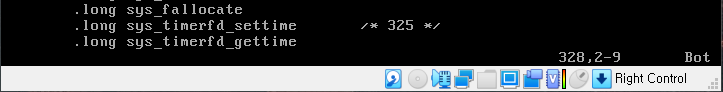
**오퍼레이팅시스템 4주차**

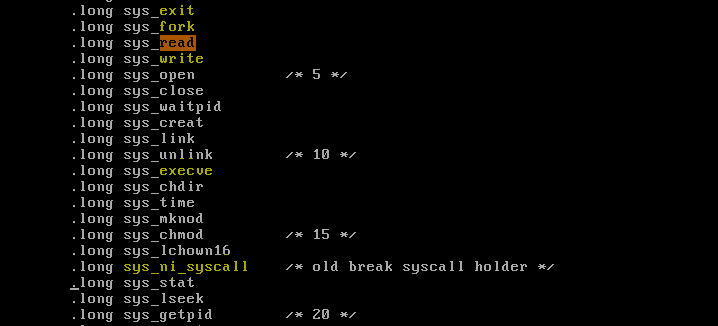
**12180626 성시열**

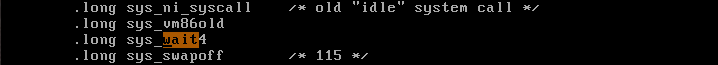
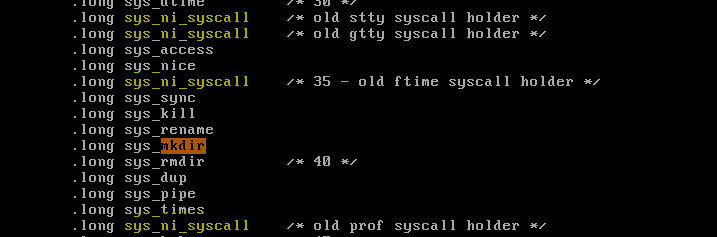
7) sys\_call\_table[] is in arch/x86/kernel/syscall\_table\_32.S. How many system calls does Linux 2.6 support? What are the system call numbers for exit, fork, execve, wait4, read, write, and mkdir? Find system call numbers for sys\_ni\_syscall, which is defined at kernel/sys\_ni.c. What is the role of sys\_ni\_syscall?





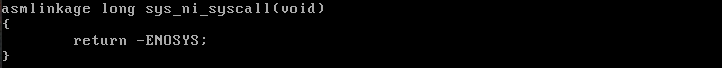
해당 위치로 이동하였고, 넘버링을 통해 326번까지 존재함을 확인하였다. 0번부터 시작하였으므로 총 개수는 327개이다.





exit - 1, fork - 2, execve - 11, wait4 - 114, read - 3, write - 4, and mkdir – 39

Find system call numbers for sys\_ni\_syscall 17, 31, 32, 35, 44, 112,,,, 곳곳에 있음을 확인하였다.



kernel/sys\_ni.c 를 확인해본 결과 -ENOSYS라는 값을 return 해주는데 이는 아무런 역할을 하지 않음을 확인하였다.

8) Change the kernel such that it prints "length 17 string found" for each printf(s) when the length of s is 17. Run a program that contains a printf() statement to see the effect. printf(s) calls write(1, s, strlen(s)) system call which in turn runs

mov eax, 4 ; eax<--4. 4 is system call number for “write”

int 128

INT 128 will make the cpu stop running current process and jump to the location written in IDT[128]. IDT[128] contains the address of system\_call (located in arch/x86/kernel/entry\_32.S). Finally, system\_call will execute

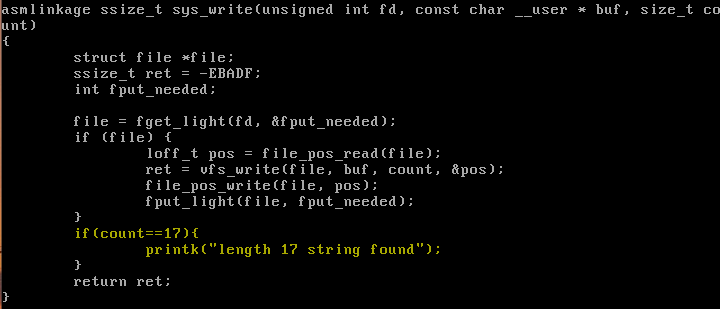
call \*sys\_call\_table(,%eax,4)

which eventually calls sys\_write() since eax=4 for write() system call (the target function location is sys\_call\_table+eax\*4).

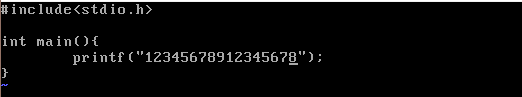
\* Sometimes the the system call runs "sysenter" instead of "int 128". In this case the cpu jumps to ia32\_sysenter\_target (also in entry\_32.S) instead of system\_call.

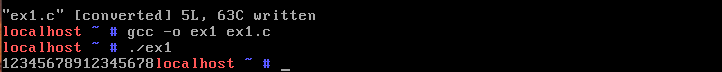


write 명령어를 수정하기 위해 해당 위치로 찾아갔다.



write가 정의된 함수에 if문을 추가하여 17글자일 때 해당 출력문이 출력되도록 하였다.





ex1.c 코드를 위와같이 작성하고 컴파일한 뒤 실행한 결과 기존 실행코드에 변화가 없음을 파악하였다.



저번 과제를 활용하여 printk의 우선순위를 변경하였더니 원하는대로 출력됨을 확인하였다.

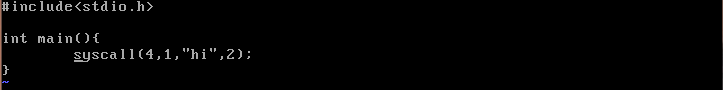
9) You can call a system call indirectly with “syscall()”.

write(1, “hi”, 2);

can be written as

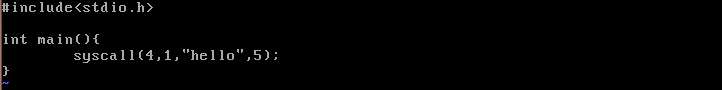
syscall(4, 1, “hi”, 2); // 4 is the system call number for “write” system call

Write a program that prints “hello” in the screen using syscall.





hi가 잘 출력됨을 알 수 있다.





hello를 출력하기 위해 뒤에 숫자를 5로 바꿔주어 5글자만큼을 출력하였다. hello가 잘 출력된 모습이다.

10) Create a new system call, my\_sys\_call with system call number 17 (system call number 17 is one that is not being used currently). Define my\_sys\_call() just before sys\_write() in read\_write.c. Write a program that uses this system call:

void main(){

syscall(17); // calls a system call with syscall number 17

}

When the above program runs, the kernel should display

hello from my\_sys\_call

To define a new system call with syscall number x

- insert the new system call name in arch/x86/kernel/syscall\_table\_32.S

at index x

- define the function in appropriate file (such as "read\_write.c")

asmlinkage void my\_sys\_call(){

printk("hello from my\_sys\_call\n");

}

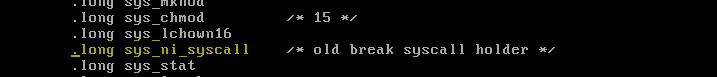
- recompile and reboot

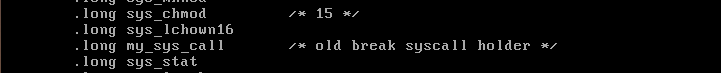
To use this system call in a user program

- void main(){

syscall(x);

}

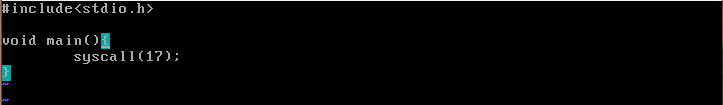
Arch/x86/kernel/syscall\_table\_32.S를 통해 system call num을 확인하고 17번을 확인해본 결과 sys\_ni\_syscall로 비어있음을 알 수 있다.



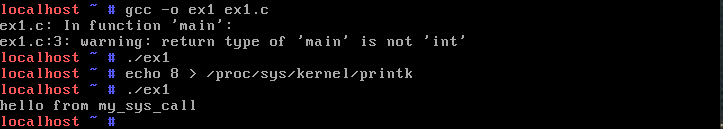
위와 같이 my\_sys\_call로 변경한 모습이다.



fs/read\_write.c로 이동하여 my\_sys\_call() 함수를 추가하고, Recompile & reboot를 하였다.

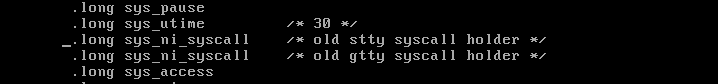


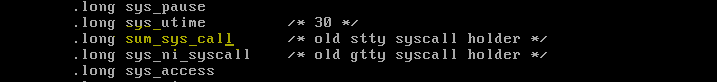
ex1.c 파일을 다음과 같이 바꿔주어 system call 17번을 호출한다.



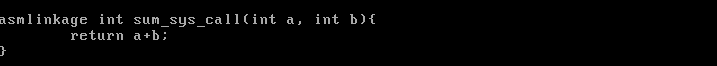
컴파일 후 실행한 결과 실행이 안됨을 확인하였고, 이전 문제와 같이 문제를 해결하여 출력이 잘 됨을 확인하였다.

10-1) Create another system call that will add two numbers given by the user.

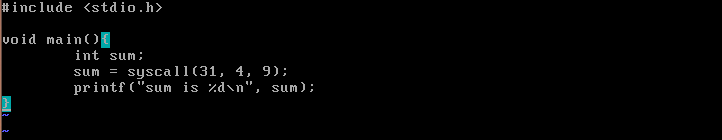




31번 위치로 가서 sum\_sys\_call로 변경하였다.



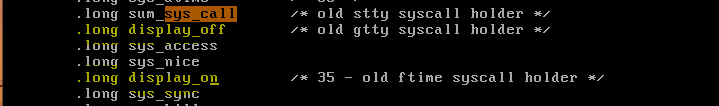
read\_write.c로 이동하여 함수를 추가하였다.



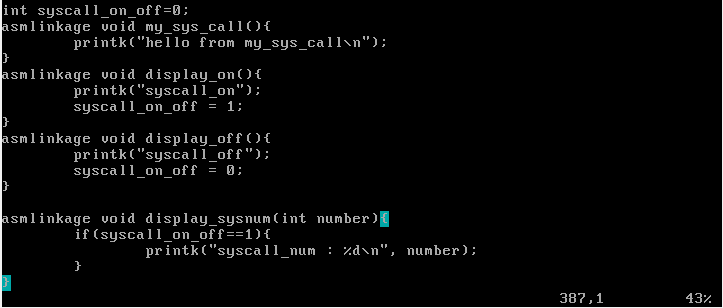


원하는 결과가 출력됨을 알 수 있다.

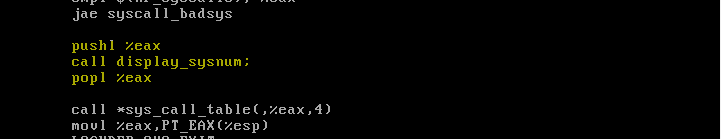
11) Modify the kernel such that it displays the system call number for all system calls. Run a simple program that displays "hello" in the screen and find out what system calls have been called. Also explain for each system call why that system call has been used.

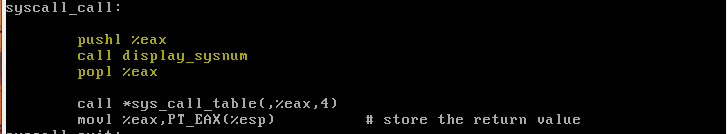


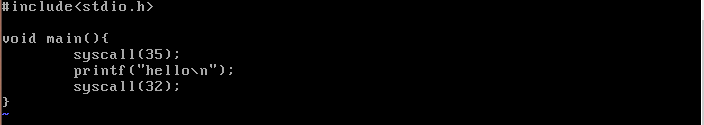
arch/x86/kernel/syscall\_table\_32.S로 이동하여 32번에 display\_off, 35번에 display\_on를 추가하였다.



fs/read\_write.c로 가서 출력을 통제하는 함수를 추가해주었다.



 arch/x86/kernel/entry\_32.S에서 call \*sys\_call\_table 직전에 다음과 같은 코드들을 추가한다.



ex1 코드에서 display\_on display\_off를 순서대로 호출한다.





실행해본 결과 위와 같이 나왔다. 각 번호를 찾아본 결과 아래와 같다.

197 : sys\_fstat64

54 : sys\_ioctl

192 : sys\_mmap2

4 : sys\_write

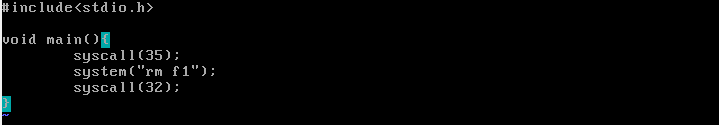
32 : display\_off

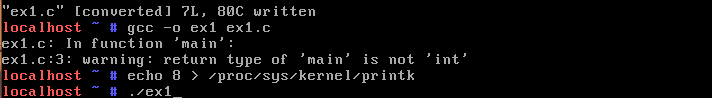
12) What system calls are being called when you remove a file? Use "system()" function to run a Linux command as below. Explain what each system call is doing. You need to make f1 file before you run it. Also explain for each system call why that system call has been used.

...........

system("rm f1");

...........



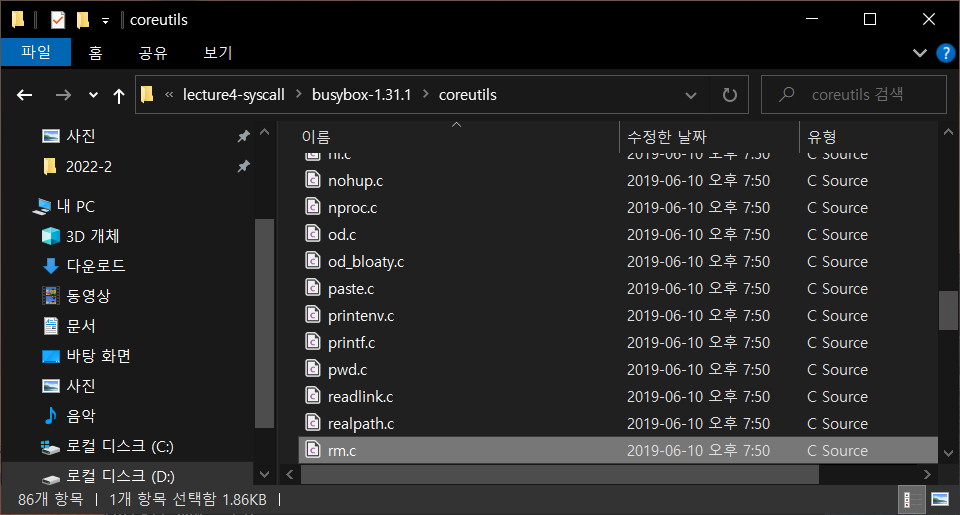


ex1을 수정하고 다시 실행해보았다.



192, 6, 192, 243, 125, 125, 125, 61, 45, 45, 54, 300, 4, 4, 4, 4, 6, 6, 252, 174, 174, 175, 32번의 명령어가 실행됨을 알 수 있다. 실행되자마자 호출되는 명령어는 보이지 않았다.

13) Find rm.c in busybox-1.31.1 and show the code that actually removes "f1". Note all linux commands are actually a program, and running "rm" command means running rm.c program. "rm" needs a system call defined in uClibc-0.9.33.2 to remove a file. You may want to continue the code tracing all the way up to "INT 0x80" in uClibc for this system call.



rm.c 코드를 찾고 활용하기 위해 Busybox 폴더 내에서 rm.c 코드를 찾았으나 리눅스 내에서 busybox를 활용하는 방법에 대해 파악하지 못하였다.