네트워크 프로그래밍

3. 저 수준(Low-level) 파일 입출력

다양한 종류의 입출력 함수

- 시스템 입출력 함수
  - 저 수준의 시스템 입출력 함수
  - 간단하게 사용할 수 있지만, 네트워크 환경에 최적화 하기 힘듦

버퍼의 장점: 표준입출력을 쓸 경우 개행단위로 구분해서 받을 수 있음
단점: 프로세스를 복사 할 경우 기존 프로세스의 버퍼까지 같이복사되어서 새 프로세스의 버퍼에 더미값이 들어있을 수 있음

- 표준 입출력 함수
  - C의 표준 입출력 함수들
  - 데이터를 쉽게 처리할 수 있음 (버퍼 기반)

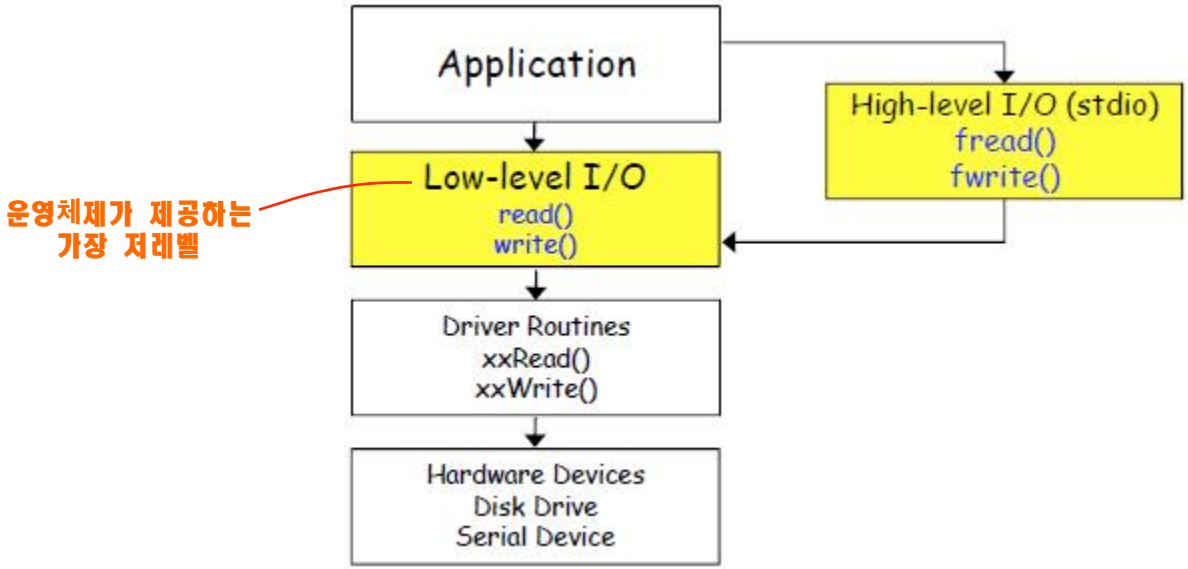
os 영역이 아닌 유저코드영역

개행단위로 읽는것은 scanf에 구현된 것이지 실제적으로 자동지원기능이 아님

- 소켓 입출력 함수
  - 소켓 API에 포함되어 있는 입출력 함수들
  - 네트워크 통신 환경에 맞게 최적화(특성을 제어) 할 수 있음

Low-level 파일 입출력

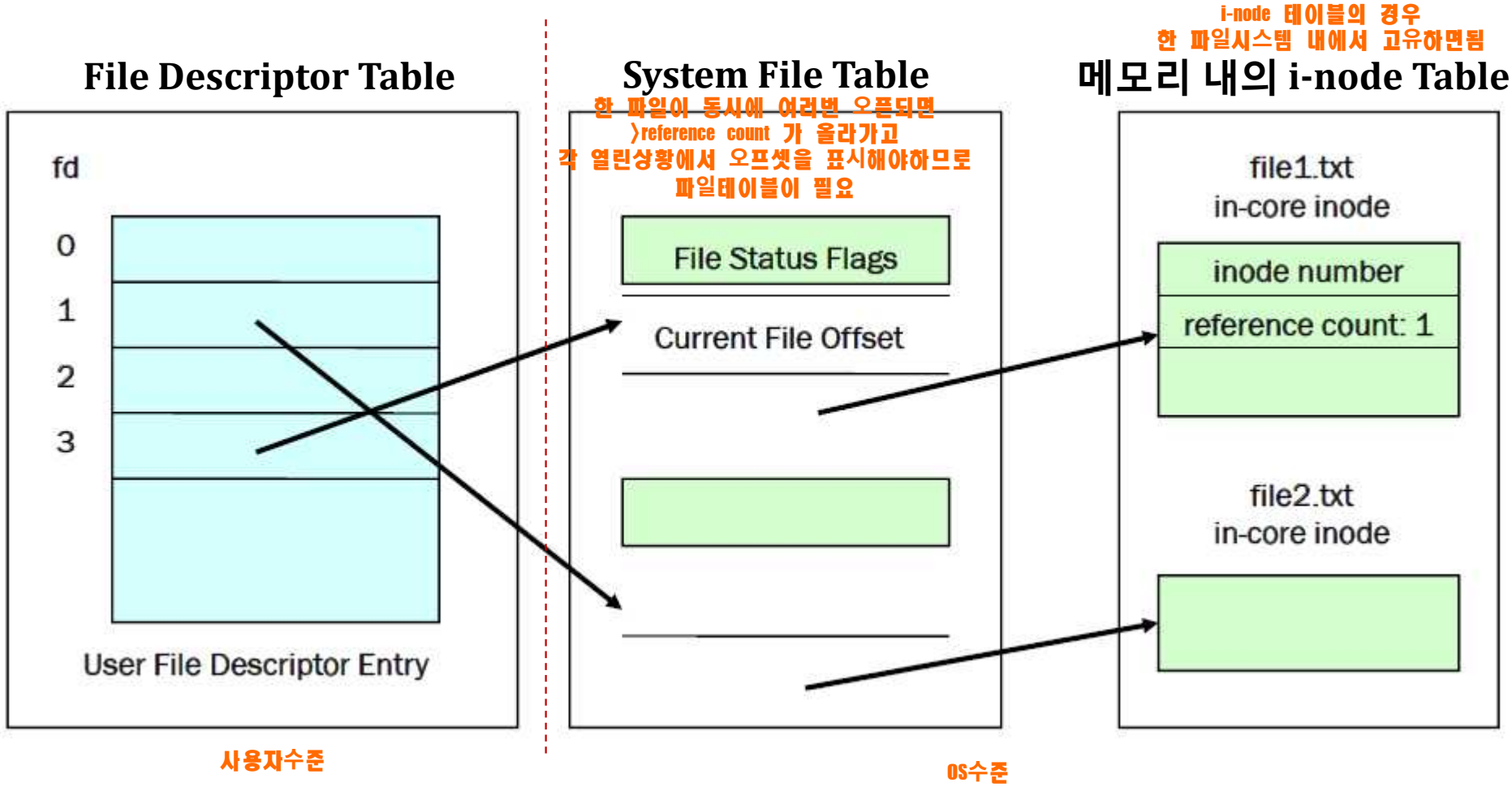
- 버퍼링을 하거나 그 밖의 역할을 하지 않는, 보다 OS수준에 근접한(저차원) 파일 입출력
- ANSI의 표준함수가 아니기 때문에 운영체제에 대한 호환성이 없다.
- open(), close(), read(), write(), fcntl()
- 파일 디스크립터(file descriptor)라는 정수값을 사용하여 파일 처리한다.
- int fd;



파일 디스크립터 (File Descriptor) – 논리적인 핸들

Tip〉 vfs: virtual file system

- 운영체제가 만든 파일(파일, 디바이스, 소켓)을 구분하기 위한 정수형 숫자
- 저 수준 파일 입출력 함수는 입출력을 목적으로 파일 디스크립터를 요구한다.



파일 열기와 닫기

외울것


```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags, mode_t mode);
```

➜ 성공 시 파일 디스크립터, 실패 시 −1 반환

- path　　파일 이름을 나타내는 문자열의 주소 값 전달.
- flag　　파일의 오픈 모드 정보 전달.

오픈 모드	의미
O_CREAT	필요하면 파일을 생성
O_TRUNC	기존 데이터 전부 삭제
O_APPEND	기존 데이터 보존하고, 뒤에 이어서 저장
O_RDONLY	읽기 전용으로 파일 오픈
O_WRONLY	쓰기 전용으로 파일 오픈
O_RDWR	읽기, 쓰기 겸용으로 파일 오픈
O_EXCL	O_CREAT와 함께 사용한다. 이미 파일이 존재하면 errno 값을 EEXIST로 설정한다.

open 함수 호출 시 반환된 파일 디스크립터를 이용해서 파일 입출력을 진행하게 된다.

```
if ((fd = open("my.txt", O_WRONLY|O_CREAT|O_EXCL)) == -1) {
        if (errno == EEXIST) {
                fprintf(stderr, "파일이 이미 존재합니다.\n");
        }
}
```

```
#include <unistd.h>

int close(int fd);
```

➜ 성공 시 0, 실패 시 −1 반환

- fd　　닫고자 하는 파일 또는 소켓의 파일 디스크립터 전달.

유닉스/리눅스의 파일권한(permission)

파일 및 디렉토리 생성시 기본 권한 설정

umask 값에 따라 기본 권한 설정이 결정되는데, 파일의 경우에는 기본 생성 최고 권한이 666이며, 디렉토리의 경우에는 777이다.
디렉토리가 777인 이유는 디렉토리에 실행 권한(x)이 없으면, 디렉토리 안으로 들어갈 수 없기 때문이다.
기본 권한은 umask 값을 ~ 연산한 값과 최고 권한을 & 연산한 값이 된다.

```
open('mydata.txt', O_CREAT, S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH);
```

```
umask(000);
open('mydata.txt', O_CREAT | O_EXCL, 0666);
```

유저의 rwx	소유 권한	권한 값(8진수)
S_IRWXU	user (file owner) has read, write and execute permission	0700
S_IRUSR	user has read permission	0400
S_IWUSR	user has write permission	0200
S_IXUSR	user has execute permission	0100
S_IRWXG	group has read, write and execute permission	0070
S_IRGRP	group has read permission	0040
S_IWGRP	group has write permission	0020
S_IXGRP	group has execute permission	0010
S_IRWXO	others have read, write and execute permission	0007
S_IROTH	others have read permission	0004
S_IWOTH	others have write permission	0002
S_IXOTH	others have execute permission	0001

파일에 데이터 쓰기


```
#include <unistd.h>

ssize_t write(int fd, const void * buf, size_t nbytes);
```

➡ 성공 시 전달한 바이트 수, 실패 시 -1 반환

- fd — 데이터 전송대상을 나타내는 파일 디스크립터 전달.
- buf — 전송할 데이터가 저장된 버퍼의 주소 값 전달.
- nbytes — 전송할 데이터의 바이트 수 전달.

```
int main(void)
{
    int fd;
    char buf[]="Let's go!\n";
    fd=open("data.txt", O_CREAT|O_WRONLY|O_TRUNC);
    if(fd==-1)
        error_handling("open() error!");
    printf("file descriptor: %d \n", fd);

    if(write(fd, buf, sizeof(buf))==-1)
        error_handling("write() error!");
    close(fd);
    return 0;
}
```

실행 결과

```
root@my_linux:/tcpip# gcc low_open.c -o lopen
root@my_linux:/tcpip# ./lopen
file descriptor: 3
root@my_linux:/tcpip# cat data.txt
Let's go!
root@my_linux:/tcpip#
```

파일에 저장된 데이터 읽기


```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbytes);
```

➜ 성공 시 수신한 바이트 수(단 파일의 끝을 만나면 0), 실패 시 -1 반환

- fd — 데이터 수신대상을 나타내는 파일 디스크립터 전달.
- buf — 수신한 데이터를 저장할 버퍼의 주소 값 전달.
- nbytes — 수신할 최대 바이트 수 전달.

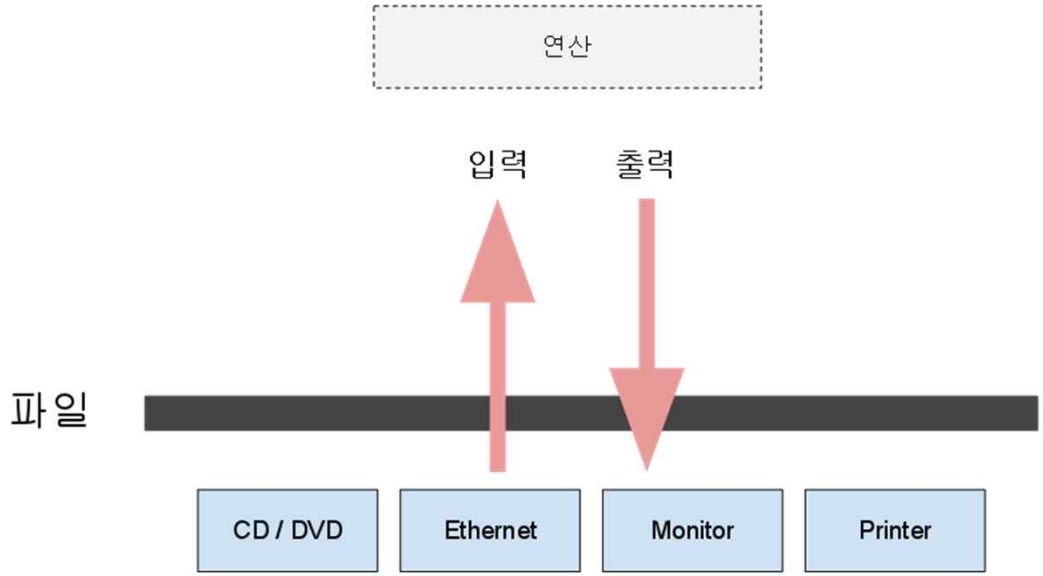
```
int main(void)
{
    int fd;
    char buf[BUF_SIZE];

    fd=open("data.txt", O_RDONLY);
    if( fd==-1)
        error_handling("open() error!");
    printf("file descriptor: %d \n" , fd);

    if(read(fd, buf, sizeof(buf))==-1)
        error_handling("read() error!");
    printf("file data: %s", buf);
    close(fd);
    return 0;
}
```

실행결과

```
root@my_linux:/tcpip# gcc low_read.c -o lread
root@my_linux:/tcpip# ./lread
file descriptor: 3
file data: Let's go!
root@my_linux:/tcpip#
```

모든 것은 파일이다

- 유닉스는 모든 자원을 파일로 본다.
- 소켓 역시 파일로 본다.



```c
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char** argv) {
        int fD, writeLen, readLen;
        char rBuff[BUFSIZ];

        if(argc != 2) {
                fprintf(stderr, "Usage: %s [Filename] \n ",argv[0]);
                return 0;
        }

        readLen = read(0, rBuff, BUFSIZ-1);

        if(readLen == -1) {
                fprintf(stderr, "Read Error \n");
                return 0;
        }

        printf("Total reading data: %d\n", readLen);
        rBuff[readLen] = '\0';
        fD = open(argv[1],O_WRONLY|O_CREAT|O_TRUNC);

        if(fD == -1) {
                fprintf(stderr, "Open Error \n");
                return 0;
        }

        writeLen = write(fD, rBuff, readLen+1);

        if(writeLen == -1) {
                fprintf(stderr, "Write Error \n");
                return 0;
        }

        printf("Total writing data: %d\n", writeLen);
        close(fD);
        return 0;
}
```

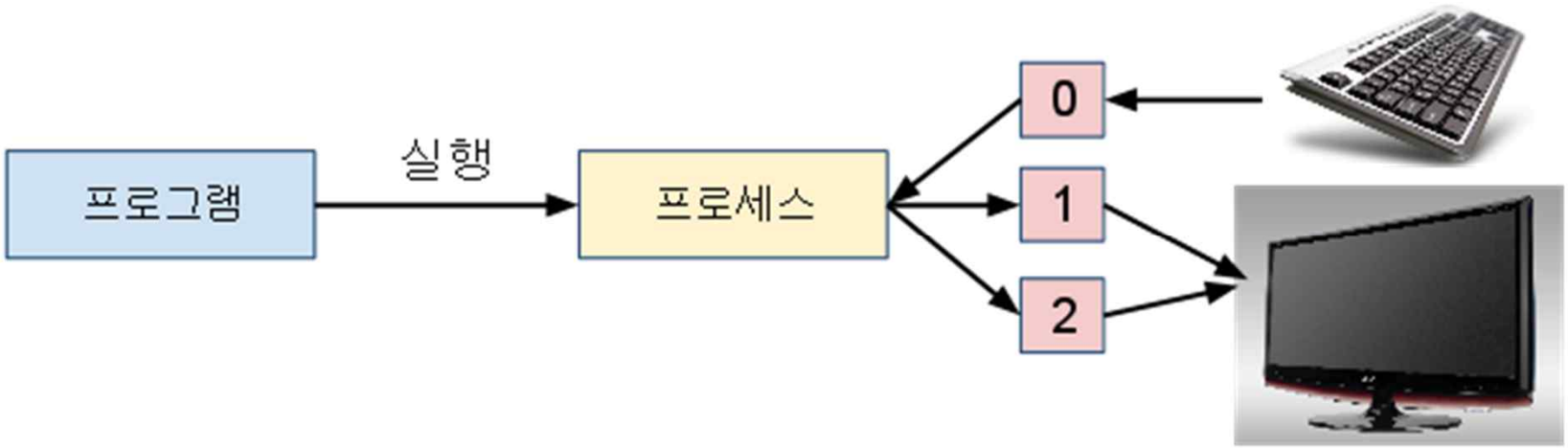
표준입출력 함수가 아니므로 내가 문자열이라는 뜻으로 널문자를 넣어준것

gcc -o io-syscall io.syscall.c

표준입력, 표준출력, 표준에러

- 리눅스는 기본적인 입출력을 위한 장치를 가진다.
- 이들 장치는 프로세스 생성과 함께 할당된다.

대상	파일 디스크립터	설명
표준입력	0	키보드 입력
표준출력	1	모니터 출력
표준에러	2	모니터 출력



io-syscall 실행결과

- 실행결과

16 + 개행문자

```
ubuntu@ip-172-30-0-185: ~/NW_source/01
ubuntu@ip-172-30-0-185:~/NW_source/01$ ./io-syscall test
How old are you?
Total reading data: 17
Total writing data: 18
ubuntu@ip-172-30-0-185:~/NW_source/01$ cat test
How old are you?
ubuntu@ip-172-30-0-185:~/NW_source/01$
```

- 실행결과

터미널이라는 뜻임 〉 찾아볼것

```
ubuntu@ip-172-30-0-185: ~/NW_source/01
ubuntu@ip-172-30-0-185:~/NW_source/01$ ./io-syscall /dev/pts/1
How old are you?
Total reading data: 17
Total writing data: 18
ubuntu@ip-172-30-0-185:~/NW_source/01$
```

위에서 엔터치는 순간 이렇게 3줄뜨고

```
ubuntu@ip-172-30-0-185: ~
ubuntu@ip-172-30-0-185:~$ tty
/dev/pts/1
ubuntu@ip-172-30-0-185:~$ How old are you?
```

터미널에서도 글이써짐

터미널도 파일이 아니지만
〉파일처럼 접근가능하다
리눅스니까!

파일 디스크립터와 소켓


```c
int main(void)
{
    int fd1, fd2, fd3;
    fd1=socket(PF_INET, SOCK_STREAM, 0);
    fd2=open("test.dat", O_CREAT|O_WRONLY|O_TRUNC);
    fd3=socket(PF_INET, SOCK_DGRAM, 0);

    printf("file descriptor 1: %d\n", fd1);
    printf("file descriptor 2: %d\n", fd2);
    printf("file descriptor 3: %d\n", fd3);

    close(fd1); close(fd2); close(fd3);
    return 0;
}
```

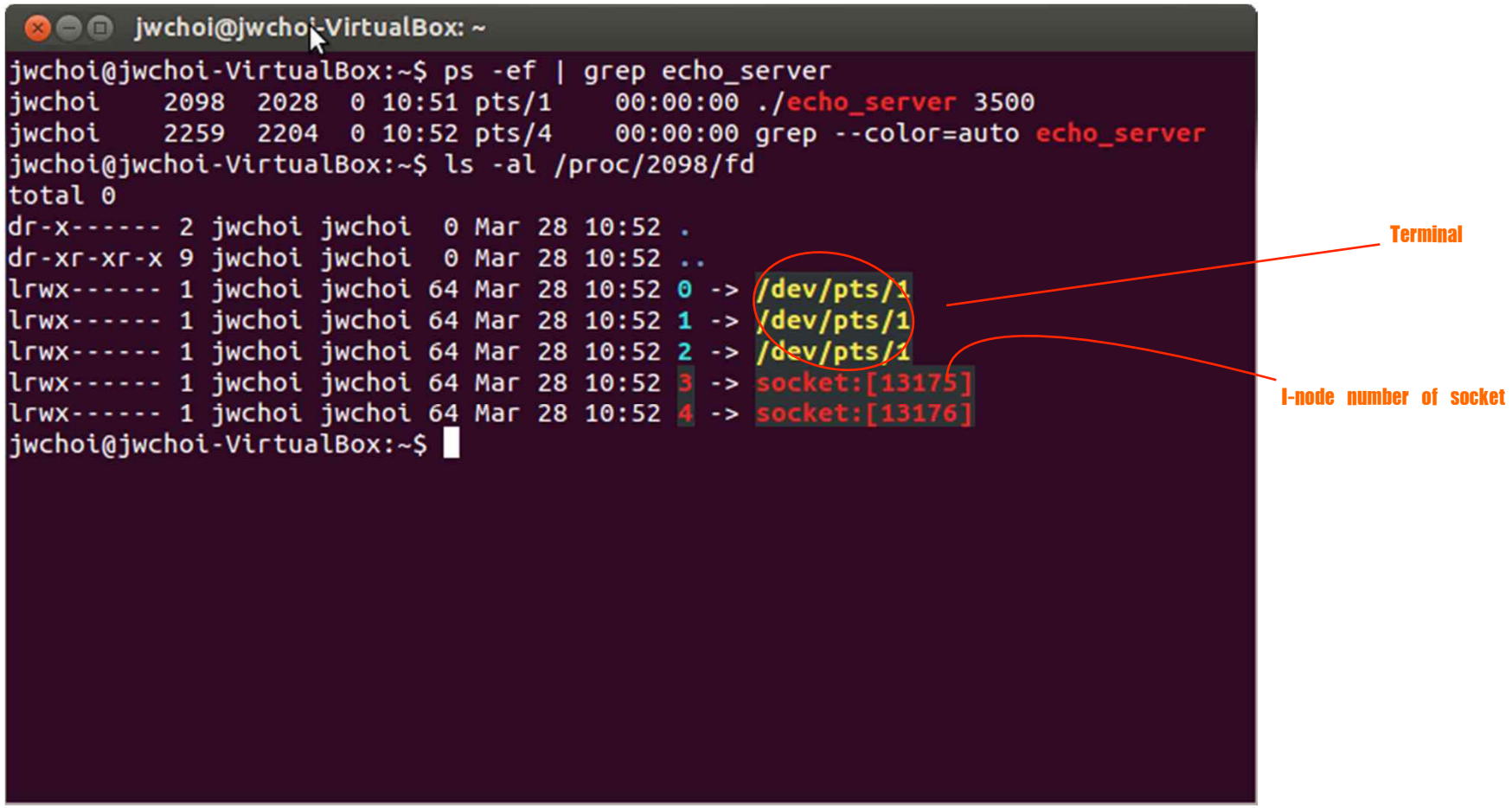
실행결과

```
root@my_linux:/tcpip# gcc fd_seri.c -o fds
root@my_linux:/tcpip# ./fds
file descriptor 1: 3
file descriptor 2: 4
file descriptor 3: 5
root@my_linux:/tcpip#
```

실행결과를 통해서 소켓과 파일에 일련의 파일 디스크립터 정수 값이 할당됨을 알 수 있다.

그리고 이를 통해서 리눅스는 파일과 소켓을 동일하게 간주함을 확인할 수 있다.

파일 디스크립터와 소켓


jwchoi@jwchoi-VirtualBox: ~
jwchoi@jwchoi-VirtualBox:~$ ps -ef | grep echo_server
jwchoi 2098 2028 0 10:51 pts/1 00:00:00 ./echo_server 3500
jwchoi 2259 2204 0 10:52 pts/4 00:00:00 grep --color=auto echo_server
jwchoi@jwchoi-VirtualBox:~$ ls -al /proc/2098/fd
total 0
dr-x------ 2 jwchoi jwchoi 0 Mar 28 10:52 .
dr-xr-xr-x 9 jwchoi jwchoi 0 Mar 28 10:52 ..
lrwx------ 1 jwchoi jwchoi 64 Mar 28 10:52 0 -> /dev/pts/1
lrwx------ 1 jwchoi jwchoi 64 Mar 28 10:52 1 -> /dev/pts/1
lrwx------ 1 jwchoi jwchoi 64 Mar 28 10:52 2 -> /dev/pts/1
lrwx------ 1 jwchoi jwchoi 64 Mar 28 10:52 3 -> socket:[13175]
lrwx------ 1 jwchoi jwchoi 64 Mar 28 10:52 4 -> socket:[13176]
jwchoi@jwchoi-VirtualBox:~$
Terminal
I-node number of socket