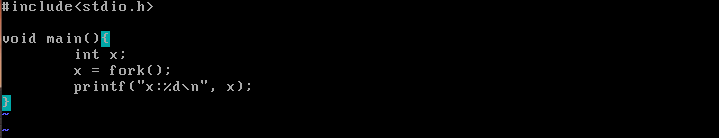
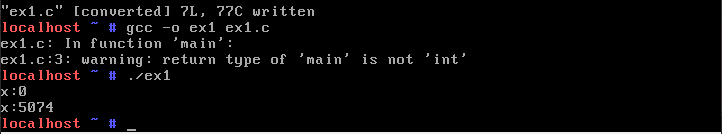
Homework

1) Run the program below. What happens? Explain the result.



vi ex1.c를 통해 코드를 수정하였다.



컴파일 후 실행결과 x가 복제됨을 확인하였다. 출력된 숫자는 pid를 의미하며, x:0이 child이고, x:5074는 parent이고 child의 pid를 return 하였다.

fork

1 int x의 body를 복사

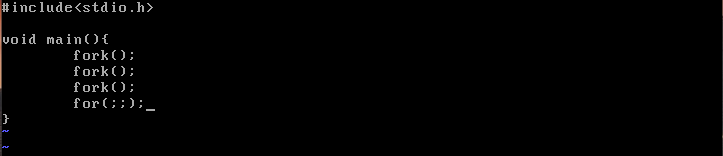
2 x의 process descriptor까지 복사

3 복사된 대상을 child로 하고, child의 process descriptor의 pid, mem위치 등을 조정

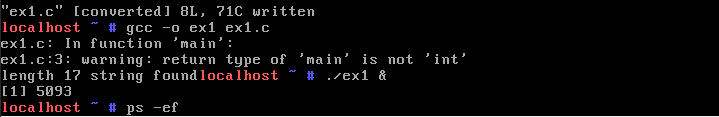
4 child를 process queue에 삽입, 부모와 자식을 링크

5 child는 0을 return, parent는 child의 pid를 return함.

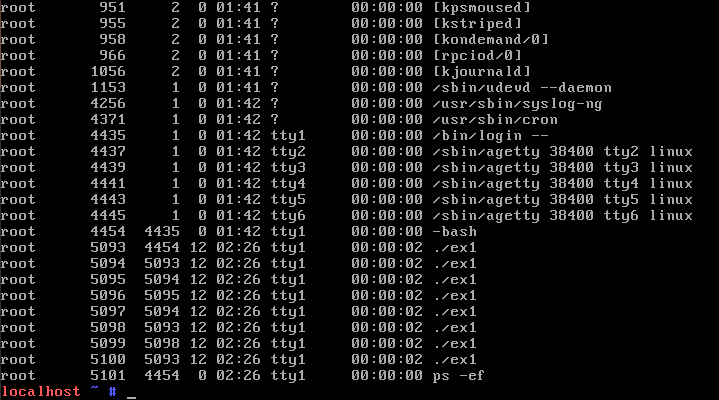
2) Try below and explain the result.



vi ex1.c를 통해 코드를 수정하였다.

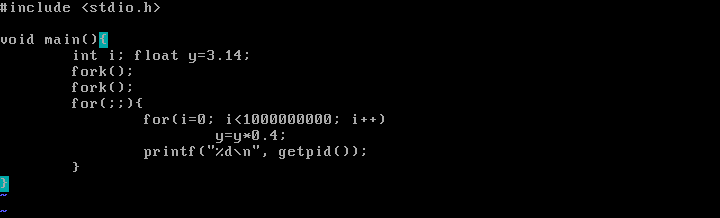


컴파일 후 실행하였다. 무한 루프가 있으므로 실행에 &를 붙여주었다. 이후 실행중인 프로세스를 확인해보았다.

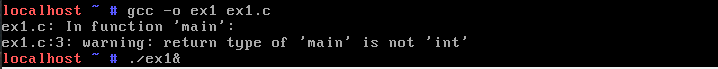


무수한 프로세스들이 출력되었고, ./ex1이 8번 실행되고 있음을 확인하였다. 처음 복제로 2개가 된 뒤, 다음 복제로 4개가 되고, 마지막 복제로 8번이 되었을 것이라 추측된다. 이후 방금 실행했던 ps -ef가 출력된다.

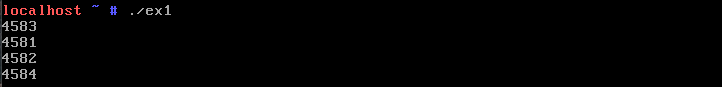
3) Run following code. What happens? Explain the result.



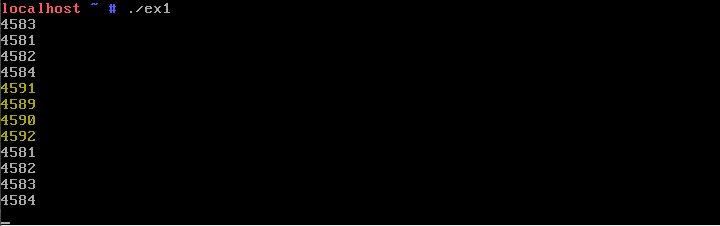
해당 코드를 vi ex1.c 를 통해 작성하였다.

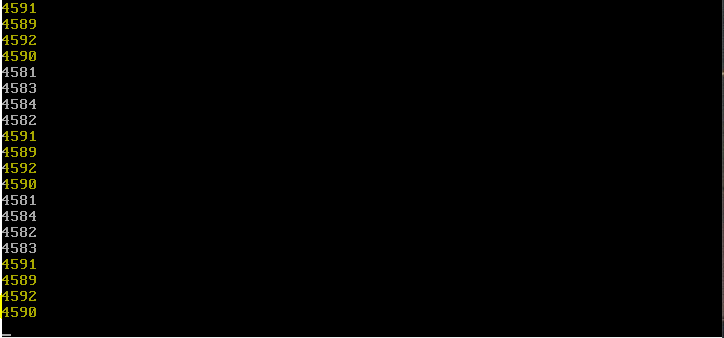


컴파일 후



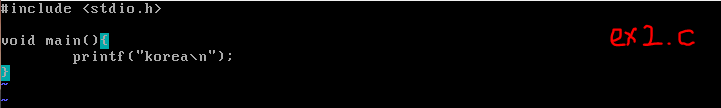
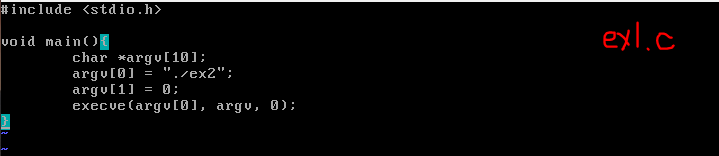
해당 코드를 실행시켜보았다. 처음엔 아무런 값이 출력되지 않았다. 일정 시간이 지난 뒤 4개의 출력문이 출력되었다.

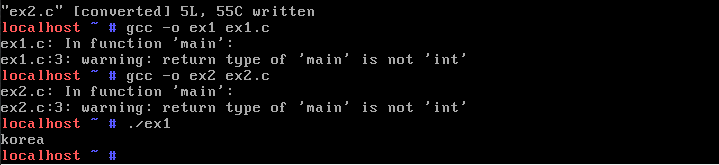




이후 위와 같이 일정 시간이 지날 때마다 4개의 pid가 출력되었다. 코드 내 무한루프로 인해 지속적으로 출력문이 출력되어 kill -9 (pid)를 통해 무한루프를 강제 종료시켰다. 해당 pid들은 4581, 4582, 4583, 4584 와 4589, 4590, 4591, 4592 로 두 세트가 있는데 이 내에서 순서는 스케줄러가 중요하게 여기는 순서대로 출력되므로 계속 순서가 바뀔 수 있다.

4) Try below and explain the result.





ex1을 실행하였으나 ex2가 실행된 모습이다. ex1 코드를 살펴보자.

execve

1. old body인 ex1의 body를 제거한다.

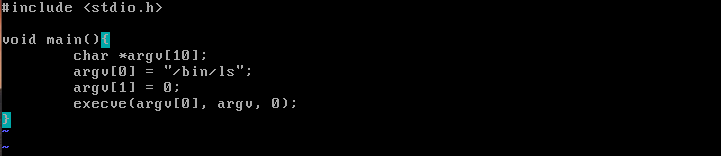
2. new body인 ex2의 body를 load한다.

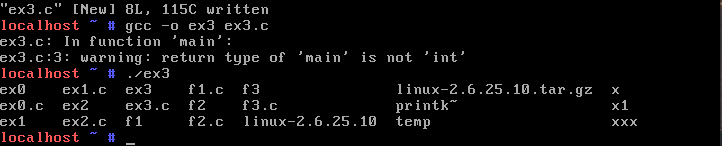
3. 기존 ex1의 process descriptor를 ex2로 연결한다.

4. new body인 ex2를 실행한다.

위 과정을 통해 ex1을 실행하더라도 ex2가 실행된 모습이다.

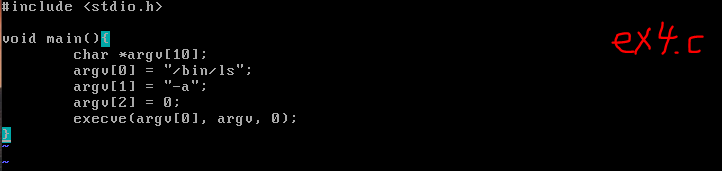
5) Run following code and explain the result.

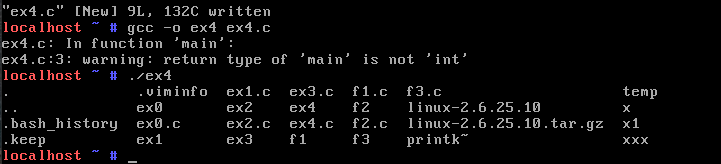




ex3로 새로운 파일을 생성하였다. 4)번에서 설명한 exevce 원리를 통해 ex3를 실행하면 /bin/ls가 실행되어 현 위치의 파일 목록을 알려준다.

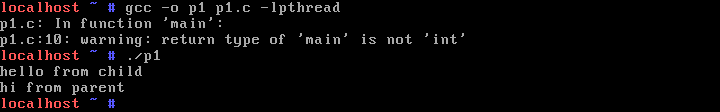
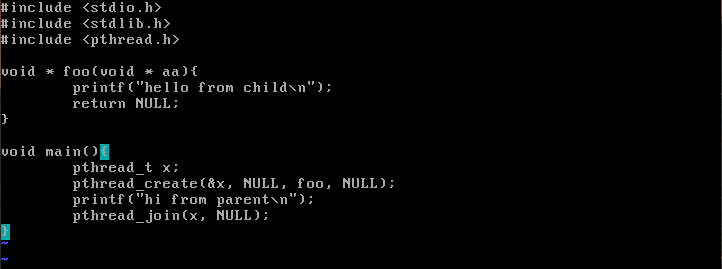
6) Run following code and explain the result.





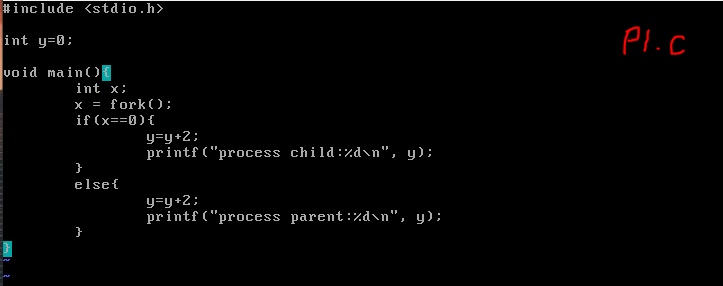
ex4로 새로운 파일을 생성하였다. -a 옵션을 추가하기 위해 argv[1]에 넣어주었고, 원하는 명령어를 모두 입력한 뒤 0을 넣어주어 명령어를 모두 입력했다는 것을 알려준다. 4)번에서 설명한 exevce 원리를 통해 ex4를 실행하면 /bin/ls -a가 실행되어 숨겨진 파일이나 디렉토리도 보여준다.

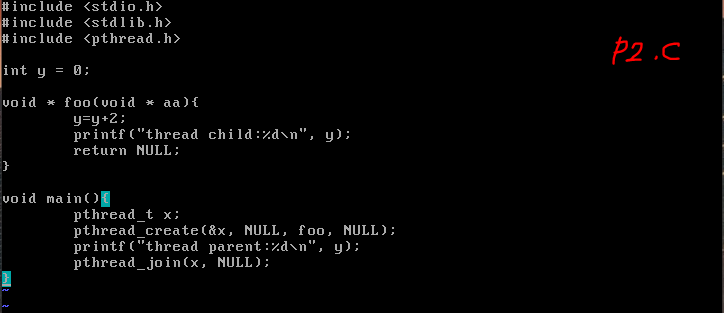
7) Run following code and explain the result.

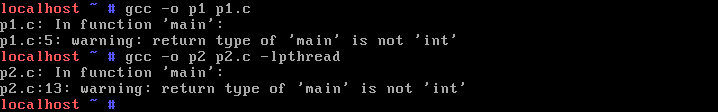


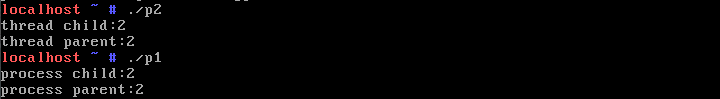
컴파일할 때 -lpthread 명령어를 추가하여 pthread와 관련된 함수가 호출된 코드를 컴파일 하였다. p1의 main 함수에서 사용된 pthread\_create는 fork와 유사하지만 body를 복사하지 않는 코드이다. 그렇기 때문에 body가 부모-자식간 공유가 되므로 pthread\_create 과정 이후에 다시 실행을 재개할 때, 자식은 pthread\_create의 지정된 시작 지점부터 실행하고, 부모는 pthread\_create 이후로 이어서 진행한다. return 값은 없다. p1에서는 pthread\_create( , , foo, )이므로 다시 실행을 재개할 때, child는 foo부터 실행된다.

8) Run following code and explain the difference.









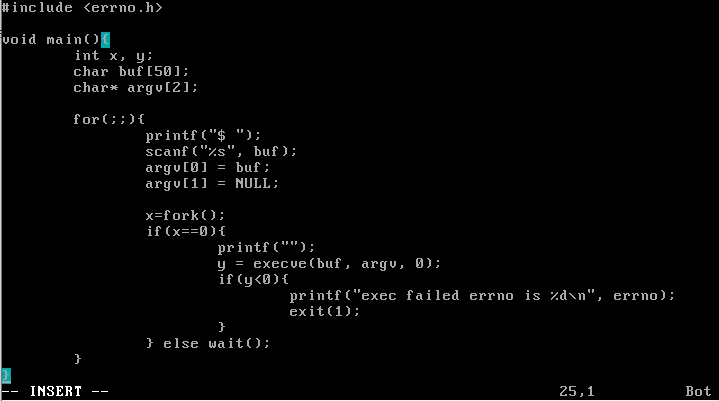
p1.c는 x를 부모와 자식으로 복제한다. 자식일 때는 y에 2를 더하고 y를 출력문과 함께 출력한 뒤 함수를 끝낸다. 부모일 때도 y에 2를 더하고 y를 출력문과 함께 출력한 뒤 함수를 끝낸다. 둘의 차이는 자식일 때는 thread child, 부모일 때는 thread parent를 출력한다는 것이다. 스케쥴러에 의해 child가 먼저 실행되고 parent가 이후에 실행된다.

반면 p2.c는 folk를 활용한 복제가 아닌 pthread\_create를 활용한 복제이다. 즉, body까지 복사하여 생성하지 않고, 하나의 body를 부모-자식 간 같이 사용한다.

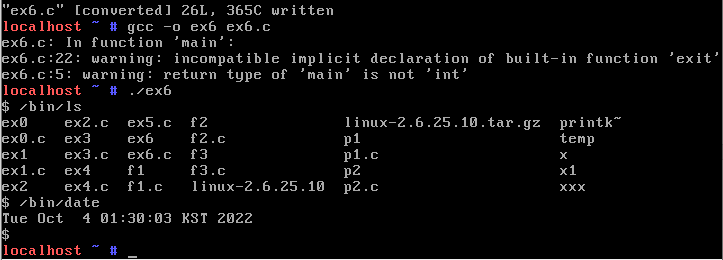
11. Homework

1) Try the shell code in section 7. Try Linux command such as "/bin/ls", "/bin/date", etc.



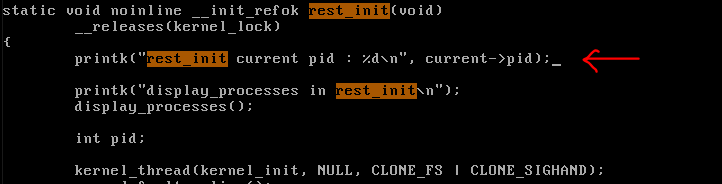


강의자료 7번에 Shell 코드를 ex6.c에 작성하였다. Fork()를 활용하여 실행창에 Shell처럼 구현한 모습이다.

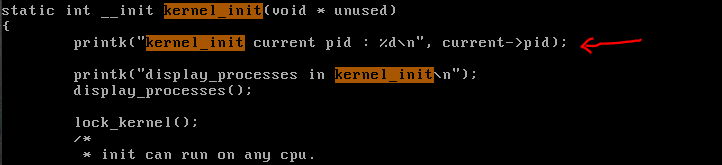


ex6.c 컴파일 후 실행하였다. $ 이후 입력을 기다렸고, 명령어를 입력해본 결과 기존과 같이 잘 실행됨을 확인하였다.

2) Print the pid of the current process (current->pid) inside rest\_init() and kernel\_init(). The pid printed inside rest\_init() will be 0, but the pid inside kernel\_init() is 1. 0 is the pid of the kernel itself. Why do we have pid=1 inside kernel\_init()? Find a location where current->pid will print 2.



init 디렉토리로 이동하여 main.c 내부의 rest\_init를 찾는다. 함수 내부에서 해당 문장을 추가하여 현재 pid를 파악한다.



kernel\_init에서도 추가한다.



Recomple & reboot





rest\_init은 0, kernel\_init은 1임을 확인하였다. Rest\_init()에서와 kernel\_init() 에서 다른 프로세스가 실행됨을 알 수 있다.

Cpu idle에서 스케줄러를 실행하는데, 실행되기 전에는 부모(0)가 실행되다가 스케줄러가 실행되고 나서는 자식이 실행되기 때문에 1로 바뀝니다.

3) The last function call in start\_kernel() is rest\_init(). If you insert printk() after rest\_init(), it is not displayed during the system booting. Explain the reason.

void start\_kernel(){

............

printk("before rest\_init\n"); // this will be printed out

rest\_init();

printk("after rest\_init\n"); // but this will not.

}

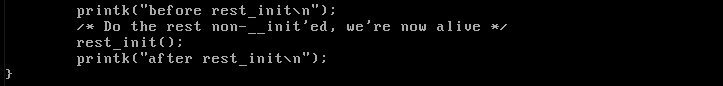


해당 위치로 이동하여

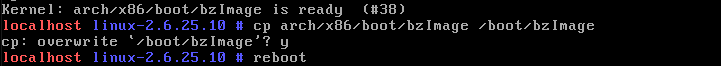


Start\_kernel 함수를 찾는다.

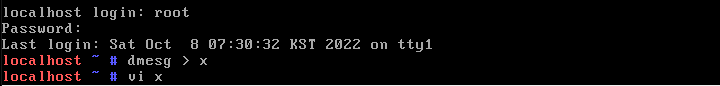


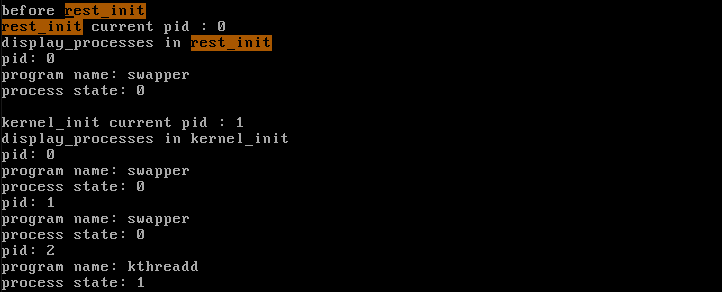


함수의 마지막에 rest\_init()이 호출됨을 확인하였고, 전후로 출력을 해주었다.



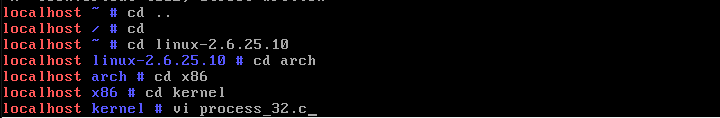
Recompile & reboot



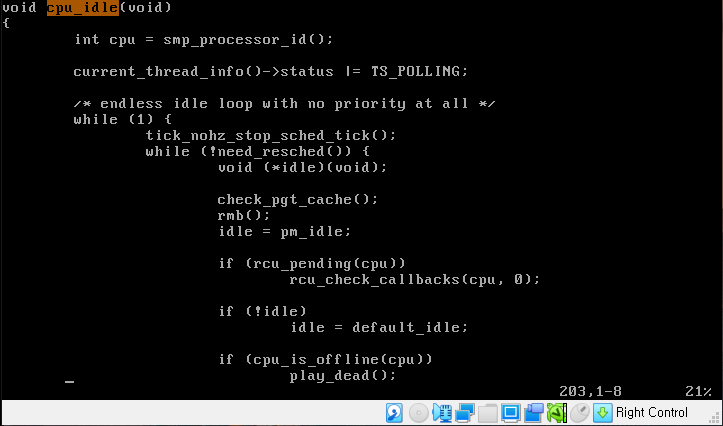


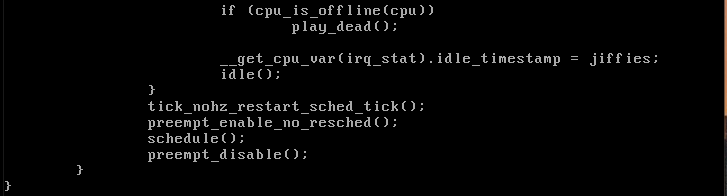
부팅 메시지를 확인해본 결과 before rest\_init은 출력되었으나 after rest\_init은 출력되지 않았다.

rest\_init()을 확인해본 결과 함수 마지막에 cpu\_idle()을 호출한다.



해당 위치로 이동하여





cpu\_idle()를 확인해보면 while(1) 을 통해 무한루프가 실행되고 있음을 알 수 있고, 이 때문에 after rest\_init이 출력되지 않은 것이다.

4) The CPU is either in some application program or in Linux kernel. You always should be able to say where is the CPU currently. Suppose we have a following program (ex1.c).

void main(){

printf("korea\n");

}

When the shell runs this, CPU could be in shell program or in ex1 or in kernel. Explain where is CPU for each major step of this program. You should indicate the CPU location whenever the cpu changes its location among these three programs. Start the tracing from the moment when the shell prints a prompt until it prints next prompt.

shell: printf("$"); // CPU is in shell

=> write(1, "$", 1) // CPU is in c library

=> INT 128 // CPU is in c library

kernel:

sys\_write() // CPU is in kernel and display '$' in screen

// after sys\_write() kernel schedules shell again

// and CPU goes back to shell

shell: scanf("%s", buf); // CPU is in shell

..............

.............

shell: printf("$"); // CPU is back in shell

Ex1.c 파일을 위와같이 작성하고 컴파일 후 실행하였다.



Shell 에서는 명령어를 기다리고, ./ex1를 scanf를 통해 입력한 뒤 실행 명령어를 작동시키면 korea가 출력됨을 알 수 있다. 이후 다시 shell은 명령어를 기다린다.

이 과정을 shell, kernel, user의 관점에서 보면 다음과 같다.

**shell**:

printf("$"); // CPU is in shell

=> write(1, "$", 1) // CPU is in c library

=> INT 128 // CPU is in c library

**kernel**:

sys\_write() // CPU is in kernel and display '$' in screen

// after sys\_write() kernel schedules shell again

// and CPU goes back to shell

**shell**:

scanf("%s", buf); // CPU is in shell

=> read(0, buf, n) // CPU is in c library

=> INT 128 // CPU is in c liabrary

**kernel**:

sys\_read() // CPU is in kernel and make the shell sleep

=> cpu\_idle() // the scheduler picks kernel and runs cpu\_idle()

// so CPU jumps to cpu\_idle

// and waits until the user types something

**user**:

press ‘.’ => INT 33 // CPU is in c library

**kernel**:

atkbd\_interrupt() // CPU is in kernel

// store ‘.’ and show “.” in screen

=> cpu\_idle() //and CPU goes back to cpu\_idle()

**user**:

release ‘.’ => INT 33 // CPU in c library

**kernel**:

atkbd\_interrupt() // CPU is in kernel

// ignore key release

=> cpu\_idle() //and CPU goes back to cpu\_idle()

**user**:

press ‘/’ => INT 33 // CPU is in c library

**kernel**:

atkbd\_interrupt() // CPU is in kernel

// store ‘/’ and show “/” in screen

=> cpu\_idle() //and CPU goes back to cpu\_idle()

**user**:

release ‘/’ => INT 33 // CPU in c library

**kernel**:

atkbd\_interrupt() // CPU is in kernel

// ignore key release

=> cpu\_idle() //and CPU goes back to cpu\_idle()

**...**

**repeat the above steps for pressing ‘e’, ‘x’, ‘1’**

**...**

**user**:

press ‘<Enter>’ => INT 33 // CPU is in c library

**kernel**:

atkbd\_interrupt() // CPU is in kernel

// store ‘./ex1’ in shell’s buf and wake up shell

// the scheduler picks shell and shell resumes execution

**shell**:

fork() // CPU is in shell and shell runs fork()

=> INT 128 // CPU is in c library

**kernel**:

sys\_fork() // CPU is in kernel

=> do\_fork() // make a child

// after sys\_fork() kernel schedules shell

// and CPU goes to shell

**shell**:

wait() // if the scheduler picks parent fisrt, parent shell runs wait()

=> INT 128 // CPU is in c library

**kernel**:

sys\_wait4() // CPU is in kernel

=> do\_wait() // make parent sleep

// after sys\_wait() kernel schedules child shell

// and CPU goes to child shell

**shell**:

execve() // CPU is in shell and child shell runs execve(“./ex1”, ...)

=> INT 128 // CPU is in c library

**kernel**:

sys\_execve() // CPU is in kernel

=> do\_execve() // change to “./ex1” program

// after sys\_execve() kernel schedules ex1

// and CPU goes to ex1

**ex1**:

printf(“korea\n”); // ex1 runs and CPU is in ex1

=> write(1, “korea\n”, 6) // CPU is in c library

=> INT 128 // CPU is in c libarary

**kernel**:

sys\_write() // CPU is in kernel and display “korea\n” in screen

// after sys\_write() kernel schedules ex1 again

// and CPU goes back to ex1

**ex1**:

exit() // CPU is in ex1 and call exit()

// exit() is make it a zombie and sends a signal to parent

// this signal wakes up parent and the scheduler picks parent

**shell**:

//parent goes back to the beginning of for(;;) loop

printf("$"); // CPU is in shell and runs printf(“$”)

=> write(1, "$", 1) // CPU is in c library

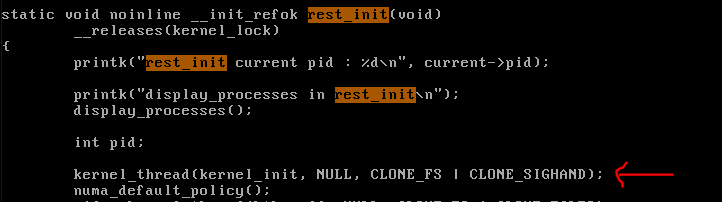
=> INT 128 // CPU is in c library

5) What happens if the kernel calls "kernel\_init" directly instead of calling kernel\_thread(kernel\_init, ...) in rest\_init()? Call kernel\_init with NULL argument and explain why the kenel falls into panic.

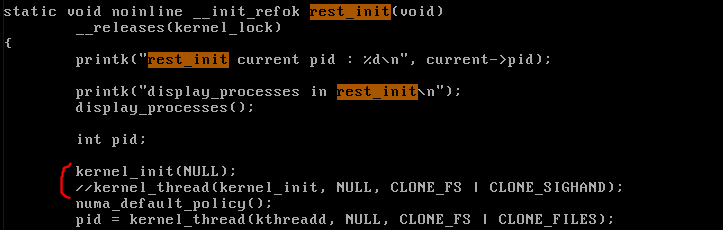
...............

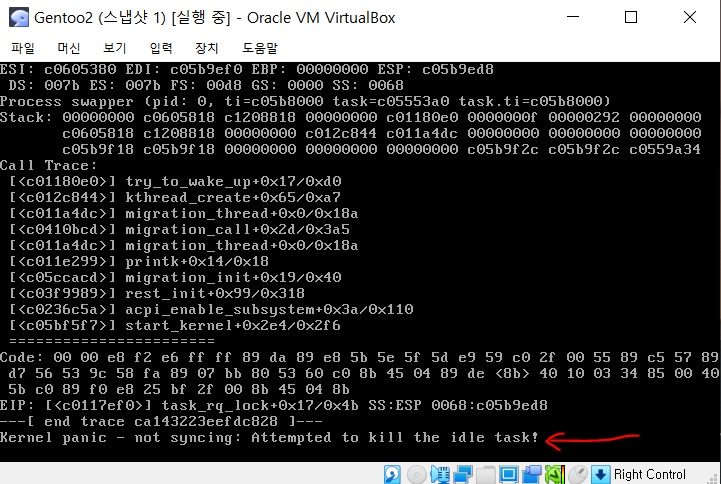
kernel\_init(NULL);

.............



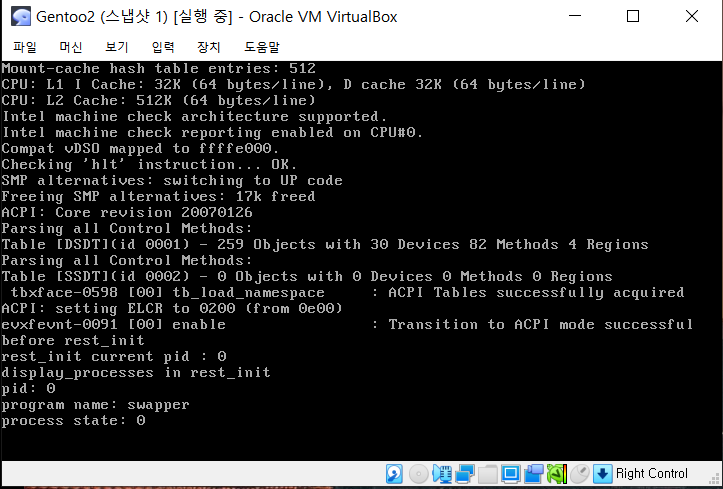
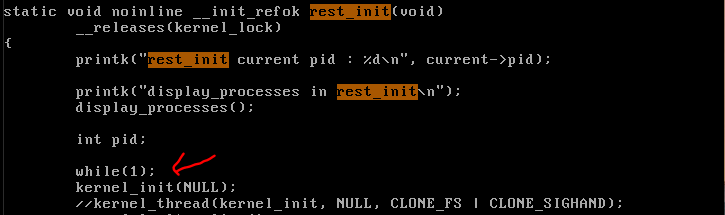
main.c로 가서 rest\_init을 찾는다. 해당 함수에서 kernel\_thread 함수를 호출하는데, 이 대신 kernel init(NULL)을 실행한다.

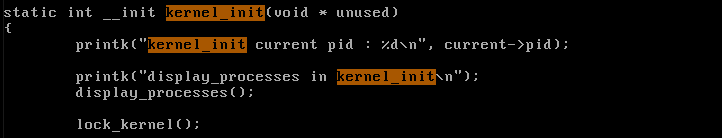


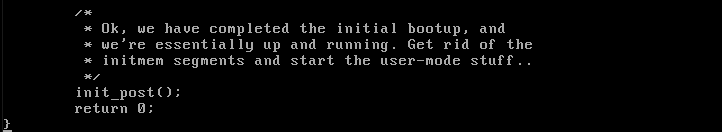


Recompile 후 reboot를 하는 과정에서 My Linux에 접속하니 위와 같이 Kernel panic이 출력되고 멈추었다.

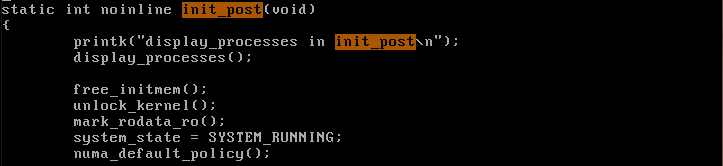
Kernel panic이 발생하는 위치를 찾기 위해 while(1)을 kernel\_init 전과 후에 모두 넣어보았다.

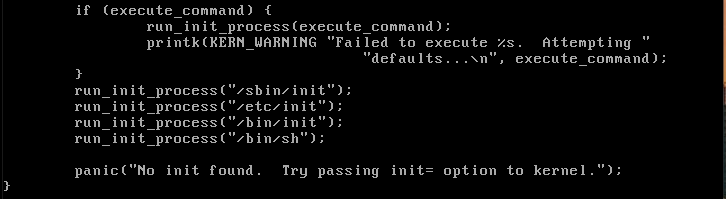
kernel\_init() 뒤에 넣을 때는 kernel panic이 발생하였으나, kernel\_panic 시작하기 전에 while(1)을 넣으니 kernel panic이 발생하지 않은 것을 확인할 수 있다. 이는 kernel\_init()에서 kernel\_panic이 발생하였음을 알 수 있다.



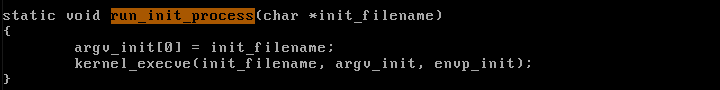


kernel\_init 함수를 찾아본 결과 init\_post()함수를 호출하며 마무리 된다.





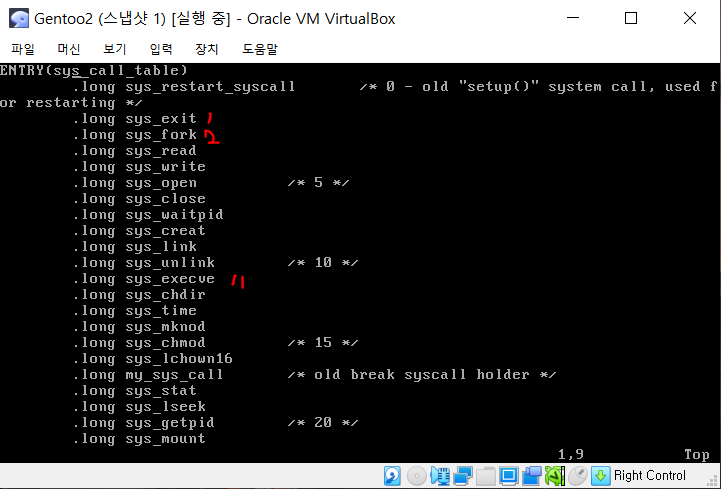
이번엔 init\_post() 함수를 찾아본 결과 run\_init\_process를 호출한다.



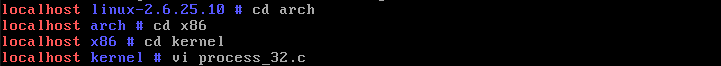
run\_init\_process 함수를 찾아보았다. 앞서 파라미터로 넘겨받은 filename을 kernel\_execve의 파라미터로 사용한다. Execve 함수는 body를 새로운 body로 변경한다. 이 때, kernel\_thread 과정을 거치지 않고 kernel\_execve를 거쳤기 때문에 panic에 걸린 것임을 알 수 있다.

6) Trace fork, exec, exit, wait system call to find the corresponding code for the major steps of each system call.

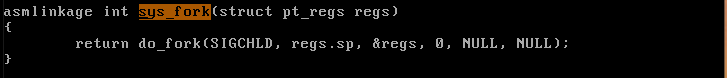
system\_call table은 arch/x86/kernel/syscall에서 1, 2, 11번에서 볼 수 있다.



<fork>



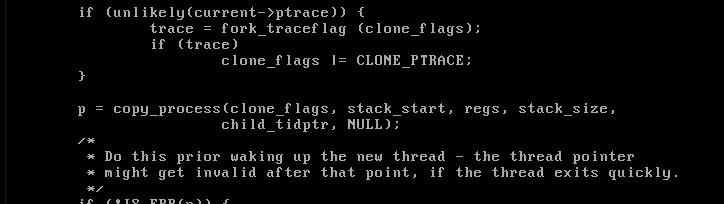
해당 위치로 가서 sys\_fork를 찾는다.



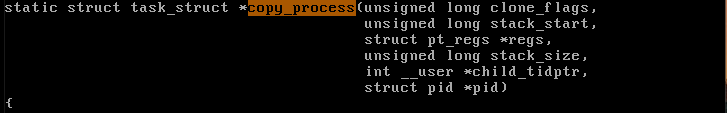
do\_fork를 실행함을 알 수 있다.



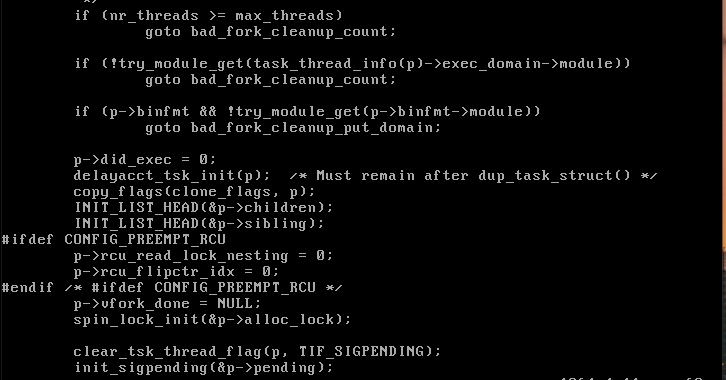
해당 위치로 가서 do\_fork를 찾는다.

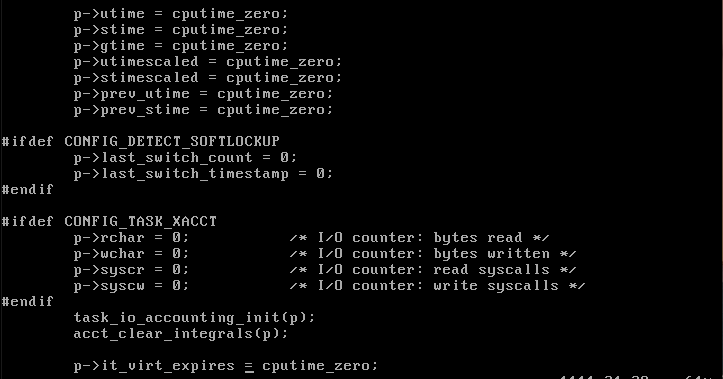


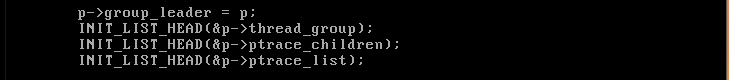
do\_fork함수 내부에 copy\_process를 호출한다. 이는 parent process의 body와 thread\_union을 복제한다.



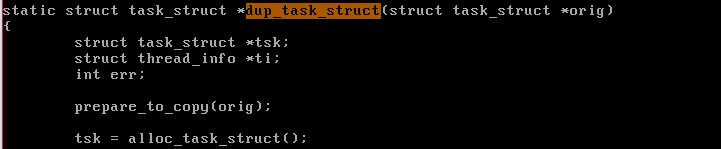








copy\_process 내에서 dup\_task\_struct를 호출한다. 이는 thread\_union을 copy하고 위와 같이 p-> 를 통해 복제한 프로세스 정보를 조정한다. 또한 process queue에 복제한 자식 프로세스를 넣어준다.

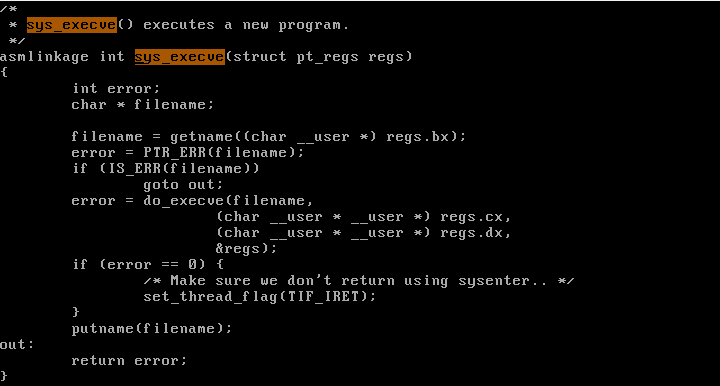


dup\_task\_struct()는 위와 같았는데, prepare\_to\_copy()를 통해 원래의 프로세스를 copy할 준비를 하는 것으로 보인다.

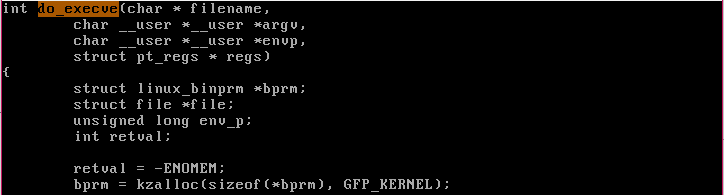
즉 fork는 부모 프로세스의 body와 thread\_union을 copy하고 이를 조정하며, process queue에 자식 프로세스를 넣어주는 동작을 한다. 그리고 자식 프로세스에게는 0을, 부모 프로세스에게는 자식 프로세스의 pid를 return한다.

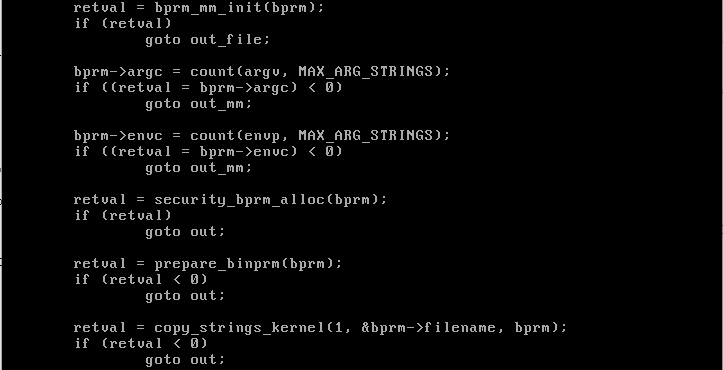
<exec>





sys\_execve()는 “arch/x86/kernel/process\_32.c”에서 찾을 수 있다. do\_execve()를 호출하는 것을 알 수 있다.



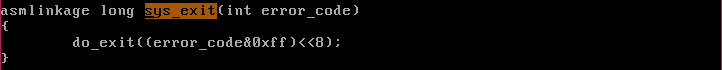


do\_execve()의 내용을 살펴보면, bprm\_mm\_init()을 통해 원래 body를 지우는 것으로 보인다. 또한 security\_bprm\_alloc()을 통해 새로운 body를 할당해주는 것 같다.

즉 exec는 원래의 body를 지우고 새로운 body를 load하며, task\_struct를 업데이트하는 동작을 한다. 또한 새로운 body의 시작 위치가 KMS.eip에 저장되도록 KMS를 업데이트한다.

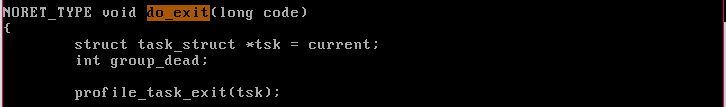
<exit>

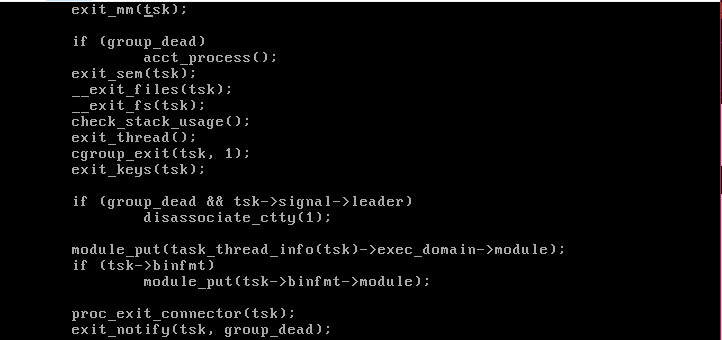


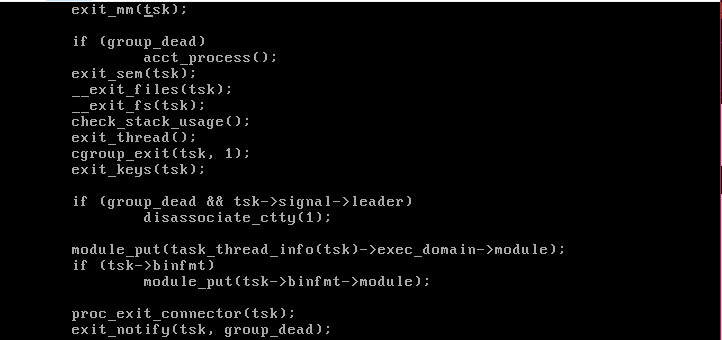


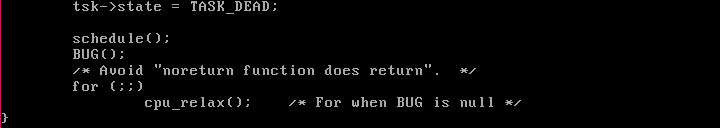
sys\_exit()는 “kernel/exit.c”에서 찾을 수 있었는데, do\_exit()을 call하는 부분을 볼 수 있었다.









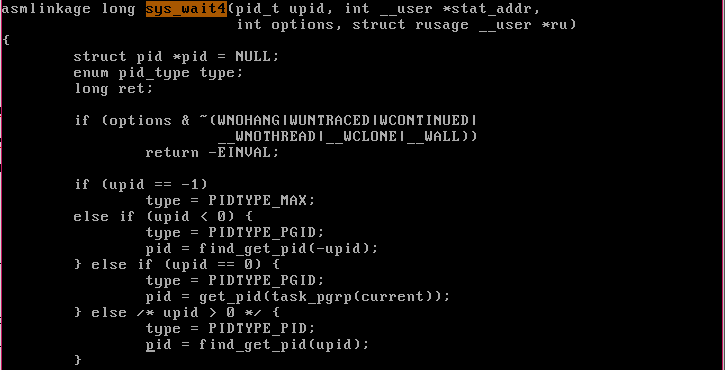


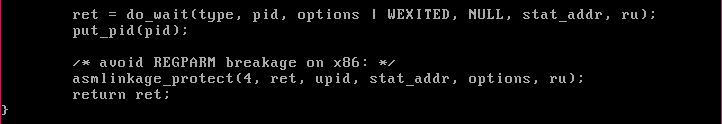
따라서 “kernel/exit.c”에 있는 do\_exit()의 내용을 살펴보았다. exit\_mm()을 통해 body를 지우고, exit\_notify()를 통해 parent에게 신호를 보내는 것임을 볼 수 있다. schdule()을 call함으로써 다음 프로세스를 schedule할 것이다..



<wait>

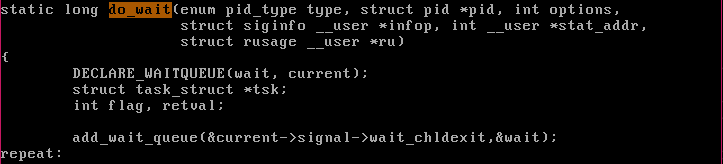


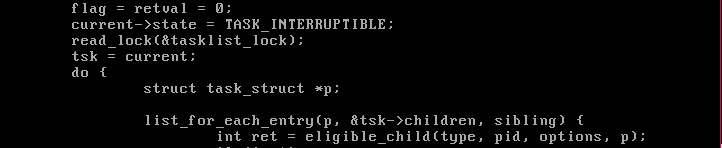


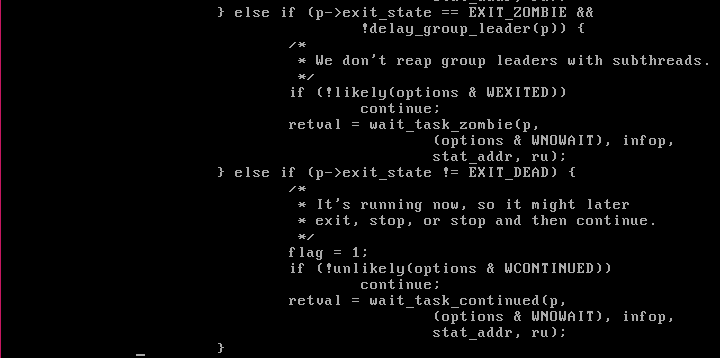


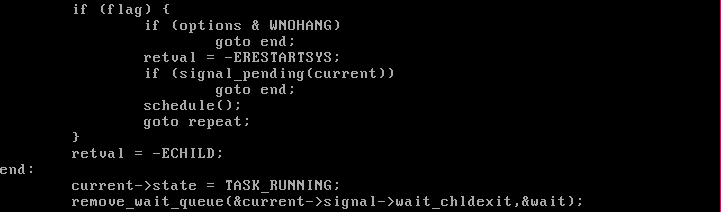
sys\_write4()는 “kernel/exit.c”에서 찾을 수 있다. do\_wait()를 호출하는 것을 알 수 있다.



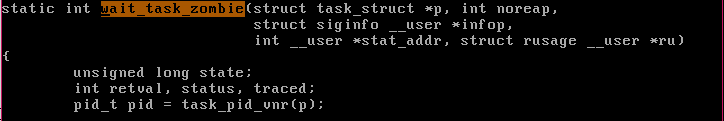


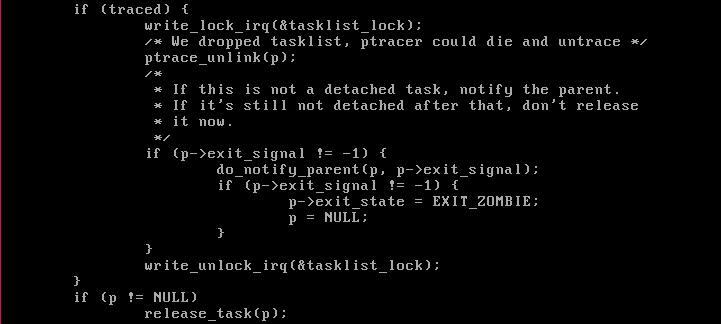




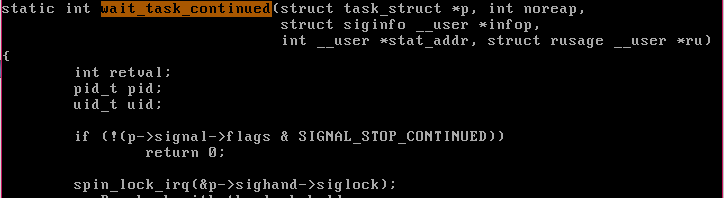


“kernel/exit.c”에서 do\_wait() 함수 정의하는 부분을 찾아보았다.시작 부분에서 “DECLARE\_ WAITQUEUE(wait, current)”를 하여 wait queue를 만들고, “current->state=TASK\_INTERRUPTIBLE”를 통해 parent를 blocking해준다. 그리고 child가 zombie인 경우 wait\_task\_zombie()가 call되고, 아직 살아있는 경우에는 wait\_task\_continued()가 call되는 것 같다. 또한 schedule()을 call함으로써 다음 프로세스를 schedule 해줄 것으로 보이며, child가 exit하여 parent의 wait가 끝나면 “current->state=TASK\_RUNNING”을 통해 parent를 다시 runnable 상태로 만들어 주는 것 같다.



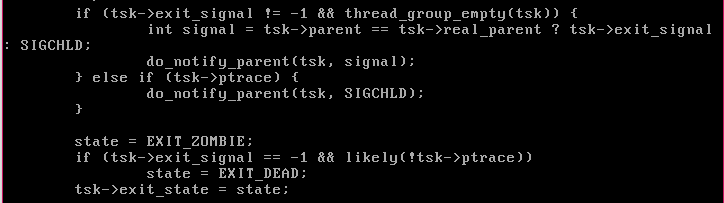


wait\_task\_zobie()에서 release\_task()를 통해 process descriptor를 지워줌을 알 수 있다.



wait\_task\_continued()에서 spin\_lock\_irq()를 통해 parent에 대해 발생하는 모든 인터럽트를 disable시킨다.

즉 wait는 자식 프로세스가 exit하여 zombie상태이면 그것의 process descriptor를 지워 없애주고, 만약 아직 자식 프로세스가 죽지 않았다면 부모 프로세스를 block한 뒤 run queue에서 부모 프로세스를 지우는 동작을 한다. 또한 그 후에 다음 프로세스를 schedule해준다.



exit\_notify()의 내용을 살펴본 결과, do\_notify\_parent()를 call하여 parent에게 SIGCHLD를 보내는 것 같다. 또한 “tsk->exit\_state = state”라는 부분을 볼 수 있는데, 위에서 “state = EXIT\_ZOMBIE”를 하였기 때문에 이를 통해 zombie 프로세스로 만드는 것 같다.

즉 exit는 body를 지우고 zombie로 만들며, 부모 프로세스에게 SIGCHLD signal을 보내는 동작을 한다. 또한 init process에 자식 프로세스를 adopt시키며, 다음 프로세스를 schedule한다.