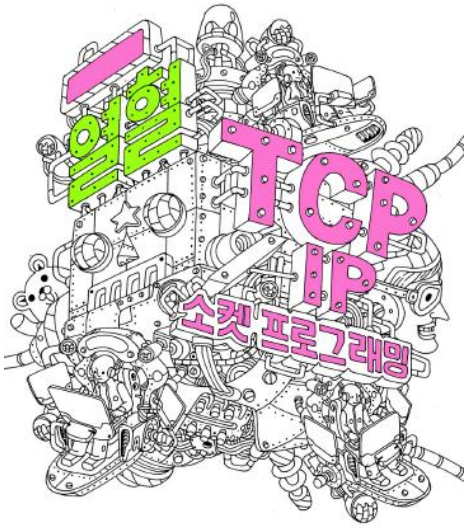
윤성우의 열혈 TCP/IP 소켓 프로그래밍

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

Chapter 12. IO 멀티플렉싱



Chapter 12-1. IO 멀티플렉싱 기반의 서버

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

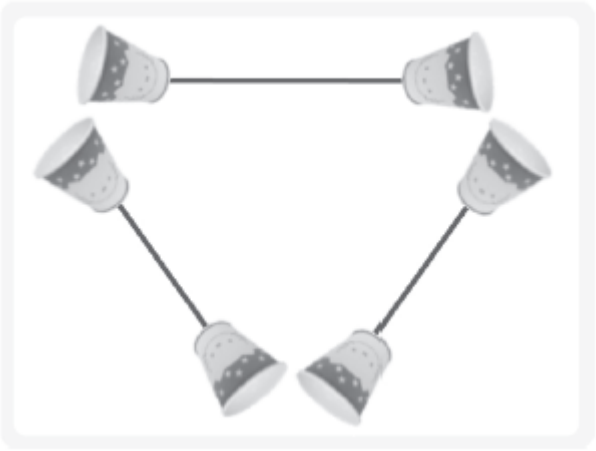
멀티 프로세스 서버의 단점과 대안

- 멀티 프로세스 서버의 단점
  - 프로세스의 빈번한 생성은 성능의 저하로 이어진다.
  - 멀티 프로세스의 흐름을 고려해서 구현해야 하기 때문에 구현이 쉽지 않다.
  - 프로세스간 통신이 필요한 상황에서는 서버의 구현이 더 복잡해진다.

- 멀티 프로세스 서버의 대안
  - 하나의 프로세스가 다수의 클라이언트에게 서비스를 할 수 있도록 한다.
  - 이를 위해서는 하나의 프로세스가 여러 개의 소켓을 핸들링 할 수 있는 방법이 존재해야 한다.
  - 바로 이것이 IO 멀티플렉싱이다.

멀티플렉싱이라는 단어의 이해

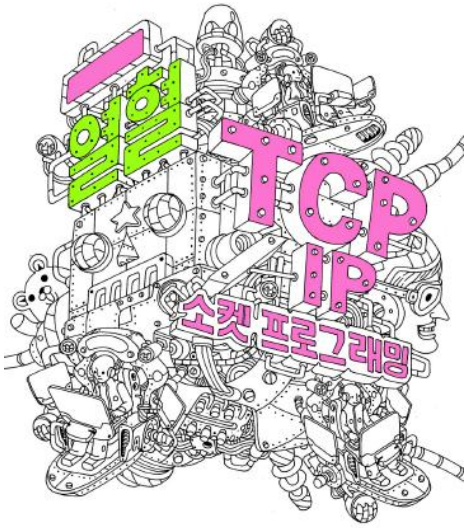
멀티 프로세스에 비유 할 수 있다.



멀티플렉싱에 비유 할 수 있다.

전화의 경우 말을 하는 순서가 일치하지 않기 때문에, 그리고 목소리의 주파수도 다르기 때문에 멀티 플렉싱이 가능하다.
유사하게 데이터의 송수신도 실시간으로 0의 지연시간을 가지고 서비스 해야 하는 것이 아니기 때문에 멀티플렉싱이 가능하다.
문제는 하나의 프로세스가 다수의 소켓을 관리하는 방법에 있다.

왼쪽의 실전화기가 보이듯이 하나의 리소스를 둘 이상의 영역에서 공유하는 것을 가리켜 멀티플렉싱이라 한다.



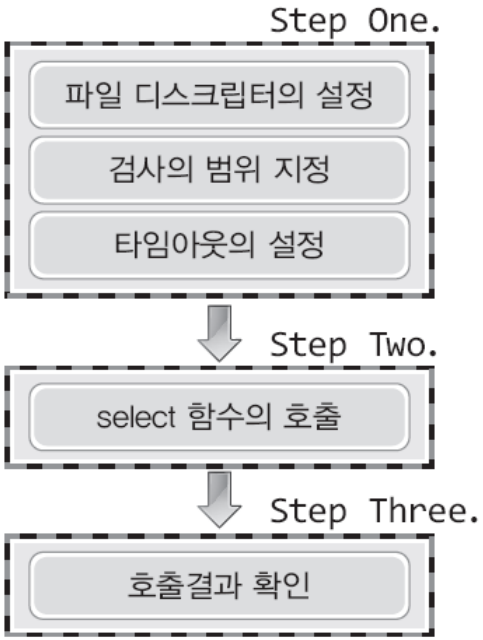
Chapter 12-2. select 함수의 이해와 서버의 구현

윤성우 저 열혈강의 TCP/IP 소켓 프로그래밍 개정판

select 함수의 기능과 호출순서

- 수신한 데이터를 지니고 있는 소켓이 존재하는가?
- 블로킹되지 않고 데이터의 전송이 가능한 소켓은 무엇인가?
- 예외상황이 발생한 소켓은 무엇인가?

select 함수를 이용하면, 배열에 저장된 다수의 파일 디스크립터를 대상으로 이와 같은 질문을 던질 수 있다.



Step One에서는 관찰의 대상을 묶고, 관찰의 유형을 지정한다.

Step Two에서는 관찰 대상의 변화를 묻는다.

Step Three에서는 물음에 대한 답을 듣는다.

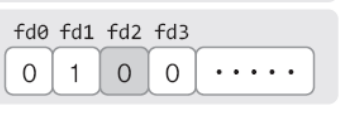
파일 디스크립터의 설정

```
int main(void)
{
   fd_set  set;
   FD_ZERO(&set);
   FD_SET(1, &set);
   FD_SET(2, &set);
   FD_CLR(2, &set);
}
```

모두 0으로 초기화

디스크립터 1을 관찰 대상으로 추가

디스크립터 2를 관찰 대상으로 추가



디스크립터 2를 관찰 대상에서 제외

fd_set형 변수에 select 함수에 전달할 디스크립터의 정보를 묶는다.

fd_set형 변수의 컨트롤 함수

- `FD_ZERO(fd_set * fdset)` — 인자로 전달된 주소의 fd_set형 변수의 모든 비트를 0으로 초기화한다.
- `FD_SET(int fd, fd_set *fdset)` — 매개변수 fdset으로 전달된 주소의 변수에 매개변수 fd로 전달된 파일 디스크립터 정보를 등록한다.
- `FD_CLR(int fd, fd_set *fdset)` — 매개변수 fdset으로 전달된 주소의 변수에서 매개변수 fd로 전달된 파일 디스크립터 정보를 삭제한다.
- `FD_ISSET(int fd, fd_set *fdset)` — 매개변수 fdset으로 전달된 주소의 변수에 매개변수 fd로 전달된 파일 디스크립터 정보가 있으면 양수를 반환한다.

select 함수의 소개

```
#include <sys/select.h>
#include <sys/time.h>

int select(
int maxfd, fd_set *readset, fd_set *writeset, fd_set *exceptset, const struct timeval * timeout);
```

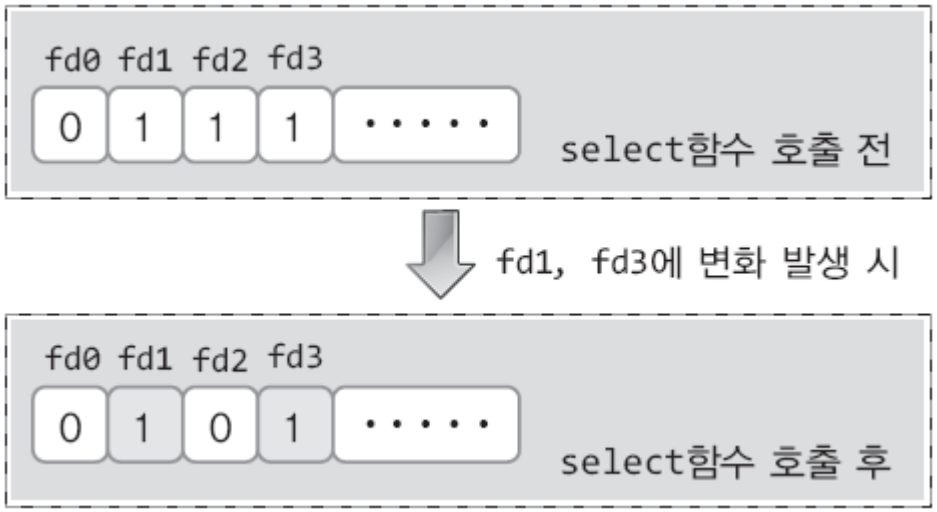
➡ 성공 시 0 이상, 실패 시 -1 반환

- maxfd — 검사 대상이 되는 파일 디스크립터의 수.
- readset — fd_set형 변수에 '수신된 데이터의 존재여부'에 관심 있는 파일 디스크립터 정보를 모두 등록해서 그 변수의 주소 값을 전달한다.
- writeset — fd_set형 변수에 '블로킹 없는 데이터 전송의 가능여부'에 관심 있는 파일 디스크립터 정보를 모두 등록해서 그 변수의 주소 값을 전달한다.
- exceptset — fd_set형 변수에 '예외상황의 발생여부'에 관심이 있는 파일 디스크립터 정보를 모두 등록해서 그 변수의 주소 값을 전달한다.
- timeout — select 함수호출 이후에 무한정 블로킹 상태에 빠지지 않도록 타임아웃(time-out)을 설정하기 위한 인자를 전달한다.
- 반환 값 — 오류발생시에는 -1이 반환되고, 타임 아웃에 의한 반환 시에는 0이 반환된다. 그리고 관심대상으로 등록된 파일 디스크립터에 해당 관심에 관련된 변화가 발생하면 0보다 큰 값이 반환되는데, 이 값은 변화가 발생한 파일 디스크립터의 수를 의미한다.

```
struct timeval
{
    long tv_sec;      // seconds
    long tv_usec;     // microseconds
}
```

관찰의 대상이 되는 디스크립터의 수는 maxfd이며, 두 번째, 세 번째, 네 번째 인자를 통해서 전달된 관찰의 대상중에서 각각 입력, 출력, 또는 오류가 발생했을 때 select 함수는 반환을 한다. 단, timeout의 지정을 통해서 무조건 반환이 되는 시간을 결정할 수 있다.

select 함수호출 이후의 결과확인



select 함수호출 이후에는 변화가 발생한(입력 받은 데이터가 존재하거나 출력이 가능한 상황 등), 소켓의 디스크립터만 1로 설정되어 있고, 나머지는 모두 0으로 초기화된다.

select 함수의 호출의 예

예제 select.c의 일부

```
int main(int argc, char *argv[])
{
    fd_set reads, temps;
    int result, str_len;
    char buf[BUF_SIZE];
    struct timeval timeout;
    FD_ZERO(&reads);
    FD_SET(0, &reads); // 0 is standard input(console)
    /*
    timeout.tv_sec=5;
    timeout.tv_usec=5000;
    */
    while(1)
    {
        temps=reads;
        timeout.tv_sec=5;
        timeout.tv_usec=0;
        result=select(1, &temps, 0, 0, &timeout);
        if(result==-1)
        {
            puts("select() error!");
            break;
        }
```

검색의 대상 지정

select 함수호출 후에는 타임아웃 발생시전의 시간이 담기므로 select 함수 호출 전에 매번 초기화!

select 함수 호출 후에 저장된 값이 바뀌니, reads 복사!

데이터 입력여부 검사

이어짐

```
        else if(result==0)
        {
            puts("Time-out!");
        }
        else
        {
            if(FD_ISSET(0, &temps))
            {
                str_len=read(0, buf, BUF_SIZE);
                buf[str_len]=0;
                printf("message from console: %s", buf);
            }
        }
    }
    return 0;
}
```

파일 디스크립터 0의 변화 관찰

수신된 데이터가 있으므로 읽는다!

```
root@my_linux:/tcpip# gcc select.c -o select
root@my_linux:/tcpip# ./select
Hi~
message from console: Hi~
Hello~
message from console: Hello~
Time-out!
Time-out!
Good bye~
message from console: Good bye~
```

실행결과

멀티플렉싱 서버의 구현 1단계

서버소켓의 생성과 관찰대상의 등록

```
int main(int argc, char *argv[])
{
    int serv_sock, clnt_sock;
    struct sockaddr_in serv_adr, clnt_adr;
    struct timeval timeout;
    fd_set reads, cpy_reads;
    socklen_t adr_sz;
    int fd_max, str_len, fd_num, i;
    char buf[BUF_SIZE];
    serv_sock=socket(PF_INET, SOCK_STREAM, 0);
    memset(&serv_adr, 0, sizeof(serv_adr));
    serv_adr.sin_family=AF_INET;
    serv_adr.sin_addr.s_addr=htonl(INADDR_ANY);
    serv_adr.sin_port=htons(atoi(argv[1]));
    if(bind(serv_sock, (struct sockaddr*) &serv_adr, sizeof(serv_adr))==-1)
        error_handling("bind() error");
    if(listen(serv_sock, 5)==-1)
        error_handling("listen() error");
    FD_ZERO(&reads);
    FD_SET(serv_sock, &reads);
    fd_max=serv_sock;
```

서버소켓(리스닝 소켓)을 통한 연결요청도 일종의 데이터 수신이기 때문에 관찰의 대상에 포함을 시킨다.

연결요청과 일반적인 데이터 전송의 차이점은 전송되는 데이터의 종류에 있다. 따라서 연결요청도 데이터의 수신으로 구분이 되어서 select 함수의 호출결과를 통해서 확인이 가능하다.

따라서 리스닝 소켓도 관찰의 대상에 포함을 시켜야 한다.

멀티플렉싱 서버의 구현 2단계

```
while(1)
{
    cpy_reads=reads;
    timeout.tv_sec=5;
    timeout.tv_usec=5000;
    if((fd_num=select(fd_max+1, &cpy_reads, 0, 0, &timeout))==-1)
        break;
    if(fd_num==0)
        continue;
```

select 함수를 호출할 때마다 타임아웃을 명시해야 한다.

수신된 데이터가 있는가에 대해서만 관찰을 하는 select 함수의 호출문이다.

select 함수를 호출하고 있다. 호출된 select 함수가 반환을 하면, 반환의 이유를 관찰하게 되지만, 타임아웃에 의한 반환이라면, 다시 select 함수를 호출하기 위해서 break문을 실행해야 한다.

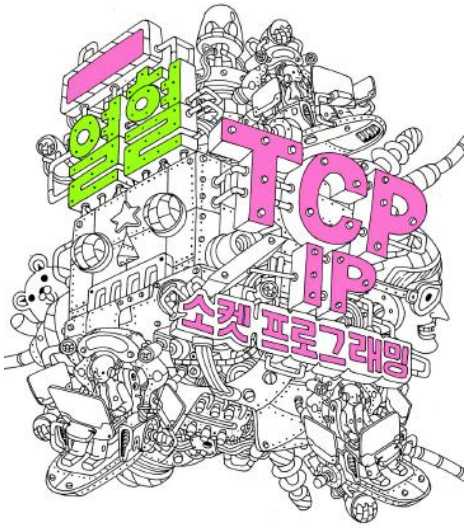
멀티플렉싱 서버의 구현 3단계

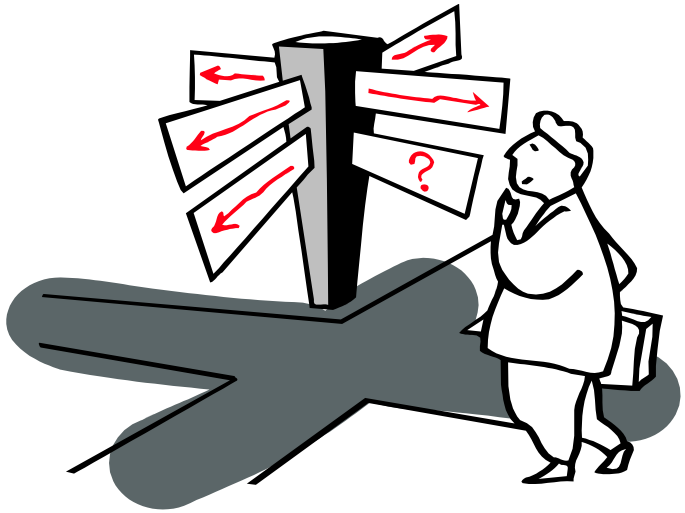
```
for(i=0; i<fd_max+1; i++)
{
    if(FD_ISSET(i, &cpy_reads))
    {
        if(i==serv_sock)    // connection request!
        {
            adr_sz=sizeof(clnt_adr);
            clnt_sock=
                accept(serv_sock, (struct sockaddr*)&clnt_adr, &adr_sz);
            FD_SET(clnt_sock, &reads);
            if(fd_max<clnt_sock)
                fd_max=clnt_sock;
            printf("connected client: %d \n", clnt_sock);
        }
        else    // read message!
        {
            str_len=read(i, buf, BUF_SIZE);
            if(str_len==0)  // close request!
            {
                FD_CLR(i, &reads);
                close(i);
                printf("closed client: %d \n", i);
            }
            else
            {
                write(i, buf, str_len);// echo!
            }
        }
    }
}
```

수신된 데이터가 serv_sock에 있다면 연결 요청이기 때문에 이에 따른 처리를 진행한다.

수신된 데이터가 클라이언트와 연결된 소켓에 있다면, 이에 따른 에코 처리를 진행한다.

select 함수의 특성상 이벤트가 발생하면, 이벤트가 발생한 대상을 찾기 위해서 반복문을 구성해야 한다. 그리고 이것이 select 함수의 단점으로 지적이 된다.





Chapter 12가 끝났습니다. 질문 있으신지요?