시스템프로그래밍

2차 과제 보고서

목차

1. 소스코드 설명

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <sys/timeb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fcntl.h>
#include <unistd.h>

#include <pthread.h>

#define MAX_PORT 9                  // maximum number of ports
#define MAX_CONNECTION 5        // maximum number of connections per ports
#define MAX_MSG 65536             // maximum size of message
```

프로그래밍에 필요한 라이브러리들을 추가한다. 효율적인 프로그래밍을 위해 client의 포트번호, 포트 당 최대 연결의 수, 메세지의 길이 등을 매크로로 선언한다.

다음은 프로그램의 main 함수에 대한 설명이다.

```
int main(int argc, char *argv[]) {

    int client_socket[MAX_PORT];                    // client socket
    int ports[MAX_PORT];                              // port number for each sockets
    char server_IP[20];                               // server ip address
    struct sockaddr_in client_addr[MAX_PORT];   // address for each client
    struct sockaddr_in server_addr;                // server address

    pthread_t p_thread[MAX_PORT];                  // thread
    int status[MAX_PORT];                            // thread status

    int i;                                                 // index for managing ports
    int port_num;                                      // the number of ports
```

위의 코드는 main 함수의 선언부이다. 서버와의 통신을 위해 클라이언트 소켓의 file descriptor를 저장할 배열, 포트 번호들을 저장할 배열, 서버의 IP 주소, 클라이언트 소켓의 주소정보를 저장할 sockaddr 구조체, 서버 소켓의 주소정보를 저장할 sockaddr 구조

체 들을 선언한다. 그리고 각 소켓 통신을 thread로 관리하기위해 p_thread 배열과, 각 thread의 상태를 저장하기 위한 상태 배열을 선언한다. 여러 개의 클라이언트 소켓을 관리하기위한 이터레이터를 위한 변수와 포트의 수를 저장할 변수를 선언한다.

```
printf("Enter server IPv4 address: ");
scanf("%s", server_IP);

// repeat until ctrl-c
while(1){
	int idx_port_mapping[MAX_PORT] = {0,};		// to keep max connection
	int socket_connection_num[MAX_PORT] = {0,}; // the number of connection per socket

	// intialize ports
	port_num = init_ports(ports);

	printf("Open : ");
	// connect each sockets to the server
	for (i = 0; i < port_num; i++){

		// create a socket at port[i]
		create_socket(idx_port_mapping, &client_socket[i], ports[i], socket_connection_num);

		// if created, print port number
		printf("%6d	",ports[i]);

		// initialize server information for connection
		configure_server(&server_addr, server_IP, ports[i]);

		// connect the socket to the server
		connect_server(&server_addr, client_socket[i], ports[i]);

	}
	printf("\n");

	// create threads
	for (i = 0; i < port_num; i++){
		create_thread(&p_thread[i], &client_socket[i], ports[i], i);
	}
```

```
            // wait threads
            for (i = 0; i < port_num; i++){
                pthread_join(p_thread[i], (void *)&status[i]);
            }

            // close sockets
            close_sockets(client_socket, port_num);
        }

        return 0;
}
```

위의 코드는 main함수의 구현부이다. 클라이언트와 서버의 연결을 위해 서버의 IP주소를 사용자로부터 입력 받는다. 서버와의 통신은 사용자가 강제로 프로그램을 종료할 때까지 반복된다.

통신 과정은 다음과 같다. 먼저 각 포트의 연결들을 관리하기위한 배열들을 초기화 한다. init_port 함수를 통해 사용자로부터 포트 번호를 입력 받아 사용할 포트 정보를 저장하고, 함수의 반환 값인 사용 가능한 포트의 수를 저장한다. 각 포트에 대해 create_socket 함수를 통해 포트를 열고 소켓 생성한다. 소켓을 생성할 때, 각 포트의 연결 수를 고려한다. 소켓이 정상적으로 열렸다면 포트 번호를 출력한다. configure_server함수를 통해 해당 소켓에 대해 서버와의 연결을 준비한다. 그리고 connect_server 함수를 통해 해당 소켓과 서버의 연결을 형성한다.

모든 포트에 대해 연결이 형성 됐다면, create_thread 함수를 통해 각 포트에 대해 p_thread를 생성하여 작업을 수행한다. create_thread함수에서는 클라이언트가 받은 서버의 메세지를 읽고, 클라이언트의 로컬에 메세지 내용을 로그로 기록한다.

각 p_thread의 작업이 끝나고 다음 코드를 진행하기 위해, pthread_join 함수를 통해 각 p_thread의 작업이 끝나기를 기다린다.

모든 p_thread의 작업이 끝났다면 close_sockets 함수를 통해 클라이언트의 소켓들을 정리한다.

이제 2017320256_client.c에 구현된 함수들을 자세히 살펴보도록 한다.

```
// initialize ports
//    returns the number of available ports
int init_ports(int *ports) {

    // input : the number of ports client use
    int port_num;
    scanf("%d",&port_num);
```

```
        // input : user defines the port numbers
        for(int i = 0; i < port_num; i++){
                int p;
                scanf("%d",&p);
                ports[i] = p;
        }

        return port_num;
}
```

위의 함수는 사용자로부터 포트 번호를 입력 받아 포트 번호를 관리하는 배열에 사용할 포트 정보를 저장하고, 사용 가능한 포트의 수를 반환한다.

```
// create socket
//    args :    idx_port_mapping -> to check connection number per ports
//              client_socket -> socket fd
//              port -> socket location,
//              socket_connection_num -> connection number per ports
void create_socket(int* idx_port_mapping, int *client_socket, int port, int* socket_connection_num) {
        int loc = -1;                                   // index of port in idx_port_mapping
        int last_idx = MAX_PORT-1;                      // last valid information in idx_port_mapping

        // table lookup for idx-port mapping to manage connection number per ports
        for(int i = 0 ; i < MAX_PORT ; i++){
                if(idx_port_mapping[i]==port){
                        loc = i;
                        break;
                }
                if(idx_port_mapping[i]==0){
                        last_idx = i;
                        break;
                }
        }

        // check if there is available connection, which means ports keeps less than 5 connections.
        if(loc==-1){
                // if the port number never appeared before,
                //    update mapping information and increase connection number.
                idx_port_mapping[last_idx] = port;
                socket_connection_num[last_idx] +=1;
```

```
    }
    else{
        if(socket_connection_num[loc] >= MAX_CONNECTION){
            printf("\nClient : Port %d has no available connection\n", port);
            exit(0);
        }
        // increase connection number of the port
        socket_connection_num[loc] +=1;
    }

    // create an endpoint for communication
    //    args : PF_INET -> IPv4, SOCK_STREAM -> TCP
    *client_socket = socket(PF_INET, SOCK_STREAM, 0);

    // error handling
    //    if impossible to create an socket on selected port
    if (*client_socket == -1)
    {
        printf("\nClient : Can't open stream socket, port %d \n", port);
        exit(0);
    }

    return;
}
```

위의 함수는 포트를 열고, 포트에 소켓 생성한다. 소켓을 생성하기 전, 해당 포트에 연결을 추가해도 되는 지를 확인한다. 만약 해당 포트에 어떠한 소켓도 존재하지 않는다면, 포트의 연결 수를 증가시킨다. 만약 해당 포트에 소켓이 존재한다면, 그 수를 확인하고 과제의 조건인 5개보다 작다면 해당 포트의 연결 수를 증가시킨다. 반면 5개보다 큰 소켓 수를 갖는다면 에러를 발생시키고 프로그램을 종료시킨다. 포트의 연결 수가 무사히 증가했다면 해당 포트에 소켓을 생성한다. 만약 소켓을 생성할 수 없는 상황이라면 에러를 발생시키고 프로그램을 종료한다.

```
// set server information
void configure_server(struct sockaddr_in *server_addr, char *server_IP, int port) {

    memset(server_addr, 0, sizeof(*server_addr));

    server_addr->sin_family = AF_INET;                    // IPv4
    server_addr->sin_port = htons(port);                  // port
```

```
    server_addr->sin_addr.s_addr = inet_addr(server_IP);        // server IP address

    return;
}
```

위의 함수는 클라이언트의 포트와 서버와의 연결을 위해 서버 정보를 세팅하는 함수이다. 클라이언트와 서버의 연결은 IPv4를 통해 이루어지므로, 연결 정보, 서버의 IP주소와 포트 번호를 서버 정보를 관리하는 sockaddr_in 구조체 변수에 저장한다.

```
// connect socket to server
void connect_server(struct sockaddr_in *server_addr, int client_socket, int port) {

    if (connect(client_socket, (struct sockaddr *)server_addr, sizeof(*server_addr)) < 0){
        printf("\nConnection failed, port %d\n", port);
        exit(0);
    }

    return;
}
```

위의 함수는 클라이언트와 서버의 연결을 형성한다. 클라이언트의 소켓의 abstraction인 소켓 file descriptor와 클라이언트의 포트정보, 서버의 주소 정보를 connect함수의 인자로 전달하여 서버에 연결을 요청한다. 서버가 요청을 받아들인다면, 서버와 클라이언트의 연결이 형성된다. 만약 서버가 요청을 받아들이지 않는다면 에러를 발생시키고 프로그램을 종료한다.

```
// for log
char *get_current_time() {

    struct timeb itb;
    struct tm *lt;
    static char s[20];

    ftime(&itb);

    lt = localtime(&itb.time);

    sprintf(s, "%02d:%02d:%02d.%03d", lt->tm_hour, lt->tm_min, lt->tm_sec, itb.millitm);

    return s;
}
```

위의 함수는 현재 시간 정보를 읽고 문자열로 반환하는 함수이다. time라이브러리와 sys/timeb라이브러리에 정의된 구조체 timeb, tm과 ftime, localtime 등의 함수들을 통해 현재 시간을 문자열로 변환하여 저장하고 이를 반환한다. 서버의 메세지를 클라이언트의 로컬에 로그로 저장할 때, 시간 정보도 함께 기록하기 위해 사용된다.

```
// a trigger function for finishing connection between server and client
//    if '@' appears more than 5 times, connection terminated.
int atsign_counting(const char * const buf, size_t len){
        int i;
        int n = 0;
        for (i = 0; i < len; i++) {
                if (buf[i] == '@')
                        n++;
        }
        return n;
}
```

위의 함수는 통신 종료를 확인하기 위한 함수이다. 클라이언트가 서버로부터 “@@@@@”를 받으면 통신을 종료한다. 서버 메세지를 저장한 버퍼를 읽어 ‘@’의 갯수를 확인하여 이를 반환한다.

```
// a struct for implementing function, pthread_create() : args in args
typedef struct _pthread_args {
        int *client_socket;
        int port;
}p_thread_args;
```

위의 구조체는 pthread_create함수를 사용하기 위해 정의한 구조체이다. 이 구조체의 멤버 변수들은 다음에 설명할 server_msg함수의 인자들이다.

```
// Clients receive server's msg
void *server_msg(void *pThreadArgs) {

        int msg_len;                    // message size
        char msg_buffer[MAX_MSG];       // message buffer

        int client_socket_fd = *(((p_thread_args *)pThreadArgs)->client_socket);
        int port_num = ((p_thread_args *)pThreadArgs)->port;

        // initialize log file
```

```
    FILE *fp;
    char log_path[20];

    // set log path
    sprintf(log_path, "./log/%d_%d.txt", port_num,client_socket_fd);

    // open log file
    fp = fopen(log_path, "w");

    if (!fp){
        printf("Client : File open failed\n");
        exit(0);
    }

    while(1){
        int atsign_count = 0;      // trigger variable for terminating connection
        char log_temp[80];    // buffer for writing msg to log, different from msg buffer

        memset(log_temp,0,sizeof(log_temp));
        memset(msg_buffer,0,sizeof(msg_buffer));

        // read msg
        //    receive msg through client_socket_fd,
        //    store it in msg_buffer,
        //    and return the length of received msg
        msg_len = recv(client_socket_fd, msg_buffer, MAX_MSG, 0);

        // write log
        //    write current time, msg size, msg to log file
        sprintf(log_temp, "%s | %d | %s", get_current_time(), msg_len, msg_buffer);
        fputs(log_temp, fp);
        fputs("\n", fp);

        // check if connection finished
        atsign_count += atsign_counting(msg_buffer, msg_len);
        if (atsign_count >= 5){
            fclose(fp);
            return;
        }
    }
```

```
    return;
}
```

위의 함수는 클라이언트의 로컬에 서버의 메세지를 로그로 기록하기 위한 함수이다. 서버의 메세지를 읽을 때 사용할 메세지 버퍼와 메세지의 길이를 저장할 변수를 선언한다. 통신에 사용될 클라이언트의 소켓 file descriptor와 포트 번호를 관리할 변수를 선언하고 인자로부터 값들을 받아온다. 이 값들을 파일 이름으로 하여 로그 파일을 생성한다. 이제 서버 메세지 읽기를 시작한다. 소켓 file descriptor로 클라이언트가 수신한 서버 메세지에 접근하여 메세지를 msg_buffer에 담는다. 현재시간과 메세지의 길이, msg_buffer의 데이터를 버퍼에 담고, 이 버퍼의 데이터를 파일에 쓴다. 이를 통신 종료 문구를 만날 때 까지 반복한다. 통신 종료 문구를 만나면 쓰기를 멈추고, 파일을 닫고 함수 실행을 종료한다.

```
// create threads
void create_thread(pthread_t *p_thread, int *client_socket, int port, int log_num) {

    int thread_ID;

    // set pthread_create args
    p_thread_args *pThreadArgs = (p_thread_args *)malloc(sizeof(p_thread_args));
    pThreadArgs->client_socket = client_socket;
    pThreadArgs->port = port;

    // create pthread : dealing with server_msg
    thread_ID = pthread_create(p_thread, NULL, server_msg, (void *)pThreadArgs);

    // error
    if (thread_ID < 0){
        perror("Client : Can't create thread.");
        exit(0);
    }


    return;
}
```

위의 함수는 server_msg 함수를 thread로 처리하기 위한 함수이다. p_thread_args 구조체 변수의 멤버 변수에 클라이언트 소켓 file descriptor와 포트 번호를 할당한다. pthread_create함수에 pthread_t 구조체변수, thread로 처리할 server_msg함수의 함수포인터, 이 함수의 인자인 p_thread_args 구조체 변수를 인자로 전달하면, pthread함수는 thread를

생성하고 thread ID를 반환한다. 만약 thread가 정상적으로 생성되지 않았다면 에러를 발생시키고 프로그램을 종료시킨다.

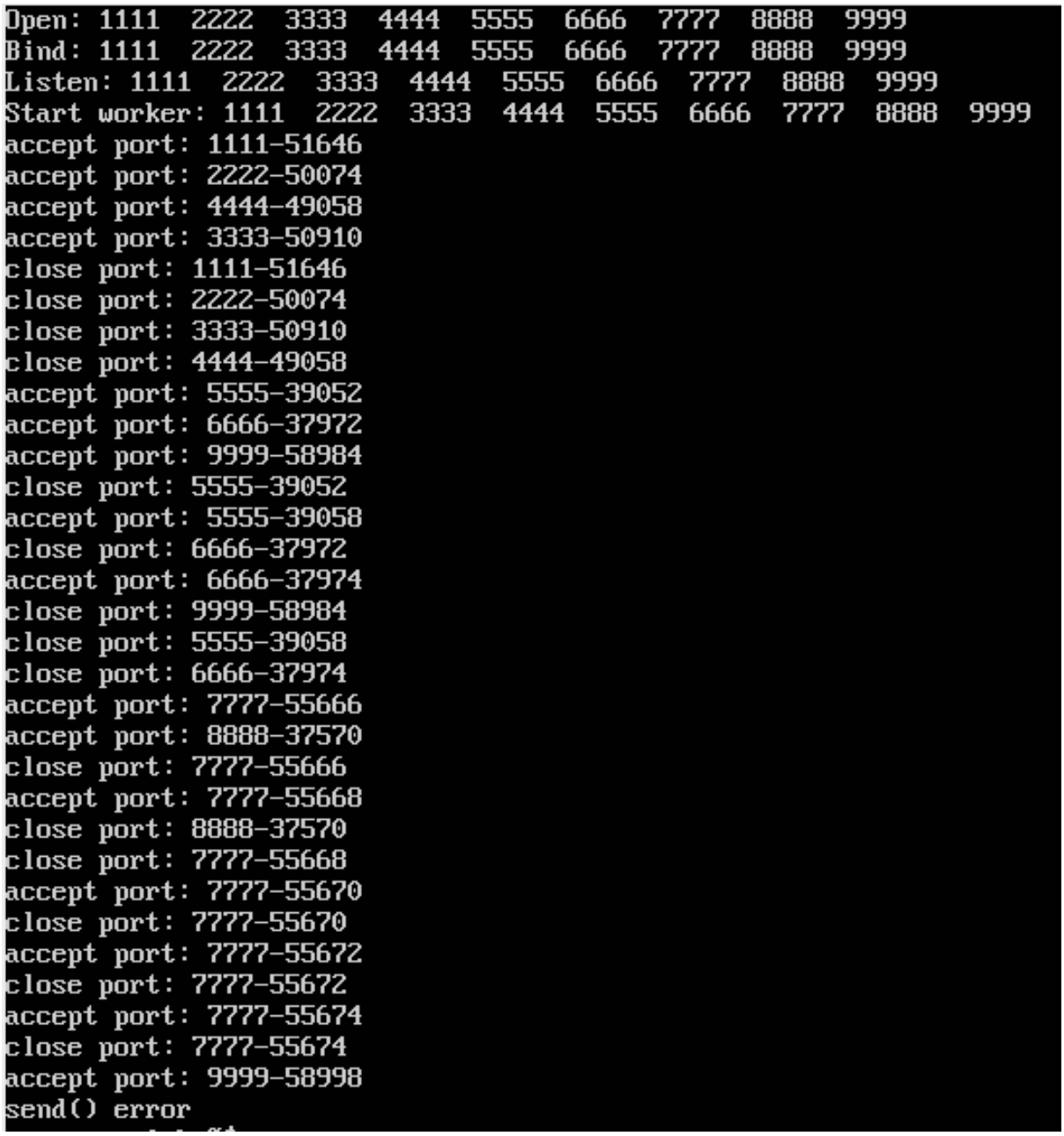
```
// Terminate connection
void close_sockets(int *client_socket, int port_num) {
        printf("Close\n");

        for (int i = 0; i < port_num; i++){
                close(client_socket[i]);
        }

        return;
}
```

위의 함수는 서버와 클라이언트의 통신을 마무리하기 위한 함수이다. 통신이 끝나면 모든 소켓을 닫는다.

다음은 프로그램 실행 결과이다.



위의 그림은 클라이언트의 요청에 따른 서버 출력 내용이다.

```
syspro@syspro-VirtualBox:~/hw$ ./2017320256_client
Enter server IPv4 address: 192.168.99.101
4 1111 2222 3333 4444
Open :   1111    2222    3333    4444
Close
5 5555 6666 6666 5555 9999
Open :   5555    6666    6666    5555    9999
Close
6 7777 7777 7777 7777 7777 8888
Open :   7777    7777    7777    7777    7777    8888
Close
6 9999 9999 9999 9999 9999 9999
Open :   9999    9999    9999    9999    9999
Client : Port 9999 has no available connection.
```

위의 그림은 클라이언트의 표준 출력 내용이다.

2. Pthread를 사용하는 이유

Thread

프로세스 내에서 실행되는 여러 흐름의 단위로 프로세스는 여러 thread들로 구성된다. Thread를 활용해 여러 task들을 동시에(concurrently) 실행할 수 있다. 이때 동일한 프로그램의 thread일지라도 서로 독립적으로 작업을 수행한다. 이러한 동작이 가능한 것은 각각의 thread들이 프로세스의 코드, 데이터 힙 영역을 공유하지만, 스택 영역은 공유하지 않기 때문이다.

Pthread

POSIX 시스템의 standard API로 thread 생성과 synchronization에 사용된다. 유닉스 계열의 운영체제는 thread를 사용하기위해 pthread를 사용한다.

Pthread를 사용하는 이유

프로세스는 자녀 프로세스를 생성할 때 fork() 시스템콜을 사용한다. fork는 콜을 호출한 프로세스의 모든 자원을 copy-on-wirte으로 복사하여 해당 프로세스와 동일한 자녀 프로세스를 생성한다. 이때 오버헤드가 발생하고, 메모리가 낭비된다. 또한 부모 프로세스와 자식 프로세스가 통신하기 위해서는 IPC를 사용해야하는데, 이 역시 오버헤드가 상당한 작업이다.

Pthread를 활용하여 thread로 프로세스를 관리한다면 fork로 인해 프로세스 생성 과정에서 발생하는 자원의 낭비를 절감할 수 있다. 앞서 언급한 바와 같이 thread는 공유 메모리 영역이 존재하기 때문에, thread 생성 과정에서 이 영역을 복사할 필요가 없기 때문이다. 따라서 fork에 비해 빠른 생성 능력을 갖는다. 또한 동일한 정보가 메모리 상에서 불필요하게 반복되지 않기 때문에 메모리를 절약할 수 있다. 공유 메모리의 존재는 thread간 통신 속도를 높인다. 공유 메모리는 IPC에 비해 thread간 정보 교환이 용이하기

때문이다.

3. 과제 수행 시의 Trouble과 Troubleshooting 과정

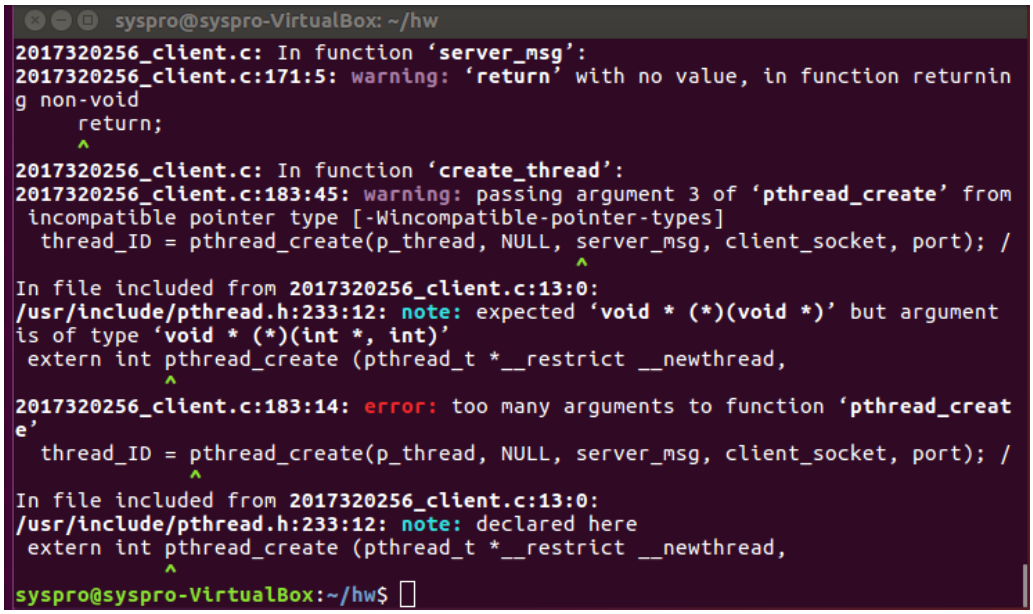
1. 통신 종료

통신 종료의 조건은 클라이언트가 "@@@@@"를 받는 것이다. 그렇지 않으면 무한 루프를 돈다. 코드 작성 과정에서 클라이언트가 "@@@@@"를 받았음에도 무한 루프를 돌며 통신이 종료되지 않는 현상이 있었다. 이는 메세지 버퍼의 초기화가 제대로 이루어 지지 않았기 때문에 발생하는 문제였다. 버퍼들을 memset 함수를 이용하여 초기화했더니 이와 같은 문제가 해결됐다.

하지만 port를 open할 때, MAX_THREAD 수 이상의 port를 open한 경우, pthread_join함수를 실행할 때 core dump가 발생하여 프로그램의 비정상적인 종료가 발생했다. 이는 pthread_join함수를 잘못 사용하여 발생한 문제였다. 초기 코드의 pthread_join을 수행할 때, 모든 thread에 대해 하나의 status 변수만이 존재했다. 이는 thread의 특성인 동시성을 고려하지 못한 프로그래밍이었다. 따라서 Status 변수의 consistency가 보장되지 않아 비정상적인 종료가 발생한 것이었다. 이에 thread 하나 당 하나의 status가 존재하도록 코드를 수정하였더니 문제가 해결됐다.

2. pthread_create() implementation

Thread프로그래밍을 위해 pthread_create 함수를 호출할 때 다음과 같은 오류가 발생했다. 함수 인자의 전달이 잘못됐다는 것이다.



pthead_create()함수는 thread를 생성하여 thread로 인수로 전달받은 함수를 실행한다. 이때 함수의 인자도 pthead_create()함수의 인자로 전달해야 한다. 이때 가장 마지막 인자인 args에 대한 오해로 문제가 발생했다. 초기의 코드는 thread화할 함수의 인자를 독립적으로 전달했다. 만약 thread화할 함수의 인자가 2개라면 2개의 인자를 모두 pthread_create 함수의 인자로 전달했다. 위와 같은 문제가 발생한 뒤 함수의 인

자들로 이루어진 구조체를 정의하여 pthread_create 함수에 마지막 인자로 하나의 변수만 전달했더니 위와 같은 문제가 해결됐다.