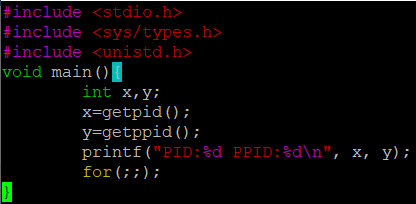
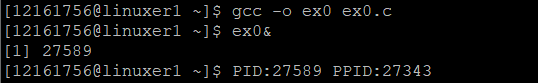
**System Programming Lect9 HW**

**001분반 / 12161756 / 윤성호**

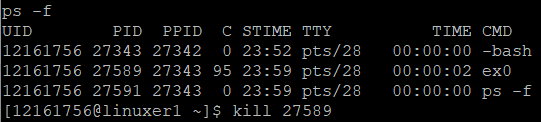
**3. Homework**

**0) Try ex0 below. Who is the parent of ex0?**

: x에는 ex0의 PID, y에는 PPID(Parent PID)가 저장되고 이를 출력한다. 프로그램이 종료되지 않도록 코드 끝에 무한루프를 만들어 둔다.



1. ex0을 컴파일한 후 &기호를 이용해 백그라운드에서 실행하자 ex0의 PID값 27589이 출력된다.이후 프로그램에서 PID와 PPID를 출력한다.

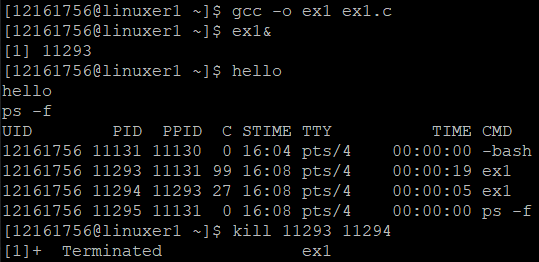


2. [ps -f]로 현재 프로세스 상태를 출력한다. ex0의 PPID값을 통해 ex0 부모의 PID값이 27343임을 알 수 있다. PID가 27343인 process는 -bash이다. 따라서 ex0의 parent는 **-bash**이다. 이후 [kill + ex0의 PID]를 입력해 ex0을 종료한다.

**1) Try ex1 below. Why do we have two hello's? What are the PID of ex1 and ex1's child? Who is the parent of ex1?**



: [x=fork()]로 ex1 프로그램을 복제한다. scheduler는 이 두 프로세스를 fork 이후 printf부터 실행하게 한다. 두 프로세스는 각각 ‘hello’를 출력하고 무한루프로 인해 종료되지 않는다.



**< 1. Why do we have two hello's? >**

: hello가 두 번 출력된 이유는 printf 코드 바로 위에서 **fork 함수로 인해 ex1 프로세스가 복제되었기 때문이다.**

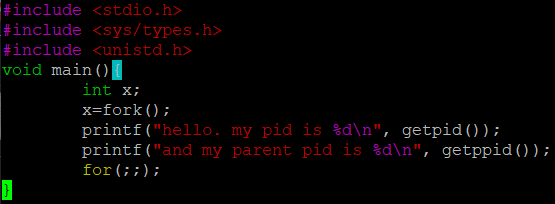
**< 2. What are the PID of ex1 and ex1's child? >**

: 컴파일 후 ex1을 백그라운드에서 실행한다. **ex1의 PID(11293)**와 hello가 두 번 출력된다. [ps -f]의 CMD에 두 개의 ex1이 표기되어 복제된 것을 확인할 수 있다. ‘ex1’s child PPID = ex1 PID(11293)’인 것을 이용하면 ex1의 자식 프로세스는 위 프로세스 목록 중 3번째 것임을 알 수 있다. 따라서 **ex1’s child PID는 11294**이다.

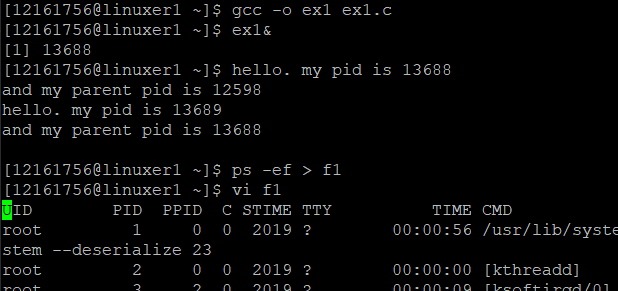
**< 3. Who is the parent of ex1? >**

: ex1의 PPID는 11131이고 이를 PID로 가지는 프로세스는 ‘-bash’이다. 따라서 ex1의 parent는 **‘-bash’**이다.

**2) Modify ex1.c such that it prints its own pid and the parent pid. Confirm the result with "ps -ef". Who is the parent of the parent of ex1? Who is the parent of the parent of the parent of ex1? Follow the parent link until you reach PID 0 and show all of them.**



: fork로 프로세스를 복제한 후 getpid, getppid로 각 프로세스의 PID, PPID를 출력한다. 이후 무한 루프로 인해 프로세스가 종료되지 않는다.



: ex1을 컴파일 후 백그라운드에서 실행하였다. ex1의 PID가 13688인 것을 알 수 있다. 이후 [ps -ef] 결과를 f1에 저장한다. vi에서 ex1의 PID를 검색하여 ex1의 PPID를 찾고 이 PPID를 PID로 가지는 프로세스를 검색해 ex1의 부모를 찾을 수 있다. 이 과정을 반복하면 ex1의 ancestor를 추적할 수 있다.













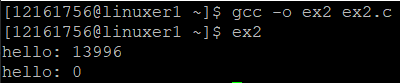
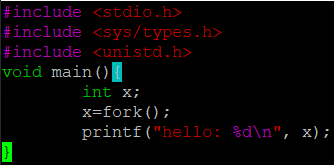


1. the parent of the parent of ex1 = **sshd: 12161756@pts/4**

2. the parent of the parent of the parent of ex1 = **sshd: 12161756 [priv]**

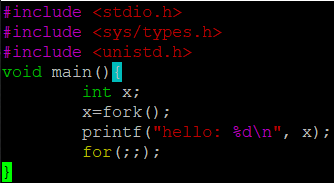
3. PID 0번은 sched라는 swapper에 할당되고 PID 1번은 init 프로세스에 할당된다. [ps -ef]에서 PID가 1번부터 시작하여 swapper은 볼 수 없었다.

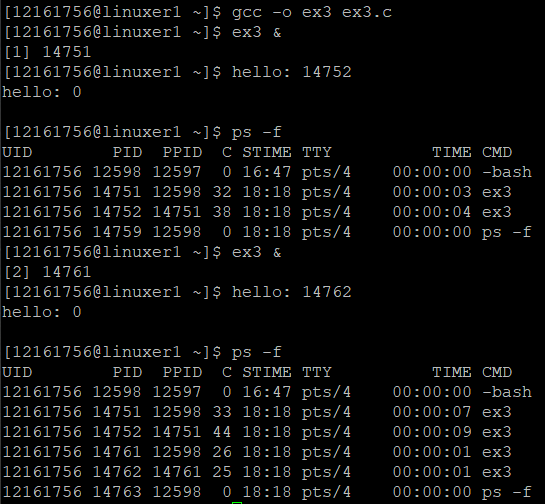
**3) Try below (ex2.c). Which hello is displayed by the parent and which hello is by the child?**



: **첫 번째 hello는 parent, 두 번째는 child의 hello**이다. fork 함수는 프로세스를 복제 후 parent에는 child의 PID를, child에는 0을 return하기 때문이다.

**4) Try below (ex3.c) and show all ancestor processes of ex3 (parent, parent of parent, etc).**

: fork 함수로 ex3 프로그램을 복제한다. scheduler는 이 두 프로세스를 fork 이후부터 실행하게 한다. 두 프로세스는 각각 ‘hello’를 출력하고 무한루프로 인해 종료되지 않는다.



: ex3을 백그라운드로 두 번 실행했다. 첫 번째 ex3의 PID는 14751, 두 번째 ex3의 PID는 14761이다. 이 두 프로세스는 각각 자식 프로세스를 1개씩 가진다.





: 첫 번째 ex3의 PPID는 12598이다.





: 두 번째 ex3의 PPID 역시 12598이다.







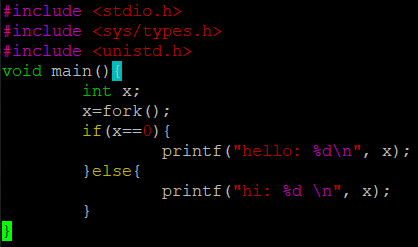




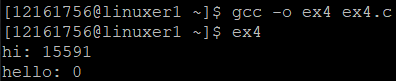


: 첫 번째, 두 번째 ex3의 parent process는 ‘-bash’로 동일하였다. 그러므로 이후 ancestor 또한 둘 다 동일하다.

**5) Try below (ex4.c). Which message was displayed by the parent and which one by the child?**

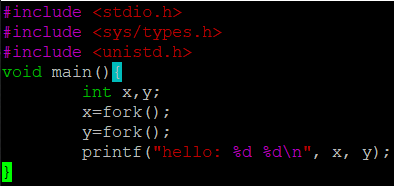


: fork로 프로그램을 복제한다. fork의 return 값인 x가 0이면 hello와 x값을 출력하고 그렇지 않으면 hi와 x값을 출력한다.

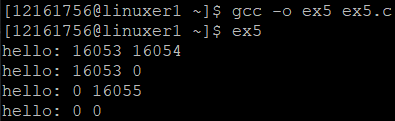


: fork는 parent에는 child의 PID를, child에게는 0을 return한다. 그러므로 **hi는 parent에 의해, hello는 child 프로세스에 의해** 출력된 것이다.

**6) Try below (ex5.c). How many hellos do you see? Explain why you have that many hellos. Draw the process tree.**



: 첫 번째 fork에서 프로그램이 복제돼 2개가 된다. 이 두 프로세스는 바로 다음 fork에서 각각 또 복제되어 ex5 프로그램은 총 4개가 된다. 이후 각 프로세스는 hello 메시지와 함께 fork를 통해 return된 x, y값을 출력한다.



1. 위에서 기술한 것처럼 두 번의 fork 함수를 거쳐 프로세스는 4개가 되어 hello가 **4번** 출력된다.

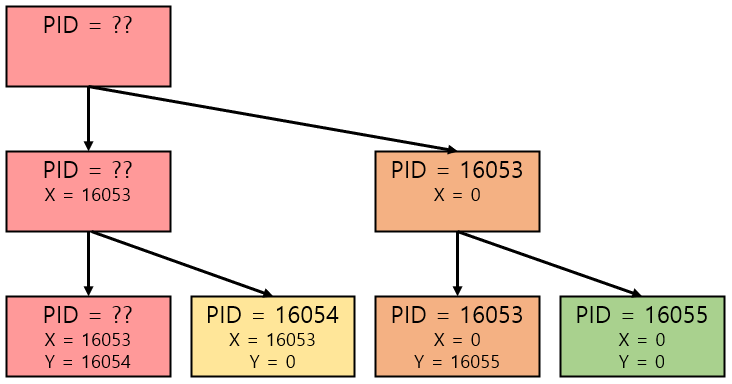
2. 첫 번째 hello는 x와 y 값이 모두 0이 아니다. 이를 통해 이 프로세스는 두 번의 fork에서 모두 parent였다는 것을 알 수 있다.

3. 두 번째는 y값만 0이므로 첫 번째 fork에서 parent인 프로세스의 child임을 알 수 있다.

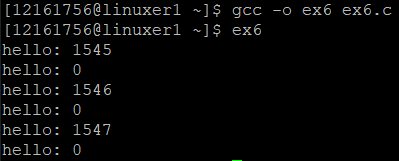
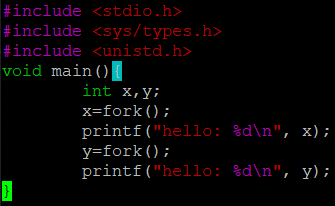
4. 세 번째는 첫 번째 fork에서의 child이다.

5. 마지막은 첫 번째 fork에서의 child인 프로세스의 child이다.

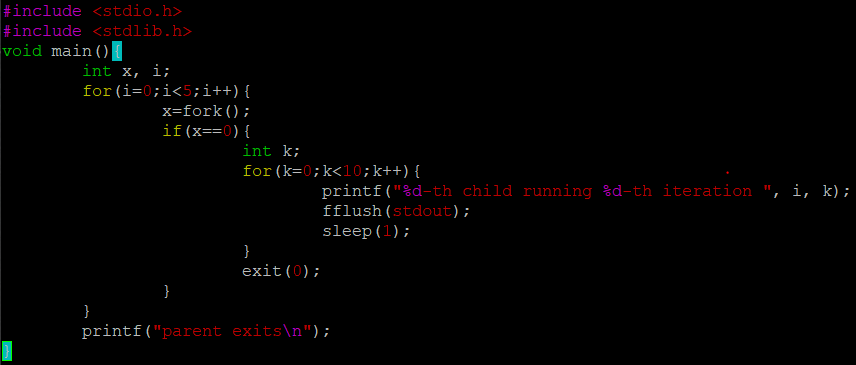
**< ex5’s Process Tree >**



**7) Try below (ex6.c). How many hellos do you see? Explain why you have that many hellos.**

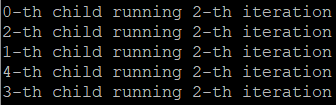
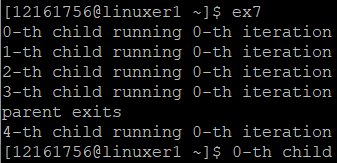


: **6개**의 hello가 출력된다. 첫 번째 fork에서 프로세스가 복제되고 두 번의 hello와 x를 출력한다. 이 두 프로세스는 두 번째 fork에서 4개가 되고 네 번의 hello와 y를 출력한다.

**8) Try below (ex7.c). When you run ex7, how many processes run at the same time? Which process finishes first and which process finishes last? Show the finishing order of the processes. Run ex7 again and compare the finishing order with that of the first run.**

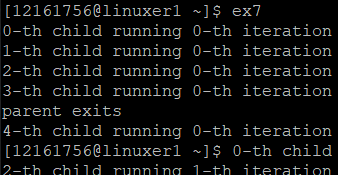
: outer for문에서 fork에 의해 프로세스가 5번 복제된다. 그래서 outer for문이 종료된 시점에는 총 6개의 프로세스가 실행 중이다. 복제된 child들은 loop를 돌지 못하고 if문 내부로 들어가 exit(0)에 의해 종료되기 때문에 child의 child는 생성되지 않는다.

**< 1. how many processes run at the same time? >**



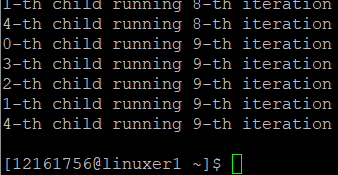
: 최초에 빨간 네모 안의 메시지가 동시에 출력되었다. 이를 통해 parent process가 종료되기 전까지는 6개가 동시에 실행되었다는 것을 알 수 있다. parent process가 종료된 이후로는 오른쪽 캡처처럼 5개씩 출력되었기 때문에 5개가 동시에 실행된 것이다.

**< 2. Which process finishes first? >**



: 6개의 프로세스 중 가장 먼저 종료되는 프로세스는 **최초의 ex6 프로세스**이다. 최초 프로세스는 fork의 리턴 값인 x에 복제된 child의 PID값이 기록되기 때문에 if 조건문을 건너 뛰고 5번의 fork 함수를 실행한 후 “parent exits” 메시지를 출력한 후 종료된다.

**< 3. which process finishes last? >**

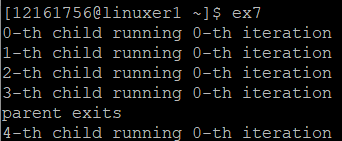
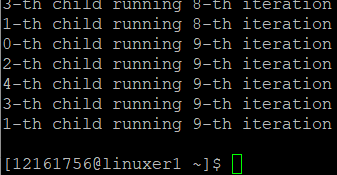


: **4-th child process (마지막에 생성된 child)**가 가장 마지막에 종료되는 것을 알 수 있다.

**< 4. Show the finishing order of the processes. >**

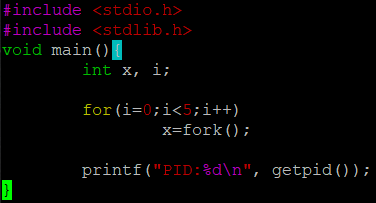
: 앞선 2번과 3번의 캡처를 통해 **parent -> 0\_child -> 3\_child -> 2\_child -> 1\_child -> 4\_child** 순으로 종료되는 것을 알 수 있다.

**< 5. Run ex7 again and compare the finishing order with that of the first run. >**

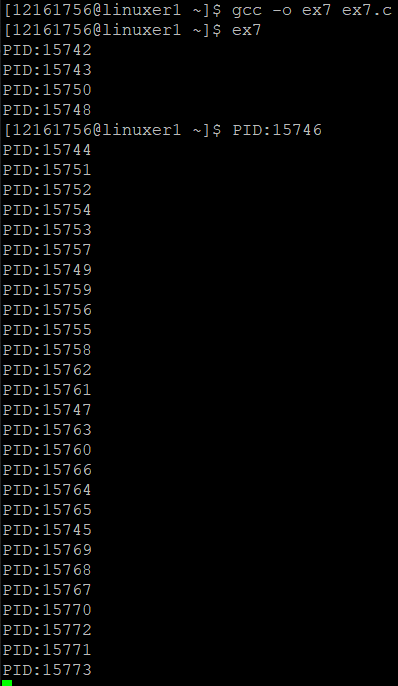
 

: 두 번째 실행에서는 **parent -> 0\_child -> 2\_child -> 4\_child -> 3\_child -> 1\_child** 순으로 종료되었고 첫 번째 실행과 결과가 달랐다. 직관적으로는 0->1->2->3->4 순으로 종료될 것 같지만 실제로는 scheduler가 그때그때 상황에 따라 process 실행 순서를 결정한다는 것을 알 수 있다.

**9) If you delete "exit(0)" in ex7.c, how many processes will be created? Confirm your answer by modifying the code such that each process displays its own pid.**



: 기존 코드에서 PID 출력에 불필요한 exit(0) 등의 코드를 제거하였다. ex7 프로그램은 for문에 의해 fork를 5번 실행한다. 첫 fork에서 프로세스는 2개, 다음 fork에서 4개, 그 다음은 8개, 16개, 32개가 된다. 각 프로세스는 종료하기 전에 자신의 PID값을 출력한다.



: 32개의 PID가 출력되었다. 이 프로그램은 **32개**의 프로세스를 생성한다.