1.

우선 gentoo를 pc에 있는 가상머신에 설치한 경우이므로 'arch'의 하위 디렉토리가 **'x86'이 아닌 경우는 모두 제외**됩니다. 그리고 32비트 cpu를 가정하고 있으므로 x86 안에서도 **'32 비트용 파일'**을 선택합니다.

위와 같이 함수를 찾은 후 그 함수가 실제 리눅스가 부팅될 때 호출되는 함수인지를 확인하기위해, 해당 함수안에 'printk()'로 메시지를 출력하는 코드를 삽입해서 이 메시지가 출력되는지를 확인합니다.

2.

리눅스가 부팅될 때 제일 처음 실행되는 함수 : **'init/main.c' 내 'start_kernel()'함수**

3.

리눅스가 부팅될 때 호출되는 초기화 함수들은 함수명 앞에 '__init'가 선언되어 있다.

4.

커널 코드 내 추가한 'printk()'를 화면에 출력하고 싶다면, **'echo 8 > /proc/sys/kernel/printk'** 명령을 먼저 쳐야함

5.

리눅스로 설명하자면, 'CPU IDLE time'은 **CPU가 'cpu_idle()'에서 무한루프를 돌고 있는 시간**을 말한다. CPU를 점유하는 다른 프로세스들이 없다면, **cpu_idle()함수를 호출하는 프로세스(pid = 0, 해당 프로세스는 top 명령어, ps -ef 명령어로 출력되지 않음)가 CPU를 점유하고 해당 CPU는 cpu_idle()함수 내 존재하는 무한루프에서 위치**하게 된다. 이와 같이 CPU가 cpu_idle()함수의 무한루프에 오랫동안 위치하게 되면, CPU IDLE time이 증가하게 된다.

6.

인터럽트가 발생하면 'IDT(Interrupt Descriptor Table) -> ISR1(entry_32.S에 선언되어 있음) -> ISR2(임의의 파일에 선언되어있음)' 순서로 명령이 실행된다.

7.

IDT in arch/x86/kernel/traps_32.c/trap_init()(for exception interrupts and system call

interrupt)

IDT in arch/x86/kernel/i8259_32.c/init_IRQ()(for hardware interrupts)

8.

하드웨어 인터럽트, 예외 인터럽트의 ISR1들도 entry_32.S에 있습니다. entry_32.S에 **ENTRY(xxx) 로 표시된 모든 함수는 xxx 라는 인터럽트 핸들러 (ISR1)입니다.** 예를 들어 'INT 0'은 ENTRY(divide_error)가 ISR1입니다.

'ENTRY()'함수(ISR1)안에 해당 ISR2 함수가 호출된다.(즉, 여기서 ISR2 함수명이 호출된다.)

ISR1이 선언되어 있는 경로 : /root/linux-2.6.25.10/arch/x86/kernel/entry_32.S

9.

'ENTRY(sys_call_table)[system call interrupt에 대한 ISR1]'은 'arch/x86/kernel/syscall_table_32.S'에 선언되어있다.

10.

ISR1에 등장하는 **'eax(매개변수)'에는 system call number가 저장**된다.

11.

'sys_write()'함수의 매개변수인 count에는 출력할 문자열의 문자 개수가 저장되고, *buf에는 출력할 문자열이 저장된다.

12.

'atkbd_interrupt()'내 int형 지역변수 code에는 스캔코드 값이 저장된다.

13.

PID(Process ID) 와 PPID(Parent Process ID) : 프로세서 식별용 값을 의미하며 리눅스는 기본적으로 0,1,2 의 프로세스를 기본으로 가지고 시작된다. **이중 1번이 거의 대부분의 프로세스들의 부모역할을 한다.** 이는 리눅스 시스템 특성에 기반을 둔 것으로, **1번 프로세서에서 fork 되어 생성되기 때문이다.** 추가적으로, 부모 프로세서가 종료 되면 하위 프로세서들은 모두 종료 된다.

14.

0 는 runnable state, 1은 block state를 의미합니다. 'include/linux/sched.h'에 각 스테이트가 정의되어 있다.

15.

sleep 상태는 sleep()함수가 호출된 후 정해진 시간만큼 sleep하고 있는 상태이고, 좀비상태는 죽은 상태입니다. 좀비상태 프로세스는 부모가 wait을 호출하면 완전히 사라집니다.

sleep 상태는 프로세스 내에서 sleep 함수를 호출했을 때 빠지게 되는 상태이고, 좀비상태는 exit을 호출했을 때 빠지게 되는 상태입니다

16.

'task_struct { }'는 process descriptor이고 '구조체'이다.

해당 구조체의 멤버 변수로는 pid, comm(현 프로세스를 실행시킨 '실행파일 이름'), files, mm 등이 있다.

선언 경로 : include/linux/sched.h

17.

init_task는 'pid = 0'인 프로세스의 process descriptor이다. 당연히, task_struct 구조체에 의해 선언된 것이다.

선언 경로 : arch/x86/kernel/init_task.c, include/linux/init_task.h

18.

```
void display_processes(void){

        struct task_struct *temp;
        temp = &init_task;

        for(;;){
                printk("PID : %4d, PPID : %4d, UID : %4d, State : %4d, ",
                temp->pid, temp->parent->pid, temp->uid, temp->state);
                printk("Command : %s\n",temp->comm);

                temp = next_task(temp);
                if(temp == &init_task)break;
        }
}
```

현재 프로세스들의 process descriptor를 출력하는 함수선언

19..

특정 커널 c파일 내 새로운 함수를 선언한다면, 파일 앞부분에 해당 함수에 대한 프로토타입을 선언하는 것이 커널 컴파일 오류 방지에 좋다.

20.

process Image는 코드, 데이타, 스택 등 3부분으로 구성됩니다.

코드에는 명령어 코드들, 데이타에는 광역 변수와 동적으로 할당되는 메모리, 그리고 스택에는 지역변수와 함수 리턴 주소가 저장됩니다.

21.

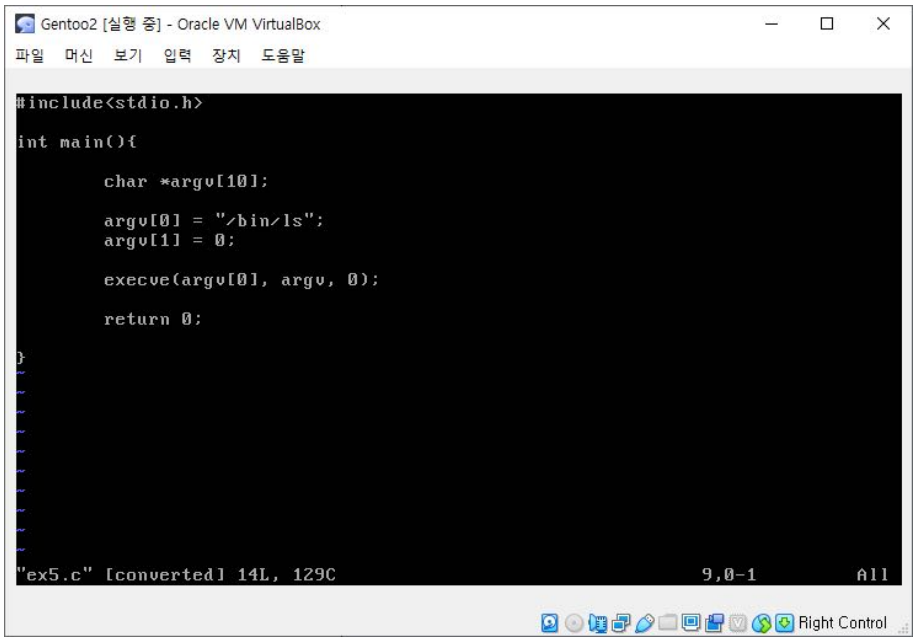
fork 는 독립적인 프로세스를 만들 때 사용합니다.

pthread_create은 데이터를 공유하는 프로세스(쓰레드)를 만들 때 사용합니다.

kernel_thread는 커널안에서 쓰레드를 만들 때 사용합니다.

exec하면 현재 프로세스는 사라지고, 사라진 프로세스가 인자로 들어온 프로그램으로 대체됩니다.

int execve(const char *path, char *const argv[], char *const envp[]);'가 execve()함수의 선언부분이다. argv와 envp는 포인터 배열이고, 해당 배열의 마지막에는 NULL 포인터가 저장되어야한다.



22.

execve()함수는 실행시킬 프로그램 경로 인식을 성공할시 아무것도 반환하지 않고, 실패할시 -1을 반환한다.

23.

'dd명령어'와 'mkfs명령어'를 실행하여 가상 디스크를 만들 수 있다. **특정 파일을 mkfs명령어로 포맷하는 순간, 해당 파일은 '디스크'의 존재가 돼버린다.** 이렇게 생성된 가상 디스크를 빈 디렉토리에 mount(mount 명령어)시켜야, 이제 해당 가상 디스크 내에 특정 파일들을 저장할 수 있게 된다.

24.

생성된 가상디스크의 정보를 16진수로 보기 위해선, 'xxd 명령어'를 사용해야한다. 그리고 xxd명령어를 사용하기 전에 반드시 가상디스크를 unmount해야 한다.

25.

1024를 16진수로 변환하면 0x400이다. 4096을 16진수로 변환하면 0x1000이다.

26.

EXT2 파일 시스템의 첫 번째 block은 사용되지 않는다. 즉 block 1부터 시작이다.

27.

block 1이 superblock이고, 해당 superblock내에는 총 아이노드 개수(사용되지 않고 있는 아이노드도 포함), 총 block 개수(사용되지 않고 있는 block도 포함), superblock의 주소상 위치, 한 개의 block size(1024 * 2^n) 등의 정보가 들어있다.

28.

superblock은 Ext2 파일시스템에서 사용되는 주요 설정 정보들이 기록되는 영역으로 block 그룹의 첫 번째 block**(block 1)**에 위치하며, 파일시스템에서 설정한 block의 크기가 1KB든, 4KB든 실제 위치하는 곳은 같다. 즉, 디폴트 block 크기인 1KB인 경우에 맞춰 4KB block일 때도 3KB가 패딩되지 않는다. **superblock은 반드시** block 그룹의 시작부터 1024Byte 내에 기록되어야 하며, **1024Byte의 크기로 저장되어야 한다.**

29.

Group Descriptor**(block 2에 위치)**는 IBM, DBM, Inode table의 block number를 담고 있고, 각각 4byte에 걸쳐서 담고 있다.

30.

inode table내에선 한 개의 inode가 128byte(0x80)의 크기를 차지한다. 그리고 inode 1은 사용되지 않는다.(즉, **inode table 내 inode는 '1'부터 시작하고, inode 1은 사용되지 않는다.**)

31.

한 개의 inode는 한 개의 파일에 대한 정보를 내포한다.

32.

한 개의 inode 내 0x4부터 4byte에 걸쳐서(즉, 0x4 ~ 0x7까지) 해당 파일의 크기가 나타나 있다.

33.

한 개의 **inode 내 0x28부터 해당 파일의 block number가 4byte에 걸쳐서 나타나있다.**

34.

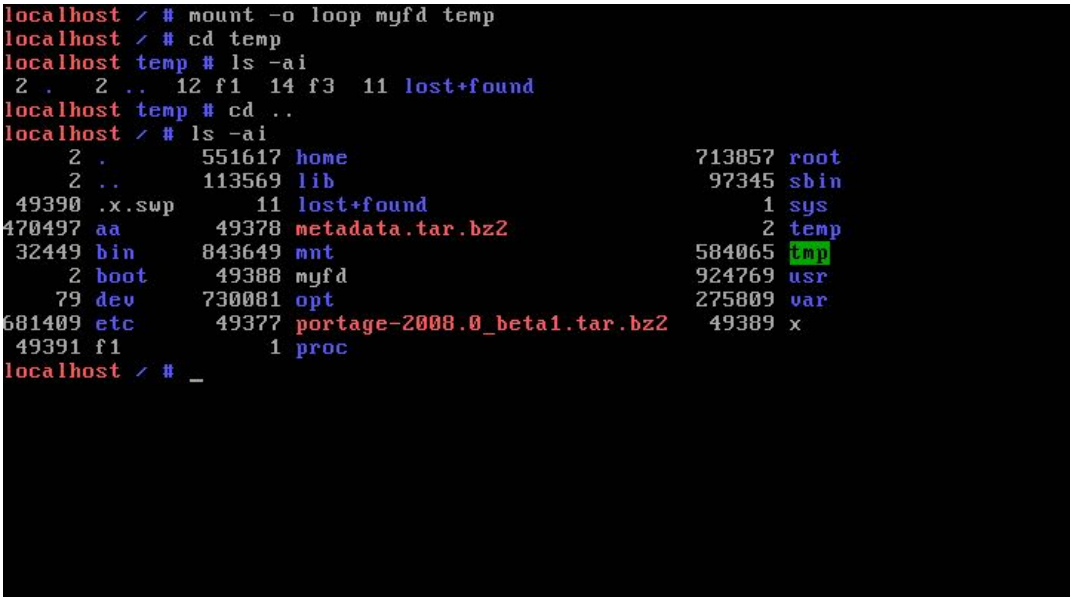
directory file의 block 내에는 inode number(4byte를 차지), record length(2byte를 차지), name length(1byte를 차지), file type(1byte를 차지), file name(유동적임)에 관한 정보가 나타나있다.

directory file의 block을 읽을 때는 record length부터 파악하는 것이 우선이다. 왜냐하면, 이를 통해 해당 directory 내에 몇 개의 파일들의 정보가 몇 byte에 걸쳐서 나타나있는지 알 수 있기 때문이다.

35. 'ls -i'명령어를 치면 해당 디렉토리 내 존재하는 파일들의 inode 번호가 출력된다.

36.

특정 디스크 내부에 존재하는 모든 파일들에 대해 inode 2부터 배정하는 것이다. 서로 다른 두 디스크 내 존재하는 파일들은 서로 독립적이다.



37.

특정 파일을 지우고 새로운 파일을 생성하면, 새로운 파일의 inode number는 지워졌던 inode number를 부여받는다.(단, 두 파일의 type이 같아야한다. 만약 두 파일의 type이 다르다면, 새로운 파일의 inode number는 지워졌던 파일의 inode number와 다르다.)

38.

mount_root()함수가 root file system(root device의 file system)을 메모리에 적재한다.

39.

- superblock 구조체

on-disk : ext2_super_block{}, on-mem : super_block{}

- inode 구조체

Individual inode is cached when a file including the inode is accessed by the system.(즉, 디바이스 내 현재 사용되어지는 file의 inode들만 caching된다.)
on-disk : ext2_inode{}, on-mem : inode{} (include/linux/fs.h)

- dentry table

For each cached directory entry, dentry{} structure is defined
For example, when reading "/aa/bb", three dentry objects are created: one for "/", another for "aa", and the last for "bb".
dentry{} (include/linux/dcache.h)

- vfsmount

어떤 디스크가 특정 mounting point에 mount되면 생성되는 구조체이다.
해당 구조체는 mount된 디스크의 정보와 mounting point에 대한 정보를 내포하고 있다.

40.

다른 디스크를 mount를 하게 되면, 해당 디스크의 정보(superblock, root directory file's inode)가 caching된다. 그리고 mounting point의 정보(해당 mounting point가 속한 root directory file's block, mounting point's inode)도 caching된다.

41.

fd table(file descriptor table)을 통해서, 특정 프로세스가 몇 개의 파일을 오픈하였는지 알 수 있다. fd table은 fild{}구조체의 주소를 담고 있는 pointer array이다.

42.

link()는 파일에 대해서만 새 이름을 생성하며, 생성된 이름으로 같은 파일을 사용할 수 있습니다. 즉, 리눅스에서는 하나의 파일에 여러 이름을 지정할 수 있으며, 생성된 이름 어느 것으로 파일 내용을 수정하면 다른 이름으로 열어 보아도 수정된 내용으로 볼 수 있습니다.

43.

chroot()와 chdir()의 매개변수로 들어가는 경로의 기준은 현재 프로세스가 속해있는 디렉토리이다.

44.

remember the location of each page in a **page table(task_struct->mm->pgd)**

45.

In default, 1 page= 1 page frame = 4K byte = 4096 byte = 0x1000 byte.

46.

printf()에 함수명만 기입해도 해당 함수의 메모리상의 주소값이 나온다.(즉, 굳이 함수에 '& 연산자'를 사용할 필요가 없음!)

47.

'**cat /proc/(pid of ex1)/maps > x1**' 해당 명령어로 해당 프로세스의 process image를 볼 수 있다. 해당 명령어에 의해 출력된 것을 memory map이라 부른다.

48.

- 대부분 프로세스의 process image는 다음과 같다.

code(명령어[함수, 제어문 등]) at 08048000-08049000 (page number 8048)
read-only (such as linking info) data at 08049000-0x80da000 (page number 8049)
data at 0804a000-0804b000 (page number 804a)
c library code at b7de8000-b7f12000 (page number b7de8 to b7f11)
c library data at b7f14000-b7f15000 (page number b7f14)

49.

Remember a page fault will raise INT 14 which will be processed by page_fault() in arch/x86/kernel/entry_32.S, which in turn calls do_page_fault() in arch/x86/mm/fault.c.(즉, page fault가 발생하면, 해당 인터럽트에 대한 ISR2인 do_page_fault()가 실행된다.)

50.

VMA(Virtual Memory Area)는 현재 프로세스가 적재되어 있는 '가상메모리공간'을 뜻한다.