



INTER-PROCESS COMMUNICATION

OPERATING SYSTEM



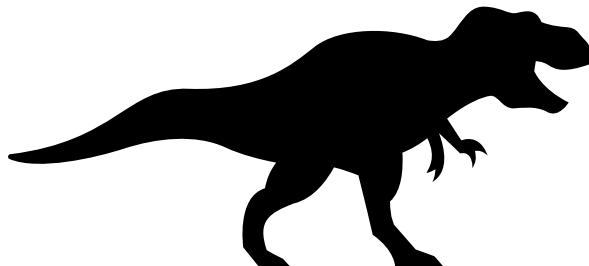
COLLABORATORS NAME & ID :
MARYAM ALIKARAMI 9731045
MAHAN AHMADVAND 9731071



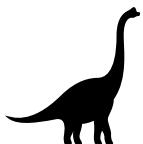


INTER-PROCESS COMMUNICATION

OPERATING SYSTEM



COLLABORATORS NAME & ID :
MARYAM ALIKARAMI 9731045
MAHAN AHMADVAND 9731071



OPERATING SYSTEM LABORATORY

INTER-PROCESS COMMUNICATION



INSTRUCTOR :
ARMAN GHEISARI

COLLABORATORS :
MARYAM ALIKARAMI
MAHAN AHMADVAND

QUESTION 1

محیطی آماده کنید که دو فرآیند در آن وجود داشته باشند و از روش حافظه مشترک برای ارتباط استفاده کنند.(برای مثال می توانید دو فرآیند در نظر بگیرید که کی مقداری بنویسد و دیگر از آن بخواند)
کد مربوط به writer :

```
1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/ipc.h>
4 #include <sys/shm.h>
5 #include <sys/stat.h>
6
7 int main (int argc, char *argv[])
8 {
9     int segment_id;
10    key_t key;
11    key = 3232;
12    char* shared_memory;
13    int shared_segment_size = 100;
14
15    struct shmid_ds shmbuffer;
16    int segment_size;
17
18
19    segment_id = shmget(key, shared_segment_size, IPC_CREAT | 0666);
20
21    shared_memory = shmat (segment_id, NULL, 0);
22    printf("shared memory attached at address %p\n", shared_memory);
23
24    shmctl(segment_id, IPC_STAT, &shmbuffer);
25    segment_size = shmbuffer.shm_segsz;
26    printf("segment size: %d\n", segment_size);
27
28    sprintf(shared_memory, "%s", argv[1]);
29    printf("This string inserted in shared memory: %s\n\n", argv[1]);
30
31    printf("Waiting for reader...\n");
32    while (*shared_memory != '*')
33        sleep(1);
34
35    shmdt(shared_memory);
36
37    return 0;
38 }
39
40 }
```

ک مریبوط پے : reader

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
    int segment_id;
    key_t key;
    char * shared_memory;
    int shared_segment_size = 100;

    key = 3232;

    segment_id = shmget(key, shared_segment_size, IPC_CREAT | 0666);

    shared_memory = shmat(segment_id, NULL, 0);

    printf("%s\n", shared_memory);

    *shared_memory = '*';

    exit(0);
}

exit(0):
    *shared_memory = '*';
    btrnft("n/segs", "shared-memory");
    shmctl(segment_id, SHM_UNRM, 0);
```

ابتدا می خواهیم توضیحاتی در مورد نحوه ای عملکرد کد های بالا بدهیم سپس خروجی موردنظر را مشاهده کنیم.

در این قسمت از ساختار **shared memory** برای ارتباط بین **process** ها استفاده کرده ایم.

برای ایجاد حافظه ای مشترک برای پردازه ها از فراخوانی سیستمی `shmget()` استفاده کرده ایم، پارامتر اول شماره ای **segment** ای را که می خواهیم اختصاص بدهیم را مشخص می کند، که در اینجا متغیر **key** که مقدار 3232 را برای آن انتخاب کرده ایم به آن اختصاص داده ایم، متغیر بعدی سایز **segment** ای که می خواهیم به عنوان حافظه ای اشتراکی اختصاص دهیم را مشخص می کند که در اینجا مقدار 100 را به آن اختصاص داده ایم یعنی 100 تا **page** و سپس باید مود دسترسی به این حافظه ای اشتراکی را تعیین کنیم، که در اینجا **IPC_CREAT|0666** را به عنوان ورودی داده ایم، در واقع 0666 برای تعیین دسترسی ها استفاده می شود و **IPC_CREAT** برای این است که به سیستم بگوییم که یک **memory segment** برای حافظه ای اشتراکی ایجاد کند.

خروجی این تابع یک عدد **integer** است.

حال باید این حافظه ای مشترک را به فضای آدرس برنامه را **attach** کنیم، این کار را با استفاده از تابع `(shmat()` انجام می دهیم، که سه تا پارامتر می گیرد، پارامتر اول خروجی `shmget()` است، سپس آدرسی که می خواهیم مپ کنیم را مشخص می کنیم که ورودی **Null** داده ایم که این به این معناست که خود **linux** یک حافظه به ما تخصیص می دهد و پارامتر آخر را نیز صفر داده ایم که این پارامتر تعیین می کند

که این حافظه ای که attach کرده ایم **read only** باشد یا

که مقدار صفر هیچکدام از پارامتر ها را سرت نمی کند.
دستور `(shmctl()` یک دستور کنترلی و اطلاعاتی است که وضعیت حافظه ای اشتراکی را برای ما بازمی گرداند که ورودی اول آن همان خروجی `(shmget()` است و ورودی دوم `IPC_STAT` داده ایم که می گوییم وضعیت حافظه را می خواهیم و ورودی آخر یک داده ساختار است به اسم `shared_memory` که آن را به این دستور پاس داده ایم، حال با استفاده از تابع `sprintf` در حافظه ای اشتراکی می نویسیم و سپس با `while` نوشته شده صبر می کنیم که `reader` این مقدار را از حافظه ای اشتراکی بخواند و زمانی که `reader` از حافظه ای اشتراکی خواند مقدار `*` را در متغیر `shared` ذخیره می کند و `writer` با دریافت این مقدار از `while` بیرون می آید و حافظه ای اشتراکی را با استفاده از دستور `(shmdt()` آزاد می کند.

با استفاده از دستور `gcc -o writer writer.c` و `gcc -o reader reader.c` این دو کد را کامپایل کرده و خروجی را مشاهد می کنیم :

```
mahan@mahan-VirtualBox:~/Desktop$ cd Desktop
mahan@mahan-VirtualBox:~/Desktop$ gcc -o writer writer.c
writer.c: In function 'main':
writer.c:33:9: warning: [enabled by default] implicit declaration of function 'sleep'; did you mean 'select'? [-Wimplicit-function-declaration]
    sleep(1);
          ^
          sleep
mahan@mahan-VirtualBox:~/Desktop$ gcc -o reader reader.c
reader.c: In function 'main':
reader.c:26:5: warning: [enabled by default] implicit declaration of function 'exit' [-Wimplicit-function-declaration]
    exit(0);
    ^
reader.c:26:5: warning: incompatible implicit declaration of built-in function 'exit'
reader.c:26:5: note: include <stdlib.h> or provide a declaration of 'exit'
reader.c:11:1: warning: include <stdlib.h>
reader.c:11:1: note: include <stdlib.h>
reader.c:26:5:
    exit(0);
    ^
mahan@mahan-VirtualBox:~/Desktop$ ./writer Hello
shared memory attached at address 0x7f0bbfd10000
segment size: 100
This string inserted in shared memory: Hello
Waiting for reader...
mahan@mahan-VirtualBox:~/Desktop$
```

همانطور که مشاهد می کنیم خروجی مورد نظر را دریافت کرده ایم.

QUESTION 2

استفاده از ارتباطات client-server یکی دیگر از راههای برقرار ارتباط بین پردازه‌ها در سیستم‌عامل است. البته این روش‌ها فراتر از ارتباط بین پردازه‌های داخل یک سیستم می‌روند و می‌توانند بین پردازه‌هایی که در سیستم‌های متفاوتی قرار دارند نیز از طریق شبکه ارتباط برقرار کنند. یکی از راههای برقراری ارتباط کلاینت-سروری استفاده از socket است.

سوکت بستره است که ارتباط منطقی بین دو پردازه را فراهم می‌کند. برای استفاده از سوکت باید پورتی را نیز به همراه آن مشخص کنیم که سوکت به روی این پورت، تبادلات داده را انجام دهد.

(از نظر ماهیت، کلاینت یک پردازه و سرور نیز یک پردازه است، این دو پردازه پردازه‌های همکاری‌کننده هستند)

در این قسمت از آزمایش قصد داریم که با استفاده از socket، چندین client را به یک server متصل کنیم و فضایی شبیه چت روم ایجاد کنیم.

در مرحله اول توضیح می‌دهیم که چگونه می‌توان یک سرور و یک کلاینت ساخت و با استفاده از سوکت آن‌ها را به هم متصل کرد:

```
server.c x client.c x
65     printf("\n Invalid address/ Address not supported \n");
66     return -1;
67 }
68
69 // connects to the server
70 if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
71 {
72     printf("\nConnection Failed \n");
73     return -1;
74 }
75 printf(" [CLIENT] client %s connected to the server [CLIENT]\n", client_name);
76 if(send(sock, client_name, strlen(client_name), 0)<0){
77     perror("send failed");
78     exit(-1);
79 }
80
81 // create a thread for listening to the server
82 pthread_t tid;
83 pthread_attr_t attr;
84 pthread_attr_init(&attr);
85 pthread_create(&tid, &attr, runner, (void *)sock);
86
87 // reading requests from commandline
88 printf(" [CLIENT] now ready to read user requests [CLIENT]\n";
89
90 char str[600] = {0};
91 while(fgets(str,600,stdin) > 0) {
92     char msg[600] = {0};
93     strcpy(msg,client_name);
94     strcat(msg,"%");
95     strcat(msg,str);
96     int val = send(sock, msg, strlen(msg), 0);
97     if(val < 0){
98         perror("message not sent");
99         return -1;
100    }
101    if(strcmp(str, "quit ") != NULL) {
102        break;
103    }
104 }
105 pthread_join(tid,NULL);
106 close(sock);
107 exit(0);
108
109 }
110
server.c x client.c x
30
31     int sock = 0, valread;
32     struct sockaddr_in serv_addr;
33     char buffer[1024] = {0};
34
35     char host_name [100];
36     char client_name[100];
37
38     if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
39         printf("\n Socket creation error \n");
40         return -1;
41     }
42
43     // sets all memory cells to zero
44     memset(&serv_addr, '0', sizeof(serv_addr));
45
46     // sets port and address family
47     strcpy(host_name, argv[1]);
48     int PORT = atoi(argv[2]);
49     strcpy(client_name, argv[3]);
50     serv_addr.sin_family = AF_INET;
51     serv_addr.sin_port = htons(PORT);
52
53
54     // Convert IPv4 and IPv6 addresses from text to binary form
55     if(inet_pton(AF_INET, host_name , &serv_addr.sin_addr) <= 0)
56     {
57         printf("\n Invalid address/ Address not supported \n");
58         return -1;
59     }
60
61
62     // connects to the server
63     if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
64     {
65         printf("\nConnection Failed \n");
66         return -1;
67     }
68
69     printf(" [CLIENT] client %s connected to the server [CLIENT]\n", client_name);
70     if(send(sock, client_name, strlen(client_name), 0)<0){
71         perror("send failed");
72         exit(-1);
73     }
74
75     // create a thread for listening to the server
76     pthead_t tid;
77     pthead_attr_t attr;
```

همانطوری که در کد مربوط به کلاینت مشاهده می‌کنید، برای تعریف سوکت از تابع `(socket()` استفاده می‌کنیم که یک متغیر `int` را بازمی‌گرداند که از آن می‌توانیم به عنوان `file descriptor` این سوکت استفاده کنیم. اگر مقدار این عدد صحیح، مثبت یا ۰ باشد یعنی سوکت به درستی ایجاد شده است در غیر این صورت با ارور مواجه شده ایم. آرگومان‌های ورودی این تابع، `AF_INET`، `SOCKET_STREAM` و ۰ هر کدام به شرح زیر، تنظیمات اولیه‌ی این سوکت را انجام می‌دهند:

مشخص می‌کند که `Address Family` این سوکت چیست. در اینجا `AF_INET` به معنی IPv4 است و اگر از `AF_INET6` استفاده می‌کردیم به معنی IPv6 می‌بود.

`SOCKET_STREAM`: نوع پرتکل لایه‌ی ترانسپورت این سوکت را به عنوان `TCP` تنظیم می‌کند. اگر از متغیر از پیش تعریف شده‌ی `SOCKET_DGRAM` استفاده می‌کردیم، سوکت با پروتکل `UDP` با لایه‌ی ترانسپورت شبکه ارتباط برقرار می‌کرد.

آرگومان سوم `protocol number` را مشخص می‌کند که ۰ مقدار اتوماتیک است و خود سیستم با توجه به دو آرگومان اول، شماره پروتکل را مشخص خواهد کرد.

در مرحله‌ی بعدی باید یک نمونه از `struct sockaddr_in` ایجاد کنیم. از این نوع استراکت برای تعریف آدرسی که سوکت می‌تواند به آن متصل شود استفاده می‌شود. نام این نمونه ایجاد شده را `serv_addr` را می‌گذاریم چرا که قرار است نشان دهنده‌ی آدرس سرور باشد. با استفاده از تابع `memset` بخشی از حافظه که به این نمونه تعلق دارد را مقداردهی اولیه ۰ می‌کنیم و بعد به صورت دستی فیلدهای این استراکت را مطابق با تنظیمات سوکت و آدرس سرسوس خود، مقداردهی می‌کنیم.

فیلد `sin_family` نوع آدرس را مشخص می‌کند. فیلد `sin_addr` شامل آدرسی آی پی است که میخواهیم به آن متصل شویم. در تیکه کد کلاینت این مقدار

را برابر قرار می‌دهیم. `host_name` را در آرگومان‌های ورودی فایل اجرایی کلاینت خودمان مقداردهی می‌کنیم، به این صورت کلاینت به آدرس دلخواه ما متصل می‌شود. البته برای اینکه استرینگ به فرمت آدرس IP تبدیل شود باید از تابع `inet_ntop` استفاده نماییم که در خط ۶۳ کد کلاینت مشاهده می‌کنید.

در تیکه کد مربوط به سرور که در انتهای این بخش آورده شده است، این مقدار را برابر `INADDR_ANY` قرار می‌دهیم. این متغیر ثابت به معنی آدرس ۰.۰.۰.۰ است که به سرور این امکان را می‌دهد که درخواست‌ها تمام آدرس آی پی‌های ممکن را بپذیرد و خود را به کلاینتی با آدرس آی پی مشخص محدود نکند. فیلد `sin_port` مشخص کنندهٔ پورت است. از تابع `(htons()` برای تبدیل فرمت اینت به فرمتی که برای این فیلد(پورت) مناسب است استفاده می‌شود.

پس از اینکه تنظیمات مربوط به آدرس و پورت به پایان رسید، در سرور باید آن را به سوکت بایند کنیم (خط ۵۵) و سپس روی آن سوکت به `listen` کردن بپردازیم ولی در کد کلاینت نیازی به بایند کردن نیست و کافی است با استفاده از سوکتی که داریم به آدرسی که ایجاد کردہ‌ایم درخواست `connection` بفرستیم (خط ۷۱)

```

server.c           x  client.c           x
20  void mysend(int socket, char *str, int len);
21
22  int main(int argc, char const *argv[]) {
23
24      pthread_t tid;
25      pthread_attr_t attr;
26      pthread_attr_init(&attr);
27
28      char buffer[100] = {0};
29
30
31      // creates socket file descriptor
32      int server_fd;
33      server_fd = socket(AF_INET, SOCK_STREAM, 0);
34      if (server_fd < 0) {
35          perror("socket failed");
36          exit(EXIT_FAILURE);
37      }
38
39      // setting up the address variable
40      int PORT = atoi(argv[1]);
41      struct sockaddr_in address;
42      address.sin_family = AF_INET;
43      address.sin_addr.s_addr = INADDR_ANY;
44      address.sin_port = htons(PORT); // host to network -- converts the ending of the given integer
45      const int addrlen = sizeof(address);
46
47
48      // binding (bind the created socket to the address variable we set)
49      if (bind(server_fd, (struct sockaddr *) &address, sizeof(address)) < 0) {
50          perror("bind failed");
51          exit(EXIT_FAILURE);
52      }
53
54
55      // listening on server socket with backlog size 3.
56      if (listen(server_fd, 3) < 0) {
57          perror("listen failed");
58          exit(EXIT_FAILURE);
59      }
60
61      printf("[SERVER] Listening on %s:%d [SERVER]\n", inet_ntoa(address.sin_addr), ntohs(address.sin_port));
62
63
64      // initialize group array
65      for (int i = 0; i < MAX_GROUP_COUNT; ++i)
66          for (int j = 0; j < MAX_CLIENT_COUNT; ++j)
67              group[i][j] = -1;
68
69      // accepting client
70      // accept returns client socket and fills given address and addrlen with client address information
71      // client socket is an identifier in order to connect to the client socket
72      int client_socket;
73      while(1){
74          if ((client_socket = accept(server_fd, (struct sockaddr *) &address, (socklen_t *) &addrlen)) < 0) {
75              perror("accept failed");
76              exit(EXIT_FAILURE);
77          }
78
79          // print initial connection information
80          printf("[SERVER] connection established for client %s:%d [SERVER]\n", inet_ntoa(address.sin_addr), ntohs(address.sin_port));
81          char *greetings = "Server : Hello Client\n";
82          send(client_socket, greetings, strlen(greetings), 0);
83
84          //setting up a thread for each client
85          client_count++;
86          bzero(buffer, 100);
87          if(read(client_socket, buffer, 100) < 0){
88              perror("reading failed");
89              exit(EXIT_FAILURE);
90          }
91          printf("Client name is: %s\n", buffer);
92          if(pthread_create(&tid, &attr, listentoClient, (void *)client_socket)){
93              perror("Thread creation failed");
94              exit(EXIT_FAILURE);
95          }
96
97      }
98
99      return 0;
100 }
101
102
103
104
105 }
```

در صورتی که درخواست connect یک کلاینت، توسط سرورس که در حال listen کردن است پذیرفته شود، بین کلاینت و سرور ارتباط برقرار می‌شود و می‌توانند از طریق توابع read و send با هم تبادل داده داشته باشند. کلاینت برای اینکه بتواند از این توابع استفاده کند باید فایل دیسکریپتور سوکت سروی را به آن connect شده است را استفاده کند و سرور نیز از فایل دیسکریپتور سوکت کلاینتی که آن را accept کرده است استفاده می‌کند.

نکته‌ی دیگری که قابل توجه است، back log سرور می‌باشد. این عدد را به عنوان دومین آرگومانتابع listen وارد می‌کنیم و نشان‌دهنده‌ی تعداد کلاینت‌هایی است که می‌توانند در صف این سرور منتظر accept بمانند.

همچنین در هنگام استفاده از توابع read و send باید توجه کنیم که چه مقداری را برمی‌گردانند. اگر مقدار ایتنی که برمی‌گردانند کوچکتر از ۰ باشد یعنی عملیات send یا read با موفقیت انجام نشده است.

حالا که با مقدمات برقراری ارتباط بین سرور و کلاینت آشنا شدیم باید به توضیح بخش‌های دیگر از جمله لزوم وجود نخ در برنامه‌های کلاینت-سرور بپردازیم.

از آن‌جایی که سرور قرار است با چند کلاینت به صورت همزمان ارتباط داشته باشد و به آن‌ها خدمات ارائه دهد، ناچار است که به ازای هر کلاینتی که Accept می‌کند برای اینکه بتواند پردازش‌های مربوط به آن کلاینت را انجام دهد یک نخ ایجاد کند. در کلاینت نیز برای از آن‌جایی که فقط منتظر دریافت پیام از سرور نیستیم و باید یک سری پردازش‌های دیگر نیز انجام دهیم، باید از حداقل دو نخ (یکی main و دیگری یک pthread استفاده کنیم)

حالا به توضیح نحوه‌ی پیاده‌سازی بخش‌های مربوط به چت روم می‌پردازیم: در اینجا بعد از اینکه ارتباط سرور با هر کلاینت برقرار می‌شود، به ازای آن کلاینت یک ترد می‌سازیم که تابع listentocClient می‌شود، به ازای آن void * این تابع یک حلقه‌ی while (۱) داریم که در آن به طور مداوم تابع read را صدا می‌زنیم تا اگر کلاینت برای ما پیامی فرستاد، آن را بخوانیم. با استفاده از چند نخ، سرور همواره در حال انتظار برای پیام تمام کلاینت‌هایی است که به آن متصل هستند و می‌تواند پردازش‌های آن‌ها را به صورت جداگانه انجام دهد.

```

166     void *listentoClient (void *c){
167
168         int socket = (Int)(long) c;
169
170         printf("[SERVER] waiting for user on socket %d [SERVER]\n", socket);
171
172         //waits for client to send sth
173         char buffer[MESSAGE_SIZE] = {0};
174         int valread = -1;
175
176         while(1){
177             bzero(buffer, MESSAGE_SIZE);
178             valread = read(socket , buffer, MESSAGE_SIZE);
179             if(valread >= 0){
180                 int ret = parseCommand(socket, buffer);
181                 if (ret == -1){
182                     break;
183                 }
184                 if (ret == 1){
185                     char str[20] = "[SERVER] : done ";
186                     if(send(socket, str , strlen(str), 0)<0){
187                         perror("unable to send");
188                         exit(-1);
189                     }
190                 }
191             } else{
192                 perror("reading failed");
193                 exit(EXIT_FAILURE);
194             }
195             if(send(socket, "$$$" , strlen("$$$"), 0)<0){
196                 perror("unable to send");
197                 exit(-1);
198             }
199             pthread_t pt = pthread_self();
200             pthread_cancel(pt);
201         }
202     }

```

از آن جایی که فایل دیسکریپتور سوکت Client در هر کانکشن یک عدد یکتا می باشد که `read` و `send` را نیز با استفاده از آن انجام می دهیم، در این پیاده سازی برای مشخص کردن هر کلاینت از شماره سوکت آن استفاده کردیم. به عبارتی شماره سوکت هم شناسه‌ی کلاینت‌ها در این چت روم است و هم رابط ارسال و دریافت پیامهای ما به سرور.

همانطوری که در تصویر نیز مشخص است، هرگاه کلاینت برای سرور درخواستی بفرستد، تابع `parseCommand()` فراخوانی می شود تا درخواست کلاینت را تجزیه و تحلیل کند و بر اساس ورودی‌ها، عملیات لازم را اجرا کند.

از آن جایی که پیاده سازی پیامهای ارسالی و دریافتی بین کلاینت و سرور کاملاً قراردادی است، در این چت روم لازم ندانستیم که هیچ اطلاعات اضافه‌ای از کلاینت را در سرور ذخیره کنیم. تنها چیزی که سرور از کلاینت ذخیره دارد یک متغیر `int` است که شماره سوکت آن کلاینت را در این ارتباط مشخص می کند. باقی اطلاعات مثل نام یا هر اطلاعات دیگری که نیاز است سرور بداند، با هر پیام برای سرور فرستاده می شوند و در قسمت `parseCommand()` تابع استخراج می شوند.

```

int parseCommand (int client, char * msg){
    char * name;
    char * gp_id;
    char * command;

    name = strtok(msg, "$");
    command = strtok(NULL, " ");

    //join request
    if (!strcmp(command, "join")){
        gp_id = strtok(NULL, "\0");
        printf("...join request by %s for group %s", name, gp_id);
        if (!validateGroupID(atoi(gp_id), client, name))
            return 0;
        if (!joinGroup(client, atoi(gp_id)))
            return 0;
    }
    // send message request
    else if (!strcmp(command, "send")){
        gp_id = strtok(NULL, " ");
        char * pm;
        pm = strtok(NULL, "\n");
        printf("...send message request by %s to group %s", name, gp_id);
        if (!validateGroupID(atoi(gp_id), client, name))
            return 0;
        sendtoAll(client, gp_id, name, pm);
        return 0;
    }
    // leave group request
    else if (!strcmp(command, "leave")){
        gp_id = strtok(NULL, "\0 ");
        printf("...leaving request by %s for group %s", name, gp_id);
        if (!validateGroupID(atoi(gp_id), client, name))
            return 0;
        if (!leaveGroup(client, atoi(gp_id)))
            return 0;
    }

    //quit chatroom
    else if (!strcmp(command, "quit")){
        printf("...quit request by %s", name);
        for (int i = 0; i < MAX_GROUP_COUNT; ++i){
            for (int j = 0; j < MAX_CLIENT_COUNT; ++j){
                if (group[i][j] == client){
                    char gid[10] = {0};
                    sprintf(gid, "%d", i+1);
                    sendtoAll(client, gid, name, "left the chat\n");
                    group[i][j] = -1;
                    break;
                }
            }
        }
        printf("done\n");
        return -1;
    }
    // wrong input
    else{
        char * str = "[SERVER] : invalid command, please try again";
        mysend(client, str, strlen(str));
        printf("[SERVER] invalid command, CLIENT: %s must try again [SERVER]\n", name);
        return 0;
    }
    return 1;
}

```

مثلا در اینجا، کلاینت با هر درخواستی که برای سرور می‌فرستد، نام خودش را نیز در رشته‌ی درخواست قرار می‌دهد و با استفاده از علامت \$\$ از باقی اطلاعات دستوری جدا می‌کند.

```

70     // connects to the server
71     if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
72     {
73         printf("\nConnection Failed \n");
74         return -1;
75     }
76     printf(" [CLIENT] client %s connected to the server [CLIENT]\n", client_name);
77     if(send(sock, client_name, strlen(client_name), 0)<0){
78         perror("send failed");
79         exit(-1);
80     }
81     // create a thread for listening to the server
82     pthread_t tid;
83     pthread_attr_t attr;
84     pthread_attr_init(&attr);
85     pthread_create(&tid, &attr, runner, (void *)sock);
86
87     // reading requests from commandline
88     printf(" [CLIENT] now ready to read user requests [CLIENT]\n");
89
90     char str[600] = {0};
91     while(fgets(str,600,stdin) > 0) {
92         char msg[600] = {0};
93         strcpy(msg,client_name);
94         strcat(msg,"$");
95         strcat(msg,str);
96         int val = send(sock, msg, strlen(msg), 0);
97         if (val < 0){
98             perror("message not sent");
99             return -1;
100        }
101        if strstr(str, "quit ") != NULL) {
102            break;
103        }
104    }

```

همانطوری که در تکه کد بالا (خط ۹۲ تا ۹۶) که متعلق به کلاینت است نیز مشخص است، کلاینت داده‌های کاربر را می‌خواند و آن‌ها را به مدلی که می‌داند سرور قرار است پارس کند در کنار هم قرار می‌دهد و می‌فرستد.

نکته‌ی قابل توجه دیگر که در همین بخش از کد کلاینت مشخص است استفاده از ترد است. کلاینت دو نخ دارد. یک نخ main که در این نخ با یک حلقه‌ی (1) تمامی اطلاعات ورودی کاربرش را دریافت و برای سرور ارسال می‌کند و یک نخ pthread runner که تابع main را همزمان با نخ runner اجرا می‌کند. در تابع runner هم در یک حلقه‌ی لایتناهی، همواره به سرور گوش می‌دهد. به این ترتیب با استفاده از دو نخ، کلاینت همزمان هم برای سرور پیام ارسال می‌کند و هم پیام‌ها آن را دریافت می‌کند. (قرارداد \$\$\$ نشان‌دهنده‌ی آن است که سرور دیگر برای کلاینت پیامی ارسال نخواهد کرد در نتیجه بعد از رویت این رشته، نخ را از بین می‌بریم)

```

11
12     void *runner(void *serv_sock)
13 {
14     //printf("thread creation was successful\n");
15     int server = (int)(long)serv_sock;
16     char buffer[1024];
17     // client thread always ready to receive message
18     while(1){
19         bzero(buffer, 1024);
20         int valread = read(server, buffer, 1024);
21         if(strstr(buffer, "$$$") != NULL) {
22             break;
23         }
24         printf(" %s\n", buffer);
25         if (valread < 0) {
26             perror("read");
27             exit(-1);
28         }
29     }
30     printf("\n~bye\n");
31     pthread_exit(0);
32 }
33 }
```

بخش‌های دیگر پیاده‌سازی که مربوط به توابع `joinGroup()` و `leaveGroup()` و `validateGroup()` می‌شود، نکته‌ی خاصی ندارد و به چک کردن اینکه آیا کاربر در گروهی که درخواست حذف از آن یا افزوده شدن به آن را می‌دهد از قبل عضو بوده است برمی‌گردد.

در این بخش اشاره‌ای به ساختمان داده‌ی گروه‌ها نیز می‌کنیم. در این چت روم ما ۵۰ گروه با ظرفیت حداقل ۱۰۰ کاربر تعریف کرده‌ایم. اطلاعات مربوط به هر گروه در یک آرایه‌ی دو بعدی از `integer` ذخیره می‌شود.

```

#define MESSAGE_SIZE 1024
#define MAX_CLIENT_COUNT 100
#define MAX_GROUP_COUNT 50

int client_count = 0;
int group_count = 1;
int group[MAX_GROUP_COUNT][MAX_CLIENT_COUNT];
```

شناسه هر گروه، همان شماره ردیف در آرایه است و محتویاتی که در هر ردیف ذخیره می‌شود، شماره‌ی شناسایی (شماره سوکت) کلاینت‌هایی است که در آن گروه عضو هستند.

آخرین تابع مهم در این پیاده سازی، تابع `sendtoAll` می باشد که پیام های کاربر را در گروه تقاضا شده، برای تمامی کاربران آن گروه ارسال کنیم.

```
void sendtoAll(int client, char * group_id, char * name, char * msg){

    int g = atoi(group_id);

    // check membership
    bool memember = false;
    for (int i = 0; i < MAX_CLIENT_COUNT; ++i){
        if (group[g-1][i] == client){
            memember = true;
        }
    }
    if (memember == false){
        printf("not a member of this group\n");
        char msg[100] = "[SERVER] : you are not a member of this group";
        mysend(client, msg, strlen(msg));
        return;
    }

    // build message
    char str1[MESSAGE_SIZE] = {0};
    strcpy(str1, " Group ");
    strcat(str1, group_id);
    strcat(str1, " | ");
    char str2[MESSAGE_SIZE] = {0};
    strcpy(str2, " [ ");
    strcat(str2, name);
    strcat(str2, " ] : ");
    strcat(str2, msg);

    // send message to all members of the group
    for (int i = 0; i < MAX_CLIENT_COUNT; ++i){
        if (group[g-1][i] != -1){
            int socket = group[g-1][i];
            mysend(socket, str1, strlen(str1));
            mysend(socket, str2, strlen(str2));
        }
    }
    printf("\nmessage sent to all\n");
    return;
}

void mysend(int socket, char * str, int len){
    char string[len+10];
    strcpy(string, str);
    if(send(socket, string, strlen(string), 0)<0){
        perror("send error");
        exit(-1);
    }
}
```

همانطور که از کد نیز مشخص است، اول چک می کنیم که کاربر عضو گروهی که تقاضا کرده هست یا نه. اگر بود، پیام برای تمامی کاربران آن گروه ارسال می شود. کاربران یک گروه با شماره سوکت مختص به خودشان در یک ردیف

از آرایه‌ی group ذخیره شده اند که شماره‌ی ردیف هم همان id گروه است پس برای ارسال پیام به تمامی کاربران یک گروه، کافی است پیام را به تمام متغیرهای غیرمنفی ردیف id_glp_group آرایه‌ی group ارسال نماییم.

تابع (mysend) نیز تعریف شده است تا مجبور به تکرار بخش ارورگیری تابع send نباشیم.

پیاده سازی دستور quit تا حد بسیار زیادی شبیه به sendtoAll است و نیازی به توضیح ندارد. کدهای کلاینت و سرور جویی پیاده سازی شده‌اند که خروج یک کلاینت از چت، هم معنی قطع ارتباط آن با سرور باشد و روی ارتباط سرور با باقی کلاینت‌ها تاثیر نگذارد.

چند خروجی از کد را در ادامه می‌آوریم:

۱) برای کامپایل از دستوری gcc -o client client.c و gcc -o server server.c استفاده می‌کنیم که استفاده از pthread -o pthread استفاده از posix thread در هنگام کامپایل به این دلیل است که در کد از استفاده کرده‌ایم.

```
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ gcc -pthread -o client client.c
client.c: In function 'main':
client.c:85:38: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
    pthread_create(&tid, &attr, runner, (void *)sock);
                                         ^
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ gcc -pthread -o serve server.c
server.c: In function 'main':
server.c:97:51: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
    if (pthread_create(&tid, &attr, listentoClient, (void *)client_socket)){
                                         ^
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$
```

۲) ساخت کلاینت و سرور با آرگومان‌های تعریف شده در دستورکار (در اجرای executable حاصل از فرآیند کامپایل باید حتماً دقت کنیم که نام فایل را به همراه / . که مشخص کننده‌ی دایرکتوری آن است بیاوریم)

server [port]

client [server address] [port] [name]

```
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./server 7021
[SERVER] Listening on 0.0.0.0:7021 [SERVER]
[SERVER] connection established for client 127.0.0.1:43714 [SERVER]
    client name is: client
[SERVER] waiting for user on socket 4 [SERVER]
```

```
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./client 127.0.0.1 7021 client
[CLIENT] client client connected to the server [CLIENT]
[CLIENT] now ready to read user requests [CLIENT]
Server : Hello Client!
```

یک سرور روی پورت ۷۰۲۱ ساختیم که کلاینت با نام client به آن متصل می‌شود. در ابتدای برقراری ارتباط، کلاینت نام خود را برای سرور می‌فرستد و سرور نیز با local address ۱۲۷.۰.۰.۱ hello + name پاسخ می‌دهد. (۱۱ نیز میباشد)

میتوانیم چندین کلاینت را با نام‌های متفاوت در چند تب از ترمینال ران کنیم و به این سرور متصل کنیم.

```
Q                                     oslab@OSLab-VirtualBox:~/Desktop/Lab 4
oslab... x   oslab... x   oslab... x   oslab... x   oslab... x   oslab... x
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ gcc -pthread -o serve server.c
server.c: In function 'main':
server.c:97:51: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
    if (pthread_create(&tid, &attr, listentoClient, (void *)client_socket)){
                                         ^
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./server 7021
[SERVER] Listening on 0.0.0.0:7021 [SERVER]
[SERVER] connection established for client 127.0.0.1:43714 [SERVER]
    client name is: client
[SERVER] waiting for user on socket 4 [SERVER]
[SERVER] connection established for client 127.0.0.1:43718 [SERVER]
    client name is: client1
[SERVER] waiting for user on socket 5 [SERVER]
[SERVER] connection established for client 127.0.0.1:43720 [SERVER]
    client name is: client2
[SERVER] waiting for user on socket 6 [SERVER]
[SERVER] connection established for client 127.0.0.1:43726 [SERVER]
    client name is: client3
[SERVER] waiting for user on socket 7 [SERVER]
[SERVER] connection established for client 127.0.0.1:43728 [SERVER]
    client name is: client4
[SERVER] waiting for user on socket 8 [SERVER]
```

۳) دستورات join و leave برای یک پردازه. (همانطوری که مشاهده می‌کنید، سرور پیام‌های مربوط به موفقیت آمیز بودن یا نبودن دستورات را برای کلاینت می‌فرستد)

```
oslab... x oslab... x oslab... x oslab... x oslab... x oslab... x
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./client 127.0.0.1 7021 client
[CLIENT] client client connected to the server [CLIENT]
[CLIENT] now ready to read user requests [CLIENT]
Server : Hello Client!

join 1
[SERVER] : done
join 57
[SERVER] : invalid group id, enter an integer between 1 and 50
join 0
[SERVER] : invalid group id, enter an integer between 1 and 50
join 8
[SERVER] : done
leave 4
[SERVER] : not a member of this group
leave 8
[SERVER] : done
```

خودش هم اجرای هر دستور با به همراه نام کلاینتی که درخواستش را داشته چاپ می‌کند (نام کلاینت را نمایش می‌دهد که در اینجا نامش همان client است)

```
oslab... x oslab... x oslab... x oslab... x oslab... x oslab... x
[SERVER] connection established for client 127.0.0.1:43718 [SERVER]
    client name is: client1
[SERVER] waiting for user on socket 5 [SERVER]
[SERVER] connection established for client 127.0.0.1:43720 [SERVER]
    client name is: client2
[SERVER] waiting for user on socket 6 [SERVER]
[SERVER] connection established for client 127.0.0.1:43726 [SERVER]
    client name is: client3
[SERVER] waiting for user on socket 7 [SERVER]
[SERVER] connection established for client 127.0.0.1:43728 [SERVER]
    client name is: client4
[SERVER] waiting for user on socket 8 [SERVER]
...join request by client for group 1
user added to the group
...join request by client for group 57
invalid group id by CLIENT: client! try again between 1 and 50
...join request by client for group 0
invalid group id by CLIENT: client! try again between 1 and 50
...join request by client for group 8
user added to the group
...leaving request by client for group 4
...leaving request by client for group 8
user removed successfully
```

۴) همهی کلاینت‌ها را در گروه ۲ جوین می‌کنیم و یکی را جوین نمی‌کنیم.
دستور send را وارد می‌کنیم تا ببینیم آیا پیام یک عضو برای تمام اعضا فرستاده
می‌شود یا خیر و می‌بینیم که پیام برای همهی اعضای گروه ۲ ارسال می‌شود
غیر از کلاینتی که عضو گروه ۲ نیست

```
oslab... x oslab... x oslab... x oslab... x oslab... x oslab... x oslab...
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./client 127.0.0.1 7021 client
[CLIENT] client client connected to the server [CLIENT]
[CLIENT] now ready to read user requests [CLIENT]
Server : Hello Client!

join 1
[SERVER] : done
join 57
[SERVER] : invalid group id, enter an integer between 1 and 50
join 0
[SERVER] : invalid group id, enter an integer between 1 and 50
join 8
[SERVER] : done
leave 4
[SERVER] : not a member of this group
leave 8
[SERVER] : done
join 2
[SERVER] : done
send 2 hi everynody
Group 2 | [ client ] : hi everynody
```

```
oslab... x oslab... x oslab... x oslab... x oslab... x oslab... x oslab...
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./client 127.0.0.1 7021 client1
[CLIENT] client client1 connected to the server [CLIENT]
[CLIENT] now ready to read user requests [CLIENT]
Server : Hello Client!

join 2
[SERVER] : done
Group 2 | [ client ] : hi everynody
```

```
oslab... x oslab... x oslab... x oslab... x oslab... x oslab... x oslab...
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./client 127.0.0.1 7021 client2
[CLIENT] client client2 connected to the server [CLIENT]
[CLIENT] now ready to read user requests [CLIENT]
Server : Hello Client!

join 2
[SERVER] : done
Group 2 | [ client ] : hi everynody
```

```

oslab... x oslab... x oslab... x oslab... x oslab... x oslab... x | oslab...
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./client 127.0.0.1 7021 client4
[CLIENT] client client4 connected to the server [CLIENT]
[CLIENT] now ready to read user requests [CLIENT]
Server : Hello Client!

join 3
[SERVER] : done
send 2 hi
[SERVER] : you are not a member of this group

```

(۵) دستور quit را برای یکی از کلاینت‌ها اجرا می‌کنیم:

```

oslab... x oslab... x oslab... x oslab... x oslab... x oslab... x | oslab...
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ ./client 127.0.0.1 7021 client2
[CLIENT] client client2 connected to the server [CLIENT]
[CLIENT] now ready to read user requests [CLIENT]
Server : Hello Client!

join 2
[SERVER] : done
Group 2 | [ client ] : hi everynody
quit

-bye
oslab@OSLab-VirtualBox:~/Desktop/Lab 4$ █

```

کلاینت‌های دیگر پیام خروج را مشاهده می‌کنند ولی اتصال آن‌ها با سرور تحت تاثیر قرار نمی‌گیرد.

```

Q oslab@OSLab-VirtualBox: ~/Desktop/Lab 4 F1 □
oslab... x oslab... x oslab... x oslab... x oslab... x oslab... x | oslab...
[CLIENT] now ready to read user requests [CLIENT]
Server : Hello Client!

join 1
[SERVER] : done
join 57
[SERVER] : invalid group id, enter an integer between 1 and 50
join 0
[SERVER] : invalid group id, enter an integer between 1 and 50
join 8
[SERVER] : done
leave 4
[SERVER] : not a memeber of this group
leave 8
[SERVER] : done
join 2
[SERVER] : done
send 2 hi everynody
Group 2 | [ client ] : hi everynody
Group 2 | [ client2 ] : left the chat!

leave 2
[SERVER] : done
█

```

نمایی از سرور در حین اجرای این دستورات:

```
[SERVER] connection established for client 127.0.0.1:43714 [SERVER]
    client name is: client
[SERVER] waiting for user on socket 4 [SERVER]
[SERVER] connection established for client 127.0.0.1:43718 [SERVER]
    client name is: client1
[SERVER] waiting for user on socket 5 [SERVER]
[SERVER] connection established for client 127.0.0.1:43720 [SERVER]
    client name is: client2
[SERVER] waiting for user on socket 6 [SERVER]
[SERVER] connection established for client 127.0.0.1:43726 [SERVER]
    client name is: client3
[SERVER] waiting for user on socket 7 [SERVER]
[SERVER] connection established for client 127.0.0.1:43728 [SERVER]
    client name is: client4
[SERVER] waiting for user on socket 8 [SERVER]
...join request by client for group 1
user added to the group
...join request by client for group 57
invalid group id by CLIENT: client! try again between 1 and 50
...join request by client for group 0
invalid group id by CLIENT: client! try again between 1 and 50
...join request by client for group 8
user added to the group
...leaving request by client for group 4
...leaving request by client for group 8
user removed successfully
...join request by client1 for group 2
user added to the group
...join request by client2 for group 2
user added to the group
...join request by client3 for group 2
user added to the group
...join request by client4 for group 3
user added to the group
...join request by client for group 2
user added to the group
....send message request by client to group 2
massage sent to all
....send message request by client4 to group 2not a memeber of this gro
...quit request by client2

massage sent to all
done
...leaving request by client for group 2
user removed successfully
```

QUESTION 3

محیطی را فراهم کنید که در آن دو فرآیند با استفاده از خط لوله به تبادل یک پیام متنی پردازنند. فرآیند اول یک پیام متنی دارای حروف بزرگ و کوچک(برای مثال : This Is First (Process) به فرآیند دوم ارسال می کند، فرآیند دوم این پیام را دریافت می کند و حروف بزرگ را به حروف کوچک و حروف کوچک را به حروف بزرگ تبدیل می کند(برای مثال : iS fIRST (pROCESS) و به فرآیند اول می فرستد. راهنمایی : برای این کار به دو خط لوله نیاز دارید.

کد مربوط به این سوال مطابق شکل زیر است :

حال به توضیح قسمت های مختلف این کد می پردازیم :
ابتدا در خط اول file descriptor های پایپ را مشخص کرده ایم سپس یک آرایه ای از کاراکترها داریم که مقادیر آن را با This is First Process پر کرده ایم، سپس در خط 13، Error handling مربوط به پایپ را انجام داده ایم و برای این کار از perror استفاده کرده ایم که ارور را در stderr می نویسد و به آن system pipe() را پاس داده ایم که یک pipe() است که برای ارتباط بین پردازه های مختلف linux function استفاده می شود.

حال با استفاده از دستور fork یک پردازه دیگر(پردازه فرزند) ایجاد کرده ایم که خروجی fork برای فرزند صفر بوده و برای پدر آن یک عدد مثبت که PID فرزند خود است را برمی گرداند، حال با استفاده از دستور switch case، کارهای پدر و فرزند را از هم تفکیک می کنیم.(که حالت default را برای پدر انتخاب کرده ایم و حالت صفر برای فرزند است)

سپس مطابق کد عمل تبدیل lowercase ها به uppercase ها و بالعکس مشخص است(توجه شود که این کار توسط فرزند انجام می شود).

سپس حاصل نهایی را برای پدر می فرستیم و آن را در صفحه ی ترمینال چاپ می کنیم.

حال می خواهیم برنامه ی موردنظر را کامپایل کرده و خروجی را مشاهده کنیم.

A screenshot of a Linux desktop environment. On the left, there's a vertical dock with various application icons. In the center, a terminal window is open with the following text:

```
mahan@mahan-VirtualBox:~/Desktop$ gcc -o pipe pipe.c
pipe.c: In function `main':
pipe.c:10:18: warning: comparison between pointer and integer
  if(str[0] == '\n')
                  ^
pipe.c:72:17: warning: format '%s' expects argument of type 'char *', but argument 3 has type
  "char"
    scanf("%s", &result);
           ^~~~~~
scanf("%s", &result);

mahan@mahan-VirtualBox:~/Desktop$ ./pipe
THIS IS FIRST PROCESS
mahan@mahan-VirtualBox:~/Desktop$
```

همانطور که مشاهده می کنیم خروجی به درستی نمایش داده شده است.

THIS IS FIRST pROCESS