory
290         while ((ent = readdir(dir)) != NULL) {
291
292             //snprintf(tempMssg, sizeof(tempMssg), "%s\n", ent->d_name); // Send serponse to client
293             if (strcmp(ent->d_name, ".") == 0 || strcmp(ent->d_name, "..") == 0) {
294                 continue;
295             }
296             snprintf(tempMssg, sizeof(tempMssg), "%s", ent->d_name);
297             if (write(clientFd, tempMssg, sizeof(tempMssg)) < 0) {
298                 perror("server write failed");
299                 close(clientFd);
300                 closedir(dir);
301                 exit(EXIT_FAILURE);
302             }
303             memset(tempMssg, 0, sizeof(tempMssg)); // clear the tempMssg
304         }
305     } else {
306         perror("");
307         return;
308     }
309     close(clientFd);
310     closedir(dir);
311 } else if (strcmp(newReq[0], "readF") == 0) {

```

Their Terminal Results;

```

ahmet@ahmet-Lenovo-V15-IIL: ~/Desktop/200104004127_ML...
ahmet@ahmet-Lenovo-V15-IIL: ~/Desktop/200104004127_Mldterm/Main$ ./serverExe ee 1
>> Server Started PID 50089
>> Waiting for clients...
>> Client PID 50125 connected as "Client 1 "

ahmet@ahmet-Lenovo-V15-IIL: ~/Desktop/200104004127_Mldterm/Main$ ./clientExe 50089
client wrong number of arguments: Success
ahmet@ahmet-Lenovo-V15-IIL: ~/Desktop/200104004127_Mldterm/Main$ ./clientExe Conn
ect 50089
>> Waiting for Que.. Connection established:
Available comments are :
help, list, readF, writeT, upload, download, archServer, quit, killServer
>> Enter comment: help
server.c
temp.txt
>> Enter comment:

```

Release, free and kill; These process save us to leak memory

```

179
180
181 // release shared memory
182 munmap(sharedSequentialClientCounter, sizeof(int));
183 shm_unlink(counterSharedMemory);
184
185 // release semaphores
186 sem_close(sem);
187 sem_unlink(semaphoreName);
188
189 // Close the FIFO
190 close(serverResponseFd);
191 unlink(mainFifo);
192 // Free the allocated memory
193 free(activeClients);
194 free(waitingQueue);
195
196 kill(getpid(), SIGKILL); // kill the server process
197 return 0;
198 }
199

```

For managing whole operation in a same root directory I create **makefile**.

```

Main > M makefile
1 All: compile run clean
2
3 compile: server/server.c client/client.c
4   @gcc server/server.c -o serverExe
5   @gcc client/client.c -o clientExe
6
7 run:
8   ./serverExe neHosServer 1 & ./clientExe Connect 10
9
10 clean:
11   @rm -f *.txt exe

```