Assigmnent 2:
Description
Part A
* "cmp" - compare two files, return 0 if qeual and 1 if not equal.
* supports -v flag which request verbal output equal or distinct.
* supports -i flag which ignores case of letters. A equal to a if this flag is on.
** USAGE:
- cmp <file1> <file2>
- cmp <file1> <file2> -v
- cmp <file1> <file2> -i
- cmp <file1> <file2> -v -i

* "copy" - copy a file to another place. return 0 if succeed else 1 on failure
* will create a new file if it doesnt exist but doesnt overwrite a file if exists
* supports -v flag to return verbal output such as succeess or target file exists
* or general failure on other problem.
* also supports -f flag which means force to overwrite a target file
** USAGE:
- copy <file1> <file2>
- copy <file1> <file2> -v
- copy <file1> <file2> -f
- copy <file1> <file2> -v -f

Part B
* Coding library that implements two methods
- CodecA - convers lower case letters to upper case and the other case. turns capital to lower case
- respects only letters which means it doesnt relate to other chars
- CodecB - convers a char to the 3rd next letter
** these methods support encode and decode
which does one way and the decode reverses it
** USAGE:
- encode <codec> <message>
- decode < codec> <message>

Part C
* implements simple shell. it should be able to:
* run CMD tools that exist on System
* be able to stop a running tool by ctrl + c while not killing the shell mid run
* redirect output with > and >> also should support piping |
* be able to stop by exit command

** USAGE:
- stshell <commands>

STSHELL.C

```
#include <sys/stat.h> #include <sys/wait.h> #include <fcntl.h> #include <stdio.h> #include <errno.h>
#include <stdlib.h> #include <unistd.h> #include <string.h> #include <signal.h>
#define MAX_COMMAND_LENGTH 1024
#define MAX_ARGUMENTS 10

void sigint_handler(int signum)
{ printf("\n"); // newline }

void execute_command(char **argv, int input_fd, int output_fd, int *pipe_fds) {
    pid_t pid = fork();
    if (pid == -1)
    {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0)
    {
        // child process
        if (input_fd != STDIN_FILENO)
        {
            dup2(input_fd, STDIN_FILENO);
            close(input_fd);
        }
        if (output_fd != STDOUT_FILENO)
        {
            dup2(output_fd, STDOUT_FILENO);
            close(output_fd);
        }
        if (pipe_fds != NULL)
        {
            close(pipe_fds[0]); // close read end of pipe
            dup2(pipe_fds[1], STDOUT_FILENO);
            close(pipe_fds[1]);
        }
        execvp(argv[0], argv);
        perror("execvp failed");
        exit(1);
    }
    else{
        // parent process
        if (pipe_fds != NULL)
        {
            close(pipe_fds[1]); // close write end of pipe
        }
        int status;
        if (waitpid(pid, &status, 0) == -1)
        {
            perror("waitpid failed");
            exit(EXIT_FAILURE);
        }
        do
        {
            waitpid(pid, &status, 0);
        } while(!WIFEXITED(status) && !WIFSIGNALED(status));
    }
}
```

```
void parse_command(char *command, char **argv, int *num_args, int *input_fd, int *output_fd, int *pipe_fds) {
    *num_args = 0;
    *input_fd = STDIN_FILENO;
    *output_fd = STDOUT_FILENO;
    *pipe_fds = -1;

    // token command string by spaces
    char *token = strtok(command, " ");
    while (token != NULL)
    {
        if (strcmp(token, ">") == 0)
        {
            // output
            token = strtok(NULL, " ");
            *output_fd = open(token, O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (*output_fd == -1)
            {
                perror("open failed");
                return; // return without executing the command
            }
        }
        else if (strcmp(token, ">>") == 0)
        {
            // output (append)
            token = strtok(NULL, " ");
            *output_fd = open(token, O_WRONLY | O_CREAT | O_APPEND, 0644);
            if (*output_fd == -1)
            {
                perror("open failed");
                return; // return without executing the command
            }
        }
        else if (strcmp(token, "|") == 0)
        {
            // pipe
            int pipefds[2];
            if (pipe(pipefds) == -1)
            {
                perror("pipe failed");
                return; // return without executing the command
            }
            *pipe_fds = pipefds[0];  // read end of pipe
            *output_fd = pipefds[1]; // write end of pipe
            argv[*num_args] = NULL;
            execute_command(argv, *input_fd, *output_fd, pipefds);
            *num_args = 0;         // reset argument count
            *input_fd = *pipe_fds; // set input to read end of pipe
            *pipe_fds = -1;        // reset pipe
        }
        else
        {
            // regular argument
            argv[*num_args] = token;
            (*num_args)++;
        }
```

```
        token = strtok(NULL, " ");
    }
    argv[*num_args] = NULL;
    execute_command(argv, *input_fd, *output_fd, pipe_fds);
}

int main()
{
    char command[MAX_COMMAND_LENGTH];
    char *argv[MAX_ARGUMENTS];
    int num_args, input_fd, output_fd, pipe_fds;

    signal(SIGINT, sigint_handler);

    while (1)
    {
        printf("stshell : ");
                fgets(command, 1024, stdin);
                command[strlen(command) - 1] = '\0'; // replace \n with \0

        if(strcmp(command, "exit") == 0) break;

        // parse command
        parse_command(command, argv, &num_args, &input_fd, &output_fd, &pipe_fds);
    }

    return 0;
}
```

MAKEFILE.C

```
GC = gcc
FLAGS = -Wall -g
all: cmp copy encode decode stshell
cmp.o: cmp.c
        $(GC) $(FLAGS) -c cmp.c
copy.o: copy.c
        $(GC) $(FLAGS) -c copy.c
encode.o: encode.c
        $(GC) $(FLAGS) -c encode.c -ldl
encode: encode.o
        $(GC) $(FLAGS) -o encode encode.o -ldl
decode.o: decode.c
        $(GC) $(FLAGS) -c decode.c -ldl
decode: decode.o
        $(GC) $(FLAGS) -o decode decode.o -ldl
codecA: codecA.c codecA.h
        $(GC) $(FLAGS) -shared -fPIC -o codecA codecA.c
codecB: codecB.c codecB.h
        $(GC) $(FLAGS) -shared -fPIC -o codecB codecB.c
stshell.o: stshell.c
        $(GC) $(FLAGS) -c stshell.c
stshell: stshell.o
        $(GC) $(FLAGS) -o stshell stshell.o
.PHONY: all clean

clean:
        rm -f *.o *.so *.a cmp copy encode decode stshell
```

ASSIGNMENT 3
We have been asked to buikd a chat cmd tool that can send messages over network to the same tool.
There are 2 sides. Communication based on IPv4 TCP. We use poll here.

client: ./stnc -c IP PORT
server: ./stnc -s PORT

Performance test
We expand the tool so it will work test utility.
Sending 100MB size of data from client to server.
The server reports the time.

TCP/UDP ipv4 and ipv6
mmap and pipe
UDS stream, dgram

client: ./stnc -c IP PORT -p TYPE PARAM
-p stands for performance
TYPE is the communication type
PARAM is the parameter
ipv4 ipv6 uds pipe mmap

server: ./stnc -s PORT -p -q
-p performance
-q quiet mode

STNC.C

#include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h>
#include <sys/socket.h> #include <netinet/in.h> #include <arpa/inet.h> #include <poll.h>
#include <time.h> #include <sys/sendfile.h> #include <fcntl.h> #include <sys/stat.h>
#include <sys/mman.h> #include <sys/un.h> #include <sys/fcntl.h> #include <sys/un.h>
#define MAX_CLIENTS 1
#define BUFFER_SIZE 1024
#define SIZE 104857600
#define PIPE "/tmp/pipe"
#define SOCKET_PATH "/tmp/mysocket"
#define SOCKET_PATH "/tmp/mysocket"

void generate_file() {

```
    const char* filename = "newfile.bin";
    FILE* file = fopen(filename, "wb");
    if (file == NULL) {
        fprintf(stderr, "Failed to open file for writing\n");
        return;
    }
    char buffer[BUFFER_SIZE];
    size_t bytes_written = 0;
    srand((unsigned int)time(NULL));
    while (bytes_written < SIZE) {
        for (int i = 0; i < BUFFER_SIZE; i++) {
            buffer[i] = (char)rand();
        }
        size_t chunk_size = 1024;
        if (bytes_written + 1024 > SIZE) {
            chunk_size = SIZE - bytes_written;
        }
```

```
        fwrite(buffer, 1, chunk_size, file);
        bytes_written += chunk_size;
    }
    printf("file created, %d bytes the file %s\n", SIZE, filename);
    fclose(file);
}

void calculate_file_checksum(const char *filename) {
    FILE *file = fopen(filename, "rb");
    if (!file)
    {
        perror("Failed to open file");
        exit(1);
    }

    unsigned char checksum = 0;
    unsigned char buffer;
    int i = 0;
    while (fread(&buffer, 1, 1, file) == 1)
    { checksum += buffer; }

    fclose(file);
    printf("checksum: %o\n", checksum);
}

void server_udsdgram(int quiet_mode){
    int serverfd;
    struct sockaddr_un addr, client_addr;
    int addr_len = sizeof(client_addr);
    char buf[BUFFER_SIZE];

    // Create a socket
    if ((serverfd = socket(AF_UNIX, SOCK_DGRAM, 0)) == -1)
    {
        perror("error socket");
        exit(1);
    }

    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SOCKET_PATH, sizeof(addr.sun_path) - 1);

    unlink(SOCKET_PATH);

    if (bind(serverfd, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        perror("error bind");
        exit(1);
    }

    struct pollfd fds[1];
    fds[0].fd = serverfd;
    fds[0].events = POLLIN;

    FILE *f = NULL;
    int pollRet;
    clock_t time = clock();
```

```
    int sum = 0;
    while (1)
    {
        pollRet = poll(fds, 1, -1);

        if (pollRet > 0)
        {
            if (fds[0].revents & POLLIN)
            {
                int rec = recvfrom(serverfd, buf, BUFFER_SIZE, 0,
                                   (struct sockaddr *)&client_addr, (socklen_t *)&addr_len);
                if (rec <= 0)
                {
                    perror("recvfrom");
                    break;
                }

                if (f == NULL)
                {
                    f = fopen("newfile_udsDgram.bin", "wb");
                    if (f == NULL)
                    {
                        perror("error open file");
                        exit(1);
                    }
                }
                sum += rec;
                fwrite(buf, 1, rec, f);
            }
        }
        else
        {
            perror("poll");
            break;
        }
    }

    time = clock() - time;
    if (f != NULL)
    {
        fclose(f);
    }
    close(serverfd);

    if (quiet_mode != 1)
    {
        printf("File transfer - %d bytes received.\n", sum);
        printf("Time taken: %.6lf\n", (double)time);
    }
    else
    { printf("uds_stream,%.6lf seconds\n", (double)time*1000); }
}
```

```
void client_udsdgram()
{
    int sock = 0;
    struct sockaddr_un server_addr;
    char buf[BUFFER_SIZE];

    // Create a socket
    if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
    {
        perror("socket");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sun_family = AF_UNIX;
    strncpy(server_addr.sun_path, SOCKET_PATH, sizeof(server_addr.sun_path) - 1);

    FILE *f = fopen("newfile.bin", "rb");
    if (f == NULL)
    {
        perror("fopen");
        exit(EXIT_FAILURE);
    }

    clock_t time = clock();
    int read;
    while ((read = fread(buf, 1, BUFFER_SIZE, f)) > 0)
    {
        sendto(sock, buf, read, 0, (struct sockaddr *)&server_addr, sizeof(server_addr));
    }

    time = clock() - time;

    fclose(f);
    close(sock);

    printf("File transfer completed in %f .\n", (double)time);
}

void server_udsstream(int quiet_mode)
{
    int serverfd, client_sock;
    struct sockaddr_un addr, client_addr;
    int addr_len = sizeof(client_addr);
    char buf[BUFFER_SIZE];

    if ((serverfd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }

    memset(&addr, 0, sizeof(addr));
    addr.sun_family = AF_UNIX;
    strncpy(addr.sun_path, SOCKET_PATH, sizeof(addr.sun_path) - 1);

    unlink(SOCKET_PATH);
```

```
if (bind(serverfd, (struct sockaddr *)&addr, sizeof(addr)) < 0)
{
    perror("bind");
    close(serverfd);
    exit(1);
}

if (listen(serverfd, 1) < 0)
{
    perror("listen");
    exit(1);
}

if ((client_sock = accept(serverfd, (struct sockaddr *)&client_addr, (socklen_t *)&addr_len)) < 0)
{
    perror("accept");
    exit(1);
}

FILE *f = NULL;
int sum = 0;
clock_t time = clock();
while (1)
{
    int rec = read(client_sock, buf, BUFFER_SIZE);
    if (rec <= 0)
    {
        if (rec < 0)
        {
            perror("read");
        }
        break;
    }

    if (f == NULL)
    {
        f = fopen("newfile_udsStream.bin", "wb");
        if (f == NULL)
        {
            perror("fopen");
            exit(1);
        }
    }
    sum += rec;
    fwrite(buf, 1, rec, f);
}

time = clock() - time;

fclose(f);

close(client_sock);
close(serverfd);
if (quiet_mode != 1)
{
    printf("File transfer - %d bytes received.\n", sum);
```

```
        printf("Time taken: %.6lf\n", (double)time);
    }
    else
    {
        printf("uds_stream,%.6lf seconds\n", (double)time);
    }
}

void client_udsstream()
{
    sleep(1);

    int sock, len;
    struct sockaddr_un client_addr;
    char buf[BUFFER_SIZE];

    if ((sock = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
    {
        perror("socket");
        exit(1);
    }

    client_addr.sun_family = AF_UNIX;
    strcpy(client_addr.sun_path, SOCKET_PATH);
    len = strlen(client_addr.sun_path) + sizeof(client_addr.sun_family);

    if (connect(sock, (struct sockaddr *)&client_addr, len) == -1)
    {
        perror("connect");
        exit(1);
    }

    generate_file();
    char c;
    calculate_file_checksum("newfile.bin");
    if (send(sock, &c, sizeof(c), 0) == -1)
    {
        perror("send");
        exit(1);
    }

    // open the file
    FILE *f;
    if ((f = fopen("newfile.bin", "rb")) == NULL)
    {
        perror("fopen");
        exit(1);
    }

    while (fgets(buf, BUFFER_SIZE, f) != NULL)
    {
        if (send(sock, buf, BUFFER_SIZE, 0) == -1)
        {
            perror("send");
            exit(1);
        }
    }
```

```
    // close file and socket
    fclose(f);
    close(sock);
}


void client_ipv4_tcp(int port, const char* ip)
{
    fflush(stdout);
    int sock_addr = socket(AF_INET, SOCK_STREAM,0);
    if(sock_addr < 0){
        perror("eror socket ");
        exit(1);

    }
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr =inet_addr(ip);
    server_addr.sin_port = htons(port);

    if(connect(sock_addr, (struct  sockaddr *)&server_addr, sizeof(server_addr)) < 0)
    {
        perror("eror connect ");
        exit(1);
    }

    generate_file();

    int f = open("newfile.bin", O_RDONLY);
    off_t offset = 0;
    int sent;
    sleep(5);
    while( (sent = sendfile(sock_addr, f, &offset,BUFFER_SIZE ))  > 0 ) {

        if(sent == -1){
            perror("eror send file ");
            exit(1);
        }
    }
    printf("file transfer\n");
    sleep(2);
    close(f);
    close(sock_addr);
}

void server_ipv4_tcp(int port, int quiet)
{
    int server_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (server_sock < 0)
    {
        perror("error socket ");
        exit(1);
    }

    struct sockaddr_in server_addr;
```

```
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(port);

    if(bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
    {
        perror("error bind");
        exit(1);
    }

    if(listen(server_sock, 5) < 0)
    {
        perror("error listen");
        exit(1);
    }

    struct sockaddr_in client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    int connect_sock = accept(server_sock, (struct sockaddr *)&client_addr, &client_addr_len);
    if (connect_sock < 0)
    {
        perror("error accept");
        exit(1);
    }

    char buf[BUFFER_SIZE];
    int sfile = SIZE;
    FILE * f = fopen("newfile_tcp4", "wb");
    int rec;
    sleep(1);
    clock_t time = clock();

    while(sfile > 0 ){
        rec = recv(connect_sock, buf, BUFFER_SIZE, 0);
        if(rec == -1){
            perror("error recive");
            exit(1);
        }
        sfile -= rec;
    }
    time = clock() - time;

    printf("IPv4 tcp: %f\n", (double)time);
    char * filename = "newfile_tcp4";
    if (quiet) calculate_file_checksum(filename);
    fclose(f);
    close(connect_sock);

}

void client_ipv6_tcp(int port,const char* ip)
{

    fflush(stdout);
    int sock_addr6 = socket(AF_INET6, SOCK_STREAM, 0);
    if(sock_addr6 < 0){
```

```
        perror("eror socket ");
        exit(1);
    }

    struct sockaddr_in6 server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin6_family = AF_INET6;
    inet_pton(AF_INET6, ip, &server_addr.sin6_addr);
    server_addr.sin6_port = htons(port);

    if(connect(sock_addr6, (struct  sockaddr *)&server_addr, sizeof(server_addr)) < 0)
    {
        perror("eror connect ");
        exit(1);    }

    generate_file();
    int f = open("newfile.bin", O_RDONLY);
    off_t offset = 0;
    int sent;
    sleep(3);
    while( (sent = sendfile(sock_addr6, f, &offset,BUFFER_SIZE ))  > 0 ) {

        if(sent == -1){
            perror("eror send file ");
            exit(1);
        }
    }
    printf("file transeref\n");
    sleep(2);
    close(f);
    close(sock_addr6);
}

void server_ipv6_tcp(int port, int quiet)
{
    int server_sock = socket(AF_INET6, SOCK_STREAM, 0);
    if (server_sock < 0)
    {
        perror("error socket ");
        exit(1);
    }

    struct sockaddr_in6 server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin6_family = AF_INET6;
    server_addr.sin6_addr = in6addr_any;
    server_addr.sin6_port = htons(port);

    if(bind(server_sock, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
    {
        perror("error bind");
        exit(1);
    }

    if(listen(server_sock, 5) < 0)
    {
        perror("error listen");
```

```
        exit(1);
    }


    struct sockaddr_in6 client_addr;
    socklen_t client_addr_len = sizeof(client_addr);
    int connect_sock = accept(server_sock, (struct sockaddr *)&client_addr, &client_addr_len);
    if (connect_sock < 0)
    {
        perror("error accept");
        exit(1);
    }

    generate_file();
    char buf[BUFFER_SIZE];
    int sfile = SIZE;
    FILE * f = fopen("newfile_tcp6", "wb");
    int rec;
    sleep(1);
    clock_t time = clock();

    while(sfile > 0 ){
        rec = recv(connect_sock, buf, BUFFER_SIZE, 0);
        if(rec == -1){
            perror("error recive");
            exit(1);
        }
        sfile -= rec;
    }
    time = clock() - time;

    printf("IPv6 tcp: %f\n", (double)time);
    if(quiet) calculate_file_checksum("newfile_tcp6");
    fclose(f);
    close(connect_sock);
}

void client_ipv4_udp(int port, const char* ip)
{
    int sock_addr = socket(AF_INET, SOCK_DGRAM ,0);
    if(sock_addr < 0){
        perror("eror socket ");
        exit(1);

    }
    struct sockaddr_in server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr =inet_addr(ip);
    server_addr.sin_port = htons(port);

    generate_file();
    char buf[BUFFER_SIZE];
    FILE * f = fopen("newfile.bin", "rb");
    int sent, len;

    sleep(5);
```

```
    while( (len = fread(buf, BUFFER_SIZE, 1, f ))  > 0 ) {
        sent = sendto(sock_addr, buf, BUFFER_SIZE, 0, (struct sockaddr*)&server_addr, sizeof(server_addr));
        if (sent < 0)
        {
            perror("error sentdo");
            exit(1);
        }
    }
    printf("sent file\n");
    sleep(3);
    fclose(f);
    close(sock_addr);
}

void server_ipv4_udp(int port, int quiet){
    int sock_addr = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock_addr < 0){
        perror("error socket");
        exit(1);
    }

    struct sockaddr_in server_addr;
    char buf[BUFFER_SIZE];
    FILE * f = fopen("newfile_udp4", "wb");
    int len;
    socklen_t client_addr_len;
    struct sockaddr_in client_addr;
    memset(&server_addr, 0 ,sizeof(server_addr));
    memset(&client_addr, 0 ,sizeof(client_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(port);


    if (bind(sock_addr, (struct sockaddr*)&server_addr, sizeof(server_addr)) < 0)
    {
        perror("err bind!");
        exit(1);
    }
    sleep(2);
    clock_t time = clock();

    while(1)
    {
        len = recvfrom(sock_addr, buf, BUFFER_SIZE, 0, (struct sockaddr*)&client_addr, &client_addr_len);
        if (len < 0)
        {
            perror("error recvfrom ");
            exit(1);
        }
        fwrite(buf, 1 , len , f);
        memset(buf, 0, BUFFER_SIZE);
        len -= len;
        if ( len <= BUFFER_SIZE) break;
    }
    time = clock() - time;
    printf("IPv4 udp time: %f\n", (double)time);
```

```
    if(quiet) calculate_file_checksum("newfile_udp4");
    fclose(f);
    close(sock_addr);
}

void client_ipv6_udp(int port, const char* ip)
{
    int sock_addr = socket(AF_INET6, SOCK_DGRAM ,0);
    if(sock_addr < 0){
        perror("eror socket ");
        exit(1);

    }
    struct sockaddr_in6 server_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin6_family = AF_INET6;
    inet_pton(AF_INET6, ip, &server_addr.sin6_addr);
    server_addr.sin6_port = htons(port);

    generate_file();
    char buf[BUFFER_SIZE];
    FILE * f = fopen("newfile.bin", "rb");
    int sent, len;
    sleep(1);
    while( (len = fread(buf, 1, BUFFER_SIZE, f ))  > 0 ) {
        sent = sendto(sock_addr, buf, len, 0, (struct sockaddr*)&server_addr, sizeof(server_addr));
        if (sent < 0)
        {
            perror("error sentdo");
            exit(1);
        }
    }
    printf("sent file\n");
    sleep(1);
    fclose(f);
    close(sock_addr);
}

void server_ipv6_udp(int port, int quiet){
    int sock_addr = socket(AF_INET6, SOCK_DGRAM, 0);
    if (sock_addr < 0){
        perror("error socket");
        exit(1);
    }

    struct sockaddr_in6 server_addr;
    struct sockaddr_in6 client_addr;
    memset(&server_addr, 0, sizeof(server_addr));
    memset(&client_addr, 0, sizeof(client_addr));
    server_addr.sin6_family = AF_INET6;
    server_addr.sin6_addr = in6addr_any;
    server_addr.sin6_port = htons(port);

    if(bind(sock_addr, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0)
    {
        perror("error bind");
```

```
        exit(1);
    }

    char buf[BUFFER_SIZE];
    FILE * f = fopen("mewfile_udp6", "wb");
    int len;
    socklen_t client_addr_len;

    clock_t time = clock();
    while(1)
    {
        len = recvfrom(sock_addr, buf, BUFFER_SIZE, 0, (struct sockaddr*)&client_addr, &client_addr_len);
        if (len < 0)
        {
            perror("error recvfrom ");
            exit(1);
        }
        fwrite(buf, 1 , len , f);
        memset(buf, 0, BUFFER_SIZE);
        len -= len;
        if ( len <= BUFFER_SIZE) break;
    }
    time = clock() - time;
    printf("IPv6 udp,%f\n", (double)time);

    if(quiet) calculate_file_checksum("newfile_udp6");
    fclose(f);
    close(sock_addr);
}

void server_mmap(char * filename, int quiet){
    int fd = open(filename, O_RDWR | O_CREAT, 0666);
    if (fd == -1)
    {
        perror("open error");
        exit(1);
    }

    if(ftruncate(fd, SIZE) < 0)
    {
        perror("error resize");
        exit(1);
    }

    void * mmmap = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd , 0);
    if (mmmap == MAP_FAILED)
    {
        perror("mmap err");
        exit(1);
    }
    sleep(2);
    int new_fd = open("mmap.bin", O_WRONLY | O_CREAT, 0666);
    clock_t time = clock();
    if (new_fd == -1)
    {
        perror("err");
        exit(1);
```

```
    }
    if(write(new_fd, mmmap, SIZE) == -1)
    {
        perror("err");
        exit(1);
    }

    time = clock() - time;
    double total_time = (double)(time);
    printf("mmap time %f\n", total_time);
    if(quiet) calculate_file_checksum("mmap.bin");
    if(munmap(mmmap, SIZE) == -1)
    {
        perror("munmap err");
        exit(1);
    }
    close(fd);
    close(new_fd);

}

void client_mmap(char * argv[]){
    int fd = open("mmap", O_RDWR, S_IRUSR | S_IWUSR);
    if (fd == -1)
    {
        perror("err");
        exit(1);
    }

    char *mmmap = mmap(NULL,SIZE,PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if ( mmmap == MAP_FAILED)
    {
        perror("mmap err");
    }

    generate_file();
    char buff[BUFFER_SIZE];
    FILE * f = fopen("mmap", "rb");
    if ( !f)
    {
        perror("open err");
        exit(1);
    }

    while(fread(buff, 1 , BUFFER_SIZE, f) > 0) memcpy(mmmap, buff, BUFFER_SIZE);

    if (munmap(mmmap, SIZE) == -1)
    {
        perror("munmap err");
        exit(1);
    }
    close(fd);
}

void client_pipe(char * argv[])
{
    printf("in client pipe\n");
```

```
    int fd = open(argv[6], O_WRONLY);
    if (fd == -1)
    {
        perror("open error");
        exit(1);
    }

    FILE * f;
    f = fopen("file.txt", "r");
    if (!f)
    {
        perror("fopen error");
        exit(1);
    }

    generate_file();
    char buff[BUFFER_SIZE];
    ssize_t bytes;
    sleep(2);
    while ((bytes = fread(buff, sizeof(char), sizeof(buff), f)) > 0)
    {
        if (write(fd, buff, bytes) != bytes)
        {
            perror("write error");
            exit(1);
        }
    }
    close(fd);
    fclose(f);
}

void server_pipe(char* filename, int quiet){
    FILE *f;
    char buff[BUFFER_SIZE];
    mkfifo(PIPE, 0666);

    f = fopen(filename, "w");
    if(f == NULL)
    {
        perror("err file");
        exit(1);
    }
    int fd = open(PIPE, O_RDONLY);
    clock_t time = clock();
    int bytes;
    while((bytes = read(fd, buff, BUFFER_SIZE))>0)
    {
        fwrite(buff, sizeof(char), bytes, f);
    }
    time = clock() - time;
    printf("pipe, %f\n", (double)time);
    if(quiet) calculate_file_checksum("pipe.bin");
    fclose(f);
    close(fd);
}
```

```
void client(const char *ip, int port)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("ERROR opening socket");
        exit(1);
    }

    struct sockaddr_in serv_addr;
    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(ip);
    if (inet_pton(AF_INET, ip, &serv_addr.sin_addr) <= 0)
    {
        perror("ERROR invalid address");
        exit(1);
    }

    if (connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        perror("ERROR connecting");
        exit(1);
    }
    printf("Connected to server %s:%d\n", ip, port);

    struct pollfd fds[2];
    fds[0].fd = STDIN_FILENO;
    fds[0].events = POLLIN;
    fds[0].revents = 0;
    fds[1].fd = sockfd;
    fds[1].events = POLLIN | POLLHUP;
    fds[1].revents = 0;

    char buffer[BUFFER_SIZE];
    while (1)
    {
        int ret = poll(fds, 2, 500); // Add timeout of 500 milliseconds
        if (ret < 0)
        {
            perror("ERROR polling");
            break;
        }
        else if (ret == 0)
        {
            // Timeout expired and no events to report
            continue;
        }

        if (fds[0].revents & POLLIN)
        {
```

```
            memset(buffer, 0, BUFFER_SIZE);
            fgets(buffer, BUFFER_SIZE - 1, stdin);
            if (write(sockfd, buffer, strlen(buffer)) < 0)
            {
                perror("ERROR writing to socket");
                break;
            }
        }

        if (fds[1].revents & POLLIN)
        {
            memset(buffer, 0, BUFFER_SIZE);
            if (read(sockfd, buffer, BUFFER_SIZE - 1) < 0)
            {
                perror("ERROR reading from socket");
                break;
            }
            printf("Server sent: %s", buffer);
        }

        if (fds[1].revents & POLLHUP)
        {
            printf("Connection closed from server side\n");
            close(sockfd);
            break;
        }
    }

    close(sockfd);
}

void client_perf(char* argv[]){
    int port = atoi(argv[3]);
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));

    serv_addr.sin_family = AF_INET;
    // serv_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
    serv_addr.sin_port = htons(port);
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if(sockfd == -1)
    {
        perror("err sock");
        exit(1);
    }

    if(connect(sockfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr))< 0)
    {
        printf("%s", argv[3]);
        perror("err connect");
        exit(1);
    }

    if(strcmp(argv[5],"ipv4") == 0)
    {
        if(strcmp(argv[6], "tcp") == 0)
        {
```

```
            send(sockfd, "ipv4 tcp", strlen("ipv4 tcp"), 0);
            client_ipv4_tcp(port+1, argv[2]);
        }
        else if (strcmp(argv[6], "udp") == 0) {
            send(sockfd, "ipv4 udp", strlen("ipv4 udp"), 0);
            client_ipv4_udp(port+1, argv[2]);
        }
    }
    else if(strcmp(argv[5], "ipv6")==0)
    {
        if(strcmp(argv[6], "tcp") == 0)
        {
            send(sockfd, "ipv6 tcp", strlen("ipv6 tcp"), 0);
            client_ipv6_tcp(port+1, argv[2]);
        }
        else if (strcmp(argv[6], "udp") == 0)
        {
            send(sockfd, "ipv6 udp", strlen("ipv6 udp"), 0);
            client_ipv6_udp(port+1, argv[2]);
        }
    }

    else if(strcmp(argv[5], "mmap") == 0)
    {
        send(sockfd, "mmap", strlen("mmap"), 0);
        client_mmap(argv);
    }
    else if(strcmp(argv[5], "pipe") == 0)
    {
        send(sockfd, "pipe", strlen("pipe"), 0);
        client_pipe(argv);
    }

    else if(strcmp(argv[5],"uds")==0){
        if(strcmp(argv[6],"stream")==0)
        {
            send(sockfd,"uds stream", strlen("uds stream"), 0);
            client_udsstream();
        }
        else if(strcmp(argv[6],"dgram")==0)
        {
            send(sockfd,"uds dgram", strlen("uds dgram"), 0);
            client_udsdgram();
        }
    }

}
```

```
void server(int port)
{
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("ERROR opening socket");
        exit(1);
    }

    int enable = 1;
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0)
    {
        perror("ERROR setting socket options");
        exit(0);
    }
    struct sockaddr_in serv_addr;
    memset(&serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(port);

    if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        perror("ERROR on binding");
        exit(1);
    }

    if (listen(sockfd, 5) < 0)
    {
        perror("ERROR on listening");
        exit(1);
    }
    printf("Server is listening on port %d\n", port);

    while (1)
    {
        int clientfd = accept(sockfd, NULL, NULL);
        if (clientfd < 0)
        {
            perror("ERROR on accept");
            exit(1);
        }
        printf("Client connected!\n");

        struct pollfd fds[2];
        fds[0].fd = STDIN_FILENO;
        fds[0].events = POLLIN;
        fds[1].fd = clientfd;
        fds[1].events = POLLIN;

        char buffer[BUFFER_SIZE];
        while (1)
        {
```

```
            if (poll(fds, 2, -1) < 0)
            {
                perror("ERROR polling");
                exit(1);
            }

            if (fds[0].revents & POLLIN)
            {
                memset(buffer, 0, BUFFER_SIZE);
                fgets(buffer, BUFFER_SIZE - 1, stdin);
                if (write(clientfd, buffer, strlen(buffer)) < 0)
                {
                    perror("ERROR writing to socket");
                    exit(1);
                }
            }

            if (fds[1].revents & POLLIN)
            {
                memset(buffer, 0, BUFFER_SIZE);
                ssize_t n = read(clientfd, buffer, BUFFER_SIZE - 1);
                if (n < 0)
                {
                    perror("ERROR reading from socket");
                    exit(1);
                }
                else if (n == 0)
                {
                    printf("Client disconnected\n");
                    close(clientfd);
                    break;
                }
                printf("Client sent: %s", buffer);
            }

            if (fds[1].revents & POLLHUP)
            {
                printf("client disconnected\n");
                close(clientfd);
                break;
            }

            if (poll(fds, 2, 0) == 0)
            { // check if timeout expired and no events to report
                if (clientfd < 0)
                { // if the socket is closed, the client has disconnected
                    printf("client disconnected\n");
                    break;
                }
            }
        }
    }

    close(sockfd);
}

void server_perf(char* argv[], int quiet)
```

```
{

    int port = atoi(argv[2]);
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        perror("ERROR opening socket");
        exit(1);
    }

    int enable = 1;
    if (setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR, &enable, sizeof(int)) < 0)
    {
        perror("ERROR setting socket options");
        exit(0);
    }

    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(port);

    if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        perror("ERROR on binding");
        exit(1);
    }

    if (listen(sockfd, 5) < 0)
    {
        perror("ERROR on listening");
        exit(1);
    }

    struct sockaddr_in client_addr;
    socklen_t client_addr_size = sizeof(client_addr);
    int client_sock = accept(sockfd, (struct sockaddr*) & client_addr, &client_addr_size);

    if(client_sock == -1)
    {
        perror("err sock accept");
        exit(1);
    }

    char buff[BUFFER_SIZE];
    int bytes = recv(client_sock, &buff, sizeof(buff), 0);
    if(bytes == -1){
        perror("err recv");
        exit(1);
    }

    buff[bytes] = '\0';

    if(strcmp(buff,"ipv4 tcp") == 0)
    {
        server_ipv4_tcp(port + 1, quiet);
```

```
        return;
    }

    else if(strcmp(buff,"ipv4 udp") == 0)
    {
        server_ipv4_udp(port + 1, quiet);
        return;
    }

    else if(strcmp(buff,"ipv6 tcp") == 0)
    {
        server_ipv6_tcp(port + 1, quiet);
        return;
    }

    else if(strcmp(buff,"ipv6 udp") == 0)
    {
        server_ipv6_udp(port + 1, quiet);
        return;
    }

    else if(strcmp(buff, "uds stream") == 0)
    {
        server_udsstream(quiet);
        return;
    }

    else if(strcmp(buff, "uds dgram") == 0)
    {
        server_udsdgram(quiet);
        return;
    }

    else if(strcmp(buff, "mmap") == 0)
    {
        char *filename = "mmap.bin";
        server_mmap(filename, quiet);
        return;
    }

    else if(strcmp(buff, "pipe") == 0)
    {
        char *filename = "pipe.bin";
        server_pipe(filename, quiet);
        return;
    }
  close(sockfd);
}
```

```
int main(int argc, char *argv[]){
    int perf = 0, quiet = 0;

    if (argc < 3 || argc > 7)
    {
        printf("Usage: %s [-c IP PORT|-s PORT]\n", argv[0]);
        exit(1);
    }

    for (int i = 0; i < argc; i++)
    {
        if (strcmp(argv[i], "-p") == 0) {
            perf = 1;

        }
        else if (strcmp(argv[i], "-q") == 0) quiet = 1;
    }

    int is_client = 0;
    const char *ip = NULL;
    int port;

    if (strcmp(argv[1], "-c") == 0)
    {
        is_client = 1;
        ip = argv[2];
        port = atoi(argv[3]);

        if(!perf)
        {
            client(ip, port);
            return 0;
        }
        client_perf(argv);
        return 0;
    }
    else if (strcmp(argv[1], "-s") == 0){
        is_client = 0;
        port = atoi(argv[2]);
    }
    else {
        printf("Usage: %s [-c IP PORT|-s PORT]\n", argv[0]);
        exit(1);
    }
    if (is_client == 1) {
        client(ip, port);
    }
    else{
        if(perf == 0) server(port);
        else server_perf(argv, quiet);
    }
    return 0;
}
```

ASSIGNMENT 4:
Description about the assignment:
This project is a chat that supports unlimited amount to clients while using poll and reactor.

Reactor library has some functinos which are:
A) void* createReactor
 • Creates a Reactor, returns a pointer to the Reactor structure that will be redirected to the following functions.
When the Reactor is created he doesnt work but some of the data structures will be init'd and allocated

B) void stopReactor
 • stops the Reactor if it's on. otherwise does nothing

C) void startReactor
 • Starts a thread of Reactor. The thread will be in a busy loop and will call select or poll. (in our project will be poll)

D) void addFd
 • When the handler_t is a pointer to a function that will call when the fd is 'hot'.
   which means add the fd to the Reactor.

E) void WaitFor
 • Waits untill the Reactor thread will finish and then join it.

F) void* handleThreads
 • Handles threads check if there is a process that has available information and predicts
   between processes.
H) void removeFD
 • responsible of removing specific file descriptor from the Reactor.

pollserver - During the process of the project we used pollserver file from slide of lesson 5 which found in the moodle.
we predicted the functions and divided it to handlers functions.
we used signal to use SIGKILL SIGINT to catch errors from the server side or it's being terminated.

HOW TO RUN??
make all - creates all the server files
gcc -o client client.c

creation of a client
client 127.0.0.1

STREACTOR.C

```
#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <pthread.h> #include "st_reactor.h"
void *createReactor()
{
   reactor_t *reactor = malloc(sizeof(reactor_t));
   reactor->fd_count = 0;
   reactor->fd_size = 5;
   reactor->pfds = malloc(sizeof(*reactor->pfds) * reactor->fd_size);
   reactor->remoteIP = malloc(INET6_ADDRSTRLEN);
   reactor->handlers = malloc(sizeof(*reactor->handlers) * reactor->fd_size);
   reactor->flag = 0;
   reactor->listener = 0;
   return reactor;
}

void *handleThreads(void *arg)
{
   reactor_t *reactor = (reactor_t *)arg;
   while (reactor->flag == 1)
   {
      int poll_count = poll(reactor->pfds, reactor->fd_count, -1);
      if (poll_count == -1)
      {
         perror("poll");
         exit(1);
      }
      for (int i = 0; i < reactor->fd_count; i++)
      {
         if (reactor->pfds[i].revents & POLLIN) // Check if someone's ready to read
         { // We got one!!
            if (reactor->pfds[i].fd == reactor->listener)
            {
               reactor->handlers[i](reactor->pfds[i].fd, reactor);
               break;
            }
            else   {
               reactor->handlers[i](reactor->pfds[i].fd, reactor);
            }
         }
      }
   }
   return NULL;
}

void startReactor(void *this){
   reactor_t *reactor = (reactor_t *)this;

   if (reactor == NULL || reactor->flag == 1)
      return;
   reactor->flag = 1;
   int num = pthread_create(&reactor->thread, NULL, handleThreads, reactor);
   if (num != 0) {
      perror("thread create");
      exit(0);
   }
}
```

```
void stopReactor(void *this){
    reactor_t *reactor = (reactor_t *)this;
    if (reactor == NULL || reactor->flag == 0)
        return;
    reactor->flag = 0;
    pthread_cancel(reactor->thread);
    pthread_join(reactor->thread, NULL);
    printf("Reactor stopped\n");
}

void addFd(void *this, int fd, handler_t handler){
    reactor_t *reactor = (reactor_t *)this;
    if(reactor->listener == 0)
        reactor->listener = fd;
    if (reactor->fd_count == reactor->fd_size)
    {
        reactor->fd_size *= 2; // Double it
        reactor->pfds = realloc(reactor->pfds, sizeof(*reactor->pfds) * (reactor->fd_size));
        reactor->handlers = realloc(reactor->handlers, sizeof(handler_t) * reactor->fd_size);
    }
    reactor->handlers[reactor->fd_count] = handler;
    reactor->pfds[reactor->fd_count].fd = fd;
    reactor->pfds[reactor->fd_count].events = POLLIN; // Check ready-to-read

    reactor->fd_count++;
}

void waitFor(void *this){
    reactor_t *reactor = (reactor_t *)this;
    if (reactor == NULL)
        return;
    if (reactor->flag)
    {
        int num = pthread_join(reactor->thread, NULL);
        if (num != 0)
        {
            perror("wait for");
            exit(0);
        }
    }
}

void removeFd(void *this, int fd){
    reactor_t *reactor = (reactor_t *)this;

    if (reactor == NULL || fd < 0 || fd >= reactor->fd_count)
    {
        return;
    }
    reactor->pfds[fd] = reactor->pfds[reactor->fd_count - 1];
    reactor->handlers[fd] = reactor->handlers[reactor->fd_count - 1];
    reactor->handlers[reactor->fd_count - 1] = NULL;
    reactor->fd_count--;
}
```

POLLSERVER.C

```
#include "st_reactor.h" #include <signal.h>
#define PORT "9034"

reactor_t* myReactor;
// Get sockaddr, IPv4 or Ipv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}
// Return a listening socket

int get_listener_socket(void)
{
    int listener;     // Listening socket descriptor
    int yes=1;        // For setsockopt() SO_REUSEADDR, below
    int rv;
    struct addrinfo hints, *ai, *p;

    // Get us a socket and bind it
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
        exit(1);
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (listener < 0) {
            continue;
        }

        // Lose the pesky "address already in use" error message
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(listener);
            continue;
        }
        break;
    }

    // If we got here, it means we didn't get bound
    if (p == NULL) {
        return -1;
    }

    freeaddrinfo(ai); // All done with this
    // Listen
    if (listen(listener, 10) == -1) {
```

```
        return -1;
    }
    return listener;
}


void sighandler(int sig)
{
   if(sig == SIGKILL || sig == SIGINT )
   {
    printf("\n");
    stopReactor(myReactor);
    if(myReactor == NULL)
        return;
    if(myReactor->flag)
    waitFor(myReactor);
    free(myReactor->handlers);
    free(myReactor->pfds);
    free(myReactor->remoteIP);
    free(myReactor);
    printf("Server closed\n");
    close()
    exit(0);
   }
}

void handleClient(int fd, void *arg){
    char buf[256];    // Buffer for client data
    reactor_t* reactor = (reactor_t*)arg;
    int nbytes = recv(fd, buf, sizeof buf, 0);
    if (nbytes <= 0) {
    // Got error or connection closed by client
        if (nbytes == 0)  // Connection closed
            printf("pollserver: socket %d hung up\n", fd);
        else
            perror("recv");
        close(fd); // Bye!
        removeFd(reactor, fd);
        }
        else {// We got some good data from a client
            printf ("Client %d: %s\n",fd,buf);
            for(int j = 0; j < reactor->fd_count; j++) { // Send to everyone!
                int dest_fd = reactor->pfds[j].fd;
                // Except the listener and ourselves
                if (dest_fd != reactor->listener && dest_fd != fd) {
                    if (send(dest_fd, buf, nbytes, 0) == -1) {
                        perror("send");
                    }
                }
            }
        }
} // END handle data from client
```

```
void handleNewClient(int fd, void *arg){
    printf("in handle new\n");
    reactor_t* reactor = (reactor_t*) arg;
    int listener = reactor->listener;     // Listening socket descriptor
    int newfd;        // Newly accept()ed socket descriptor
    struct sockaddr_storage remoteaddr; // Client address
    socklen_t addrlen;
    addrlen = sizeof remoteaddr;
    newfd = accept(listener,(struct sockaddr *)&remoteaddr,&addrlen);
    if (newfd == -1) {
        perror("accept eror");
    }
    else {
    addFd(reactor, newfd,handleClient);
    printf("pollserver: new connection from %s on socket %d\n",
        inet_ntop(remoteaddr.ss_family,get_in_addr((struct sockaddr*)&remoteaddr),
        reactor->remoteIP, INET6_ADDRSTRLEN),newfd);
    }
}

int main(void)
{
    printf("Start\n");
    int listener;
    myReactor = createReactor();
    signal(SIGINT, sighandler);
    listener = get_listener_socket();
    if (listener == -1) {
        fprintf(stderr, "error getting listening socket\n");
        exit(1);
    }
    printf("listen...\n");
    addFd(myReactor, listener, handleNewClient);
    startReactor(myReactor);
    waitFor(myReactor);
    return 0;
}
```

POLLSERVER.C

```
#include "st_reactor.h" #include <signal.h>
#define PORT "9034"

reactor_t* myReactor;
// Get sockaddr, IPv4 or IPv6:
void *get_in_addr(struct sockaddr *sa)
{
    if (sa->sa_family == AF_INET) {
        return &(((struct sockaddr_in*)sa)->sin_addr);
    }

    return &(((struct sockaddr_in6*)sa)->sin6_addr);
}

// Return a listening socket
int get_listener_socket(void)
{
    int listener;     // Listening socket descriptor
    int yes=1;        // For setsockopt() SO_REUSEADDR, below
    int rv;
    struct addrinfo hints, *ai, *p;

    // Get us a socket and bind it
    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;
    if ((rv = getaddrinfo(NULL, PORT, &hints, &ai)) != 0) {
        fprintf(stderr, "selectserver: %s\n", gai_strerror(rv));
        exit(1);
    }

    for(p = ai; p != NULL; p = p->ai_next) {
        listener = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
        if (listener < 0) {
            continue;
        }

        // Lose the pesky "address already in use" error message
        setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int));

        if (bind(listener, p->ai_addr, p->ai_addrlen) < 0) {
            close(listener);
            continue;
        }

        break;
    }

    // If we got here, it means we didn't get bound
    if (p == NULL) {
        return -1;
    }

    freeaddrinfo(ai); // All done with this
```

```
    // Listen
    if (listen(listener, 10) == -1) {
        return -1;
    }

    return listener;
}

void sighandler(int sig)
{
   if(sig == SIGKILL || sig == SIGINT )
   {
    printf("\n");
    stopReactor(myReactor);
    if(myReactor == NULL)
        return;
    if(myReactor->flag)
    waitFor(myReactor);
    free(myReactor->handlers);
    free(myReactor->pfds);
    free(myReactor->remoteIP);
    free(myReactor);
    printf("Server closed\n");
    close()
    exit(0);
   }
}

void handleClient(int fd, void *arg){
    char buf[256];    // Buffer for client data
    reactor_t* reactor = (reactor_t*)arg;
    int nbytes = recv(fd, buf, sizeof buf, 0);
    if (nbytes <= 0) {
    // Got error or connection closed by client
        if (nbytes == 0)  // Connection closed
            printf("pollserver: socket %d hung up\n", fd);
        else
            perror("recv");
        close(fd); // Bye!
        removeFd(reactor, fd);
        }
        else {// We got some good data from a client
            printf ("Client %d: %s\n",fd,buf);
            for(int j = 0; j < reactor->fd_count; j++) { // Send to everyone!
                int dest_fd = reactor->pfds[j].fd;
                // Except the listener and ourselves
                if (dest_fd != reactor->listener && dest_fd != fd) {
                    if (send(dest_fd, buf, nbytes, 0) == -1) {
                        perror("send");
                    }
                }
            }
        }
} // END handle data from client
```

```
void handleNewClient(int fd, void *arg){
    printf("in handle new\n");
    reactor_t* reactor = (reactor_t*) arg;
    int listener = reactor->listener;     // Listening socket descriptor
    int newfd;        // Newly accept()ed socket descriptor
    struct sockaddr_storage remoteaddr; // Client address
    socklen_t addrlen;
    addrlen = sizeof remoteaddr;
    newfd = accept(listener,(struct sockaddr *)&remoteaddr,&addrlen);
    if (newfd == -1) {
        perror("accept eror");
    }
    else {
    addFd(reactor, newfd,handleClient);
    printf("pollserver: new connection from %s on socket %d\n",
        inet_ntop(remoteaddr.ss_family,get_in_addr((struct sockaddr*)&remoteaddr),
        reactor->remoteIP, INET6_ADDRSTRLEN),newfd);
    }
}

int main(void){
    printf("Start\n");
    int listener;
    myReactor = createReactor();
    signal(SIGINT, sighandler);
    listener = get_listener_socket();

    if (listener == -1) {
        fprintf(stderr, "error getting listening socket\n");
        exit(1);
    }
    printf("listen...\n");
    addFd(myReactor, listener, handleNewClient);
    startReactor(myReactor);
    waitFor(myReactor);
    return 0;
}
```

MAKEFILE

```
CC = gcc
WALL = -Wall -Werror -fpic
LFLAGS = -shared
LIBRARY = st_reactor.so
TARGET = react_server
SRCS = pollserver.c st_reactor.c
OBJS = $(SRCS:.c=.o)
all: $(LIBRARY) $(TARGET)
$(LIBRARY): st_reactor.o
        $(CC) $(LFLAGS) -o $@ $<
$(TARGET): $(OBJS)
        $(CC) $(WALL) -o $@ $(OBJS) -L. -l:$(LIBRARY)
%.o: %.c
        $(CC) $(WALL) -c $< -o $@

clean:
        rm -f $(OBJS) $(LIBRARY) $(TARGET)
```

ASSIGMNENT 5:

multi-threading, synchronization,active object design patterns. involves implementing a multi-threaded pipeline of active objects that perform tasks on a sequence of numbers, checking if a number is prime, manipulating the numbers, and passing them between the active objects.

TASK A:
Implement a function that takes an unsigned int as input and checks if the number is prime. The function should return 0 if the number is not prime. We check wether the number is prime without using Miller-Rabin method.

Task B:
Implement a thread-safe queue in a multi-threaded environment. The queue should have mutex and allow waiting for an item in the queue without busy waiting. You can use a conditional variable (cond) to achieve this. The queue should hold elements of type void*.
- cond: initialized everytime one thread at a time from the threads that await by the signal, and also be responsible for the threads conditions.
- mutex: At most one thread uses a particular resouce at any given time in critical time. This requirement is concurrency control and the purpose is to prevent thread race situation.

Task C:
Implement an active object that supports the following functions:
a. CreateActiveObject: This function should create and run a thread for the ActiveObject. the function should enqueue a task and receive a pointer to a function to be called for each item in the queue.
b. getQueue: This function return a pointer to the queue of the ActiveObject passed as a parameter (this). It can be used to add an item to the queue.
c. stop: This function should stop the ActiveObject passed as a parameter (this). Additionally, it should release all memory resources of the object.
- Queue: this is normal queue that uses mutex and cond as explained above, that represents multi-thread environment.
-Active Object: creating an Active Object that posesses function and AO for continuation, it runs untill it gets NULL.

Task D:
Use the functions implemented in the previous tasks to create a program called st_pipeline. The program takes one or two arguments from the command line: N, representing the number of tasks, and an optional random seed. If no random seed is provided, it can be generated using null or time function.

A pipeline is built using Active Objects (AO), forming a collection of Active Objects. The pipeline should be constructed as follows:

The first AO:

Initialize the random number generator with the given seed and generate N six-digit random numbers (e.g., 3 + 3 using .rand).
Pass each number one by one to the next AO with a time delay of one millisecond (1ms).
The second AO:

Print the number received.
Check if the number is prime and print "true" or "false" accordingly.
Add 11 to the number and pass it to the next AO.
The third AO:

Print the number received.
Check if the number is prime and print "true" or "false" accordingly.
Subtract 13 from the number and pass it to the next AO.
The fourth AO:

Print the received number.
Add 2 to the number and print the new number.
If implemented correctly, the last number printed should be the same as the first number.

HOW TO RUN??
make all
./st_pipeline 3

ACTIVEOBJECT.C

```
typedef struct AO
{
   pqueue queue;
   pthread_t thread;
   void (*func)( void*);
   struct AO* next;
   int flag;
}ao, *pao;
void* run(void* arg);
pao createActiveObject( void (*func)(void*), pao next);
pqueue getQueue(pao obj);
void stop(pao obj);
```

```
void* run(void* arg)
{
   pao this = (pao) arg;
   void * obj;
   while((obj = dequeue(this->queue)) != NULL) {
      this->func(obj);
      if(this->flag != 1 && this->next != NULL){
         enqueue(this->next->queue, obj);
      }
   }
   if(this->next != NULL)
      enqueue(this->next->queue, obj);
   return NULL;
}

pao createActiveObject(void  (*func)(void*), pao next)
{
   pao obj = (pao)malloc(sizeof(ao));
   if(obj == NULL) return NULL;
   obj->queue = createQueue();
   obj->next = next;
   obj->func = func;
   pthread_create(&obj->thread, NULL, run, obj);
   return obj;
}

pqueue getQueue(pao obj)
{
   return obj->queue;
}

void stop(pao obj)
{
   pthread_join(obj->thread, NULL);
   deleteQueue(obj->queue);
   free(obj);
}
```

QUEUE.C

```
typedef struct Node
{
   struct Node *next;
   void *task;
} Node, *pnode;

typedef struct Queue
{
   pnode head;
   pnode tail;
   pthread_mutex_t mutex;
   pthread_cond_t cond;
} Queue, *pqueue;
```

```
pqueue createQueue() {
   pqueue queue = malloc(sizeof(Queue));
   if (queue == NULL)
   {
      perror("malloc");
      exit(1);
   }

   queue->head = NULL;
   queue->tail = NULL;
   pthread_mutex_init(&queue->mutex, NULL);
   pthread_cond_init(&queue->cond, NULL);

   return queue;
}

void deleteQueue(pqueue queue){
   pthread_mutex_lock(&queue->mutex);
   pnode save = queue->head;
   while (save != NULL)
   {
      pnode next = save->next;
      free(save);
      save = next;
   }
   pthread_mutex_unlock(&queue->mutex);
   pthread_mutex_destroy(&queue->mutex);
   pthread_cond_destroy(&queue->cond);
   free(queue);
}

void enqueue(pqueue queue, void *task)
{
   pnode node = malloc(sizeof(Node));
   if (node == NULL)
   {
      exit(1);
   }
   node->task = task;
   node->next = NULL;
```

```
    pthread_mutex_lock(&queue->mutex);

    if (queue->head == NULL)
    {
        queue->head = node;
        queue->tail = node;
    }
    else
    {
        queue->tail->next = node;
        queue->tail = node;
    }

    pthread_cond_signal(&queue->cond);
    pthread_mutex_unlock(&queue->mutex);
}

void *dequeue(pqueue queue)
{

    pthread_mutex_lock(&queue->mutex);

    while (queue->head == NULL)
    {
        pthread_cond_wait(&queue->cond, &queue->mutex);
    }

    pnode node = queue->head;
    void *save = node->task;
    queue->head = node->next;
    if (queue->head == NULL)
    {
        queue->tail = NULL;
    }

    free(node);
    pthread_mutex_unlock(&queue->mutex);
    return save;
}

int isEmpty(pqueue queue)
{
    pthread_mutex_lock(&queue->mutex);
    int isempty = (queue->head == NULL);
    pthread_mutex_unlock(&queue->mutex);
    return isempty;
}
```

STPIPELINE.C

```
// PART A:
int isPrime(unsigned int number)
{
   if (number == 2)
      return 1;
   if (number < 2 || number % 2 == 0)
      return 0;
   for (unsigned int i = 3; i * i <= number; i += 2)
   {
      if (number % i == 0)
      {
         return 0;
      }
   }
   return 1;
}

// PART D:

void first(void* arg)
{
   void **arr = (void **)arg;
   int N = *(int *)arr[0];
   int seed = *(int *)arr[1];
   pao ao1 = (pao)arr[2];
   srand(seed);
   for (int i = 0; i < N; i++) {
      int number = rand() % 900000 + 100000;
      enqueue(ao1->next->queue,(void *) &number);
      sleep(1);
   }
   enqueue(ao1->queue,NULL);

}

void second( void * arg)
{
   int *number = (int*)arg;
   printf("%d\n", *number);
   if(isPrime(*number))
      printf("true\n");
   else
      printf("false\n");
   *number += 11;
}

void third(void* arg){
   int * number = (int*)arg;
   printf("%d\n", *number);
   if(isPrime(*number))
      printf("true\n");
   else
      printf("false\n");
   *number -= 13;

}
```

```
void fourth(void * arg)
{
    int * number = (int*)arg;
    printf("%d\n", *number);
    if(isPrime(*number))
        printf("true\n");
    else
        printf("false\n");
    *number += 2;
    printf("%d\n", *number);

}


int main(int argc, char *argv[])
{
    if (argc < 2 || argc >3)
    {
        perror("eror with elements\n");
        return 1;
    }
    int N = atoi(argv[1]);
    int seed = argc > 2 ? atoi(argv[2]) : time(NULL);
    void **arr = (void **)malloc(sizeof(void *) * 3);
    if (arr == NULL)
    {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }
    arr[0] =(void*)& N;
    arr[1] = (void*)& seed;
    pao ao4 = createActiveObject(fourth, NULL);
    pao ao3 = createActiveObject(third, ao4);
    pao ao2 = createActiveObject(second, ao3);
    pao ao1 = createActiveObject(first, ao2);
    ao1->flag = 1;
    arr[2] = ao1;
    enqueue(ao1->queue, (void *)arr);
    sleep(1);
    pthread_join(ao1->thread, NULL);
    pthread_join(ao2->thread, NULL);
    pthread_join(ao3->thread, NULL);
    pthread_join(ao4->thread, NULL);
    stop(ao1);
    stop(ao2);
    stop(ao3);
    stop(ao4);
    free(arr);
    return 0;
}
```