

운영체제 보고서

과제 #04

강좌 명: 운영체제

교수님: 정준호 교수님

학과: 컴퓨터공학과

학년: 3학년

학번: 2017112138

이름: 정여준

문제 3.5

Including the initial parent process, how many processes are created by the program shown in Figure 3.31?

<Figure 3.31>

```
#include<stdio.h>

#include<unistd.h>

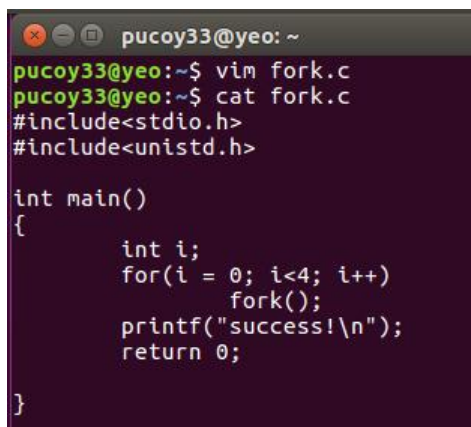
int main()
{
    int i;

    for ( i = 0; i<4; i++)

        fork();

    return 0;
}
```

이러한 코드를 vim 명령어를 통해서 생성해준다. 생성할 때 몇 개의 프로세스가 생성되었는지 확인하기 위해 printf()함수를 삽입한다.

A terminal window titled 'pucoy33@yeo: ~' shows the following commands and output: 'vim fork.c' is executed, followed by 'cat fork.c' which displays the C code from Figure 3.31. The code includes <stdio.h> and <unistd.h>, and a main function that loops from i=0 to i=3, calling fork() in each iteration, and then prints 'success!\n' and returns 0.

```
pucoy33@yeo: ~
pucoy33@yeo:~$ vim fork.c
pucoy33@yeo:~$ cat fork.c
#include<stdio.h>
#include<unistd.h>

int main()
{
    int i;
    for(i = 0; i<4; i++)
        fork();
    printf("success!\n");
    return 0;
}
```

이제 gcc -o fork fork.c 명령어를 통해서 fork.c 코드로 실행가능한 파일 fork를 생성해준다. 그리고 ./fork 명령어를 통해서 실행한다.

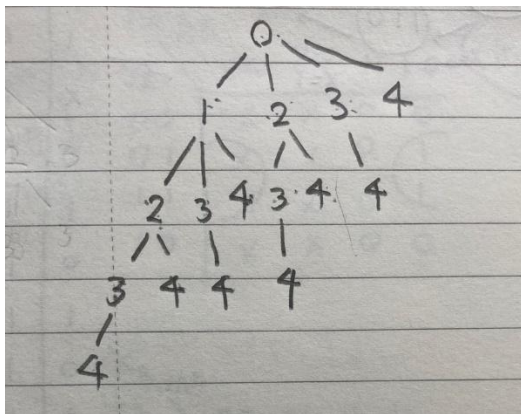
```

pucoy33@yeo:~$ gcc -o fork fork.c
pucoy33@yeo:~$ ./fork
success!
pucoy33@yeo:~$ success!
success!
success!
success!
success!
success!
success!
success!
success!
success!
success!
success!
success!
success!
success!
success!

```

success!가 16번 출력된 것을 확인할 수 있다. 이로써 이 프로그램에 의해서 프로세스가 총 16개 생성된 것을 확인할 수 있다. 추가적으로 fork()함수를 한 횟수를 n 이라 하였을 때, 생성되는 프로세스의 수는 2^n 임을 알게 되었다.

이를 손으로 풀어서 확인해보면 다음과 같다.



총 16개의 프로세스가 생성되는 것을 확인할 수 있다.

문제 3.6

Explain the circumstances under which the line of code marked **printf("Line J")** in Figure 3.32 will be reached.

<Figure 3.32>

```
#include <sys/types.h>

#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    pid = fork();

    if(pid < 0){
        fprintf(stderr, "Fork Failed");

        return 1;
    }

    else if(pid == 0){
        execlp("/bin/ls", "ls", NULL);

        printf("LINE J");
    }

    else{
        wait(NULL);

        printf("Child Complete\n");
    }

    return 0;
}
```

vim line.c 명령어를 통해서 위의 Figure 3.32 코드를 생성한다.

```
pucoy33@yeo: ~  
pucoy33@yeo:~$ vim line.c  
pucoy33@yeo:~$ cat line.c  
#include<sys/types.h>  
#include<stdio.h>  
#include<unistd.h>  
  
int main()  
{  
    pid_t pid;  
  
    pid =fork();  
  
    if(pid<0){  
        fprintf(stderr,"Fork Failed");  
        return 1;  
    }  
    else if(pid == 0){  
        execlp("/bin/ls","ls",NULL);  
        printf("Line J");  
    }  
    else{  
        wait(NULL);  
        printf("Child Complete\n");  
    }  
  
    return 0;  
}
```

이제 gcc -o line line.c 명령어를 통해서 생성한 line.c 파일을 이용해서 실행가능한 파일 line을 생성한다.

```
pucoy33@yeo:~$ gcc -o line line.c  
line.c: In function 'main':  
line.c:20:3: warning: implicit declaration of function 'wait' [-Wimplicit-function-declaration]  
    wait(NULL);  
    ^
```

이제 ./line 명령어를 통해서 실행가능한 파일 line을 실행한다.

```
pucoy33@yeo:~$ ./line  
examples.desktop  fork.c  line.c  다운로드  바탕화면  사진  템플릿  
fork              line   공개   문서      비디오    음악  
Child Complete  
pucoy33@yeo:~$
```

자식 프로세스가 execlp() 시스템 콜을 실행할 때 성공적으로 실행되면 자식 프로세스에 할당된 메모리 공간이 해당 프로세스로 대체되며 이 프로세스에는 execlp() 시스템 콜에 대한 매개변수로 지정된다. 이 경우에 printf("LINE J") 줄은 새 프로세스에 의해 덮어쓰기 때문에 절대 도달하지 않는다. 하지만 execlp()함수의 실행이 실패한다면 프로세스의 교체가 없고 printf("LINE J") 줄이 실행된다.

문제 3.7

Using the program in Figure 3.33, identify the values of pid at lines A, B, C, and D. (Assume that the actual pids of the parent and child are 2600 and 2603, respectively.)

<Figure 3.33>

```
#include <sys/types.h>

#include <stdio.h>

#include <unistd.h>

int main()
{
    pid_t pid, pid1;

    pid = fork();

    if(pid < 0){
        fprintf(stderr, "Fork Failed");

        return 1;
    }

    else if(pid == 0){
        pid1 = getpid();

        printf("child: pid = %d", pid); /*A*/

        printf("child: pid1 = %d", pid1); /*B*/
    }

    else{
        pid1 = getpid();

        printf("parent: pid = %d", pid); /*C*/

        printf("parent: pid1 = %d", pid1); /*D*/
    }
}
```

```

        wait(NULL);

    }

    return 0;

}

```

vim pid.c 명령어를 통해 Figure 3.33에 있는 코드를 파일로 생성한다.

```

pucoy33@yeo:~$ vim pid.c
pucoy33@yeo:~$ cat pid.c
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>

int main()
{
    pid_t pid,pid1;
    pid = fork();

    if(pid<0){
        fprintf(stderr,"Fork Failed");
        return 1;
    }
    else if(pid == 0){
        pid1 = getpid();
        printf("child: pid = %d",pid); /*A*/
        printf("child: pid1 = %d",pid1); /*B*/
    }
    else{
        pid1 = getpid();
        printf("parent: pid = %d",pid); /*C*/
        printf("parent: pid1 = %d",pid1); /*D*/
        wait(NULL);
    }
    return 0;
}

```

gcc -o pid pid.c 명령어를 통해 생성한 파일 pid.c를 사용해서 실행가능한 파일 pid를 생성한다.

```

pucoy33@yeo:~$ gcc -o pid pid.c
pid.c: In function 'main':
pid.c:23:3: warning: implicit declaration of function 'wait' [-Wimplicit-function-declaration]
    wait(NULL);
    ^

```

이제 생성된 실행가능한 파일 pid를 ./pid 명령어를 통해서 실행시킨다.

```

pucoy33@yeo:~$ ./pid
child: pid = 0child: pid1 = 3120parent: pid = 3120parent: pid1 = 3119

```

각각 부모와 자녀의 실제 피드를 2600과 2603으로 가정했으므로 답은 다음과 같다.

A = 0

B = 2603

C = 2603

D = 2600

문제 3.10

Using the program shown in Figure 3.34, explain what the output will be at X and Y.

<Figure 3.34>

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 5

int nums[SIZE] = {0, 1, 2, 3, 4};

int main()
{
    int i;
    pid_t pid;

    pid = fork()

    if (pid == 0) {
        for(i = 0; i < SIZE; i++) {
            nums[i] *= -i;
            printf("CHILD: %d ", nums[i]); /* LINE X */
        }
    }
    else if (pid > 0) {
        wait(NULL);
        for(i = 0; i < SIZE; i++)
            printf("PARENT: %d ", nums[i]); /* LINE Y */
    }
    return 0;
}
```


vim q4.c 명령어를 통해서 Figure 3.34 코드파일을 생성한다.

```
pucoy33@yeo:~$ vim q4.c
pucoy33@yeo:~$ cat q4.c
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
    int i;
    pid_t pid;
    pid = fork();

    if(pid == 0){
        for(i=0; i<SIZE; i++){
            nums[i] *= -i;
            printf("CHILD: %d ",nums[i]); /*LINE X*/
        }
    }
    else if(pid>0){
        wait(NULL);
        for(i=0; i<SIZE; i++)
            printf("PARENT: %d ",nums[i]); /*LINE Y*/
    }
    return 0;
}
```

gcc -o q4 q4.c 명령어를 통해서 생성한 q4.c파일을 이용해서 실행가능한 파일 q4를 생성한다.

```
pucoy33@yeo:~$ gcc -o q4 q4.c
q4.c: In function 'main':
q4.c:22:3: warning: implicit declaration of function 'wait' [-Wimplicit-function-declaration]
    wait(NULL);
    ^
```

./q4 명령어를 통해 실행가능한 파일 q4를 실행시킨다.

```
pucoy33@yeo:~$ ./q4
CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16 PARENT: 0 PARENT: 1 PARENT: 2
PARENT: 3 PARENT: 4 pucoy33@yeo:~$
```

결과는 다음과 같다.

X : 0 * (-0) = 0 / 1 * (-1) = -1 / 2 * (-2) = -4 / 3 * (-3) = -9 / 4 * (-4) = -16

Y: 0 / 1 / 2 / 3 / 4

X : 0 -1 -4 -9 -16

Y : 0 1 2 3 4

문제 3.12

Using either a UNIX or a Linux system, write a C program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds. Process states can be obtained from the command. `ps -1`. The process states are shown below the S column; processes with a state of Z are zombies. The process identifier (pid) of the child process is listed in the PID column, and that of the parent is listed in the PPID column. Perhaps the easiest way to determine that the child process is indeed a zombie is to run the program that you have written in the background (using the `&`) and then run the command `ps -1` to determine whether the child is a zombie process. Because you do not want too many zombie processes existing in the system, you will need to remove the one that you have created. The easiest way to do that is to terminate the parent process using the `kill` command. For example, if the process id of the parent is 4884, you would enter `kill -9 4884`

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>

int main()
{
    pid_t pid;

    pid = fork();
    if (pid > 0)
    {
        printf("Parent process");
        sleep(10);
    }
    else if (pid == 0)
```

```

    {
        printf("Child process");
        exit(0);
    }
    return 0;
}

```

vim zombie.c 명령어를 통해서 위와 같은 코드를 가진 zombie.c파일을 생성한다.

```

pucoy33@yeo:~$ vim zombie.c
pucoy33@yeo:~$ cat zombie.c
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>

int main()
{
    pid_t pid;
    pid = fork();

    if(pid>0)
    {
        printf("Parent Process");
        sleep(10);
    }
    else if(pid == 0)
    {
        printf("Child Process");
        exit(0);
    }

    return 0;
}

```

이렇게 생성한 파일을 이용해서 실행가능한 파일을 만들어준다.

gcc -o zombie zombie.c 명령어를 이용해서 위의 파일을 사용한 실행가능한 파일 zombie를 생성한다.

```

pucoy33@yeo:~$ gcc -o zombie zombie.c

```

이제 ./zombie& 명령어를 통해서 zombie 파일을 실행시킨다.

그 이후 ps -e -o pid,ppid,stat,cmd 명령어를 통해서 프로세스를 확인한다.

```

pucoy33@yeo:~$ ./zombie&
[1] 15366
pucoy33@yeo:~$ Child Process
pucoy33@yeo:~$ ps -e -o pid,ppid,stat,cmd
  PID  PPID  STAT  CMD
    1     0  Ss    /sbin/init splash
    2     0  S      [kthreadd]
    4     2  I<    [kworker/0:0H]
    6     2  I<    [mm_percpu_wq]
    7     2  S      [ksoftirqd/0]
    8     2  R      [rcu_sched]
    9     2  I      [rcu_bh]
   10     2  S      [migration/0]
   11     2  S      [watchdog/0]
   12     2  S      [cpuhp/0]
   13     2  S      [kdevtmpfs]
   14     2  I<    [netns]
   15     2  S      [rcu_tasks_kthre]
   16     2  S      [kauditd]
   17     2  S      [khungtaskd]
   18     2  S      [oom_reaper]
   19     2  I<    [writeback]
   20     2  S      [kcompactd0]
   21     2  SN      [ksmd]
   22     2  SN      [khugepaged]
   23     2  I<    [crypto]
   24     2  I<    [kintegrityd]
   25     2  I<    [kblockd]
   26     2  I<    [ata_sff]
   27     2  I<    [md]
   28     2  I<    [edac-poller]
   29     2  I<    [devfreq_wq]
   30     2  I<    [watchdogd]
   34     2  S      [kswapd0]
   35     2  I<    [kworker/u3:0]
   36     2  S      [ecryptfs-kthrea]
   78     2  I<    [kthrotld]
   79     2  I<    [acpi_thermal_pm]
   80     2  S      [scsi_eh_0]
   81     2  I<    [scsi_tmf_0]
   82     2  S      [scsi_eh_1]
   83     2  I<    [scsi_tmf_1]
   89     2  I<    [ipv6_addrconf]
   98     2  I<    [kstrp]
  115     2  I<    [charger_manager]
  158     2  I<    [kworker/0:1H]
  160     2  S      [scsi_eh_2]
  161     2  I<    [scsi_tmf_2]
  162     2  I<    [ttm_swap]

```

```

14363     2  I<    [xfsalloc]
14364     2  I<    [xfs_mru_cache]
14368     2  S      [jfsIO]
14371     2  S      [jfsCommit]
14372     2  S      [jfsSync]
15207     2  I      [kworker/u2:1]
15260     2  I      [kworker/u2:0]
15319   943  Sl     /usr/lib/gnome-terminal/gnome-terminal-server
15324 15319  Ss     bash
15359     2  R      [kworker/u2:2]
15366 15324  S      ./zombie
15367 15366  Z      [zombie] <defunct>
15368 15324  R+     ps -e -o pid,ppid,stat,cmd
27803     2  I      [kworker/0:0]
27865     1  Ssl    /usr/lib/accountsservice/accounts-daemon
pucoy33@yeo:~$
[1]+  완료                  ./zombie

```

프로세스를 확인하면 위의 형광펜으로 칠한 부분과 같이 파일이 실행되고 끝나기 전까지 자식 프로세스가 좀비 프로세스가 되는 것을 확인할 수 있다. wait()이 실행되지 않아서 아이 프로세스가 좀비 프로세스가 되는 것이다. 즉 리눅스에서 좀비 프로세스는 exit()함수에 의해서 완료되지만 여전히 프로세스 테이블에 존재하는 프로세스를 의미하는 것으로 보인다. 부모가 wait() 함수를 호출하지 않으면 부모 프로세스가 살아있는 동안에 자식 프로세스도 좀비가 되어 살아있는 것을 확

인할 수 있다. 이를 kill -9 프로세스 명령어를 통해서 강제 종료시켜 좀비 프로세스를 없앨 수 있다. 위에 화면에서 좀비 프로세스는 15366이기 때문에 kill -9 15366 명령어를 입력하면 강제 종료한다. 이를 확인하기 위해서 프로세스가 유지되는 시간을 좀 늘려서 sleep(30);으로 바꾼 후에 실행해 보았다.

```
15378      2 I      [kworker/u2:0]
15436 15324 S      ./zombie
15437 15436 Z      [zombie] <defunct>
15438 15324 R+     ps -e -o pid,ppid,stat,cmd
27803      2 I      [kworker/0:0]
27865      1 Ssl    /usr/lib/accountsservice/accounts-daemon
pucoy33@yeo:~$ kill -9 15436
pucoy33@yeo:~$ ps -e -o pid,ppid,stat,cmd
PID  PPID  STAT  CMD
```

좀비 프로세스 15436을 강제종료 시키기 위해서 kill -9 15436 명령어를 입력하고 다시 프로세스의 상태를 확인해보았다.

```
15359      2 I      [kworker/u2:2]
15378      2 I      [kworker/u2:0]
15439 15324 R+     ps -e -o pid,ppid,stat,cmd
27803      2 I      [kworker/0:0]
27865      1 Ssl    /usr/lib/accountsservice/accounts-daemon
[1]+  죽었음      ./zombie
pucoy33@yeo:~$
```

좀비 프로세스가 죽었다고 나오는 것을 확인할 수 있다.

<소감>

이번 과제를 통해서 수업시간에 배웠던 프로세스가 생성될 때의 특성과 어떤 식으로 종료되는지 확인할 수 있었고 수업시간에 배웠을 때 살짝 이해하기 어려웠던 좀비 프로세스에 대한 이해도 확실히 할 수 있었다. 또한 `wait()` `exit()` `sleep()` 함수를 사용하면 어떻게 되는지도 직접 실행해보면서 알게 되었다. 또한 `execlp()` 함수를 사용하면서 어떤 현상이 발생해서 어떤 결과가 나오는지도 알 수 있게 되었다. 수업시간에 배웠던 프로세스에 대해서 제대로 복습할 수 있었던 시간을 가지게 된 것 같아서 굉장히 좋았다. 또한 코드를 직접 구현해보면서 코드를 짜는 실력도 향상된 것 같아서 기분이 좋았다. 여러모로 많은 부분에 있어서 배울 수 있었던 과제였던 것 같다.