3.3 Operations on Processes

- In UNIX-like O/S,
  - A new process is created by the `fork()` system call.
  - The *child* process consists of
    - a ***copy of the address space*** of the *parent* process.
  - Both processes continue execution
    - at the instruction after the `fork()` system call.
  - With one difference:
    - the return code for the `fork()` is ***zero*** for the child process, whereas
    - the ***nonzero*** **`pid`** of the child is returned to the parent process.

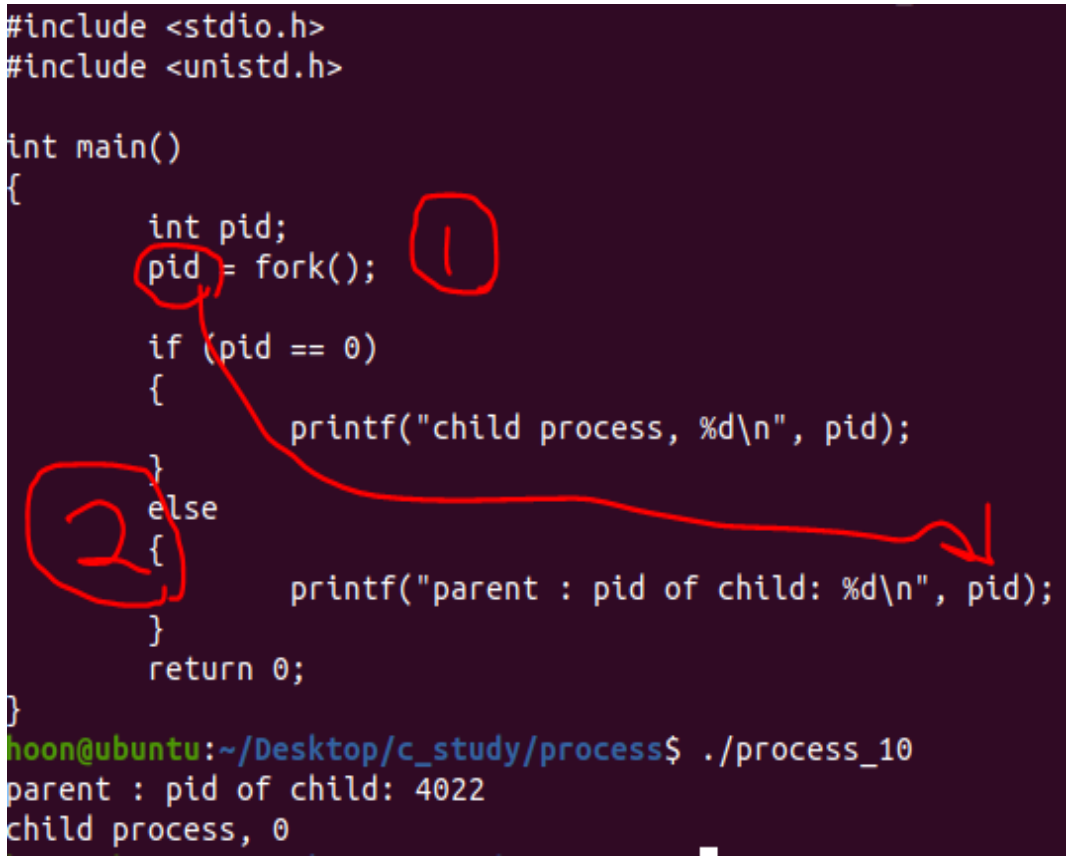
핵심!

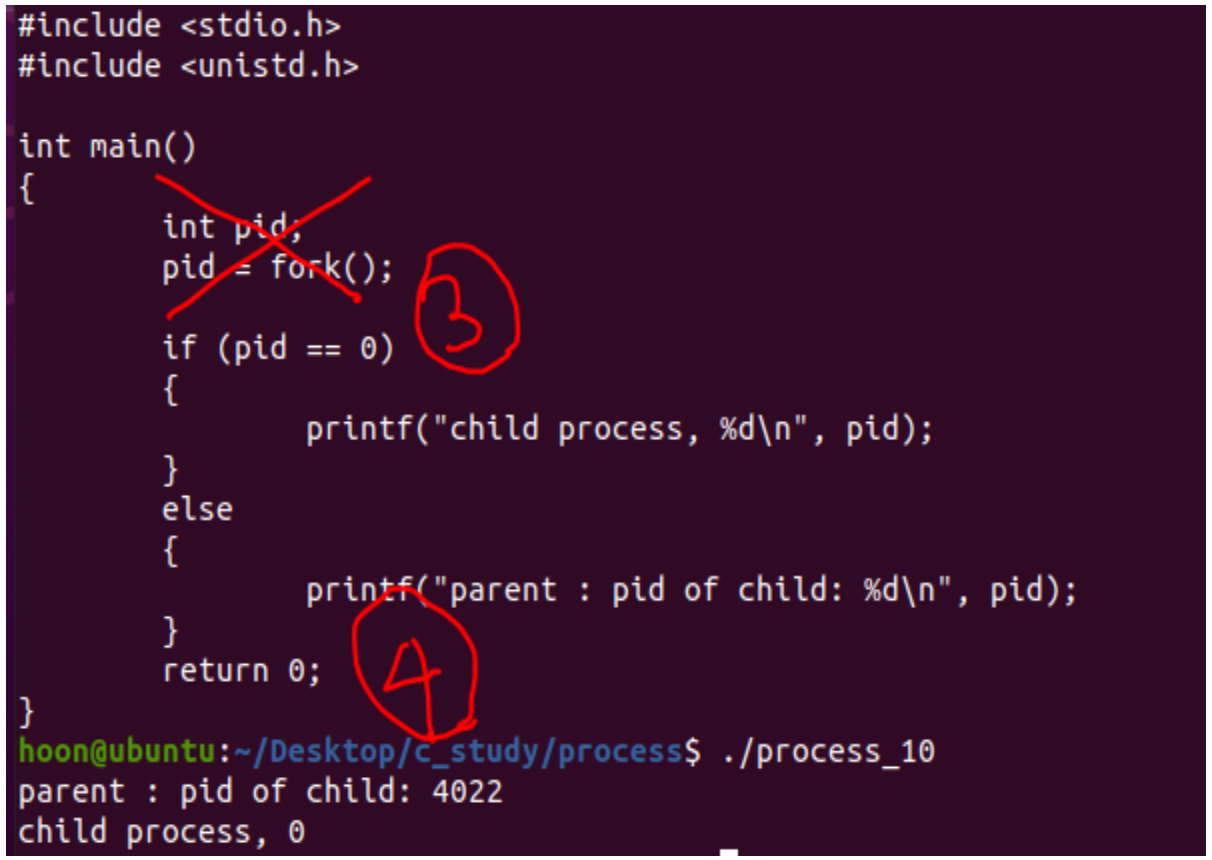
1. 부모 프로세스에서 fork()가 되면 자식 프로세스의 코드 실행은 fork() 아래 부터 실행된다 (pid == 0)

2. 보통 부모 프로세스에서 wait이 없으면 부모프로세스 리턴 -> 자식프로세스 실행

3. 부모프로세스와 자식프로세스는 메모리를 공유하지 않는다. (Text는 동일)

```
-rwxrwxr-x 1 hoon hoon 16744 Jul 18 19:51 process_1
-rw-rw-r-- 1 hoon hoon   125 Jul 18 04:44 process_1.c
-rwxrwxr-x 1 hoon hoon 16744 Jul 18 19:51 process_2
-rw-rw-r-- 1 hoon hoon   134 Jul 18 04:49 process_2.c
-rwxrwxr-x 1 hoon hoon 16784 Jul 18 19:52 process_3
-rw-rw-r-- 1 hoon hoon   180 Jul 18 19:52 process_3.c
-rwxrwxr-x 1 hoon hoon 16816 Jul 18 19:53 process_4
-rw-rw-r-- 1 hoon hoon   272 Jul 18 19:53 process_4.c
-rwxrwxr-x 1 hoon hoon 16704 Jul 18 19:53 process_5
-rw-rw-r-- 1 hoon hoon   229 Jul 18 19:35 process_5.c
-rwxrwxr-x 1 hoon hoon 16704 Jul 18 19:53 process_6
-rw-rw-r-- 1 hoon hoon   151 Jul 18 19:42 process_6.c
-rwxrwxr-x 1 hoon hoon 16832 Jul 18 19:54 process_7
-rw-rw-r-- 1 hoon hoon   293 Jul 18 19:54 process_7.c
-rwxrwxr-x 1 hoon hoon 16832 Jul 18 19:55 process_8
-rw-rw-r-- 1 hoon hoon   418 Jul 18 19:55 process_8.c
-rwxrwxr-x 1 hoon hoon 16832 Jul 18 19:55 process_9
-rw-rw-r-- 1 hoon hoon   433 Jul 18 19:55 process_9.c
```


#include <stdio.h>
#include <unistd.h>
int main()
{
int pid;
pid = fork();
if (pid == 0)
{
printf("child process, %d\n", pid);
}
else
{
printf("parent : pid of child: %d\n", pid);
}
return 0;
}
hoon@ubuntu:~/Desktop/c_study/process$ ./process_10
parent : pid of child: 4022
child process, 0
1
2


#include <stdio.h>
#include <unistd.h>
int main()
{
int pid;
pid = fork();
if (pid == 0)
{
printf("child process, %d\n", pid);
}
else
{
printf("parent : pid of child: %d\n", pid);
}
return 0;
}
hoon@ubuntu:~/Desktop/c_study/process$ ./process_10
parent : pid of child: 4022
child process, 0
3
4

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>

int global = 0; // global data
int main()
{
        int *heap = (int*)malloc(sizeof(int*)); // memory allocation
        heap[0] = 0;
        int stack = 0; // local variable
        int pid = fork();

        if (pid == 0) // child process
        {
                heap[0] = heap[0] + 2;
                stack = stack + 2;
                global = global + 2;
                printf("child process\n");
                printf("%d %d %d\n", heap[0], stack, global);
        }

        else if (pid > 0) // child process
        {
                wait(NULL);
                heap[0] = heap[0] + 1;
                stack = stack + 1;
                global = global + 1;
                printf("parent process\n");
                printf("%d %d %d\n", heap[0], stack, global);
        }
        return 0;
}
```

```
hoon@ubuntu:~/Desktop/c_study/process$ ./process_11
child process
2 2 2
parent process
1 1 1
```

메모리 공간은 (text, data, heap, stack) 영역으로 구성 되어있음

부모프로세스와 자식프로세스가 메모리 공간을 고유한다면 프로세스가 끝나기 전에 부모 프로세스가 출력하는 값은 3이 되어야 한다.

그런데 실제 출력값은
2 2 2
1 1 1로

이는 fork가 호출이 되면, 스택, 힙, 데이터 영역과 같은 것을 share 하지 않는다는 이야기입니다. 즉, **이 함수로 프로세스를 생성하면, 메모리 공간은 copy가 된다는 겁니다.**

부모에서의 global과
자식에서의 global은 다르다.

process_1.c

```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_1.c
#include <stdio.h>
#include <unistd.h>

int main()
{

        pid_t pid;
        pid = fork();
        printf("Hello, Process!\n");

        return 0;
}
hoon@ubuntu:~/Desktop/c_study/process$ ./process_1
Hello, Process!
Hello, Process!
```


```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_1.c
#include <stdio.h>
#include <unistd.h>

int main()
{

        pid_t pid;
        pid = fork();
        printf("Hello, Process!\n");

        return 0;
}
hoon@ubuntu:~/Desktop/c_study/process$ ./process_1
Hello, Process!
Hello, Process!
```

process_2.c

```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_2.c
#include <stdio.h>
#include <unistd.h>

int main()
{

        pid_t pid;
        pid = fork();
        printf("Hello, Process! %d\n", pid);

        return 0;
}

hoon@ubuntu:~/Desktop/c_study/process$ ./process_2
Hello, Process! 5114
Hello, Process! 0
```


```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_2.c
#include <stdio.h>
#include <unistd.h>

int main()
{

        pid_t pid;
        pid = fork();
        printf("Hello, Process! %d\n", pid);

        return 0;
}

hoon@ubuntu:~/Desktop/c_study/process$ ./process_2
Hello, Process! 5114
Hello, Process! 0
```

process_3.c

```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_3.c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main()
{

        pid_t pid;
        pid = fork();
        if (pid > 0)
                wait(NULL);
        printf("Hello, Process! %d\n", pid);

        return 0;
}

hoon@ubuntu:~/Desktop/c_study/process$ ./process_3
Hello, Process! 0
Hello, Process! 5118
```


```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_3.c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main()
{

        pid_t pid;
        pid = fork();
        if (pid > 0)
                wait(NULL);
        printf("Hello, Process! %d\n", pid);

        return 0;
}

hoon@ubuntu:~/Desktop/c_study/process$ ./process_3
Hello, Process! 0
Hello, Process! 5118
```

process_4.c

```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_4.c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int value = 5;

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

        if (pid == 0) // child process
        {
                value += 15;
                return 0;
        }
        else if (pid > 0)
        {
                wait(0);
                printf("Parent: value = %d\n", value); // output?

        }
}
hoon@ubuntu:~/Desktop/c_study/process$ ./process_4
Parent: value = 5
```


```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_4.c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int value = 5;

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

        if (pid == 0) // child process
        {
                value += 15;
                return 0;
        }
        else if (pid > 0)
        {
                wait(0);
                printf("Parent: value = %d\n", value); // output?

        }
}
hoon@ubuntu:~/Desktop/c_study/process$ ./process_4
Parent: value = 5
```

process_5.c

```
8   {
9       int i;
10      pid_t pid;
11
12      for (i = 0; i < 4; i++)
13          pid = fork();
14
15      return 0;
16  }
17
```

PROBLEMS 출력 DEBUG CONSOLE 터미널 1: bash

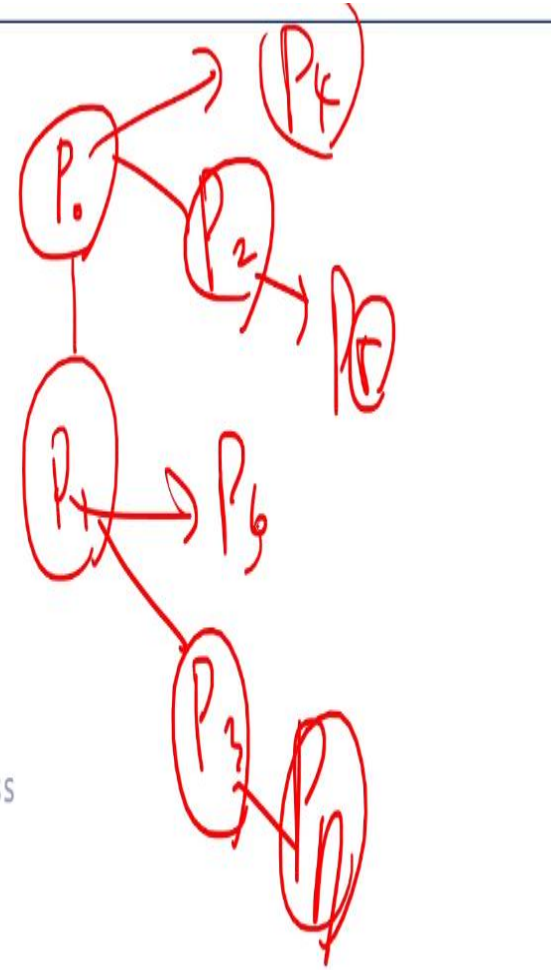
```
joonion@joonionpc:~/VSCode/OperatingSystemConcepts$ gcc 3.31_process_creation5.c
joonion@joonionpc:~/VSCode/OperatingSystemConcepts$ ./a.out
Hello, fork()!
Hello, fork()!
Hello, fork()!
Hello, fork()!
Hello, fork()!
Hello, fork()!
Hello, fork()!
```

Exercise 3.2 (p. 154)

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

/*
 * How many processes are created?
 */
int main()
{
    fork(); // fork a child process
    fork(); // fork another child process
    fork(); // and fork another

    return 0;
}
```



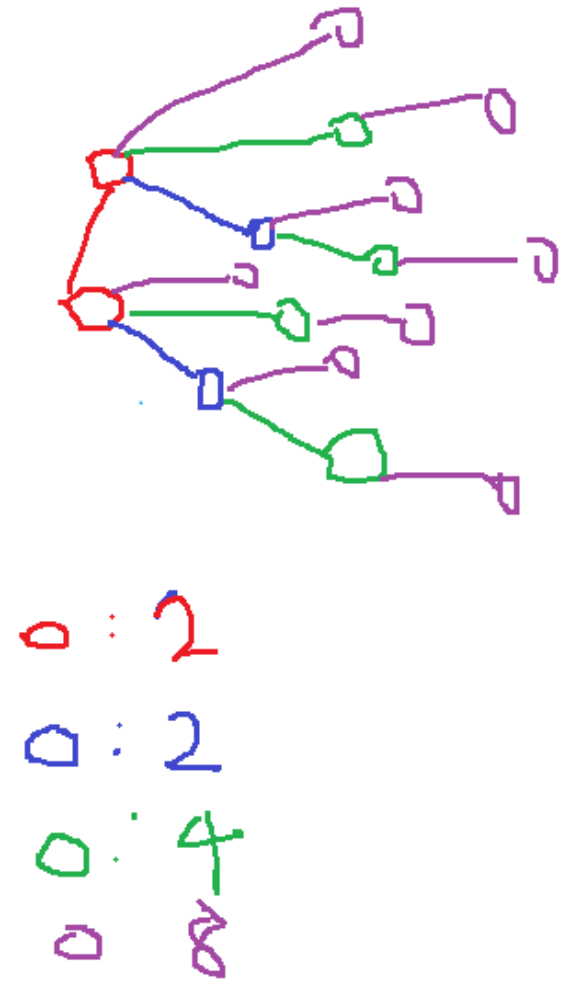
process_6.c

```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_6.c
#include <stdio.h>
#include <unistd.h>

/*
 * How many processes are created?
 */

int main()
{
        int i;

        for (i=0; i < 4; i++)
                fork();
        return 0;
}
hoon@ubuntu:~/Desktop/c_study/process$ ./process_6
```



process_7.c

```
hoon@ubuntu:~/Desktop/c_study/process$ cat process_7.c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

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

        if (pid == 0) // child process
        {
                execlp("/bin/ls", "ls", NULL);
                printf("LINE J\n");
        }
        else if (pid > 0) // parent process
        {
                wait(NULL);
                printf("Child Complete\n");
        }
        return 0;
}
hoon@ubuntu:~/Desktop/c_study/process$ ./process_7
process_1    process_2.c  process_4    process_5.c  process_7    process_8.c
process_1.c  process_3    process_4.c  process_6    process_7.c  process_9
process_2    process_3.c  process_5    process_6.c  process_8    process_9.c
Child Complete
```

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

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

        if (pid == 0) // child process
        {
                printf("LINE A\n");
                printf("LINE B\n");
                printf("LINE C\n");
        }
        else if (pid > 0) // parent process
        {
                wait(NULL);
                printf("Child Complete\n");
        }
        return 0;
}

hoon@ubuntu:~/Desktop/c_study/process$ ./process_7_1
LINE A
LINE B
LINE C
Child Complete
```

process_8.c

```c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

int main()
{
        pid_t pid, pid1;
        pid = fork();
        if (pid == 0)  // child precess
        {
                pid1 = getpid();
                printf("child: pid = %d\n", pid);
                printf("child: pid1 = %d\n", pid1);
        }
        else if (pid > 0) // parent process
        {
                pid1 = getpid();
                printf("parent pid = %d\n", pid);
                printf("parent pid1 = %d\n", pid1);
                wait(NULL);
        }
        return 0;
}
```

```
hoon@ubuntu:~/Desktop/c_study/process$ ./process_8
parent pid = 3281
parent pid1 = 3280
child: pid = 0
child: pid1 = 3281
```

child process에서 자기 pid값을 알기 위해
pid1 = getpid()

// child process
pid는 0을 리턴받은 것
pid1은 자기 pid값(자식)

// parent process
Pid는 자식 프로세스의 pid
Pid1은 부모 프로세스의 pid

process_9.c

```c
#include <stdio.h>
#include <unistd.h>
#include <wait.h>

#define SIZE 5
int nums[SIZE] = {0,1,2,3,4};

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

        if (pid == 0) // child process
        {
                for(i = 0; i < SIZE; i++)
                {
                        nums[i] *= i;
                        printf("Child: %d \n", nums[i]); // LINE X
                }
        }
        else if (pid > 0) // parent process
        {
                wait(NULL);
                for (i = 0; i < SIZE; i++)
                {
                        printf("Parent: %d \n", nums[i]); // LINE X
                }
        }
}
```

```
Child: 0
Child: 1
Child: 4
Child: 9
Child: 16
Parent: 0
Parent: 1
Parent: 2
Parent: 3
Parent: 4
```

안녕하세요, 교수님
먼저 좋은 강의 제공해 주셔서 정말 감사하다는 말씀 드립니다.
Q1. 새로운 프로그램을 process에 올리는 것도 fork() 라는 시스템 콜에 의해 진행 되는 것인가요?
예를 들어,
1. 인프런 영상을 플레이 중
2. 동시에 메모장에 오늘 배운 내용을 정리
여기서 2번이 실행 되기 위해서 시스템 내부적으로는 fork()라는 시스템콜이 있고, 복제된 프로세스위에 메모장과 관련된 데이터, 코드 등등이 덮어 써지게 되는 건가요?
Q2. 위의 말이 맞다면 최초의 복제 되상이 되는 parent 는 무엇인가요?

Q1 질문: 맞습니다.
fork() 시스템 콜은 리눅스 기준으로 질문한 것과 같이 동작합니다.
운영체제 커널이 먼저 fork()를 하고,
fork()를 한 프로세스 영역에 실행을 요청한 프로세스의 코드와 데이터 영역을 덮어 쓰는 것이죠.
Q2 질문:
fork()를 할 때는 자기 자신을 복제합니다.
메모장을 실행했을 때 fork()를 호출하는 프로세스를 복제하겠지요?
메모장 실행을 요청하는 것은 GUI의 마우스 클릭을 하겠지만,
이것은 커맨드창에서 memo.exe를 입력한 것과 동일합니다.
따라서, 리눅스라면 쉘 프로세스(sh, bash, zsh 등)가 parent 프로세스가 되어
메모장을 위한 child process를 생성하는 것이라 보면 됩니다.