[Operating System] System Call

---

fork( ), exec( ), wait( )와 같은 것들은 Process 생성과 제어를 위한 System call임.

- fork, exec는 새로운 Process 생성과 관련이 되어 있다.
- wait는 Process (Parent)가 만든 다른 Process(child) 가 끝날 때까지 기다리는 명령어임.

Fork

> 새로운 Process를 생성할 때 사용.
>
> 그러나, 이상한 방식임.

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

int main(int argc, char *argv[]) {
    printf("pid : %d", (int) getpid()); // pid : 29146

    int rc = fork();                    // 주목

    if (rc < 0) {
        exit(1);
    }                                   // (1) fork 실패
    else if (rc == 0) {                 // (2) child 인 경우 (fork 값이 0)
        printf("child (pid : %d)", (int) getpid());
    }
    else {                              // (3) parent case
        printf("parent of %d (pid : %d)", rc, (int)getpid());
    }
}
```

> pid : 29146
>
> parent of 29147 (pid : 29146)
>
> child (pid : 29147)

을 출력함 (parent와 child의 순서는 non-deterministic함. 즉, 확신할 수 없음. scheduler가 결정하는 일임.)

[해석]

PID : 프로세스 식별자. UNIX 시스템에서는 PID는 프로세스에게 명령을 할 때 사용함.

Fork()가 실행되는 순간. 프로세스가 하나 더 생기는데, 이 때 생긴 프로세스(Child)는 fork를 만든 프로세스(Parent)와 (almost) 동일한 복사본을 갖게 된다. **이 때 OS는 위와 똑같은 2개의 프로그램이 동작한다고 생각하**

고, fork()가 return될 차례라고 생각한다. 그 때문에 새로 생성된 Process (child)는 main에서 시작하지 않고, if 문부터 시작하게 된다.

그러나, 차이점이 있었다. 바로 child와 parent의 fork() 값이 다르다는 점이다. 따라서, 완전히 동일한 복사본이라 할 수 없다.

> Parent의 fork()값 => child의 pid 값
>
> Child의 fork()값 => 0

Parent와 child의 fork 값이 다르다는 점은 매우 유용한 방식이다.

그러나! Scheduler가 부모를 먼저 수행할지 아닐지 확신할 수 없다. 따라서 아래와 같이 출력될 수 있다.

> pid : 29146
>
> child (pid : 29147)
>
> parent of 29147 (pid : 29146)

---

wait

> child 프로세스가 종료될 때까지 기다리는 작업

위의 예시에 int wc = wait(NULL)만 추가함.

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

int main(int argc, char *argv[]) {
    printf("pid : %d", (int) getpid()); // pid : 29146

    int rc = fork();                    // 주목

    if (rc < 0) {
        exit(1);
    }                                   // (1) fork 실패
    else if (rc == 0) {                 // (2) child 인 경우 (fork 값이 0)
        printf("child (pid : %d)", (int) getpid());
    }
    else {                              // (3) parent case
        int wc = wait(NULL)             // 추가된 부분
        printf("parent of %d (wc : %d / pid : %d)", wc, rc, (int)getpid());
    }
}
```

> pid : 29146
>
> child (pid : 29147)

> parent of 29147 (wc : 29147 / pid : 29146)

wait를 통해서, child의 실행이 끝날 때까지 기다려줌. parent가 먼저 실행되더라도, wait ()는 child가 끝나기 전에는 return하지 않으므로, 반드시 child가 먼저 실행됨.

---

exec

단순 fork는 동일한 프로세스의 내용을 여러 번 동작할 때 사용함.

child에서는 parent와 다른 동작을 하고 싶을 때는 exec를 사용할 수 있음.

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

int main(int argc, char *argv[]) {
    printf("pid : %d", (int) getpid()); // pid : 29146

    int rc = fork();                    // 주목

    if (rc < 0) {
        exit(1);
    }                                   // (1) fork 실패
    else if (rc == 0) {                 // (2) child 인 경우 (fork 값이 0)
        printf("child (pid : %d)", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");       // 내가 실행할 파일 이름
        myargs[1] = strdup("p3.c");     // 실행할 파일에 넘겨줄 argument
        myargs[2] = NULL;               // end of array
        execvp(myarges[0], myargs);     // wc 파일 실행.
        printf("this shouldn't print out") // 실행되지 않음.
    }
    else {                              // (3) parent case
        int wc = wait(NULL)             // 추가된 부분
        printf("parent of %d (wc : %d / pid : %d)", wc, rc, (int)getpid());
    }
}
```

exec가 실행되면,

execvp(실행 파일, 전달 인자) 함수는, code segment 영역에 실행 파일의 코드를 읽어와서 덮어 씌운다.

씌운 이후에는, heap, stack, 다른 메모리 영역이 초기화되고, OS는 그냥 실행한다. 즉, 새로운 Process를 생성하지 않고, 현재 프로그램에 wc라는 파일을 실행한다. 그로인해서, execvp() 이후의 부분은 실행되지 않는다.