#include<time.h>

#include<stdio.h>

#include<sys/socket.h>

#include<netinet/in.h>

#include<string.h>

#include<sys/select.h>

#include<pthread.h>

#include<signal.h>

#include<stdlib.h>

#include<fcntl.h>

#include<pcap.h>

#include<sys/shm.h>

#include<unistd.h>

#include<sys/un.h>

#include<netinet/ip.h>

#include<arpa/inet.h>

#include<errno.h>

#include<netinet/if\_ether.h>

#include<net/ethernet.h>

#include<netinet/ether.h>

#include<netinet/udp.h>

#include<sys/ipc.h>

#include<sys/msg.h>

#include <sys/sem.h>

#include<poll.h>

#include <sys/types.h>

#include <sys/stat.h>

//#include<bits/stdc++.h>

//using namespace std;

Getpid()

Getppid()

Signal(0,number) sends to all children as well as group

**Read, write and open**  
if (write(wfd, buff, sizeof(buff)) == -1) { perror("Parent: Failed to write to pipe"); break; }

// Reading from child

if (read(rfd, buff2, sizeof(buff2)) == -1)

{ perror("Parent: Failed to read from pipe"); break; }

int ffd = open("ABC", O\_RDWR);

if (ffd == -1) {

perror("Failed to open FIFO");

exit(EXIT\_FAILURE);

}

**Pipe and fork**

int pp1[2], pp2[2];

// Error checking for pipe creation

if (pipe(pp1) == -1) {

perror("Failed to create pipe pp1");

exit(EXIT\_FAILURE);

}

if (pipe(pp2) == -1) {

perror("Failed to create pipe pp2");

exit(EXIT\_FAILURE);

}

int c = fork();

// Error checking for fork

if (c == -1) {

perror("Fork failed");

exit(EXIT\_FAILURE);

}

**Dup2**

if (dup2(c1, STDIN\_FILENO) == -1 || dup2(c2, STDOUT\_FILENO) == -1) {

perror("Failed to restore stdin and stdout");

close(ffd);

exit(EXIT\_FAILURE);

}

**Exec**

execl("./P", "P", NULL);

// If execl() fails, we should handle it

perror("Failed to execute program P");

**Fifo**

char name[50];

if(mkfifo(name,0666)==-1)

{

perror("mkfifo()1");

exit(1);

}

if((wfd=open("./wellknownfifo",O\_WRONLY))==-1)

{

perror("open()");

exit(1);

}

write(wfd,buffer,sizeof(buffer));

char buffer[50];

if(mkfifo("./wellknownfifo",0666)==-1)

{

perror("mkfifo()");

exit(1);

}

if((rfd=open("./wellknownfifo",O\_RDONLY))==-1)

{

perror("open()");

exit(1);

}

read(rfd,buffer,50);

**Poll**

int size;

struct pollfd fds[size];

fds[i]=open(" ", 0666);

fds[i].events=POLLIN;

int ret=poll(fds, size, timeout);

if(fds[i].revents & POLLIN)

{

}

**To know pid of a program by knowing its name**

int fd = fileno(popen("pidof ./S", "r"));

char s[1000];

read(fd, &s, 1000);

X = atoi(s);

int fd = fileno(popen("pidof ./P2.exe", "r"));

char s[1000];

read(fd, &s, 1000);

X = atoi(s);

**POPEN**

char buffer[1024];

ssize\_t bytesRead;

// Use popen() and immediately get the file descriptor using fileno()

int fd = fileno(popen("ls -l", "r"));

//int fd=fileno(popen(“./Prog”,”r”));

if (fd == -1) {

perror("Failed to run command or get file descriptor");

exit(EXIT\_FAILURE);

}

// Read from the file descriptor directly

while ((bytesRead = read(fd, buffer, sizeof(buffer) - 1)) > 0) {

buffer[bytesRead] = '\0'; // Null-terminate the buffer

printf("%s", buffer);

}

**Normal Signal**

void sigusr1\_handler(int sig) {

printf("Caught SIGUSR1 signal: %d\n", sig);

}

int main() {

// Set up a signal handler for SIGUSR1

if (signal(SIGUSR1, sigusr1\_handler) == SIG\_ERR) {

perror("Error setting up signal handler");

exit(EXIT\_FAILURE);

}

// Raise SIGUSR1 signal to self

if (raise(SIGUSR1) != 0) {

perror("Error raising SIGUSR1");

exit(EXIT\_FAILURE);

}

// Send a signal to another process (in this case, PID 1008)

// Replace SIGTRM with SIGTERM (correct signal name)

if (kill(1008, SIGTERM) == -1) {

// Error handling for the kill() system call

fprintf(stderr, "Error sending SIGTERM to process 1008: %s\n", strerror(errno));

exit(EXIT\_FAILURE);

}

return 0;

}  
  
**With sigaction**

#include <stdio.h>

#include <stdlib.h>

#include <signal.h>

#include <unistd.h>

// Signal handler with extended information

void handle\_signal(int sig, siginfo\_t \*siginfo, void \*context) {

printf("Received signal %d\n", sig);

printf("Sent by process with PID: %d\n", siginfo->si\_pid);

printf("User ID of sender: %d\n", siginfo->si\_uid);

}

int main() {

struct sigaction sa;

// Use the sa\_sigaction field and the SA\_SIGINFO flag

sa.sa\_sigaction = handle\_signal;

sigemptyset(&sa.sa\_mask); // No signals will be blocked during the handler

sa.sa\_flags = SA\_SIGINFO; // Use sa\_sigaction to get extended info

// Register the handler for SIGUSR1

if (sigaction(SIGUSR1, &sa, NULL) == -1) {

perror("sigaction");

exit(EXIT\_FAILURE);

}

printf("Waiting for SIGUSR1...\n");

// Infinite loop waiting for SIGUSR1

while (1) {

sleep(1);

}

return 0;

}

**Message queues**

Sender

// Define message buffer structure

struct msgbuf {

long mtype; // Message type

char mtext[100]; // Message text

};

int main() {

key\_t key;

int msgid;

struct msgbuf message;

// Create a unique key for the message queue

key = ftok("progfile", 65);

// Create the message queue

msgid = msgget(key, 0666 | IPC\_CREAT);

// Prepare message

message.mtype = 1; // Message type 1

printf("Enter message: ");

fgets(message.mtext, sizeof(message.mtext), stdin);

// Send message to the queue

if (msgsnd(msgid, &message, sizeof(message.mtext), 0) == -1) {

perror("msgsnd");

exit(1);

}

printf("Message sent: %s\n", message.mtext);

return 0;

}

Receiver

// Define message buffer structure

struct msgbuf {

long mtype; // Message type

char mtext[100]; // Message text

};

int main() {

key\_t key;

int msgid;

struct msgbuf message;

// Create a unique key for the message queue

key = ftok("progfile", 65);

// Access the message queue

msgid = msgget(key, 0666 | IPC\_CREAT);

// Receive the message from the queue (only messages of type 1)

if (msgrcv(msgid, &message, sizeof(message.mtext), 1, 0) == -1) {

perror("msgrcv");

exit(1);

}

printf("Message received: %s\n", message.mtext);

// Remove the message queue after receiving

msgctl(msgid, IPC\_RMID, NULL);

return 0;

}

**Shared Memory and Semaphore**

void sem\_wait(int semid)

{

struct sembuf sb;

sb.sem\_num=0;

sb.sem\_op=-1;

sb.sem\_flg=0;

if((semop(semid,&sb,1))==-1)

{

perror("\nFailed to acquire semaphore.");

exit(0);

}

}

void sem\_try\_wait(int semid)

{

struct sembuf sb;

sb.sem\_num=0;

sb.sem\_op=-1;

sb.sem\_flg=IPC\_NOWAIT;;

return semop(semid,&sb,1);

}

void sem\_signal(int semid)

{

struct sembuf sb;

sb.sem\_num=0;

sb.sem\_op=1;

sb.sem\_flg=0;

if((semop(semid,&sb,1))==-1)

{

perror("\nFailed to release semaphore.");

exit(0);

}

}

void sem\_wait(int semid, int smno) {

struct sembuf sb;

sb.sem\_num = smno;

sb.sem\_op = -1;

sb.sem\_flg = 0;

if (semop(semid, &sb, 1) == -1) {

perror("Failed to acquire semaphore");

exit(EXIT\_FAILURE);

}

}

int sem\_try\_wait(int semid, int smno) {

struct sembuf sb;

sb.sem\_num = smno;

sb.sem\_op = -1;

sb.sem\_flg = IPC\_NOWAIT;

return semop(semid, &sb, 1);

}

void sem\_signal(int semid, int smno) {

struct sembuf sb;

sb.sem\_num = smno;

sb.sem\_op = 1;

sb.sem\_flg = 0;

if (semop(semid, &sb, 1) == -1) {

perror("Failed to release semaphore");

exit(EXIT\_FAILURE);

}

}

int main() {

key\_t key;

// Get key for shared memory 1

key = ftok("x", 65);

if (key == -1) {

perror("ftok failed for key 'x'");

exit(EXIT\_FAILURE);

}

// Get shared memory 1

int shmid1 = shmget(key, sizeof(int), IPC\_CREAT | 0666);

if (shmid1 == -1) {

perror("shmget failed for shared memory 1");

exit(EXIT\_FAILURE);

}

// Get key for shared memory 2

key = ftok("y", 65);

if (key == -1) {

perror("ftok failed for key 'y'");

exit(EXIT\_FAILURE);

}

// Get shared memory 2

int shmid2 = shmget(key, sizeof(int), IPC\_CREAT | 0666);

if (shmid2 == -1) {

perror("shmget failed for shared memory 2");

exit(EXIT\_FAILURE);

}

// Attach shared memory 1

int \*x = shmat(shmid1, (const void \*)0, 0);

if (x == (void \*)-1) {

perror("shmat failed for shared memory 1");

exit(EXIT\_FAILURE);

}

// Attach shared memory 2

int \*y = shmat(shmid2, (const void \*)0, 0);

if (y == (void \*)-1) {

perror("shmat failed for shared memory 2");

exit(EXIT\_FAILURE);

}

// Initialize shared memory values

\*x = 1;

\*y = 1;

// Get key for semaphore

int state = 1;

key = ftok(".", state++);

if (key == -1) {

perror("ftok failed for semaphore");

exit(EXIT\_FAILURE);

}

// Get semaphore set

int sem\_id = semget(key, 2, 0666 | IPC\_CREAT);

if (sem\_id == -1) {

perror("semget failed");

exit(EXIT\_FAILURE);

}

// Set initial semaphore values

int semvalue = 0;

if (semctl(sem\_id, 0, SETVAL, semvalue) == -1) {

perror("semctl failed for semaphore 0");

exit(EXIT\_FAILURE);

}

if (semctl(sem\_id, 1, SETVAL, semvalue) == -1) {

perror("semctl failed for semaphore 1");

exit(EXIT\_FAILURE);

}

int temp = 1;

// Example loop

for (int i = 0; i < 5; i++) {

\*x = temp;

// Signal semaphore 0

sem\_signal(sem\_id, 0);

// Wait on semaphore 1

sem\_wait(sem\_id, 1);

// Output the value of y

printf("y: %d\n", \*y);

// Increment temp by the value of y

temp = \*y + 1;

}

// Detach shared memory

if (shmdt(x) == -1) {

perror("shmdt failed for shared memory 1");

}

if (shmdt(y) == -1) {

perror("shmdt failed for shared memory 2");

}

return 0;

}

**Attaching to Shared Memory:**

Copy code

void\* shmat(int shmid, const void \*shmaddr, int shmflg);

* **shmid**: Shared memory ID returned by shmget.
* **shmaddr**: Attach address (NULL for automatic).
* **shmflg**: 0 for read/write, SHM\_RDONLY for read-only.

**Example:**

Copy code

int \*shared\_memory = (int\*) shmat(shmid, NULL, 0); // Attach shared memory

if (shared\_memory == (void\*)-1) {

perror("shmat failed");

exit(EXIT\_FAILURE);

}

**Creating semaphore**

int semget(key\_t key, int nsems, int semflg);

* **key**: Semaphore key (can use ftok).
* **nsems**: Number of semaphores in the set.
* **semflg**: Flags (like IPC\_CREAT or 0666 for permissions).

int semctl(int semid, int semnum, int cmd, union semun arg);

* **cmd**: SETVAL to set the value of the semaphore.
* **semun arg**: A union that holds the semaphore value.

**BSD Connectionless //Wrong**

**Client**

int sfd = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sfd == -1) {

perror("Socket creation failed");

return 1;

}

struct sockaddr\_in serveraddress;

memset(&serveraddress, 0, sizeof(serveraddress)); // Zero out the structure

serveraddress.sin\_family = AF\_INET;

serveraddress.sin\_port = htons(8080);

serveraddress.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

if (bind(sfd, (struct sockaddr \*)&serveraddress, sizeof(serveraddress)) == -1) {

perror("Bind failed");

close(sfd);

return 1;

}

char buff[100];

char \*msg = "Sent from client";

if (sendto(sfd, msg, strlen(msg), 0, (struct sockaddr \*)&serveraddress, sizeof(serveraddress)) == -1) {

perror("Sendto failed");

close(sfd);

return 1;

}

socklen\_t addr\_len = sizeof(serveraddress);

int n = recvfrom(sfd, buff, sizeof(buff), 0, (struct sockaddr \*)&serveraddress, &addr\_len);

if (n == -1) {

perror("Recvfrom failed");

close(sfd);

return 1;

}

buff[n] = '\0'; // Null terminate the buffer

printf("Received from server: %s\n", buff);

close(sfd);

**Server**

int sfd = socket(AF\_INET, SOCK\_DGRAM, 0);

if (sfd == -1) {

perror("Socket creation failed");

return 1;

}

struct sockaddr\_in serveraddress, clientaddress;

socklen\_t addr\_len = sizeof(clientaddress);

serveraddress.sin\_family = AF\_INET;

serveraddress.sin\_port = htons(8080);

serveraddress.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

if (bind(sfd, (struct sockaddr \*)&serveraddress, sizeof(serveraddress)) == -1) {

perror("Bind failed");

close(sfd);

return 1;

}

char buff[100];

// Receive data from client

int n = recvfrom(sfd, buff, sizeof(buff) - 1, 0, (struct sockaddr \*)&clientaddress, &addr\_len);

if (n == -1) {

perror("Recvfrom failed");

close(sfd);

return 1;

}

buff[n] = '\0'; // Null-terminate the received buffer

printf("Received from client: %s\n", buff);

// Prepare message to send to the client

char \*msg = "Sent from server";

// Send response to client

if (sendto(sfd, msg, strlen(msg), 0, (struct sockaddr \*)&clientaddress, addr\_len) == -1) {

perror("Sendto failed");

close(sfd);

return 1;

}

// Close the socket

close(sfd);

**BSD Connection Oriented**

**Server**

int sfd[3];

for (int i = 0; i < 3; i++)

{

sfd[i] = socket(AF\_INET, SOCK\_STREAM, 0); // 0 given for default

if (sfd[i] < 0) {

perror("Socket creation failed");

exit(EXIT\_FAILURE);

}

struct sockaddr\_in serveaddr;

serveaddr.sin\_family = AF\_INET;

if (i == 0)

serveaddr.sin\_port = htons(7070);

else if (i == 1)

serveaddr.sin\_port = htons(8080);

else

serveaddr.sin\_port = htons(9090);

serveaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

if (bind(sfd[i], (struct sockaddr \*)&serveaddr, sizeof(serveaddr)) < 0) {

perror("Bind failed");

close(sfd[i]);

exit(EXIT\_FAILURE);

}

if (listen(sfd[i], 5) < 0) { // 5 is the backlog

perror("Listen failed");

close(sfd[i]);

exit(EXIT\_FAILURE);

}

}

struct pollfd P[3];

for (int i = 0; i < 3; i++)

{

P[i].fd = sfd[i];

P[i].events = POLLIN;

}

while (1)

{

int t = poll(P, 3, 100); // timeout set to 100ms

if (t < 0) {

perror("Poll failed");

exit(EXIT\_FAILURE);

}

if (t > 0)

{

for (int i = 0; i < 3; i++)

{

if (P[i].revents & POLLIN)

{

struct sockaddr\_in clientaddr;

socklen\_t clientsize = sizeof(clientaddr);

int nsfd = accept(sfd[i], (struct sockaddr \*)&clientaddr, &clientsize);

if (nsfd < 0) {

perror("Accept failed");

continue;

}

int c = fork();

if (c < 0) {

perror("Fork failed");

close(nsfd);

continue;

}

if (c > 0) {

close(nsfd); // Parent process closes the new socket

}

else

{

close(sfd[i]); // Child process closes the listening socket

dup2(nsfd, 0); // Redirect input to the new socket

dup2(nsfd, 1); // Redirect output to the new socket

if (i == 0)

execl("./A", "A", NULL);

else if (i == 1)

execl("./B", "B", NULL);

else

execl("./C", "C", NULL);

perror("Exec failed"); // If execl fails, report error

close(nsfd);

exit(EXIT\_FAILURE);

}

}

}

}

}

// Clean up (although this part is unlikely to be reached in a typical server loop)

for (int i = 0; i < 3; i++) {

close(sfd[i]);

}

**Client**

int sfd = socket(AF\_INET, SOCK\_STREAM, 0); // 0 given for default

if (sfd < 0) {

perror("Socket creation failed");

exit(EXIT\_FAILURE);

}

// Define server address

struct sockaddr\_in serveaddr;

serveaddr.sin\_family = AF\_INET;

serveaddr.sin\_port = htons(9090); // change port no for different processes

serveaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

// Connect to the server

if (connect(sfd, (struct sockaddr \*)&serveaddr, sizeof(serveaddr)) < 0) {

perror("Connection to server failed");

close(sfd);

exit(EXIT\_FAILURE);

}

// Send message to server

char \*str = "Hello Server";

if (send(sfd, str, strlen(str), 0) < 0) {

perror("Send failed");

close(sfd);

exit(EXIT\_FAILURE);

}

// Receive response from server

char buff[100];

int t = recv(sfd, buff, sizeof(buff), 0);

if (t < 0) {

perror("Receive failed");

close(sfd);

exit(EXIT\_FAILURE);

}

// Null-terminate the received string

buff[t] = '\0';

printf("Received from Server: %s\n", buff);

// Close the socket

close(sfd);

**Unix Domain Socket**

int send\_fd(int socket, int fd\_to\_send)

{

struct msghdr socket\_message;

struct iovec io\_vector[1];

struct cmsghdr \*control\_message = NULL;

char message\_buffer[1];

/\* storage space needed for an ancillary element with a paylod of length is CMSG\_SPACE(sizeof(length)) \*/

char ancillary\_element\_buffer[CMSG\_SPACE(sizeof(int))];

int available\_ancillary\_element\_buffer\_space;

/\* at least one vector of one byte must be sent \*/

message\_buffer[0] = 'F';

io\_vector[0].iov\_base = message\_buffer;

io\_vector[0].iov\_len = 1;

/\* initialize socket message \*/

memset(&socket\_message, 0, sizeof(struct msghdr));

socket\_message.msg\_iov = io\_vector;

socket\_message.msg\_iovlen = 1;

/\* provide space for the ancillary data \*/

available\_ancillary\_element\_buffer\_space = CMSG\_SPACE(sizeof(int));

memset(ancillary\_element\_buffer, 0, available\_ancillary\_element\_buffer\_space);

socket\_message.msg\_control = ancillary\_element\_buffer;

socket\_message.msg\_controllen = available\_ancillary\_element\_buffer\_space;

/\* initialize a single ancillary data element for fd passing \*/

control\_message = CMSG\_FIRSTHDR(&socket\_message);

control\_message->cmsg\_level = SOL\_SOCKET;

control\_message->cmsg\_type = SCM\_RIGHTS;

control\_message->cmsg\_len = CMSG\_LEN(sizeof(int));

\*((int \*) CMSG\_DATA(control\_message)) = fd\_to\_send;

return sendmsg(socket, &socket\_message, 0);

}

int recv\_fd(int socket)

{

int sent\_fd, available\_ancillary\_element\_buffer\_space;

struct msghdr socket\_message;

struct iovec io\_vector[1];

struct cmsghdr \*control\_message = NULL;

char message\_buffer[1];

char ancillary\_element\_buffer[CMSG\_SPACE(sizeof(int))];

/\* start clean \*/

memset(&socket\_message, 0, sizeof(struct msghdr));

memset(ancillary\_element\_buffer, 0, CMSG\_SPACE(sizeof(int)));

/\* setup a place to fill in message contents \*/

io\_vector[0].iov\_base = message\_buffer;

io\_vector[0].iov\_len = 1;

socket\_message.msg\_iov = io\_vector;

socket\_message.msg\_iovlen = 1;

/\* provide space for the ancillary data \*/

socket\_message.msg\_control = ancillary\_element\_buffer;

socket\_message.msg\_controllen = CMSG\_SPACE(sizeof(int));

if(recvmsg(socket, &socket\_message, MSG\_CMSG\_CLOEXEC) < 0)

return -1;

if(message\_buffer[0] != 'F')

{

/\* this did not originate from the above function \*/

return -1;

}

if((socket\_message.msg\_flags & MSG\_CTRUNC) == MSG\_CTRUNC)

{

/\* we did not provide enough space for the ancillary element array \*/

return -1;

}

/\* iterate ancillary elements \*/

for(control\_message = CMSG\_FIRSTHDR(&socket\_message);

control\_message != NULL;

control\_message = CMSG\_NXTHDR(&socket\_message, control\_message))

{

if( (control\_message->cmsg\_level == SOL\_SOCKET) &&

(control\_message->cmsg\_type == SCM\_RIGHTS) )

{

sent\_fd = \*((int \*) CMSG\_DATA(control\_message));

return sent\_fd;

}

}

return -1;

}

#define ADDRESS "mysocket"

int main()

{

int usfd;

struct sockaddr\_un userv\_addr,ucli\_addr;

int userv\_len,ucli\_len;

usfd = socket(AF\_UNIX , SOCK\_STREAM , 0);

perror("socket");

bzero(&userv\_addr,sizeof(userv\_addr));

userv\_addr.sun\_family = AF\_UNIX;

strcpy(userv\_addr.sun\_path, ADDRESS);

unlink(ADDRESS);

userv\_len = sizeof(userv\_addr);

if(bind(usfd, (struct sockaddr \*)&userv\_addr, userv\_len)==-1)

perror("server: bind");

if(listen(usfd, 10) < 0) {

perror("listen");

exit(1);

}

int nufd;

if((nufd = accept(usfd, (struct sockaddr\*)&ucli\_addr, &ucli\_len)) < 0) {

perror("accept");

exit(1);

}

int fd=open("file.txt",O\_RDONLY);

char S[5];

read(fd,S,sizeof(S));

printf("%s",S);

send\_fd(nufd,fd);

}

#define ADDRESS "mysocket"

int main()

{

int usfd;

struct sockaddr\_un userv\_addr;

int userv\_len;

// Create Unix domain socket

usfd = socket(AF\_UNIX, SOCK\_STREAM, 0); // Change to SOCK\_STREAM

if (usfd == -1) {

perror("socket");

exit(1);

}

bzero(&userv\_addr, sizeof(userv\_addr));

userv\_addr.sun\_family = AF\_UNIX;

strcpy(userv\_addr.sun\_path, ADDRESS);

userv\_len = sizeof(userv\_addr);

// Connect to the server

if (connect(usfd, (struct sockaddr \*)&userv\_addr, userv\_len) < 0) {

perror("connect");

exit(1);

}

int t=recv\_fd(usfd);

char S[5];

read(t,S,sizeof(S));

printf("%s",S);

}

**Pthread**

#include <netinet/in.h>

#include <string.h>

#include <sys/select.h>

#include <pthread.h>

#include <signal.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/shm.h>

#include <unistd.h>

#include <sys/un.h>

#include <netinet/ip.h>

#include <arpa/inet.h>

#include <errno.h>

#include <netinet/if\_ether.h>

#include <net/ethernet.h>

#include <netinet/ether.h>

#include <netinet/udp.h>

#include <sys/ipc.h>

#include <sys/msg.h>

#include <sys/sem.h>

typedef struct {

int thread\_id;

char message[20];

} thread\_data\_t;

void\* thread\_function(void\* arg) {

thread\_data\_t\* data = (thread\_data\_t\*)arg; // Cast the argument back to the original type

printf("Thread ID: %d, Message: %s\n", data->thread\_id, data->message);

pthread\_exit(NULL); // Exit the thread

}

int main() {

pthread\_t threads[5];

thread\_data\_t thread\_data[5]; // Array to hold thread data

for (int i = 0; i < 5; i++) {

thread\_data[i].thread\_id = i;

snprintf(thread\_data[i].message, sizeof(thread\_data[i].message), "Hello from %d", i);

// Create a thread

int rc = pthread\_create(&threads[i], NULL, thread\_function, (void\*)&thread\_data[i]);

if (rc) {

fprintf(stderr, "Error creating thread %d: %d\n", i, rc);

exit(-1);

}

}

// Wait for all threads to finish

for (int i = 0; i < 5; i++) {

pthread\_join(threads[i], NULL);

}

return 0;

}

**GET Peer name**

int sockfd; // Assume this is an already connected socket

struct sockaddr\_in addr;

socklen\_t addr\_len = sizeof(addr);

if (getpeername(sockfd, (struct sockaddr \*)&addr, &addr\_len) == -1) {

perror("getpeername failed");

return -1;

}

char ip[INET\_ADDRSTRLEN];

inet\_ntop(AF\_INET, &(addr.sin\_addr), ip, sizeof(ip));

printf("Connected to %s:%d\n", ip, ntohs(addr.sin\_port));