11. Process Management

Hyunchan, Park

http://oslab.jbnu.ac.kr

Division of Computer Science and Engineering

Jeonbuk National University

학습 내용

- Process Creation
- Program Execution
- Process Termination

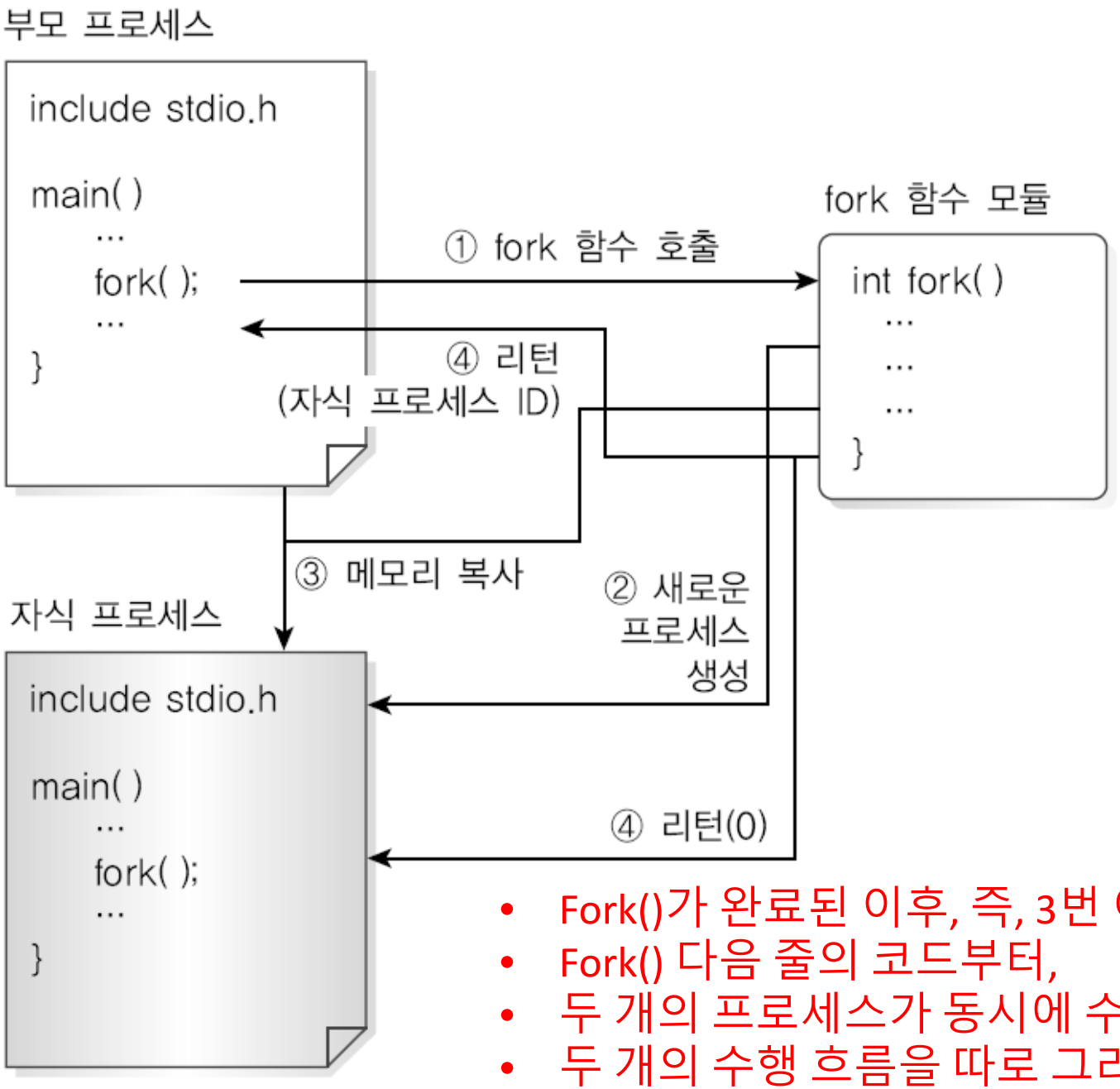
Process Creation

프로세스 생성: fork(2)

- 프로세스 생성: fork(2)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

  - 새로운 프로세스를 생성 : 자식 프로세스
  - fork 함수를 호출한 프로세스 : 부모 프로세스
  - 자식 프로세스는 부모 프로세스의 메모리 주소 공간을 그대로 복사
    - 메모리 공간이 똑같이 복제됨에 따라, 아래 자원들도 그대로 복사됨
    - RUID, EUID, RGID, EGID, 환경변수
    - 열린 파일기술자, 시그널 처리, setuid, setgid
    - 현재 작업 디렉토리, umask, 사용가능자원 제한



- Fork()가 완료된 이후, 즉, 3번 이후부터는,
- Fork() 다음 줄의 코드부터,
- 두 개의 프로세스가 동시에 수행되고 있음
- 두 개의 수행 흐름을 따로 그리면서 따라가야 함

[그림 6-1] fork 함수를 이용한 새로운 프로세스 생성

프로세스 생성: fork(2)

- 부모 프로세스와 다른 점
  - 자식 프로세스는 유일한 PID를 갖는다
  - 자식 프로세스는 유일한 PPID를 갖는다. (Parent PID)
  - 프로세스 잠금, 파일 잠금, 기타 메모리 잠금은 상속 안함
  - 자식 프로세스의 tms구조체 값은 0으로 설정 (수행 시간 관련 구조체)
- 부모 프로세스와 자식 프로세스는 열린 파일을 공유하므로 읽거나 쓸 때 주의해야 한다. (동기화 문제 발생 가능)

[예제 1] fork()

hw10 > C proc1.c

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

int main(void) {
    pid_t pid;
    /* fork another process */
    printf ("Parent: mypid = %d Fork!\n", getpid());

    pid = fork();

    if (pid < 0) { /* error occurred */
        perror("Fork Failed");
        return 1;
    } else if (pid == 0) { /* child process */
        sleep(1);
        printf("Child: I am your child! pid=%d\n", getpid());
        return 0;
    } else { /* parent process */
        sleep(1);
        printf ("Parent: I am your father! pid=%d\n", getpid());
        return 0;
    }
}
```

```
ubuntu@41983:~/hw10$ gcc -o proc1 proc1.c
ubuntu@41983:~/hw10$ ./proc1 & ps
[1] 3051776
Parent: mypid = 3051776 Fork!
    PID TTY          TIME CMD
3041613 pts/0    00:00:00 bash
3051776 pts/0    00:00:00 proc1
3051777 pts/0    00:00:00 ps
3051778 pts/0    00:00:00 proc1
ubuntu@41983:~/hw10$ Parent: I am your father! pid=3051776
Child: I am your child! pid=3051778

[1]+  Done                    ./proc1
```

프로세스 종료: exit(2)

- 프로세스 종료: exit(2)

```
#include <stdlib.h>
void exit(int status);
```

  - status : 종료 상태값
    - 부모 프로세스가 이 상태값을 보고,
      0이면 정상, 그 외 다른 값이면 비정상 종료되었다고 판단함.
  - Main() 함수에서 return 하는 것은 exit()와 동일한 효과
    - Exit()는 문맥 어디에서든 수행하여 프로세스를 종료할 수 있음
- 프로세스 종료시 수행할 작업 등록: atexit(2)

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

  - func : 종료 시, 수행할 작업을 지정한 함수명
  - 등록한 함수들이 등록 순서의 역순으로 수행됨
  - Main() 에서 return 할 때도 수행됨

프로세스 종료: exit(2)

- 프로그램 종료 함수의 일반적 종료 절차
  0. atexit() 로 등록된 함수들을 수행한다.
  1. 모든 파일 기술자를 닫는다.
  2. 부모 프로세스에 종료 상태를 알린다.
  3. 자식 프로세스들에 SIGHUP 시그널을 보낸다.
  4. 부모 프로세스에 SIGCHLD 시그널을 보낸다.
  5. 프로세스간 통신에 사용한 자원을 반납한다.

[예제2] exit() and atexit()

hw10 > C proc2.c

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

void cleanup1(void) {
    printf("Cleanup 1 is called.\n");
}
void cleanup2(void) {
    printf("Cleanup 2 is called.\n");
}

int main(void) {
    atexit(cleanup1);
    atexit(cleanup2);

    exit(0);
}
```

```
ubuntu@41983:~$ cd hw10
ubuntu@41983:~/hw10$ gcc -o proc2 proc2.c
ubuntu@41983:~/hw10$ ./proc2 2
Cleanup 2 is called.
Cleanup 1 is called.
ubuntu@41983:~/hw10$
```

Program Execution

다른 프로그램을 실행시키려면? system(3)

- 프로그램 실행 : system(3)

```
#include <stdlib.h>
int system(const char *string);
```

  - 새로운 프로그램을 실행하는 가장 간단한 방법
  - 실행할 프로그램명을 인자로 지정
    - 콘솔에 직접 입력하는 것과 마찬가지 효과를 냄
    - 예) system("ls -al");
  - 위험하고 비효율적이므로 남용하지 말 것
    - 환경 변수 해킹 등에 의해 전혀 다른 명령이 수행되어 보안 상 위험할 수 있음
    - https://www.joinc.co.kr/w/Site/system_programing/Unix_Env/secure_prog#AEN58

환경변수의 이해

- 환경변수
  - 프로세스가 실행되는 기본 환경을 설정하는 변수
  - 로그인명, 로그인 쉘, 터미널에 설정된 언어, 경로명 등
  - 환경변수는 “환경변수=값”의 형태로 구성되며 관례적으로 대문자로 사용
  - 현재 쉘의 환경 설정을 보려면 env 명령을 사용

```
# env
_=/usr/bin/env
LANG=ko
HZ=100
PATH=/usr/sbin:/usr/bin:/usr/local/bin:.
LOGNAME=jw
MAIL=/usr/mail/jw
SHELL=/bin/ksh
HOME=/export/home/jw
TERM=ansi
PWD=/export/home/jw/syspro/ch5
TZ=ROK
...
```

다른 프로그램의 실행: exec(3) series

- exec 함수군
  - exec로 시작하는 함수들로, 명령이나 실행 파일을 실행할 수 있다.
  - exec 함수가 실행되면 프로세스의 메모리 이미지는 해당 실행파일로 바뀐다.
- Exec 함수군의 형태 6가지

```
#include <unistd.h>
int execl(const char *path, const char *arg0, ..., const char *argn,
(char *)0);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ..., const char *argn,
(char *)0, char *const envp[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn,
(char *)0);
int execvp(const char *file, char *const argv[]);
```

  - Path or file : 실행할 명령의 파일 경로 지정
  - arg#, argv : main 함수에 전달할 인자 지정
  - envp : main 함수에 전달할 환경변수 지정 * 함수의 형태에 따라 NULL 값 지정에 유의!

Execlp() 함수 사용하기

```
01  #include <unistd.h>
02  #include <stdlib.h>
03  #include <stdio.h>
04
05  int main(void) {
06      printf("--> Before exec function\n");
07
08      if (execlp("ls", "ls", "-a", (char *)NULL) == -1) {
09          perror("execlp");
10          exit(1);
11      }
12
13      printf("--> After exec function\n");
14
15      return 0;
16  }
```

인자의 끝을 표시하는 NULL 포인터

첫 인자는 실행파일명 지정

메모리 이미지가 'ls' 명령으로 바뀌어 13행은 실행안됨

```
# ex6_4.out
--> Before exec function
.       ex6_1.c      ex6_3.c      ex6_4.out
..      ex6_2.c      ex6_4.c      han.txt
```

Execv() 함수 사용하기

```
01  #include <unistd.h>
02  #include <stdlib.h>
03  #include <stdio.h>
04
05  int main(void) {
06      char *argv[3];
07
08      printf("Before exec function\n");
09
10      argv[0] = "ls";
11      argv[1] = "-a";
12      argv[2] = NULL;
13      if (execv("/usr/bin/ls", argv) == -1) {
14          perror("execv");
15          exit(1);
16      }
17
18      printf("After exec function\n");
19
20      return 0;
21  }
```

첫 인자는 실행파일명 지정

인자의 끝을 표시하는 NULL 포인터

경로로 명령 지정

역시 실행안 됨

```
# ex6_5.out
--> Before exec function
.    ex6_1.c   ex6_3.c   ex6_5.c     han.txt
..   ex6_2.c   ex6_4.c   ex6_5.out
```

Execve() 함수 사용하기

```
...
05  int main(void) {
06      char *argv[3];
07      char *envp[2];
08
09      printf("Before exec function\n");
10
11      argv[0] = "arg.out";
12      argv[1] = "100";
13      argv[2] = NULL;
14
15      envp[0] = "MYENV=hanbit";
16      envp[1] = NULL;
17
18      if (execve("./arg.out", argv, envp) == -1) {
19          perror("execve");
20          exit(1);
21      }
22
23      printf("After exec function\n");
24
25      return 0;
26  }
```

실행파일명 지정

인자의 끝을 표시하는 NULL 포인터

환경변수 설정

ex6_6_arg.c를 컴파일하여 생성

```
# ex6_6.out
--> Before exec function
--> In ex6_6_arg.c Main
argc = 2
argv[0] = arg.out
argv[1] = 100
MYENV=hanbit
```

일반적인 exec() 사용 방법

- Fork()로 생성한 자식 프로세스에서 exec() 함수군을 호출
  - 자식 프로세스의 메모리 이미지가 부모 프로세스 이미지에서 exec() 함수로 호출한 새로운 명령으로 대체
    - 이를 이용해 자식 프로세스는 부모 프로세스와 다른 프로그램을 실행
    - Exec() 수행이 완료된 이후, 해당 코드 이하의 내용은 해당 프로세스에서는 삭제되므로 실행되지 않음
  - 부모 프로세스와 자식 프로세스가 각기 다른 작업을 수행해야 할 때 fork() 와 exec() 함수를 함께 사용

[예제 3] fork()와 execlp()

hw10 > C proc3.c

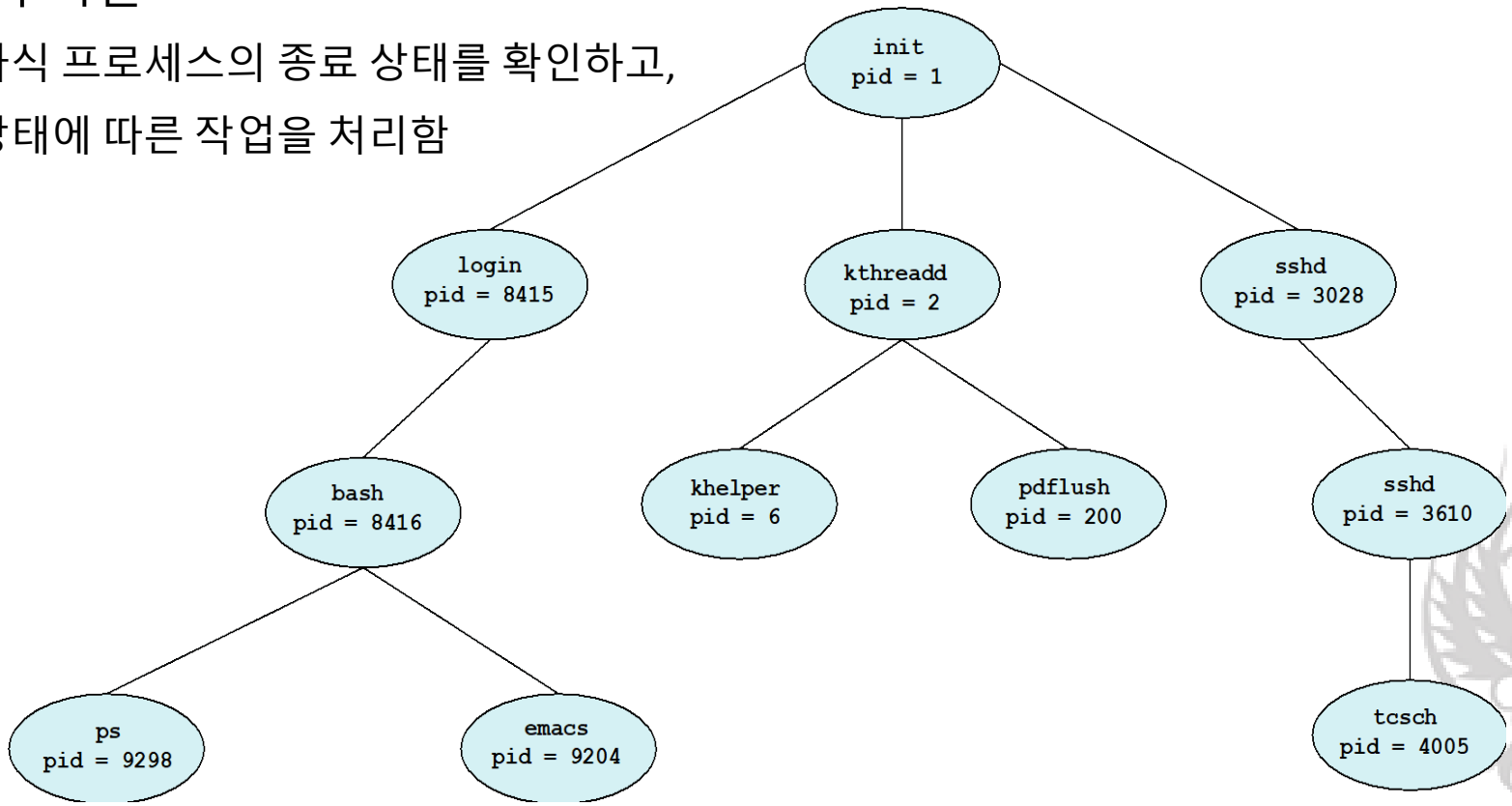
```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void) {
    pid_t pid;
    /* fork another process */
    printf ("Parent: mypid = %d Fork!\n", getpid());

    pid = fork();

    if (pid < 0) { /* error occurred */
        perror("Fork Failed");
        return 1;
    } else if (pid == 0) { /* child process */
        printf("Child: I am your child! pid=%d\n", getpid());

        if (execlp("/usr/bin/cat", "cat", "unix.txt", (char *)NULL) == -1) {
            perror("execlp");
            exit(1);
        }

        printf ("Child: This message is never shown!!\n");

        return 0;
    } else { /* parent process */
        printf ("Parent: I am your father! pid=%d\n", getpid());
        return 0;
    }
}
```

```
ubuntu@41983:~/hw10$ gcc -o proc3 proc3.c
ubuntu@41983:~/hw10$ ./proc3
Parent: mypid = 511292 Fork!
Parent: I am your father! pid=511292
Child: I am your child! pid=511293
ubuntu@41983:~/hw10$ hello world
```

Process Termination

프로세스 트리

- 프로세스는 항상 부모로부터 태어남
  - 즉, 모든 프로세스는 가족으로 family tree 가 형성됨
- 부모의 역할
  - 자식 프로세스의 종료 상태를 확인하고,
  - 상태에 따른 작업을 처리함



부모의 역할: 자식의 종료 상태 확인

- 예) 파일 전송을 수행하는 서버 프로세스
  - 각 사용자의 요청에 따라 실제 파일 전송을 담당하는 child process 생성
  - 만약 child 가 파일 전송에 실패했다고 하면?
  - 에러 코드에 따라 문제 상황을 해결하고,
    - 에러 코드: main() 함수의 return 값. 혹은 exit() 로 전달되는 값
    - 이걸 어떻게 전달 받을 수 있을까? -> wait() 시스템콜
  - 다른 child 를 만들어 파일 전송을 다시 시도함

  - 만약 에러 코드를 전달하지 못하고, (아무도 에러 코드를 받아주지 않고) 프로세스가 종료되면?
    - Zombie 프로세스!
      - 프로세스는 종료되었지만, PID 등의 자원이 반환되지 않음
- 따라서 parent는 항상 wait() 를 이용해 child process의 종료를 확인하여야 함

자식 프로세스 상태 대기: wait(3)

- wait(3)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *wstatus);
```

  - 부모 프로세스는 wait() 함수에서 멈춘 상태로,
  - 프로세스의 상태 변화를 대기: 종료, (시그널에 의한) 중단, 재개 등
    - Wait() 함수를 호출 전에 child가 이미 종료했다면 wait() 함수는 호출 즉시 리턴
  - wstatus : 자식 프로세스가 전달한 상태 정보를 저장할 주소
  - wait 함수의 리턴값은 자식 프로세스의 PID
  - 리턴값이 -1이면 살아있는 자식 프로세스가 하나도 없다는 의미

[예제 4] fork(), exec(), and wait() : complete set!

hw10 > C proc4.c

```c

int main(void) {
    pid_t pid;
    int wstatus;
    /* fork another process */
    printf ("Parent: mypid = %d Fork!\n", getpid());

    pid = fork();

    if (pid < 0) { /* error occurred */
        perror("Fork Failed");
        return 1;
    } else if (pid == 0) { /* child process */
        printf("Child: I am your child! pid=%d\n", getpid());

        if (execlp("/usr/bin/cat", "cat", "unix.txt", (char *
            perror("execlp");
            exit(1);
        }

        printf ("Child: This message is never shown!!\n");

        return 0;
    } else { /* parent process */
        printf ("Parent: I am waiting my child!\n");
        wait(&wstatus);
        printf ("Parent: Child says %d\n", wstatus);

        return 0;
    }
}
```

```
ubuntu@41983:~/hw10$ gcc -o proc4 proc4.c
ubuntu@41983:~/hw10$ ./proc4
Parent: mypid = 528574 Fork!
Parent: I am waiting my child!
Child: I am your child! pid=528575
hello world
Parent: Child says 0
ubuntu@41983:~/hw10$ rm unix.txt
ubuntu@41983:~/hw10$ ./proc4
Parent: mypid = 528757 Fork!
Parent: I am waiting my child!
Child: I am your child! pid=528758
cat: unix.txt: No such file or directory
Parent: Child says 256
```

부모-자식 프로세스의 관리

- 부모 프로세스와 자식 프로세스의 종료 절차
  - 부모, 자식 프로세스는 순서와 상관없이 실행하고, 먼저 실행을 마친 프로세스는 종료
  - wait() 이 제대로 진행되지 않는 상황에 따라 좀비 or 고아 프로세스 발생
- 좀비 프로세스
  - 실행을 종료한 자식 프로세스의 종료 상태를 부모 프로세스가 가져가지 않는 경우
  - 좀비 프로세스는 프로세스 테이블에만 존재 (제한된 슬롯: PID는 모두 65536개)
    - 웹서버 등 사용자 요청에 따라 많은 프로세스를 생성해 사용하는 경우, 문제가 될 수 있음
  - 좀비 프로세스는 일반적인 제거 방법은 없음
- 고아 프로세스 (orphan process)
  - 자식 프로세스보다 부모 프로세스가 먼저 종료할 경우,
    자식 프로세스들은 고아 프로세스가 됨
  - 이를 해결하기 위해, 고아 프로세스는 1번 프로세스(init)의 자식 프로세스로 등록
    - Init 프로세스가 wait() 를 호출해서 정상 종료시켜 줌