

12201922

이규민

3. Homework

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

ex0.c:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main(){
    int x,y;
    x=getpid();          // my pid
    y=getppid();         // parent's pid
    printf("PID:%d PPID:%d\n", x, y);
    for(;;); // to make this process alive
}
```

```
$ gcc -o ex0 ex0.c
```

Run ex0 with &. "&" puts process in the background so that you can issue next command to the shell.

```
$ ex0&
```

.....

Now confirm with "ps -f".

```
$ ps -f
```

.....

Now kill ex0.

```
$ kill xxxx
```

,where xxxx is the pid of ex0.

```
kyumin@DESKTOP-NUDFAPK ~  
$ ps -f  
    UID      PID    PPID  TTY          STIME COMMAND  
kyumin    1434    1433  pty0    09:06:03 /usr/bin/bash  
kyumin    1457    1434  pty0    09:17:35 /usr/bin/ps  
kyumin    1433         1  ?       09:06:02 /usr/bin/mintty  
  
kyumin@DESKTOP-NUDFAPK ~  
$ ex0 &  
[1] 1458  
  
kyumin@DESKTOP-NUDFAPK ~  
$ PID:1458 PPID:1434  
ps -f  
    UID      PID    PPID  TTY          STIME COMMAND  
kyumin    1459    1434  pty0    09:17:57 /usr/bin/ps  
kyumin    1434    1433  pty0    09:06:03 /usr/bin/bash  
kyumin    1458    1434  pty0    09:17:48 /cygdrive/c/Users/kyumin/AppData/Roam  
ing/SPB_Data/ex0  
kyumin    1433         1  ?       09:06:02 /usr/bin/mintty
```

Ex0 프로그램을 실행하면 pid는 1458, ppid는 1434라고 출력된다. Ps -f 명령어로 확인해보면 동일한 pid임을 확인할 수 있다.

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?

ex1.c:

```
#include <stdio.h>  
#include <sys/types.h>  
#include <unistd.h>  
void main(){  
    int x;  
    x=fork();  
    printf("hello\n");  
    for(;;);  
}
```

```
$ gcc -o ex1 ex1.c
```

```
$ ex1&
```

```
hello
```

```
hello
```

```
$ ps -f
```

.....

\$ kill xxxx(pid of ex1) yyyy(pid of ex1's child)

```
kyumin@DESKTOP-NUDFAPK ~
$ ex1 &
[1] 1797

kyumin@DESKTOP-NUDFAPK ~
$ hello
hello
ps -f
```

UID	PID	PPID	TTY	STIME	COMMAND
kyumin	1784	1783	pty0	09:22:06	/usr/bin/bash
kyumin	1783	1	?	09:22:05	/usr/bin/mintty
kyumin	1798	1797	pty0	09:24:30	/cygdrive/c/Users/kyumin/AppData/Roam
ing/SPB_Data/ex1					
kyumin	1797	1784	pty0	09:24:30	/cygdrive/c/Users/kyumin/AppData/Roam
ing/SPB_Data/ex1					
kyumin	1799	1784	pty0	09:24:33	/usr/bin/ps

```
kyumin@DESKTOP-NUDFAPK ~
$ kill 1797 1798

kyumin@DESKTOP-NUDFAPK ~
$ ps -f
```

UID	PID	PPID	TTY	STIME	COMMAND
kyumin	1784	1783	pty0	09:22:06	/usr/bin/bash
kyumin	1783	1	?	09:22:05	/usr/bin/mintty
kyumin	1800	1784	pty0	09:26:39	/usr/bin/ps

```
[1]+  Terminated                  ex1
```

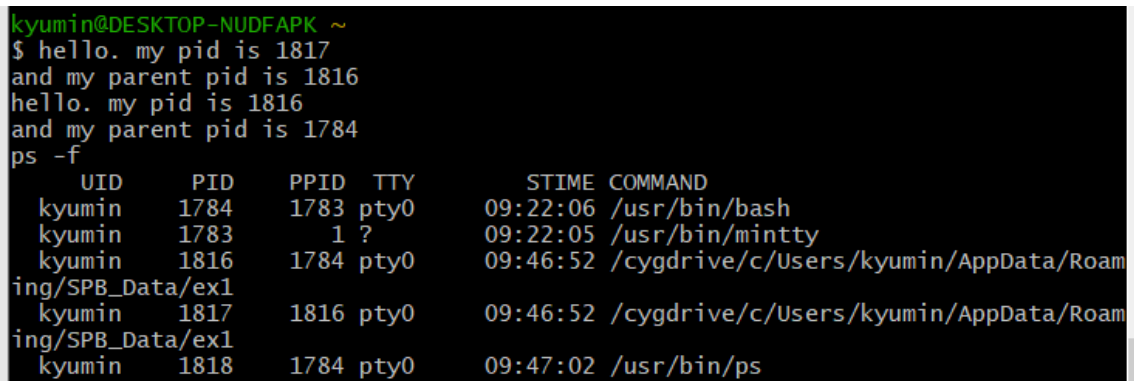
Ex1을 실행하면 hello 메시지가 두 번 출력된다. 그 이유는 fork 코드가 실행되면서 프로세스를 복제하게 된다. 그리고 스케줄러가 프로세스를 모두 실행하면서 부모와 자식 모두 printf("hello\n") 코드가 실행되므로 hello가 두 번 출력된다.

Ps -f를 통해 확인해보면 ex1 프로세스가 두 개 생성된 것을 볼 수 있다. 이 중에 pid가 1798인 프로세스를 보면 ppid가 다른 것과 달리 1797이다.

즉 부모의 pid는 1797이고, 자식의 pid는 1798이다.

2) Modify ex1.c such that it prints its own pid and the parent pid. Confirm the result with "ps -f". Who is the parent of ex1? Who is the parent of the parent of ex1? Follow the parent link until you reach PID 1 and show all of them.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main(){
    int x;
    x=fork();
    printf("hello. my pid is %d\n", getpid());
    printf("and my parent pid is %d\n", getppid());
    for(;;);
}
```



The terminal screenshot shows the execution of the program. The user runs the program, and it prints its own PID (1817) and its parent's PID (1816). Then, the user runs 'ps -f', which displays a list of processes. The output shows the following processes:

UID	PID	PPID	TTY	STIME	COMMAND
kyumin	1784	1783	pty0	09:22:06	/usr/bin/bash
kyumin	1783	1	?	09:22:05	/usr/bin/mintty
kyumin	1816	1784	pty0	09:46:52	/cygdrive/c/Users/kyumin/AppData/Roam
kyumin	1817	1816	pty0	09:46:52	/cygdrive/c/Users/kyumin/AppData/Roam
kyumin	1818	1784	pty0	09:47:02	/usr/bin/ps

출력된 메시지를 보면 1행과 2행에서는 자신의 pid가 1817, 부모의 pid가 1816이라고 출력되었다. Ps -f 명령어를 통해 확인해보면 pid가 1816인 프로세스의 ppid는 1784다.

즉 ex0의 부모 pid는 1816이고, 자식의 pid는 1817이다.

자식의 pid부터 부모의 pid를 계속 따라가보면 1817 -> 1816 -> 1784 -> 1783 -> 1이다.

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

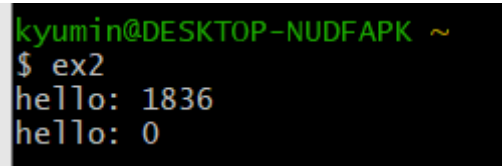
```
void main(){  
    int x;  
    x=fork();  
    printf("hello: %d\n", x);  
}
```

```
$ gcc -o ex2 ex2.c
```

```
$ ex2
```

```
hello: 22644
```

```
hello: 0
```

A terminal window with a black background and green text. The prompt is 'kyumin@DESKTOP-NUDFAPK ~'. The user enters '\$ ex2'. The output is 'hello: 1836' followed by 'hello: 0' on the next line.

```
kyumin@DESKTOP-NUDFAPK ~  
$ ex2  
hello: 1836  
hello: 0
```

Fork 함수를 사용할 경우 자식은 0을 리턴하고, 부모의 경우 자식의 pid를 리턴한다.

그러므로 x를 출력하는 코드가 실행되었을 때 0이 출력된다면 그것은 자식인 경우다.

위 결과를 보면 “hello: 1836”은 부모임을 알 수 있고, “hello: 0”은 자식임을 알 수 있다.

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

```
void main(){
    int x;
    x=fork();
    printf("hello: %d\n", x);
    for(;;) // make the parent and child alive
}
```

```
$ gcc -o ex3 ex3.c
```

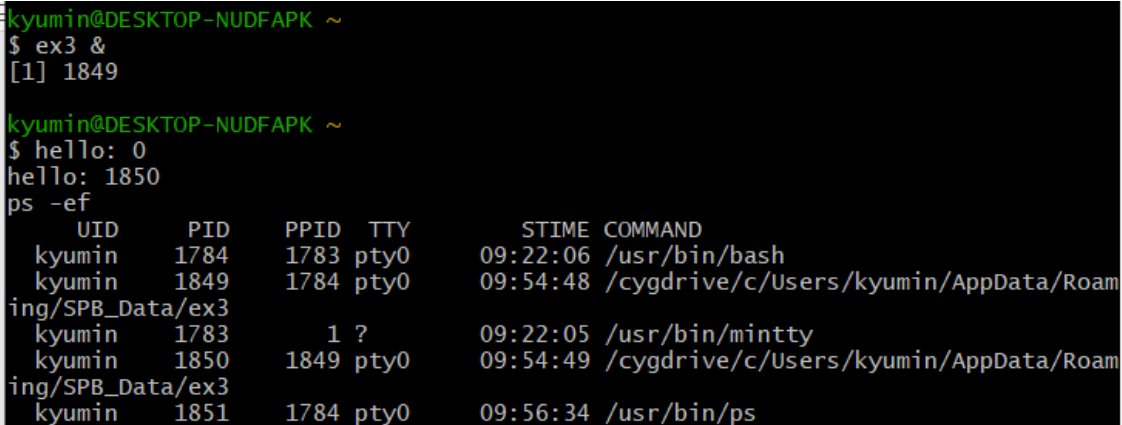
```
$ ex3 &
```

```
hello: 22644
```

```
hello: 0
```

```
$ ps -ef
```

```
.....
```



```
kyumin@DESKTOP-NUDFAPK ~
$ ex3 &
[1] 1849

kyumin@DESKTOP-NUDFAPK ~
$ hello: 0
hello: 1850
ps -ef
  UID      PID    PPID  TTY          STIME   COMMAND
  kyumin   1784    1783  pty0      09:22:06 /usr/bin/bash
  kyumin   1849    1784  pty0      09:54:48 /cygdrive/c/Users/kyumin/AppData/Roam
ing/SPB_Data/ex3
  kyumin   1783      1  ?          09:22:05 /usr/bin/mintty
  kyumin   1850    1849  pty0      09:54:49 /cygdrive/c/Users/kyumin/AppData/Roam
ing/SPB_Data/ex3
  kyumin   1851    1784  pty0      09:56:34 /usr/bin/ps
```

Ex3의 부모의 int x값은 1850이므로 자식의 pid는 1850이다. Ps -ef를 사용해 프로세스를 확인해봤다. 자식부터 pid를 따라가보면 1850 -> 1849 -> 1784 -> 1783 -> 1임을 볼 수 있다.

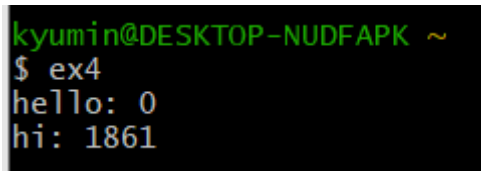
5) Try below (ex4.c). Which message was displayed by the parent and which one by the child?

```
void main(){
    int x;
    x=fork();
    if (x==0){
        printf("hello: %d\n", x);
    }else{
        printf("hi: %d \n", x);
    }
}
```

```
$ gcc -o ex4 ex4.c
```

```
$ ex4
```

.....



```
kyumin@DESKTOP-NUDFAPK ~
$ ex4
hello: 0
hi: 1861
```

위 코드는 x의 값이 0이라면 hello를 출력하고, 0이 아니라면 hi를 출력한다.

자식의 경우 x는 0이므로 hello를 출력할 것이고, 부모의 경우 x는 자식의 pid를 받으므로 hi가 출력될 것이다.

위 사진의 경우에는 위줄은 자식이 출력한 결과고, 아랫줄이 부모가 출력한 결과다.

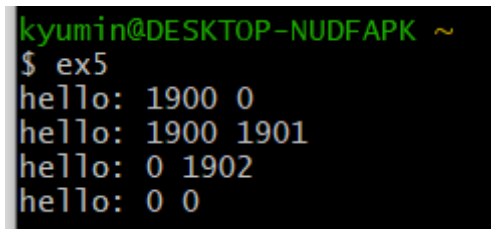
6) Try below (ex5.c). How many hellos do you see? Explain why you have that many hellos.
Draw the process tree.

```
void main(){
    int x,y;
    x=fork();
    y=fork();
    printf("hello: %d %d\n", x, y);
}
```

```
$ gcc -o ex5 ex5.c
```

```
$ ex5
```

.....



부모(1900 1901)

1900(0 1902)

1901(1900 0)

1902(0 0)

총 4줄이 출력되는데 그 이유는 fork가 두 번 실행되므로 복제된 자식이 다시 복제를 하기 때문이다.

우선 부모는 복제를 하고 자식의 pid를 x에 저장한다. 그러면 자식은 y=fork()를 실행하므로 자식의 자식이 생긴다. 다시 부모의 코드로 돌아가보면 y=fork()를 실행하면 또 다른 자식이 생기고, 그 자식의 pid를 y에 저장한다.

출력된 메시지 중에서 x와 y 모두가 0이 아니라면 부모가 출력한 메시지라는 뜻이다. 그러므로 부모의 자식의 pid는 1900과 1901이다.

Pid가 1900인 자식은 x의 값은 0으로 저장되고, 그 상태로 y=fork()를 실행한다. 그러면 자식의 x는 0이고 y는 자식의 자식의 pid를 갖을 것이다. 그러므로 hello: 0 1902가 자식이 출력한 메시지고, 자식의 자식의 pid는 1902임을 알 수 있다. 그리고 자식의 자식이 printf 할 때는 x와 y모두 0이 출력된다.

부모가 y=fork()를 실행할 때 x의 값은 1900으로 저장된 상태로 실행한다. 그러므로 부모의 또 다른 자식이 x와 y를 출력한다면 1900 0이 출력되는 것이다.

위 사진에서 우측 사진은 각각의 관계를 정리한 것이다. PID(x y) 형식으로 정리했다.

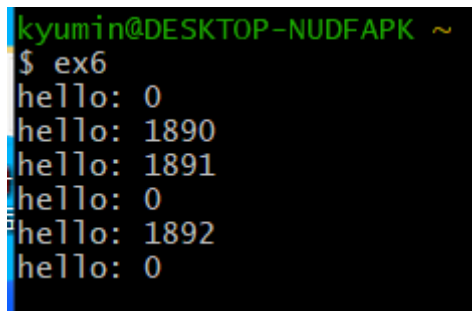
7) Try below (ex6.c). How many hellos do you see? Explain why you have that many hellos.

```
void main(){
    int x,y;
    x=fork();
    printf("hello: %d\n", x);
    y=fork();
    printf("hello: %d\n", y);
}
```

```
$ gcc -o ex6 ex6.c
```

```
$ ex6
```

.....



```
kyumin@DESKTOP-NUDFAPK ~
$ ex6
hello: 0
hello: 1890
hello: 1891
hello: 0
hello: 1892
hello: 0
```

printf("hello: %d\n", x);는 두 번 실행되고, printf("hello: %d\n", y);는 네 번 실행된다.

우선 x=fork()가 실행된 후 자식 하나만 복제되고, printf("hello: %d\n", x);가 실행되니 hello는 두 번 출력된다.

그리고 부모와 자식 모두 fork를 한 번 더 실행하므로 총 네 개의 프로세스가 되고, y와함께 hello가 4번 출력된다.

즉 hello가 총 6번 출력된다.

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. If different, explain why.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
void main(){
    int x, i;
    for(i=0;i<5;i++){
        x=fork();
        if (x==0){ // child
            int k;
            for(k=0;k<1000;k++){
                printf("%d-th child running %d-th iteration\n", i, k);
                fflush(stdout);          // to make printf work immediately
                usleep(10000);           // sleep 10000 microseconds
            }
            exit(0);  // child exits after 1000 iterations
        }
    }
    // now parent
    printf("parent exits\n");
}
```

```

kyumin@DESKTOP-NUDFAPK ~
$ ex7
0-th child running 0-th iteration
1-th child running 0-th iteration
0-th child running 1-th iteration
1-th child running 1-th iteration
0-th child running 2-th iteration
2-th child running 0-th iteration
2-th child running 1-th iteration
1-th child running 2-th iteration
0-th child running 3-th iteration
3-th child running 0-th iteration
0-th child running 4-th iteration
1-th child running 3-th iteration
2-th child running 2-th iteration
1-th child running 4-th iteration
0-th child running 5-th iteration
2-th child running 3-th iteration
3-th child running 1-th iteration
parent exits
4-th child running 0-th iteration
1-th child running 5-th iteration

```

프로그램을 실행하면 부모 프로세스는 복제를 5번 진행한다. 부모 프로세스가 종료된다면 “parent exits”라는 메시지가 출력되는데 위 사진을 보면 프로그램을 실행하자마자 출력된 것을 볼 수 있다.

자식 프로세스들은 모두 x 가 0이므로 $\text{if}(x==0)$ 을 실행하게 되는데 for문을 1000번을 돌아야하고, 그 안에 `usleep(10000)`라는 코드가 있다. 즉 자식 프로세스는 이론적으로 아무리 빨라도 10초 동안은 프로그램이 돌아야한다는 것이다. 그리고 자식 프로세스는 `exit(0)`가 실행되므로 자식이 복제를 하는 경우는 없다.

부모 프로세스보다 자식 프로세스가 먼저 끝날 가능성은 없으므로 프로그램을 시작하자마자 동시에 실행되는 프로세스는 6개다.

```

1-th child running 10-th iteration
0-th child running 11-th iteration
3-th child running 7-th iteration
4-th child running 6-th iteration

kyumin@DESKTOP-NUDFAPK ~
$ 0-th child running 12-th iteration
3-th child running 8-th iteration
1-th child running 11-th iteration
2-th child running 10-th iteration
4-th child running 7-th iteration
2-th child running 11-th iteration
4-th child running 8-th iteration

```

```

0-th child running 998-th iteration
3-th child running 994-th iteration
2-th child running 996-th iteration
1-th child running 997-th iteration
4-th child running 992-th iteration
4-th child running 993-th iteration
1-th child running 998-th iteration
2-th child running 997-th iteration
3-th child running 995-th iteration
0-th child running 999-th iteration
2-th child running 998-th iteration
3-th child running 996-th iteration
1-th child running 999-th iteration
4-th child running 994-th iteration
4-th child running 995-th iteration
3-th child running 997-th iteration
2-th child running 999-th iteration
3-th child running 998-th iteration
4-th child running 996-th iteration
3-th child running 999-th iteration
4-th child running 997-th iteration
4-th child running 998-th iteration
4-th child running 999-th iteration

```

```

3-th child running 994-th iteration
1-th child running 997-th iteration
0-th child running 998-th iteration
4-th child running 992-th iteration
2-th child running 996-th iteration
4-th child running 993-th iteration
1-th child running 998-th iteration
0-th child running 999-th iteration
3-th child running 995-th iteration
3-th child running 996-th iteration
1-th child running 999-th iteration
4-th child running 994-th iteration
2-th child running 997-th iteration
2-th child running 998-th iteration
4-th child running 995-th iteration
3-th child running 997-th iteration
3-th child running 998-th iteration
4-th child running 996-th iteration
2-th child running 999-th iteration
3-th child running 999-th iteration
4-th child running 997-th iteration
4-th child running 998-th iteration
4-th child running 999-th iteration

```

프로그램을 두 번 실행해보았고, 위의 두 사진은 그 결과다. 여기서 k의 값이 999라면 그 프로세스는 종료된 것이다.(종료될 때까지의 시간 격차는 있을 수 있다.)

왼쪽 사진의 경우 i가 0 - 1 - 2 - 3 - 4 순서로 종료되었고, 0 - 1 - 2 - 3 - 4 순서로 종료되었다. 이 외에도 여러 번 시도해봤으나 순서대로 종료되었다.

```
2-th child running 995-th iteration
4-th child running 994-th iteration
1-th child running 996-th iteration
3-th child running 992-th iteration
2-th child running 996-th iteration
1-th child running 997-th iteration
3-th child running 993-th iteration
4-th child running 995-th iteration
2-th child running 997-th iteration
1-th child running 998-th iteration
4-th child running 996-th iteration
2-th child running 998-th iteration
3-th child running 994-th iteration
1-th child running 999-th iteration
3-th child running 995-th iteration
2-th child running 999-th iteration
4-th child running 997-th iteration
4-th child running 998-th iteration
3-th child running 996-th iteration
3-th child running 997-th iteration
4-th child running 999-th iteration
3-th child running 998-th iteration
3-th child running 999-th iteration
```

이 사진의 경우 0번은 위쪽에 있어 안보이지만 0 - 1 - 2 - 4 - 3 순서로 종료되었다. 스케줄러는 프로세스는 하나씩 실행되는 것을 말하는데, 이 때 먼저 실행된 프로세스보다 나중에 실행된 프로세스가 더 빠르게 진행되는 경우가 있다. 예를 들어 세 번째 프로세스는 `usleep(10000)`가 끝났어도 스케줄러가 다른 프로세스를 돌리느라 코드를 진행하지 못하고 느려지고, 반대로 4번째 프로세스는 `usleep`이 끝나자마자 바로 다음 코드를 실행할 수 있다. 이렇게 먼저 실행된 프로세스여도 타이밍이 맞지 않아 더 늦게 끝나는 경우가 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. You may want to decrease the size of inner loop (k loop) to "k<10" to shorten the run time.

```
void main(){
    int x, i;
    int repeatNum = 5;
    for(i=0; i < repeatNum; i++){
        x=fork();
        if (x==0){ // child
            int k;
            for(k=0; k<10; k++){
                printf("%d-th\n", k);
                fflush(stdout);
                usleep(10000);
            }
        }
        //printf("%d\n", getpid());
    }
}
```

```
kyumin@DESKTOP-NUDFAPK ~
$ ex7
0-th child running 0-th iteration
0-th child running 1-th iteration
1-th child running 0-th iteration
0-th child running 2-th iteration
1-th child running 1-th iteration
2-th child running 0-th iteration
2-th child running 1-th iteration
1-th child running 2-th iteration
0-th child running 3-th iteration
3-th child running 0-th iteration
3-th child running 1-th iteration
2-th child running 2-th iteration
0-th child running 4-th iteration
1-th child running 3-th iteration
4-th child running 0-th iteration
1-th child running 4-th iteration
0-th child running 5-th iteration
2-th child running 3-th iteration
3-th child running 2-th iteration
3-th child running 3-th iteration
2-th child running 4-th iteration
4-th child running 1-th iteration
0-th child running 6-th iteration
1-th child running 5-th iteration
0-th child running 7-th iteration
2-th child running 5-th iteration
4-th child running 2-th iteration
1-th child running 6-th iteration
3-th child running 4-th iteration
3-th child running 5-th iteration
1-th child running 7-th iteration
4-th child running 3-th iteration
2-th child running 6-th iteration
0-th child running 8-th iteration
1-th child running 8-th iteration
3-th child running 6-th iteration
4-th child running 4-th iteration
0-th child running 9-th iteration
2-th child running 7-th iteration
2-th child running 8-th iteration
4-th child running 5-th iteration
3-th child running 7-th iteration
1-th child running 9-th iteration
kyumin@DESKTOP-NUDFAPK ~
$ 3-th child running 8-th iteration
4-th child running 6-th iteration
2-th child running 9-th iteration
```

```
3-th child running 6-th iteration
4-th child running 5-th iteration
4-th child running 5-th iteration
4-th child running 6-th iteration
4-th child running 5-th iteration
4-th child running 6-th iteration
3-th child running 7-th iteration
4-th child running 7-th iteration
3-th child running 8-th iteration
4-th child running 6-th iteration
4-th child running 6-th iteration
4-th child running 7-th iteration
4-th child running 7-th iteration
4-th child running 8-th iteration
4-th child running 8-th iteration
3-th child running 9-th iteration
4-th child running 8-th iteration
4-th child running 8-th iteration
4-th child running 9-th iteration
4-th child running 9-th iteration
4-th child running 9-th iteration
4-th child running 0-th iteration
4-th child running 1-th iteration
4-th child running 2-th iteration
4-th child running 3-th iteration
4-th child running 4-th iteration
4-th child running 5-th iteration
4-th child running 6-th iteration
4-th child running 7-th iteration
4-th child running 8-th iteration
4-th child running 9-th iteration
```

반복 회수만 5번으로 설정하여 출력해보면 다음과 같은 결과가 나온다. i와 j가 같은 값인 경우가 있기 때문에 동일한 메시지가 나오는 경우가 있다. 그래서 다음과 같이 보기 좋게 수정했다.

```

void main(){
    int x, i;
    int repeatNum = 5;
    for(i=0; i < repeatNum; i++){
        x=fork();
        /*if (x==0){ // child
            int k;
            for(k=0; k<10; k++){
                printf("
                fflush(s
                usleep(1
            }
        }*/
    }
    printf("%d\n", getpid());
}

```

```

kyumin@DESKTOP-NUDFAPK ~
$ ex7
630
614
631
622
618
632
624
633
623
634
625
635
636
621
628
637
638
639
620
629
640
616
617
641
642
627
643
615
619
644
645
626

```

For문 밖에 자신의 pid를 출력하는 코드를 만들었다. 오른쪽 사진처럼 pid들이 출력되었고 개수를 세보면 32개다. 즉 5번 반복할 때 총 pid 수는 32다.

10) Write a program that creates n child processes where n is specified by the user. Let each process prints some message when it has finished.

```
$ ex1
```

```
how many child processes?
```

```
5
```

```
child 1 finished
```

```
child 2 finished
```

```
child 4 finished
```

```
child 3 finished
```

```
child 5 finished
```

```
parent finished
```

```
void main(){
    int x, i, n;
    printf("how many child processes?\n");
    scanf("%d", &n);
    for(i = 0; i < n; i++){
        x = fork();
        if (x == 0){ // child
            printf("child %d finished\n", i + 1);
            exit(0);
        }
    }
    printf("parent finished\n");
}
```

```
kyumin@DESKTOP-NUDFAPK ~
$ ex8
how many child processes?
5
child 1 finished
child 2 finished
child 3 finished
child 4 finished
parent finished
child 5 finished
```

Scanf를 통해 복제 수를 n에 입력받았다. for문을 n만큼 반복했다. 그러나 자식이 복제를 하는 문제가 있을 수 있기 때문에 x가 0인 경우 exit(0)을 이용해 프로그램을 종료했다.

결과를 보면 입력받은 수만큼 복제가 되었지만 마지막에 child5보다 부모가 먼저 종료되었다.

11) Repeat Problem 10, but let the processes end in the reverse order in their creation time as shown below. Use sleep() function to force the order.

\$ ex1

how many child processes?

5

parent finished

child 5 finished

child 4 finished

child 3 finished

child 2 finished

child 1 finished

```
void main(){
    int x, i, n;
    printf("how many child processes?\n");
    scanf("%d", &n);
    for(i = 0; i < n; i++){
        x = fork();
        if (x == 0){ // child
            usleep((n - i)*50000);
            printf("child %d finished\n", i + 1);
            exit(0);
        }
    }
    printf("parent finished\n");
    usleep(n * 50000);
}
```

```
kyumin@DESKTOP-NUDFAPK ~
$ ex8
how many child processes?
5
parent finished
child 5 finished
child 4 finished
child 3 finished
child 2 finished
child 1 finished
```

프로세스를 생성한 순서의 반대로 출력을 하기 위해서 sleep 함수를 이용해줬고, 모두 다른 딜레이를 줬다. 처음 복제된 프로세스가 마지막에 복제된 프로세스보다 더 늦게 작동해야하므로 sleep((n-i)*50000) 코드를 만들어줬다. Sleep에 들어갈 변수가 i가 커질수록 작아지게 만들기 위함이다. 50ms를 곱해준 이유는 n이 작을 때 순서가 바뀌지 않고 출력될 위험이 있어서다. 만약 10ms를 곱한다면 순서대로 출력되고, 50ms정도로해줘야 순서가 바뀌어 출력된다.

12) Write a program that the parent writes some 1-digit number x into a file and the child reads it and prints $x + 1$. Assume x is less than 10. Check this file with `xxd` if it has the number written by the parent.

\$ `ex1`

parent writes 7

child prints 8

```
void main(){
    int fp, cp, y;
    char *x = "7";
    char buf[20];
    fp = open("digitfile", O_RDWR | O_CREAT | O_TRUNC, 00777);
    write(fp, x, 1);
    lseek(fp, 0, SEEK_SET);

    y = read(fp, buf, 1);
    buf[1] = '\0';
    printf("parent prints %s\n", buf);
    lseek(fp, 0, SEEK_SET);

    cp = fork();
    if(cp == 0) {
        y = read(fp, buf, 1);
        int R = atoi(buf);
        printf("child prints %d\n", ++R);
    }
}
```

```
kyumin@DESKTOP-NUDFAPK ~
$ gcc -g -o ex9 ex9.c

kyumin@DESKTOP-NUDFAPK ~
$ ex9
Parent writes 7
child prints 8

kyumin@DESKTOP-NUDFAPK ~
$ xxd digitfile
00000000: 37
```

우선 x 값은 7로 지정이 되었기 때문에 스트링을 포인터 x 에 저장해줬다. 이후 파일을 생성 또는 초기화할 수 있도록 `open` 함수를 사용하였다. `Write` 함수를 사용하여 x 의 내용을 파일에 입력해줬다. 입력을 하면 커서가 입력된 글자 뒤를 향하기 때문에 나중에 `read` 함수를 사용하면 출력이 되지 않는 문제가 발생한다. 그래서 `lseek` 함수를 사용해서 커서를 0으로

즉 처음으로 돌려주었다.

그리고 파일을 다시 read해서 읽은 내용을 buf 배열에 저장해줬다. Read 함수를 사용하면 buf에 문자 종료가 없다면 문제가 발생할 수 있으므로 안전을 위해서 buf[1] = '\0'으로 저장해줬다. Buf 를 출력해보면 7이 출력되는 것을 볼 수 있다. 다시 커서를 처음으로 돌려주었다.

그리고 fork 함수를 사용해서 프로세스 복제를 했고, 부모 프로세스의 변수들은 그대로 자식 프로세스에 저장된다. 그래서 read 함수를 사용해서 buf에 파일 내용을 저장해줬다. 그러나 1이 더해진 값을 출력해야한다. 파일의 내용은 스트링이므로 int로 전환하기 위해서 atoi 함수를 사용했다. 그렇게 얻은 결과값 7에 1을 더한 후 출력을 했다.

그리고 출력 결과를 보면 정상적으로 7과 8이 출력된 것을 볼 수 있고, xxd를 통해 파일의 내용을 확인해보면 7이 저장되어있는 것을 볼 수 있다.