[COSE342-TP] 2019320097-조이강

1. 소스코드 빌드 방법

```
make all // client와 server를 모두 빌드합니다.
make client // client만 빌드합니다.
make server // server만 빌드합니다.
make clean // 빌드를 모두 제거합니다.
```

2. 구현 상세

- server
  - 서버는 총 10개의 Key에 해당하는 Value를 저장하며, 총 10개의 클라이언트와 연결될 수 있습니다.

```
#define MAX_KEY 10
#define MAX_CLIENT 10
```

  - 서버는 소켓을 생성한 후, 입력 받은 포트 번호에 소켓을 바인딩합니다.

```
server_socket = socket(AF_INET, SOCK_STREAM, 0)

address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = htons(port);
bind(server_socket, (struct sockaddr *)&address, sizeof(address))
```

  - 그 후 클라이언트의 연결 요청을 기다리는 listen 상태로 진입합니다.

```
listen(server_socket, 3)
```

  - listen 상태에서 키보드 인터럽트를 감지하면, stop값을 1로 만들어 listen을 종료, 서버와 클라이언트 소켓을 모두 닫습니다.

```
void handle_sigint(int sig) {
    (void)sig;
    stop = 1;
}

...
    struct sigaction sa;
    sa.sa_handler = handle_sigint;
    sa.sa_flags = 0;
    sigemptyset(&sa.sa_mask);

...

    for (int i = 0; i < MAX_CLIENT; i++) {
        sd = client_sockets[i];
        if (sd > 0) {
            server_func(sd, "EXIT");
            close(sd);
        }
    }
    close(server_socket);
```

  - while 문에서 select함수를 통해 여러 클라이언트의 요청을 처리할 수 있게 구현하였습니다.
    - 서버는 client_socket를 순회하며, 각 소켓의 유효성을 확인하고 유효한 소켓의 fd를 reads에 추가합니다.
    - 만약 소켓의 fd가 현재 서버 소켓의 fd = max_sd보다 크다면, 업데이트합니다.

```
FD_ZERO(&reads);
FD_SET(server_socket, &reads);
```

```
max_sd = server_socket;
for (int i = 0; i < MAX_CLIENT; i++) {
            client_socket = client_sockets[i];
            if (client_socket > 0) {
                FD_SET(client_socket, &reads);
            }
            if (client_socket > max_sd) {
                max_sd = client_socket;
            }
        }
```

- select 함수를 사용하여, reads에 있는 유효한 소켓들에서 변화를 감지합니다.
- 키보드 인터럽트를 처리할 때, select에서 빠져나오기 위해서 타임 아웃 값을 1초로 설정하였습니다.

```
timeout.tv_sec = 1;
timeout.tv_usec = 0;

select(max_sd + 1, &reads, NULL, NULL, &timeout);
```

- select 함수가 반환하고, 만약 server_socket이 reads에 set되어 있다면 이는 클라이언트가 서버에 연결 요청을 했음을 의미합니다.
- 따라서 accept를 사용하여 새로운 클라이언트를 서버에 추가합니다.

```
if (FD_ISSET(server_socket, &reads)){
        new_socket =
        accept(server_socket, (struct sockaddr *)&address,
        (socklen_t*)&addrlen);
}
...
for (int i = 0; i < MAX_CLIENT; i++) {
    if (client_sockets[i] == 0) {
        client_sockets[i] = new_socket;
        printf("%d번 소켓에 클라이언트가 추가되었습니다.\n", i);
        break;
    }
}
```

- client_sockets를 순회하면서 reads에 set되어있는 소켓을 찾습니다.
- client_socket가 reads에 set 되어있는 경우는 클라이언트에서 연결을 해제했거나, 클라이언트가 서버에 요청함을 의미합니다.
- 클라이언트가 연결을 해제한 경우, close로 클라이언트 소켓을 닫습니다.
- 클라이언트가 연결을 통해 요청했을 경우, buffer에 이를 담아 server_func에 전달합니다.

```
for (int i = 0; i < MAX_CLIENT; i++) {
    client_socket = client_sockets[i];

    if (FD_ISSET(client_socket, &reads)) {
        int client_act;
        if ((client_act = read(client_socket, buffer, STRING_SIZE)) == 0) {
            getpeername(client_socket,
            (struct sockaddr*)&address, (socklen_t*)&addrlen);
            close(client_socket);
            client_sockets[i] = 0;
        } else {
            buffer[client_act] = '\0';
            server_func(client_socket, buffer);
        }
    }
}
```

- 이 과정을 통해 select 함수를 사용한 다중 클라이언트의 요청을 처리할 수 있습니다.

- server는 클라이언트의 요청을 server_func를 통해 처리하여 응답합니다.
- server_func는 Key_Value 구조체와 store를 사용해 Key-Value를 저장합니다.

```
typedef struct {
    char key[STRING_SIZE];
    char value[STRING_SIZE];
} Key_Value;

Key_Value store[MAX_KEY];
int len_store = 0;
void server_func(int client_socket, char *input);
```

- server_func는 클라이언트의 요청에 따라 세 가지 응답을 반환합니다.
  - 클라이언트가 GET <Key> 요청을 보내고, 유효한 key 값일 경우,
  - $<value> 의 형태로 응답합니다.
  - 유효하지 않은 Key 값에 대해선 $-1로 응답합니다.

```
if (strcmp(cmd, "get") == 0 || strcmp(cmd, "GET") == 0) {
        if (parsed_items < 2) {
            strcpy(buffer, "-ERR: GET 명령에 Key가 입력되지 않았습니다.");
        } else {
            int found = 0;
            for (int i = 0; i < len_store; i++) {
                if (strcmp(store[i].key, key) == 0) {
                    snprintf(buffer, sizeof(buffer), "$%s", store[i].value);
                    found = 1;
                    break;
                }
                if (!found) {
                strcpy(buffer, "$-1");
                    }
            }
```

  - 클라이언트가 SET <Key> <Value> 요청을 보낼 경우 store에 Key-Value를 업데이트하고,
  - +OK 로 응답합니다.

```
if (strcmp(cmd, "set") == 0 || strcmp(cmd, "SET") == 0){
     int found = 0;
            for (int i = 0; i < len_store; i++) {
                if (strcmp(store[i].key, key) == 0) {
                    strcpy(store[i].value, value);
                    found = 1;
                    break;
                }
            }
            if (!found) {
                strcpy(store[len_store].key, key);
                strcpy(store[len_store].value, value);
                len_store++;
            }
             strcpy(buffer, "+OK");
}
```

  - 만약 존재하지 않는 명령을 보내는 경우, 명령의 구조가 유효하지 않는 경우, GET 요청에 Key가 유효하지 않는 경우 등 클라이언트의 요청에 문제가 있는 상황이나 서버가 이미 종료된 경우,
  - -ERR 로 응답하여 클라이언트가 종료되게 만듭니다.

```
if (parsed_items < 1) {
        strcpy(buffer, "-ERR: 유효하지 않은 입력입니다.");
    }
...
 if (parsed_items < 2) {
            strcpy(buffer, "-ERR: GET 명령에 Key가 입력되지 않았습니다.");
    }
```

```
...
if (parsed_items < 3) {
            strcpy(buffer, "-ERR: SET 명령에 Key 혹은 Value가 입력되지 않았습니다.");
        }
...
else if (strcmp(cmd, "EXIT") == 0){
        strcpy(buffer, "-ERR: 서버가 종료되었습니다.");
    }
 else {
    strcpy(buffer, "-ERR: 유효하지 않은 입력입니다.");
}
```

- Client
  - 클라이언트는 ip 주소와 서버의 포트 번호를 입력받습니다.
  - 서버와 마찬가지로 socket을 통해 클라이언트 소켓을 만듭니다.

```
(sock = socket(AF_INET, SOCK_STREAM, 0));
```

  - 입력 받은 ip 주소를 inet_pton 함수를 사용하여 네트워크 바이트의 형태로 바꿉니다.

```
inet_pton(AF_INET, ip, &serv_addr.sin_addr)
```

  - connect를 통해 서버와 연결합니다.

```
connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr))
```

  - while문을 계속해서 반복하며 입력을 받습니다.
  - 입력이 EXIT인 경우 클라이언트를 종료합니다.
  - 그 이외의 경우, send_command 함수를 호출하여 서버에 요청합니다.

```
    while (1) {
        printf("입력 대기중...\n");
        if (fgets(input, BUFFER_SIZE, stdin) != NULL) {
            if (strncmp(input, "EXIT", 4) == 0){
                printf("Goodbye.");
                break;
            }
            if (send_command(sock, input) < 0)
                break;
        }
    }
```

  - 클라이언트는 send_command 함수를 통해 서버와 통신하게 됩니다.
  - send 함수를 통해 서버에 요청을 전달합니다.

```
send(sock, input, strlen(input), 0);
```

  - recv 함수를 통해 서버의 응답을 전달받습니다.

```
int bytes_read = recv(sock, buffer, BUFFER_SIZE - 1, 0);
```

  - 만약 서버로부터 어떤 응답도 없었을 경우, 서버와의 연결이 끊겼으므로 클라이언트를 종료합니다.

```
 if (bytes_read <= 0) {
        printf("서버와의 연결이 끊겼습니다.\n");
        return -1;
    }
```

  - 만약 서버가 -ERR 로 시작하는 응답을 보냈을 경우, 클라이언트를 종료합니다.

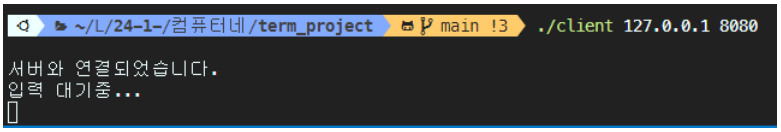
```
    if (strncmp(buffer, "-ERR", 4) == 0){
        printf("%s\n서버 에러 발생, 클라이언트를 종료합니다.\n", buffer);
        return -1;
    }
```

- 그 이외의 경우, 서버로부터 받은 응답을 그대로 출력하고 다시 입력 대기 상태로 돌아갑니다.

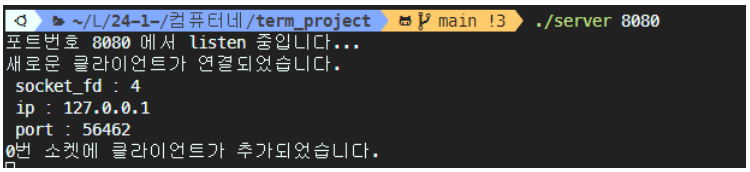
```
    printf("%s\n", buffer);
    return 0;
```

3. 예제 시나리오.
    1. 서버 실행
    - 서버

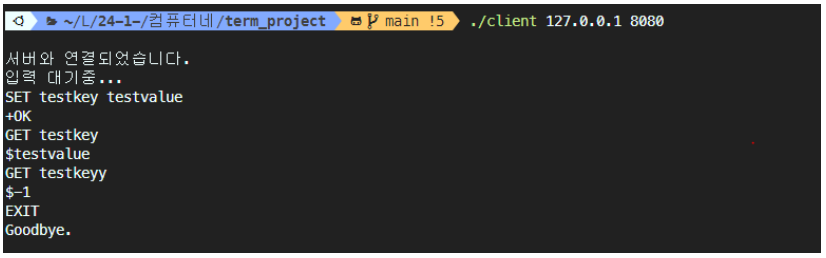
    2. 클라이언트 실행
    - 클라이언트



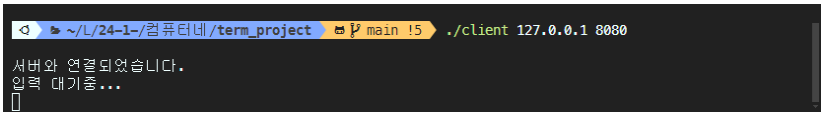
    - 서버



    3. 클라이언트 SET, GET 요청 후 EXIT
    - 클라이언트



    4. 다중 클라이언트 연결
    - 클라이언트



- 서버

```
~/L/24-1-/컴퓨터네/term_project  main !5  ./server 8080  1m 31s  21:06:33
포트번호 8080 에서 listen 중입니다...
새로운 클라이언트가 연결되었습니다.
 socket_fd : 4
 ip : 127.0.0.1
 port : 56544
0번 소켓에 클라이언트가 추가되었습니다.
새로운 클라이언트가 연결되었습니다.
 socket_fd : 5
 ip : 127.0.0.1
 port : 56546
1번 소켓에 클라이언트가 추가되었습니다.
새로운 클라이언트가 연결되었습니다.
 socket_fd : 6
 ip : 127.0.0.1
 port : 56548
2번 소켓에 클라이언트가 추가되었습니다.
```

5. 다중 클라이언트의 요청

- 0, 1, 2번째 소켓 클라이언트

```
~/L/24-1-/컴퓨터네/term_project  main !5  ./client 127.0.0.1 8080

서버와 연결되었습니다.
입력 대기중...
set testkey testvalue
+OK
```

```
~/L/24-1-/컴퓨터네/term_project  main !5  ./client 127.0.0.1 8080

서버와 연결되었습니다.
입력 대기중...
get testkey
$testvalue
```

```
~/L/24-1-/컴퓨터네/term_project  main !5  ./client 127.0.0.1 8080

서버와 연결되었습니다.
입력 대기중...
set testkey testvalue2
+OK
```

- 서버

```
~/L/24-1-/컴퓨터네/term_project  main !5  ./server 8080  1m 31s  21:06:33
포트번호 8080 에서 listen 중입니다...
새로운 클라이언트가 연결되었습니다.
 socket_fd : 4
 ip : 127.0.0.1
 port : 56544
0번 소켓에 클라이언트가 추가되었습니다.
새로운 클라이언트가 연결되었습니다.
 socket_fd : 5
 ip : 127.0.0.1
 port : 56546
1번 소켓에 클라이언트가 추가되었습니다.
새로운 클라이언트가 연결되었습니다.
 socket_fd : 6
 ip : 127.0.0.1
 port : 56548
2번 소켓에 클라이언트가 추가되었습니다.
0번 소켓의 클라이언트의 요청을 처리합니다.
1번 소켓의 클라이언트의 요청을 처리합니다.
2번 소켓의 클라이언트의 요청을 처리합니다.
```

6. 서버가 -ERR을 반환해 클라이언트 종료

- 클라이언트

```
~/L/24-1-/컴퓨터네/term_project  main !5  ./client 127.0.0.1 8080

서버와 연결되었습니다.
입력 대기중...
set testkey testvalue
+OK
set 235235
-ERR: SET 명령에 Key 혹은 Value가 입력되지 않았습니다.
서버 에러 발생, 클라이언트를 종료합니다.
```

- 서버

```
0번 소켓의 클라이언트의 요청을 처리합니다.
0번 소켓의 클라이언트가 연결 해제되었습니다.
 ip : 127.0.0.1
 port 56544
```

7. 키보드 인터럽트를 통한 서버 종료

- 서버

```
^C
서버가 종료되었습니다.
```

- 클라이언트

```
~/L/24-1-/컴퓨터네/term_project  main !5  ./client 127.0.0.1 8080

서버와 연결되었습니다.
입력 대기중...
get testkey
$testvalue

-ERR: 서버가 종료되었습니다.
서버 에러 발생, 클라이언트를 종료합니다.
```

4. API 설명

- socket()
  - 서버와 클라이언트에서 통신을 위한 소켓을 만듭니다. 입력으로 세 개의 인자를 받습니다.
  - 첫 번째 인자는 주소 체계를 받습니다. 텀 프로젝트에서는 IPv4 프로토콜을 사용했으므로 AF_INET 을 인자로 받습니다.
  - 두 번째 인자는 소켓의 타입을 받습니다. TCP를 사용했으므로 SOCK_STREAM 을 인자로 사용했으며, 만약 UDP의 경우에는 SOCK_DGRAM을 사용합니다.
  - 세 번째 인자는 프로토콜 번호이며, 기본 프로토콜을 사용하기 위해 0을 입력했습니다.
- connect()
  - 클라이언트가 만들어진 소켓을 서버에 연결하는데 사용되는 함수입니다.
  - 첫 번째 인자는 클라이언트의 소켓 fd 값을 받습니다.
  - 두 번째 인자는 서버 주소의 구조체를 가리키는 포인터를 받습니다.
  - 세 번째 인자로는 주소 구조체의 크기를 받습니다.
  - connet 함수를 통해 클라이언트와 서버가 연결되면 통신을 시작할 수 있습니다.
- bind()
  - 서버에서 만들어진 소켓에 로컬 주소를 할당하는 함수입니다.
  - 소켓에 IP 주소와 포트 번호를 bind하여 네트워크에서 식별할 수 있게 만듭니다.
  - 첫 번째 인자는 서버 소켓의 파일 디스크립터(fd) 값을 받습니다.
  - 두 번째 인자는 로컬 주소 구조체의 포인터 값을 받습니다.
  - 세 번째 인자는 로컬 주소 구조체의 크기를 받습니다.
  - bind 함수로 소켓에 IP 주소와 포트 번호를 bind 해주어야 클라이언트가 서버에 연결 요청을 할 수 있습니다.
- listen()
  - 서버에서 클라이언트의 연결 요청을 대기하게 하는 함수입니다.

    - 연결 요청이 들어오게 되면, 대기열 큐에 연결 요청을 저장하여 후에 accept될 수 있게 합니다.
    - 첫 번째 인자로는 서버 소켓의 fd를 받습니다.
    - 두 번째 인자는 대기열의 최대 길이를 받습니다.
- accept()
    - 서버에서 클라이언트의 연결 요청을 받아 새로운 소켓을 만드는 함수입니다.
    - 새롭게 만들어진 소켓을 통해 클라이언트와 연결되며, 통신을 진행할 수 있습니다.
    - 첫 번째 인자로 서버 소켓 fd를 받습니다.
    - 두 번째 인자로는 클라이언트의 주소 구조체 포인터를 받습니다.
    - 세 번째 인자는 주소 구조체의 크기를 받습니다.
    - 소켓 생성에 성공하면 만들어진 소켓의 fd를 반환하여 서버에서 통신할 수 있게 만듭니다.
- send()
    - 소켓을 통한 데이터 전송에 사용되는 함수입니다.
    - 첫 번째 인자로 소켓 fd를 받습니다.
    - 두 번째 인자는 전송할 데이터의 포인터를 받습니다.
    - 세 번째 인자는 데이터의 길이를 받습니다.
    - 네 번째 인자는 플래그를 받으나 일반적으로 0 값을 사용합니다.
    - 이를 통해 연결된 상대방의 소켓에 데이터를 전송할 수 있습니다.
- receive()
    - 소켓에 전송된 데이터를 수신하는데 사용되는 함수입니다.
    - 첫 번째 인자로 소켓 fd를 받습니다.
    - 두 번째 인자는 수신 데이터를 저장할 버퍼의 포인터를 받습니다.
    - 세 번째 인자는 버퍼의 크기를 받습니다.
    - 네 번째 인자는 플래그를 받으나 일반적으로 0 값을 사용합니다.
    - 소켓에 전달된 데이터를 버퍼에 저장하여 사용할 수 있게 만듭니다.
- close()
    - 소켓을 닫고, 할당된 자원을 free하게 만듭니다.
    - 소켓 fd 를 받아 해당하는 소켓을 close하는 함수입니다.
    - close 된 소켓은 읽기(수신) 버퍼와 쓰기(송신) 버퍼가 모두 닫혀 아예 사용할 수 없습니다.
- shutdown()
    - close와 비슷하지만 소켓을 닫을 때 읽기, 쓰기 버퍼를 닫을 지 선택할 수 있습니다.
    - 첫 번째 인자로 소켓 fd를 받습니다.
    - 두 번째로 버퍼를 종료할 지 선택할 수 있습니다. SHUT_RD를 입력하면 읽기 버퍼만 차단하고, SHUT_WR 은 쓰기 버퍼만 차단합니다. SHUT_RDWR 은 close와 같이 읽기 쓰기 버퍼를 모두 차단합니다.
- select()
    - 서버에서 여러 소켓을 지속적으로 모니터링하여 소켓에 IO 이벤트가 발생했는지 감지하는 함수입니다.
    - 소켓에 읽기, 쓰기, 예외와 같은 이벤트가 발생할 때 까지 블록하게 하는 함수입니다.
    - 이 함수를 사용하여 다중 클라이언트의 IO Multiplexing을 구현할 수 있으며, 이번 텀 프로젝트에 사용하였습니다.
    - 첫 번째 인자로 모니터링할 fd 수의 최대값을 받습니다.
    - 두 번째 인자와 세 번째 인자는 각각 읽기 이벤트와 쓰기 이벤트를 모니터링할 fd_set을 받습니다.
    - 네 번째 인자는 예외 이벤트를 모니터링할 fd_set을 받습니다.
    - 다섯 번째 인자는 block될 시간을 결정하는 타임 아웃 값이며, 텀 프로젝트 내부에선 1초로 지정되어 있습니다.
- epoll()
    - select와 비슷하게 여러 소켓을 모니터링하여 IO 이벤트를 감지하는데 사용되는 함수입니다.
    - epoll은 세 가지의 함수인 epoll_create()와 epoll_ctl(), epoll_wait() 로 구성되어있습니다.
    - epoll_create를 통해 새로운 epoll 인스턴스를 만듭니다.
    - epoll_ctl 을 통해 epoll 인스턴스에 모니터링할 fd를 추가, 삭제, 수정합니다.

- epoll_wait 를 통해 epoll 인스턴스에 포함된 소켓들에서 이벤트가 발생할 때 까지 기다립니다.
- select와 거의 비슷한 기능을 수행하나, 대형 서버에서 많은 양의 fd를 처리할 때 더 효율적인 함수입니다.