8) Try following code. Make /aa/bb and type some text with length longer than 50 bytes. Explain the result.

x=open("/aa/bb", O\_RDONLY, 00777);

y=read(x, buf, 10);

buf[y]=0;

printf("we read %s\n", buf);

lseek(x, 20, SEEK\_SET);

y=read(x, buf, 10);

buf[y]=0;

printf("we read %s\n", buf);

x1=dup(x);

y=read(x1, buf, 10);

buf[y]=0;

printf("we read %s\n", buf);

link("/aa/bb", "/aa/newbb");

x2=open("/aa/newbb", O\_RDONLY, 00777);

y=read(x2, buf, 10);

buf[y]=0;

printf("we read %s\n", buf);

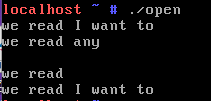
/aa/bb



open.c







/aa/bb에는 I want to go to company라는 문장이있고 x로 그것을 불러들인다. 첫번째로는 x로부터 10바이트를 읽고 출력해내기 때문에 “I want to “까지 출력된다. 그 뒤로 lseek를 통해 x의 20바이트가 되는 부분으로 포인터를 이동시키고 거기서부터 다시 10바이트를 읽기 때문에 “any”가 출력된다. 그 뒤로 dup()를 통해서 x1도 x와 같이 /aa/bb를 가리키게 한 다음 x가 읽었던 부분의 다음 부분부터 읽기 때문에 아무것도 출력하지 않게 된다. 그 다음에 link를 사용해서 /aa/newbb가 /aa/bb와 같은 파일을 가리키도록 만들고 x2를 사용해서 /aa/newbb를 불러들였다. 그러고 나서 10바이트를 읽어들이면 /aa/bb의 처음부터 10바이트를 읽어들이는 것과 같기 때문에 처음과 똑같이 I want to가 출력된다.

9) Check the inode number of /aa/bb and /aa/newbb and confirm they are same.

# ls –i /aa/\*



/aa/bb와 /aa/newbb는 같은 inode 번호를 가지고 있단걸 확인할 수 있었다.

10) Try fork() and confirm the parent and child can access the same file.

x=open("/aa/bb", ...);

y=fork();

if (y==0){

z=read(x, buf, 10);

buf[z]=0;

printf("child read %s\n", buf);

}else{

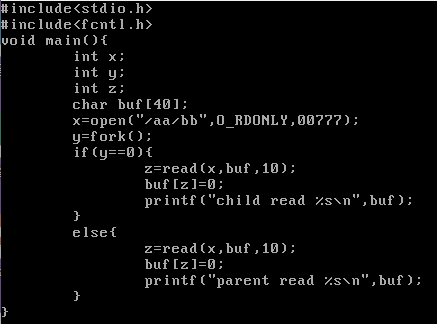
z=read(x, buf, 10);

buf[z]=0;

printf("parent read %s\n", buf);

}

fork.c





fork를 하면 child에게 부모의 body와 process descriptor가 그대로 복사되고 구조체도 그대로 상속이 돼서 같은 file table의 f\_count를 가리키게 되기 때문에 자식이 먼저 10바이트를 읽어서 “I want to “가 출력되고 그 뒤에 부모가 10바이트를 읽어서 “go to comp”가 출력됐다.

11) (Using "chroot" and "chdir") Do following and explain the result of "ex1".

a. Make f1 in several places with different content (in "/", in "/root", and in "/root/d1") as follows.

# cd /

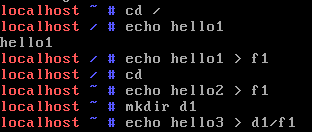
# echo hello1 > f1

# cd

# echo hello2 > f1

# mkdir d1

# echo hello3 > d1/f1



/의 f1에는 hello1을 넣고 /root의 f1에는 hello2를 넣고 /root/d1의 f1에는 hello3를 넣어주었다.

b. Make ex1.c that will display "/f1" before and after "chroot", and "f1" before and after "chdir" as follows.

display\_root\_f1(); // display the content of "/f1"

chroot(".");

display\_root\_f1();

display\_f1(); // display the content of "f1"

chdir("d1");

display\_f1();

where "display\_root\_f1()" is

x=open("/f1", ...);

y=read(x, buf, 100);

buf[y]=0;

printf("%s\n", buf);

and "display\_f1()" is

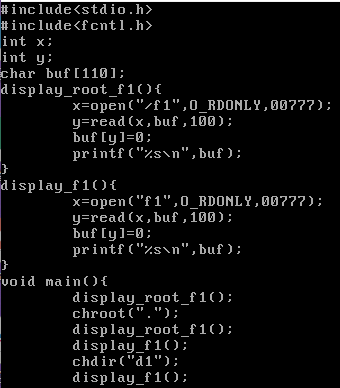
x=open("f1", ...);

y=read(x, buf, 100);

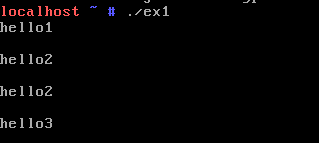
buf[y]=0;

printf("%s\n", buf);

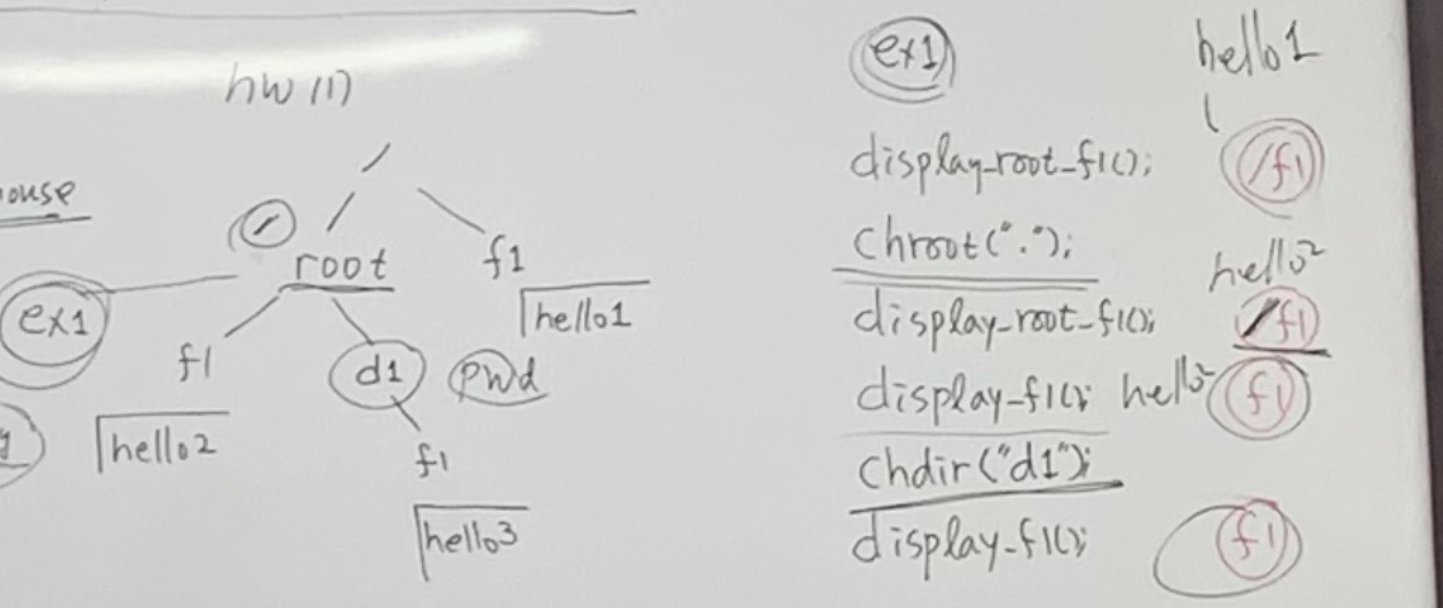
ex1.c







ex1은 기본적으로 /안에 있는 루트에 있다. /f1을 open한 뒤 출력하면 /에 있는 f1을 여는 것이 되기 때문에 hello1이 출력되고 chroot 명령어를 쓰면 /가 아닌 프로세서가 알고있는 루트로 가게된다. 그렇기 때문에 chroot 명령어 이후에 f1를 열면 프로세서가 알고있는 루트의 f1을 열게돼서 hello2가 출력된다. “/f1”은 절대경로이고 “f1”은 상대경로인데, 상대경로를 쓰면 항상 pwd의 위치에서 파일을 찾고 현재 pwd의 위치는 root이기 때문에 거기에서 f1을 열고 hello2를 출력하게 된다. chdir(“d1”) 명령어는 pwd를 d1으로 바꿔주고, 그 안에서 f1을 열면 현재 위치가 바꼈기 때문에 d1의 f1 내용인 hello3이 출력된다.



12) Make a new system call, “show\_fpos()”, which will display the current process ID and the file position for fd=3 and fd=4 of the current process. Use this system call to examine file position as follows.

x=open("f1", .............);

y=open("f2", .............);

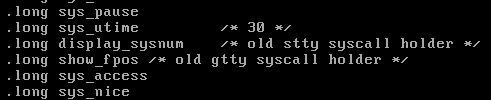
show\_fpos(); // f\_pos right after opening two files

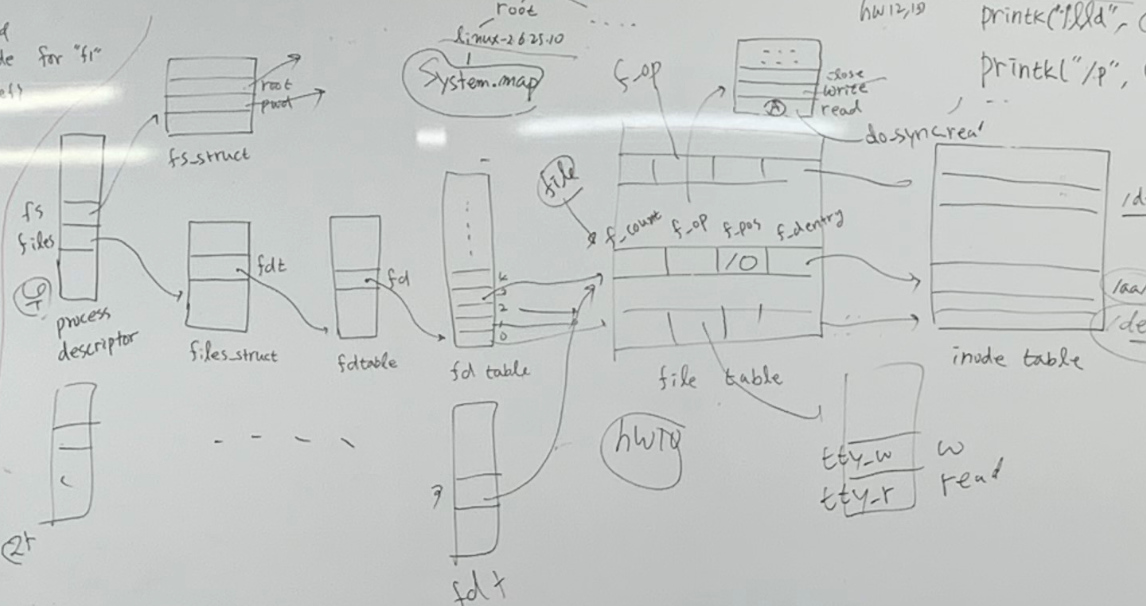
read(x, buf, 10);

read(y, buf, 20);

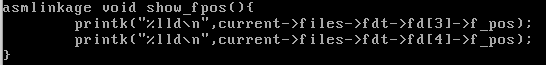
show\_fpos(); // f\_pos after reading some bytes

arch/x86/kernel/syscall\_table\_32.S



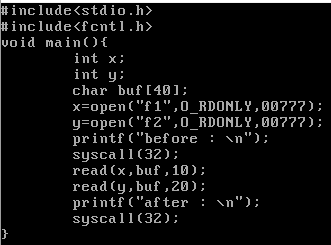


fs/read\_write.c



show\_fpos()는 파일과 관련된 함수여서 fs/read\_write.c에 정의해주고 syscall\_table\_32.S의 32번째에 명시해주었다.

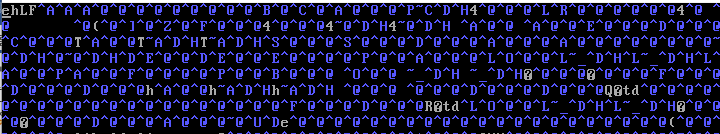
ex2.c



f1



f2

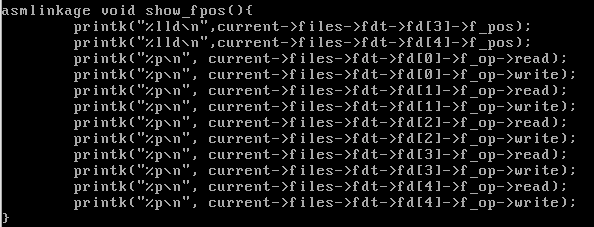


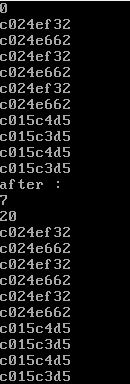


f1은 3번 파일이고 f2는 4번 파일이다. x는 f1을 열고 거기에서 10바이트를 읽어들이는데 f1 안에는 “hello2.”으로 7바이트밖에 없기 때문에 6바이트를 읽어들이고 f\_pos는 7이 된다. 반면 f2에는 많은 정보가 들어있기 때문에 20바이트 전체를 다 읽게 돼서 y가 f2를 열고 거기에서 20바이트를 읽으면 f\_pos는 20이 된다.

13) Modify your show\_fpos() such that it also displays the address of f\_op->read and f\_op->write function for fd 0, fd 1, fd 2, fd 3, and fd 4, respectively. Find the corresponding function names in System.map. Why the system uses different functions for fd 0, 1, 2 and fd 3 or 4?

fs/read\_write.c





linux-2.6.25.10/System.map







show\_fpos 함수를 고쳐서 fd 0, fd 1, fd 2, fd 3, 그리고 fd 4의 f\_op->read와 f\_op->write의 주소도 출력되도록 만들어줬다. System.map에서 출력된 주소에 대응되는 함수 이름을 찾아본 결과, fd 0, 1, 2까지는 읽고 쓰는데 tty\_read와 tty\_write함수가 사용되고 fd 3, 4에서는 do\_sync\_write와 do\_sync\_read 함수가 사용된걸 볼 수 있었다. tty\_read와 tty\_write 함수는 키보드에서 읽고 쓸 때 사용되는 함수이고 do\_sync\_write와 do\_sync\_read 함수는 regular file에서 읽고 쓸 때 사용되는 함수이기 때문에 fd 0부터 fd 4까지 다른 함수가 사용되게 된다.

14) Use show\_fpos() to explain the result of the following code. File f1 has “ab” and File f2 has “q”. When you run the program, File f2 will have “ba”. Explain why f2 have “ba” after the execution.

int f1, f2, x; char buf[10];

f1=open(“./f1”, O\_RDONLY, 00777);

f2=open(“./f2”,O\_WRONLY, 00777);

printf(“f1 and f2 are %d %d\n”, f1, f2); // make sure they are 3 and 4

x=fork();

if (x==0){

show\_fpos();

read(f1,buf,1);

sleep(2);

show\_fpos();

write(f2, buf, 1);

}else{

sleep(1);

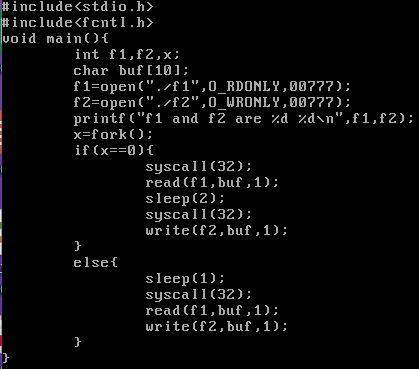
show\_fpos();

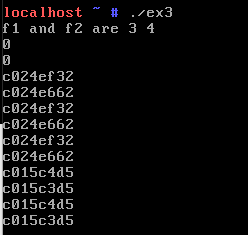
read(f1,buf,1);

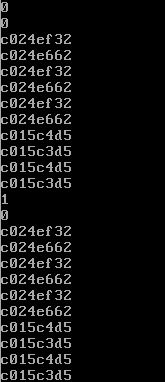
write(f2,buf,1);

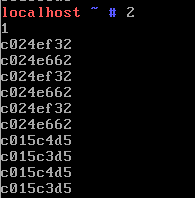
}

ex3.c









f2



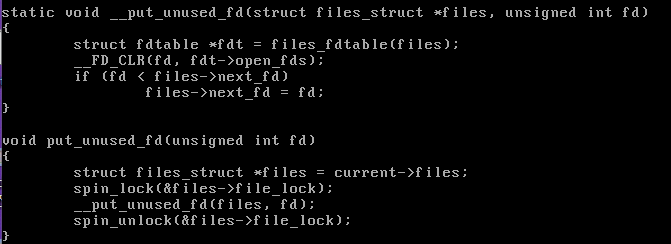
코드 중간중간에 printf를 넣어서 f3을 여러 번 실행해본 결과, fork를 쓰면 부모와 자식은 같은 file table을 가리키지만 buf까지 같은걸 쓰지는 않는단걸 알았다. 맨 처음 자식이 실행될 때 read(f1, buf, 1)에서 a를 읽어들여서 자식의 buf에 넣는다. 그 뒤로 부모가 실행되는데 read(f1,buf,1)이 실행되면 f1의 f\_pos는 1이기 때문에 a가 아닌 b가 부모의 buf에 들어가게 된다. 그 상태로 write(f2,buf,1)이 실행되면 f2에는 b가 먼저 찍히게 된다. 다시 자식으로 돌아가서 write(f2, buf, 1)이 실행되면 자식의 buf에는 아직까지 a만 들어있기 때문에 그대로 a가 찍히게 된다.

15) Find corresponding kernel code for each step below in open and read system calls:

x=open(fpath, .......);

fs/open.c

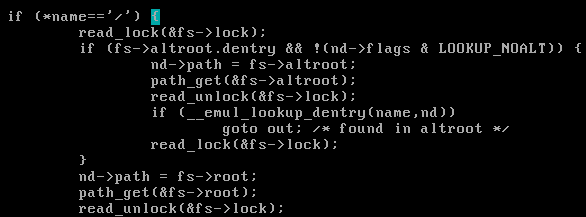
1) find empty fd



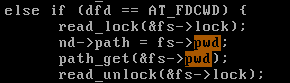
2) search the inode for "fpath"

fs/namei.c/do\_path\_lookup()

2-1) if "fpath" starts with "/", start from "fs->root" of the current process



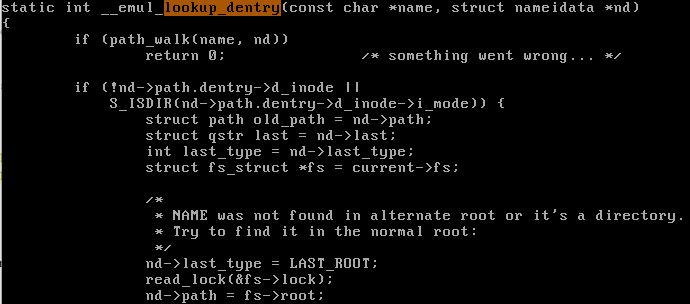
2-2) otherwise, start from "fs->pwd"

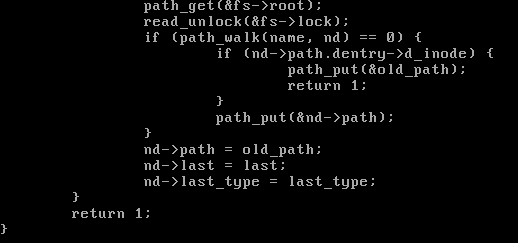


fs/namei.c/lookup\_dentry()

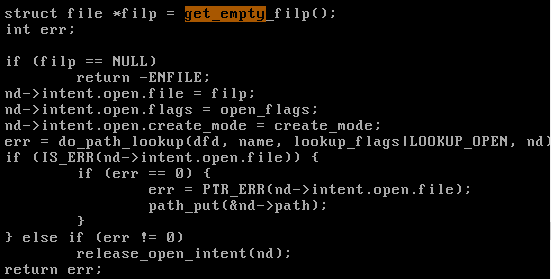
2-3) visit each directory in "fpath" to find the inode of the "fpath"

2-4) while following mounted file path if it is a mounting point.

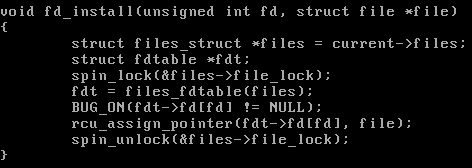




3) find empty file{} entry and fill-in relevant information.



4) chaining



5) return fd



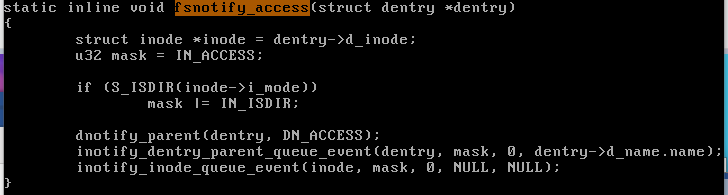
read(x, buf, n);

1) go to the inode for x

fs/read\_write.c/vfs\_read()

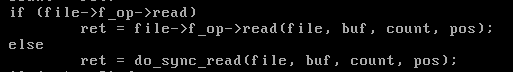


include/linux/fsnotify.h

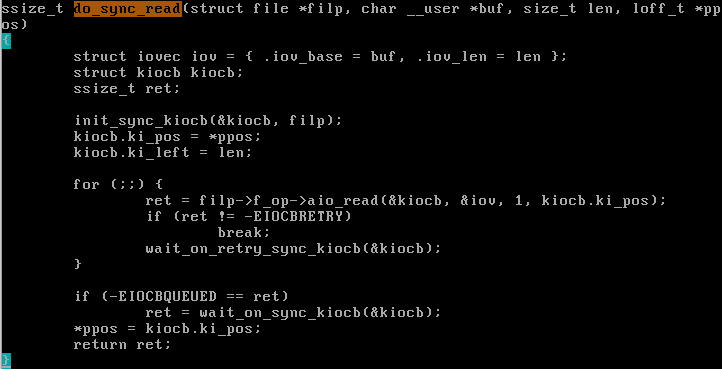


2,3) read n bytes starting from the current file position. save the data in buf.

fs/read\_write.c/vfs\_read()



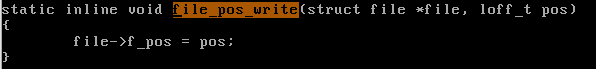
fs/read\_write.c/do\_sync\_read()



4) increase the file position by n

fs/read\_write.c/sys\_read()





vfs\_read가 실행된 이후 업데이트된 새로운 pos 값을 file->f\_pos에 대입시켜주는걸 확인할 수 있다.