10-1) Run the program below. What happens? Explain the result.

ex1.c:

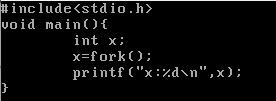
void main(){

int x;

x=fork();

printf("x:%d\n", x);

}





ex1이 실행되면 fork를 부르고, interrupt를 발생시키고, system\_call을 부르고 sys\_fork를 부른 뒤 do\_fork를 부르게 된다. 이 때 do\_fork를 부르면 body와 processor descriptor가 복사된 뒤 child의 pid 정보가 조정된다. 그러고 나서 부모에게는 child의 pid를 리턴하고 자식에게는 0을 리턴해준다. interrupt가 일어난 뒤로는 scheduler가 항상 어떤 프로세스를 실행할지 schedule 하게 되는데, 내 경우 출력결과를 보면 자식이 먼저 실행되도록 schedule 됐단걸 알 수 있다.

10-2) Try below and explain the result.

ex1.c:

void main(){

fork();

fork();

for(;;);

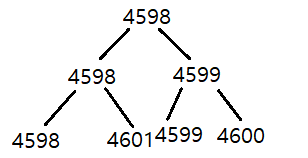
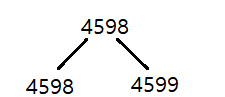
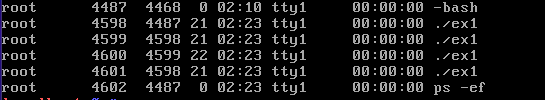
}

# gcc –o ex1 ex1.c

# ./ex1 &

# ps –ef





fork 함수의 특성상 fork가 불러지면 프로세스는 자기 자신을 복사한 뒤에 child를 만든다. 처음 fork에서는 4598이 자기 자신을 복사하고 child인 4599를 만들고, 그 다음 fork에서는 복사된 4598과 4599가 각각 또 자기 자신을 복사하고 child를 만들게 된다.

10-3) Run following code. What happens? Explain the result.

ex1.c:

void main(){

int i; float y=3.14;

fork();

fork();

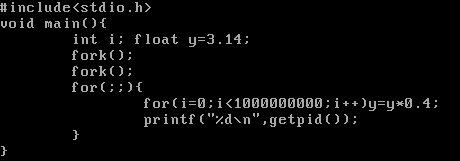
for(;;){

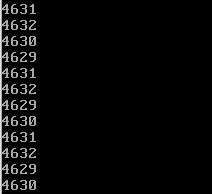
for(i=0;i<1000000000;i++) y=y\*0.4;

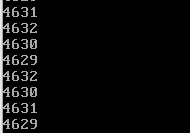
printf("%d\n", getpid());

}

}







실행 시킨 결과, 4631, 4632, 4630, 4629가 계속해서 번갈아가면서 나왔다. 순서가 번갈아가면서 나오는 이유는 interrupt가 발생한 이후 모든게 원래대로 되돌아가기 전에 scheduler가 어느 프로세스를 실행시킬지 schedule 하는데, 이 때 scheduler가 어떤 프로세스를 schedule하는지에 따라 네개의 프로세스의 실행 순서가 바뀌기 때문이다.

10-4) Try below and explain the result.

ex1.c:

void main(){

char \*argv[10];

argv[0]=”./ex2”;

argv[1]=0;

execve(argv[0], argv, 0);

}

ex2.c

void main(){

printf("korea\n");

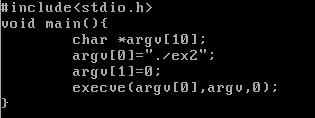
}

#gcc –o ex1 ex1.c

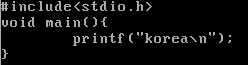
#gcc –o ex2 ex2.c

#./ex1

ex1.c



ex2.c





execve 함수는 이전 process의 body는 없애고 execve 함수의 첫번째 인자를 새로운 body로 만들어주는 함수이고, pid는 이전 body의 pid와 똑같이 쓴다. 따라서 ex1을 실행시켜주면 ex2가 실행되고, korea가 찍히게 된다.

10-5) Run following code and explain the result.

void main(){

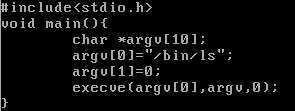
char \*argv[10];

argv[0]=”/bin/ls”;

argv[1]=0;

execve(argv[0], argv, 0);

}





실행되는건 ex1이지만 프로세스의 body는 /bin/ls가 된다. 따라서 ex1을 실행시켰을 때 실행되는건 /bin/ls이고, 그 결과 ex1을 실행시킨 결과는 /bin/ls를 실행시킨 결과와 똑같이 나왔다.

10-6) Run following code and explain the result.

void main(){

char \*argv[10];

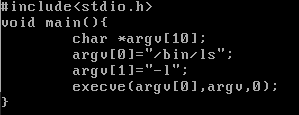
argv[0]=”/bin/ls”;

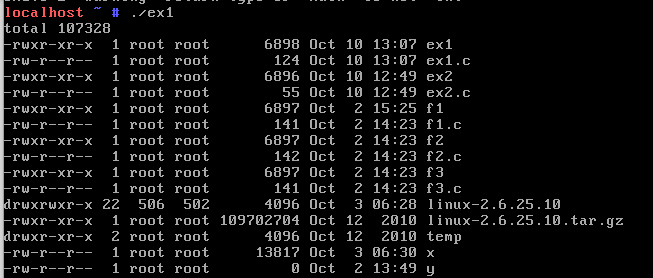
argv[1]="-l";

argv[2]=0;

execve(argv[0], argv, 0);

}





여기에서는 execve의 인자가 /bin/ls 말고도 -l이 있는데, 이는 /bin/ls가 -l을 인자로 가진다는 뜻이다. ex1을 실행시켜보면 결과는 /bin/ls -l을 실행시켰을 때의 결과와 같게 나온다.

10-7) Run following code and explain the result.

p1.c:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void \* foo(void \* aa){

printf("hello from child\n");

return NULL;

}

void main(){

pthread\_t x;

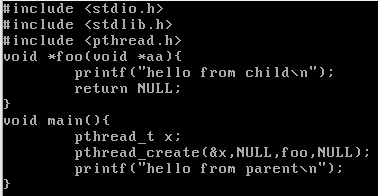
pthread\_create(&x, NULL, foo, NULL); // make a child which starts at foo

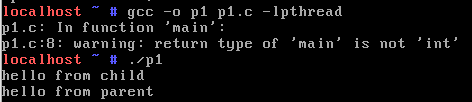
printf("hello from parent\n");

}

$ gcc –o p1 p1.c –lpthread

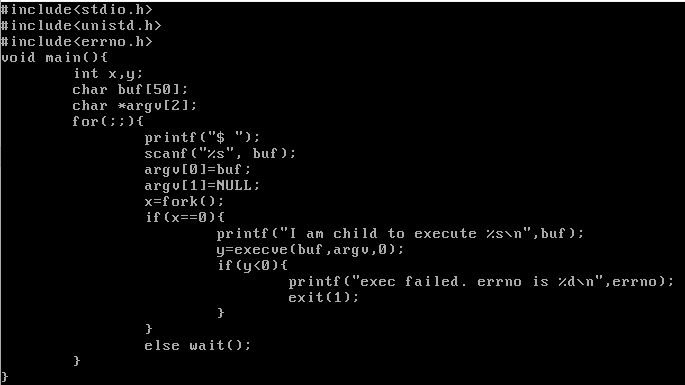
$./p1

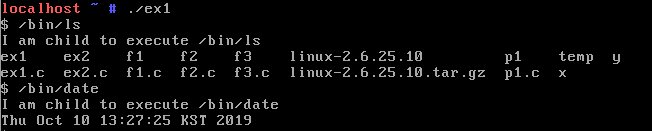




pthread를 쓰게 되면 body를 복제하는 과정이 생략되고, 부모와 자식간에 body를 공유하게 된다. 부모는 그대로 놔두면 되지만 자식은 어디서부터 시작할지를 직접 정해줘야하고, 위 코드에서는 그 시작 위치를 foo로 정해줬다. 이렇게 하면 부모는 pthread\_create 함수가 호출된 다음부터 실행되고, 자식은 내가 지정해준 시작위치인 foo부터 실행하게 된다. 그 이후 스케쥴러가 어떻게 스케쥴 하는지에 따라 부모와 자식의 실행순서가 달라지는데, 나의 경우는 자식이 먼저 스케쥴돼서 호출됐다.

11-1) Try the shell code in section 7. Try Linux command such as "/bin/ls", "/bin/date", etc.

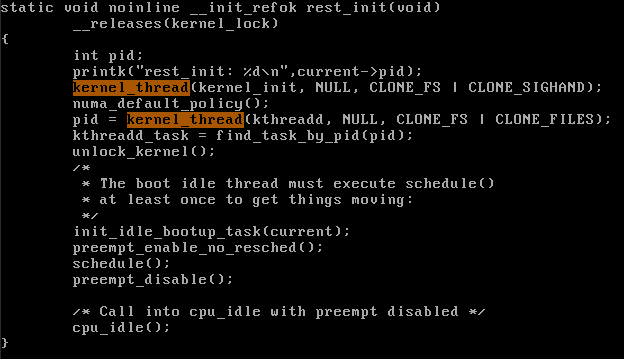


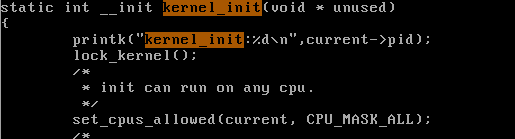


shell은 fork와 exec의 사용 예제중 하나이다. shell에서 명령 실행은 자식이 해주고 parent는 자식이 일을 다 끝낼 때까지 기다려준다. ex1을 실행시키고 명령을 입력 했을 때 결과가 정상적으로 출력됐다.

11-2) 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()?

init/main.c









원래 linux에 있던 프로세스의 pid는 0인데, rest\_init 함수를 쭉 따라가다보면 kernel\_thread 함수를 호출하게 된다. 커널에서의 kernel\_thread는 pthread와 같은 역할을 하기 때문에 process descriptor를 복제하게 되고, 이 때 만들어지는게 pid가 1인 kernel\_init이다. 원래 pthread는 호출하면 interrupt에 걸려서 커널 안으로 들어가서 처리를 한다면, kernel\_thread 함수는 이미 커널 안에 있기 때문에 interrupt에 걸리지 않는다. 하지만 rest\_init 안에서 kernel\_thread 함수가 불러진 후, 부모는 함수가 불러진 뒤부터 계속 실행하게 되고, rest\_init 함수의 마지막을 보면 cpu\_idle 함수가 호출된다. cpu\_idle 함수를 보면 while 루프 안에서 schedule 함수가 호출되고 바로 이 때 자식 프로세스인 kernel\_init이 스케쥴 된다. 그렇기 때문에 rest\_init에서는 pid가 0이지만, kernel\_init에 들어왔을 때 pid는 1로 바뀌게 된다. (priority가 init\_task가 가장 낮음)