Introduction to Computer Networks

Lab 03: select 함수를 이용한 Multi-client Server Implementation

1. 실습의 목표

Lab 03에서 만든 서버는 하나의 클라이언트만 서비스할 수 있는 서버이다. 하지만, 실제로 일반적인 서버는 다수의 클라이언트를 동시에 서비스할 수 있어야 한다. 다수의 클라이언트를 동시에 서비스할 경우, 어떤 클라이언트로부터 메시지가 도착할 지 알 수 없으므로, 여러 개의 소켓을 관찰하고 있다가 메시지가 오면 그때 그때 처리해주어야 한다. 이런 역할을 하는 함수가 select이다.

2. 작성해 볼 프로그램

selectserver: socket 통신을 위한 서버 쪽 프로그램. select 함수를 이용한 multi-client 서버이다.

3. 프로그램

아래의 예제를 따라 selectserver.cc 프로그램을 작성한다. 음영처리 된 부분이 server.cc에서 달라진 부분이다.

```
// selectserver.cc – a multi-client socket server – serves multiple clients using select()

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <arpa/inet.h>

#define MAXDATASIZE 1000
#define BACKLOG 10

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

    int sockfd;
    int newfd;
    struct addrinfo hints, *servinfo;
    struct sockaddr_storage their_addr;
    socklen_t sin_size;
    char s[INET_ADDRSTRLEN];
    int rv;

    fd_set master;
    fd_set read_fds;
    int fdmax;


    char buf[MAXDATASIZE];
```

```
    int numbytes;

    if(argc != 2) {
        printf("usage: server portnum\n");
        exit(1);
    }

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, argv[1], &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        exit(1);
    }

    if((sockfd = socket(servinfo->ai_family, servinfo->ai_socktype, servinfo-
>ai_protocol)) == -1) {
        perror("server: socket");
        exit(1);
    }

    if (bind(sockfd, servinfo->ai_addr, servinfo->ai_addrlen) == -1) {
        close(sockfd);
        perror("server: bind");
        exit(1);
    }

    freeaddrinfo(servinfo);

    if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
    }

    FD_SET(sockfd, &master);
    fdmax = sockfd;

    while(1) {
        read_fds = master;
        if(select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(1);
        }

        for(int i=0; i<=fdmax; i++) {
            if(FD_ISSET(i, &read_fds)) {
                if(i == sockfd) {
                    sin_size = sizeof their_addr;
                    newfd = accept(sockfd, (struct sockaddr*)&their_addr,
&sin_size);

                    if(newfd == -1) {
                        perror("accept");
                    } else {
                        FD_SET(newfd, &master);
                        if(newfd > fdmax) {
```

```
                        fdmax = newfd;
                    }
                    printf("selectserver: new connection from %s on socket %d\n",
                        inet_ntop(their_addr.ss_family, &((struct
sockaddr_in*)&their_addr)->sin_addr, s, sizeof s), newfd);
                }
            }
            else {
                if((numbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
                    if(numbytes == 0) {
                        printf("selectserver: socket %d hung up\n", i);
                    } else {
                        perror("recv");
                    }
                    close(i);
                    FD_CLR(i, &master);
                }
                else {
                    buf[numbytes] = '\0';
                    printf("server received: %s\n", buf);

                    if(send(i, buf, strlen(buf), 0) == -1) {
                        perror("send");
                    }
                }
            }
        }
    }
}
```

4. 프로그램 예제 설명

아래에는 selectserver의 내용을 라인별로 설명하였다.

```
    fd_set master;
    fd_set read_fds;
    int fdmax;
```

- select 함수를 사용하기 위하여 필요한 자료 구조. fd_set은 소켓의 집합이라고 보면 된다. 소켓의 집합이 master와 read_fds 두 개가 필요한 이유는 뒤에 설명한다. fdmax는 현재 사용중인 소켓 중에서 제일 큰 값으로, 관찰해야 되는 소켓의 범위를 결정해준다.

fd_set을 다음과 같은 그림으로 표현할 수 있다.

0	1	2	3	4	5	6	7	8	...

fd_set에 포함된 소켓의 최대값은 지정되지 않는다. 각 숫자는 socket id를 의미한다. (각각의 소켓은 자신만의 socket id를 갖는다는 것을 기억할 것!)

일반적으로 소켓을 만들면 socket id는 3번부터 부여된다. 그런 이유는 0, 1, 2번은 이미 STDIN,

STDOUT, STDERR로 지정되어 있기 때문이다.

STDIN: 사용자 입력
STDOUT: 출력
STDERR: 에러 출력

```
    FD_SET(sockfd, &master);
    fdmax = sockfd;
```

- FD_SET 함수는 소켓 집합 (fd_set)에 액티브 소켓을 등록시키는 일을 한다. 나중에 select 함수를 사용한 loop에서는 액티브 소켓에 대해서만 메시지가 도착했는지 확인한다. 예를 들어 sockfd가 3번이라면, FD_SET을 부르고 난 다음 다음과 같이 된다. 불이 켜졌다는 뜻은 액티브 소켓이 되었다는 뜻이다.

0	1	2	3	4	5	6	7	8	...

fdmax는 액티브 소켓 중 가장 값이 큰 값으로 지정한다. fdmax가 있어야 나중에 for loop을 이용하여 액티브 소켓을 검색할 때 범위를 지정할 수가 있다.

```
    while(1) {
        read_fds = master;
        if(select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1) {
            perror("select");
            exit(1);
        }
```

- 서버는 계속해서 새로운 클라이언트를 받기도 하고, 연결된 클라이언트에 대해 서비스 해 주어야 하므로 무한루프를 돈다.

select 함수의 원형은 다음과 같다.

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
timeval *timeout);
```

이 중에서 현재로 중요한 것은 1번과 2번 인자이다. 1번 인자는 **관찰할 때 사용하기 원하는 socket id의 최대값+1**로 지정해야 한다. 2번 인자는 관찰하고자 하는 소켓 집합이다. 나머지는 현재로는 NULL로 설정해 주면 된다.

무한루프에 들어와서 맨 처음에 read_fds = master; 와 같이 해주는 이유는, **select 함수가 소켓 집합의 원형을 변화시키기 때문**이다. 따라서 항상 select 함수를 호출하기 전에 복사본을 만들고, 그 복사본을 이용하여 select를 호출한다. 좀더 자세히 설명하면, 현재 관찰해야 하는 소켓 집합

(master)이 다음과 같다고 하자.

0	1	2	3	4	5	6	7	8	...

select 함수를 부르면, 액티브 소켓 중에서 현재 메시지가 도착한 소켓에 대해서는 불을 켜놓고, 나머지에 대해서는 불을 끈다. 다시 말해서 3번 소켓을 관찰하는데 메시지가 도착하지 않았다면 다음과 같이 변한다.

0	1	2	3	4	5	6	7	8	...

이렇게 변하기 때문에 원본인 master를 사용하지 않고 항상 복사본인 read_fds를 만들어서 사용하는 것이다.

```
     for(int i=0; i<=fdmax; i++) {
          if(FD_ISSET(i, &read_fds)) {
```

- select 함수를 호출한 다음에는, 불이 켜져 있는 (메시지가 도착한) 소켓에 대해서 메시지를 처리해주어야 할 것이다. 따라서 for loop를 통해서 불이 켜져 있는 소켓을 찾는다. 찾는 범위는 0부터 fdmax까지이다. (이를 위하여 fdmax가 필요하다.) FD_ISSET은 해당 소켓의 불이 켜져 있는 경우 1을 리턴하는 함수이다. 따라서 메시지가 도착한 소켓에 대해서만 if문 안으로 들어간다.

```
               if(i == sockfd) {
                   sin_size = sizeof their_addr;
                   newfd = accept(sockfd, (struct sockaddr*)&their_addr,
&sin_size);

                   if(newfd == -1) {
                       perror("accept");
                   } else {
                       FD_SET(newfd, &master);
                       if(newfd > fdmax) {
                           fdmax = newfd;
                       }
                       printf("selectserver: new connection from %s on socket %d\n",
                           inet_ntop(their_addr.ss_family, &((struct
sockaddr_in*)&their_addr)->sin_addr, s, sizeof s), newfd);
                   }
               }
```

- 이 블록은 메시지가 도착한 소켓이 sockfd인 경우이다. 기억해야 할 것은 sockfd는 새로운 클라이언트를 기다리는 소켓이고 (일반적으로 3번), 클라이언트가 연결되면 실제로 그 클라이언트를 담당하는 소켓은 새로 만들어진다는 사실이다. (일반적으로 4번부터 배정)
따라서 sockfd로부터 메시지가 왔다는 뜻은 **새로운 클라이언트로부터 연결 요청이 왔다는 뜻**이다. 이런 경우에는 다음과 같이 처리해준다.

- 먼저 accept를 호출하여 클라이언트를 받아들이고 새로운 소켓 아이디 (newfd)를 부여해준다.
- 소켓 집합 원본에 새로운 소켓을 등록시킨다. (FD_SET)
- 만약 새로운 소켓이 현재 socket id의 최대값보다 큰 경우 fdmax를 갱신해준다.
- 새로운 클라이언트가 연결했다는 메시지를 출력한다.

```
                else {
                    if((numbytes = recv(i, buf, sizeof buf, 0)) <= 0) {
                        if(numbytes == 0) {
                            printf("selectserver: socket %d hung up\n", i);
                        } else {
                            perror("recv");
                        }
                        close(i);
                        FD_CLR(i, &master);
                    }
```

- else 블록으로 들어가는 경우에는 도착한 메시지가 sockfd로부터 온 것이 아니라 다른 소켓에서 왔다는 뜻이다. 즉 **특정 클라이언트로부터 메시지가 왔다는 뜻**이므로, 메시지를 처리해주면 된다. 이는 Lab02에서 작성한 server.cc와 유사하므로, 자세한 설명은 이를 참고하면 된다.

- recv 함수를 호출하여 메시지를 받는다.
- 만약 recv의 리턴값 (numbytes)이 0이면 클라이언트가 연결을 끊었다는 뜻이다. 이런 경우에는 다음과 같이 처리해 주어야 한다.

먼저 해당 서버소켓을 닫아준다. (close)
그런 다음 소켓 집합 원본에서 해당 서버 소켓을 등록 취소시켜준다. (FD_CLR) 이제 더 이상 이 소켓은 액티브 소켓이 아니다.

```
                    else {
                        buf[numbytes] = '\0';
                        printf("server received: %s\n", buf);

                        if(send(i, buf, strlen(buf), 0) == -1) {
                            perror("send");
                        }
                    }
                }
            }
        }
    }
}
```

- 만약 numbytes가 0이 아닌 경우에는 정상적으로 메시지가 도착했다는 뜻이다. 이러한 경우에는 메시지를 출력하고 서버로 다시 보내준다. (echo)

이 때 주의해야 할 것은, 해당 소켓으로 메시지를 보내주어야 한다는 것이다. 현재의 경우 해당 소켓은 i이다.

5. 컴파일 및 실행

- 프로그램을 컴파일 하여 실행파일을 얻는다.
$ g++ -o selectserver selectserver.cc

- **세 개**의 ssh 창을 띄운다. 이번 실습에서 작성한 server를 먼저 띄우고, 지난 실습에서 만들었던 client를 **두 개 띄워서 둘 다 접속이 되는지 확인한다.**
$ ./selectserver 55555

- 클라이언트를 실행시킨다. 클라이언트는 lab3에서 만든 client를 사용한다. (만약 완료하지 못한 경우 Lab 01에서 만든 client를 사용해도 무방하다.)
$ ./lab3_client 127.0.0.1 55555

- 또 하나의 클라이언트를 실행시킨다.
$ ./lab3_client 127.0.0.1 55555

각각의 클라이언트에서 메시지를 입력해보고 서버가 echo server로서 잘 동작하는지 확인한다.

6. 과제
selectserver.cc를 다음과 같이 수정하여 lab3_server.cc 파일로 만든다.

- 현재 selectserver는 클라이언트로부터 받은 메시지를 해당 클라이언트로 그대로 보내주는 echo server이다. 이번 과제는 이를 수정하여 클라이언트가 요청하는 정보를 서버가 처리해서 보내주는 기능을 추가할 것이다.

- 만약 클라이언트가 "user" 라는 스트링을 보낼 때는, 서버는 현재 접속해 있는 사용자가 몇 명인지 클라이언트에게 알려주어야 한다. 즉, 다음과 같은 답을 보내주면 된다. 만약 접속해 있는 사용자가 3명인 경우:

number of online users: 3

라는 스트링을 클라이언트로 보내준다. 또한, 서버의 화면에도 이를 출력해준다.

이 때, 사용자의 수에는 요청을 보낸 클라이언트도 포함한다. (아래 팁 참조)

- 다른 스트링에 대해서는 원래처럼 그대로 다시 해당 클라이언트로 보내준다. (echo)

- 아래와 같이 컴파일 하여 서버를 켜놓는다.

```
$ g++ -o lab3_server lab3_server.cc
$ ./lab3_server 55555
```

- lab1_client 프로그램을 서로 다른 2개의 창에서 (혹은 3개 이상) 띄워서 서버에 접속한다.

- user 명령을 이용하여 현재 서버에 접속한 사용자의 수가 몇 명인지 확인한다.

- 하나의 클라이언트를 접속 종료시킨다.

- 아직 살아있는 클라이언트에서 user 명령을 실행시켜 사용자의 수가 제대로 업데이트 되었는지 확인한다.

7. 팁

서버에서 현재 사용자의 수를 확인하기 위해서는 FD_ISSET 함수를 이용하면 된다. FD_ISSET은 2개의 인자를 받으며 1번 인자는 불이 켜져 있는지 확인하고자 하는 소켓 id, 2번 인자는 소켓 집합의 주소이다. (위의 코드에서 사용법을 확인한다.)

현재 사용자의 개수는 원본 소켓 집합인 master에서 불이 켜져 있는 소켓의 개수일 것이다. 하지만 여기서 주의해야 할 것은, **새로운 클라이언트를 받는 소켓, 즉 sockfd는 개수에서 빼 주어야 한다는 것이다.**

만약 master 소켓 집합의 상태가 다음과 같다면, 접속해 있는 사용자는 **2명**일 것이다.

0	1	2	3	4	5	6	7	8	...

물론 FD_ISSET을 이용해서 user의 수를 제대로 얻기 위해서는, 새로운 사용자가 접속할 때와 접속을 종료할 때 master 소켓 집합에 대한 업데이트를 제대로 해 주어야 한다.