

# 9장 프로세스 제어

## 9.1 프로세스 생성

# 프로세스 생성

---

- fork() 시스템 호출
  - 부모 프로세스를 똑같이 복제하여 새로운 자식 프로세스를 생성
  - 새로운 프로세스를 생성하는 유일한 방법
  - 자식 프로세스, 부모 프로세스
  - 자기복제 : 코드, 데이터, 스택, 힙 등을 똑같이 복제

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

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

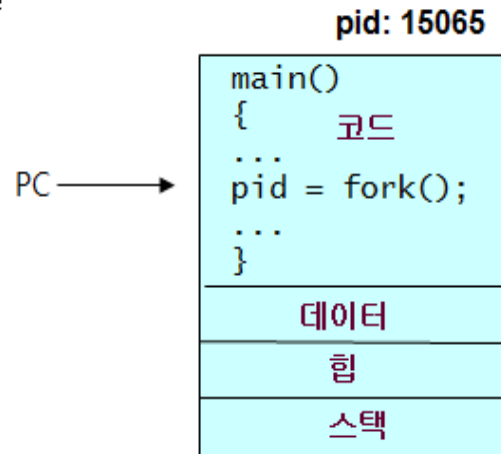
```
pid_t fork(void);
```

새로운 자식 프로세스를 생성한다. 자식 프로세스에게는 0을 리턴하고 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.

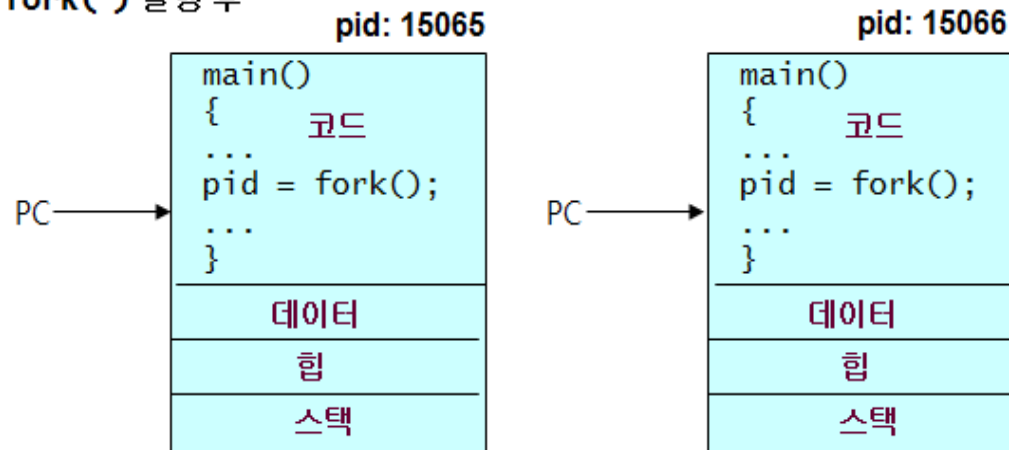
(두 번 반환됨에 주의)

# 프로세스 생성

fork( ) 실행 전



fork( ) 실행 후



PID만 다를 뿐, PC까지도 동일하게 생성됨

# 프로세스 생성

---

- `fork()`는 한 번 호출되면 두 번 리턴한다.
  - 호출 전에는 프로세스가 하나이지만, 호출 후에는 프로세서가 둘이기 때문
  - 자식 프로세스에게는 0을 리턴하고
  - 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.
  - 부모 프로세스와 자식 프로세스는 병행적으로 각각 실행을 계속
    - 이들의 실행 순서는 CPU 스케줄러가 결정

# fork1.c

---

```
#include <stdio.h>
#include <unistd.h> // fork & exec 호출을 위한 헤더파일
/* 자식 프로세스를 생성한다. */
int main()
{
    int pid;
    printf("[%d] 프로세스 시작 %d\n", getpid());
    pid = fork();
    printf("[%d] 프로세스 : 반환값 %d\n", getpid(), pid);
}
```

-> p	p
-> f	f
-> p	-> p

## 실행결과

결과가 총 몇 문장으로 찍혀야 하는가? 3문장  
그 이유는?

```
[15065] 프로세스 시작
[15065] 프로세스 : 반환값 15066
[15066] 프로세스 : 반환값 0
```

# 부모 프로세스와 자식 프로세스 구분

---

- `fork()` 호출 후에 리턴값이 다르므로 이 리턴값을 이용하여 부모 프로세스와 자식 프로세스를 구별하고 서로 다른 일을 하도록 할 수 있다.

```
pid = fork();
```

부모는 자식 생성

```
if ( pid == 0 )
```

```
{ 자식 프로세스의 실행 코드 }
```

```
else
```

```
{ 부모 프로세스의 실행 코드 }
```

# fork2.c

[Parent] : Hello, world pid=15065

[Child] : Hello, world pid=15066

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
/* 부모 프로세스가 자식 프로세스를 생성하고 서로 다른 메시지를 프린트 */
int main()
{
    int pid;
    pid = fork();
    if (pid == 0) { // 자식 프로세스
        printf("[Child] : Hello, world pid=%d\n", getpid());
    }
    else { // 부모 프로세스
        printf("[Parent] : Hello, world pid=%d\n", getpid());
    }
}
```

If문과 else문이 모두 출력된다.  
그 이유는?



# fork3.c: 두 개의 자식 프로세스 생성

부모는 두 자식 프로세스를 생성하는 기능만 수행하고 종료  
두 자식들은 각각 해당 문장을 출력 후 종료

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
/* 하나의 부모 프로세스가 두 개의 자식 프로세스를 생성한다. */
int main()
{
    int pid1, pid2;
    pid1 = fork();
    if (pid1 == 0) {    //첫번째 자식만 실행, 부모는 실행부분 없음
        printf("[Child 1] : Hello, world ! pid=%d\n", getpid());
        exit(0);        //첫번째 자식 종료
    }
    pid2 = fork();    //부모의 두번째 자식 생성
    if (pid2 == 0) {
        printf("[Child 2] : Hello, world ! pid=%d\n", getpid());
        exit(0);        //두번째 자식 종료
    }
}
```

# 프로세스 기다리기: wait()

---

- 자식 프로세스 중의 하나가 끝날 때까지 기다린다.
  - 실행이 끝난 자식 프로세스의 종료 코드(가령, `exit(0)` or `exit(1)` 등)를 `status`에 저장하고, 해당 자식 프로세스의 PID를 리턴한다.

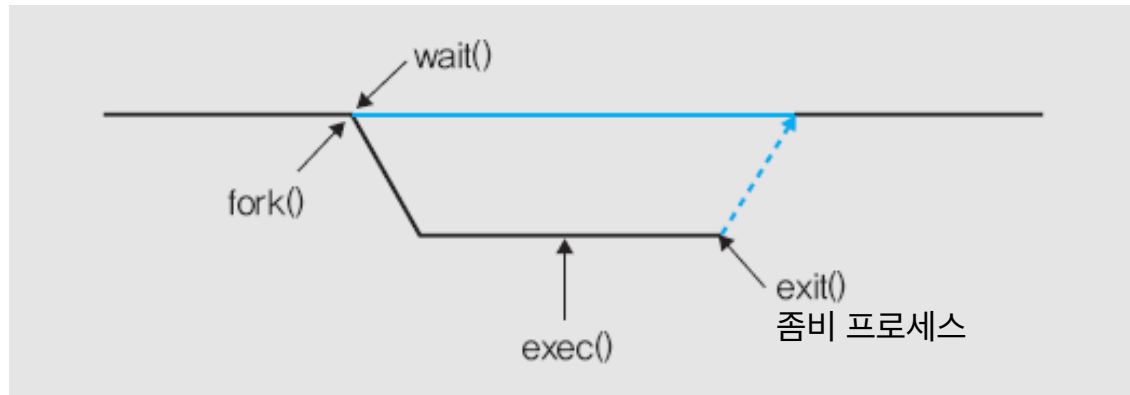
```
#include <sys/types.h>
```

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

```
pid_t wait(int *status); //자식 프로세스의 종료 대기
```

```
pid_t waitpid(pid_t pid, int *status, int options); //특정 pid 프로세스의 종료 대기  
waitpid의 세번째 옵션은 주로 0
```

# 프로세스 기다리기: wait()



프로세서의 생성 및 기다리는 과정

(참고)

종료 코드 값은 status 변수 값 4 바이트 가운데, 3번째 바이트에 저장  
따라서 우측 shift 8 bit 수행 후 출력



# forkwait.c

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

```
/* 부모 프로세스가 자식 프로세스를 생성하고 끝나기를 기다린다. */
```

```
int main()
```

```
{
```

```
    int pid, child, status;          // 정수형 변수 status (4바이트값)
```

```
    printf("[%d] 부모 프로세스 시작 \n", getpid( ));
```

16진수 0x0100 = status

```
    pid = fork();
```

2진수 0000 0001 0000 0000  
-> 0000 0000 0000 0001

```
    if (pid == 0) {
```

```
        printf("[%d] 자식 프로세스 시작 \n", getpid( ));
```

16진수 0x0001

10진수 1

```
        exit(1);                      // 종료 코드 1이 wait 함수 인자(여기서는 status)에 저장
```

```
    }
```

```
    child = wait(&status); // 자식 프로세스가 끝나기를 기다리고 자식 PID 반환
```

```
    printf("[%d] 자식 프로세스 %d 종료 \n", getpid(), child); //부모가 수행
```

```
    printf("\t종료 코드 %d\n", status >> 8);
```

특정 프로세스 기다리기 :

wait( ) 함수 대신 waitpid( ) 함수 사용

# waitpid.c

```
1  #include <sys/types.h>      #include <sys/wait.h>
                                   #include <sys/types.h>
                                   #include <unistd.h>
                                   #include <stdlib.h>
                                   #include <stdio.h>
...
7  /* 부모 프로세스가 특정 자식 프로세스를 생성하고 끝나기를 기다린다. */
8  int main()
9  {
10     int pid1, pid2, child, status;
11
12     printf("[%d] 부모 프로세스 시작 %n", getpid( ));
13     pid1 = fork();    //첫번째 프로세스 생성
14     if (pid1 == 0) {
15         printf("[%d] 자식 프로세스[1] 시작 %n", getpid( ));
16         sleep(1);
17         printf("[%d] 자식 프로세스[1] 종료 %n", getpid( ));
18         exit(1);
19     }
```

# waitpid.c

---

```
20
21     pid2 = fork();    //두번째 프로세스 생성
22     if (pid2 == 0) {
23         printf("[%d] 자식 프로세스 #2 시작 \n", getpid( ));
24         sleep(2);
25         printf("[%d] 자식 프로세스 #2 종료 \n", getpid( ));
26         exit(2);
27     }
28     // 자식 프로세스 #1의 종료를 기다린다.
29     child = waitpid(pid1, &status, 0);
30     printf("[%d] 자식 프로세스 #1 %d 종료 \n", getpid( ), child);
31     printf("\t종료 코드 %d\n", status >> 8);
32 }
```

waitpid()의 세번째 옵션은 보통 0을 사용,  
아무 실행 없이 자식 종료시 까지 대기

참고) 두번째 프로세스는 종료하였으나,  
부모 프로세스가 대기하지 않은 상태 → 좀비 프로세스

# waitpid.c 코드 분석

---

- 첫번째 프로세스는 1초 수면, 두번째는 2초 수면
  - 따라서 첫번째 자식 프로세스가 먼저 종료 될 것임
- 부모는 waitpid() 함수를 통해 프로세스 1의 종료를 대기한 후, 프로세스 1이 종료하면 바로 실행 완료 후 종료, 그리고 추후 두번째 프로세스가 종료
- 두번째 프로세스는 종료하였으나, 두번째 프로세스는 부모 프로세스가 대기하지 않은 상태 → 좀비 프로세스 사례
  - 운영체제(init process, PID == 1)가 주기적으로 좀비 프로세스 처리

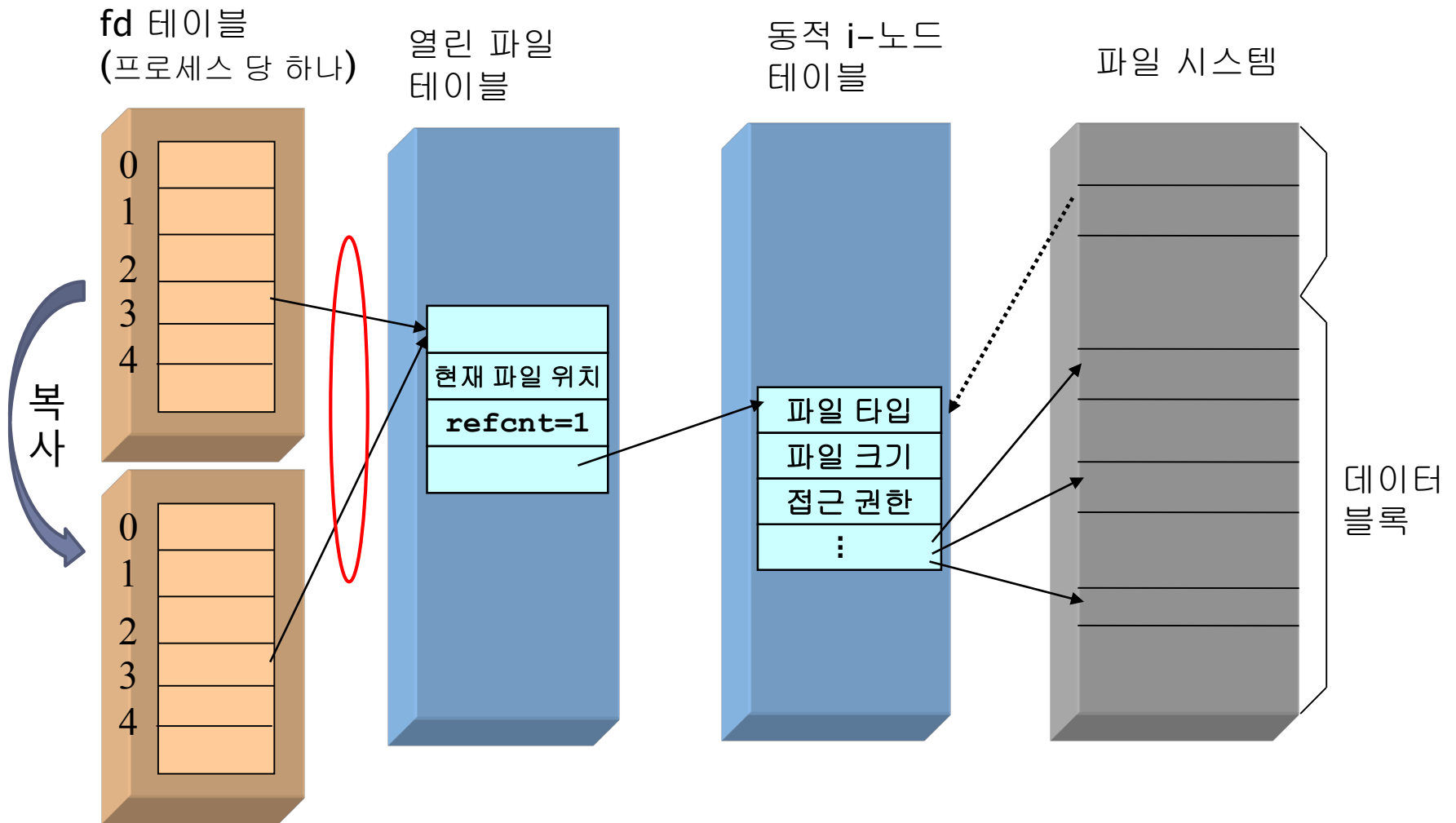
# fork() 후에 파일 공유

---

- 자식은 부모의 fd 테이블을 복사한다.
  - 부모와 자식이 **같은 파일 디스크립터**를 공유
  - **같은 파일 오프셋**을 공유
  - 따라서, 부모와 자식으로부터 출력이 서로 섞이게 됨
  - 즉, 일반적으로 상속되지 않는 몇몇 성질을 제외하고 부모와 자식은 fork( ) 후 모든 속성을 공유 함
  - Program counter, file offset 등 모든 속성을 공유
- 자식에게 상속되지 않는 성질
  - fork()의 반환값
  - 프로세스 ID
  - 파일 잠금 (lock) 속성 등



# fork()



## 9.2 프로그램 실행

# 프로그램 실행

---

- fork() 후
  - 자식 프로세스는 부모 프로세스와 똑같은 코드 실행
- 자식 프로세스에게 새 프로그램을 실행 시켜야 함
  - exec() 시스템 호출 사용
  - 프로세스 내의 프로그램을 새 프로그램으로 대체
  - 프로세스 내에서 새로운 프로그램을 실행시키는 유일한 방법
- 보통 fork() 후에 exec()를 실행 함      부모와 자식 프로세스 다름

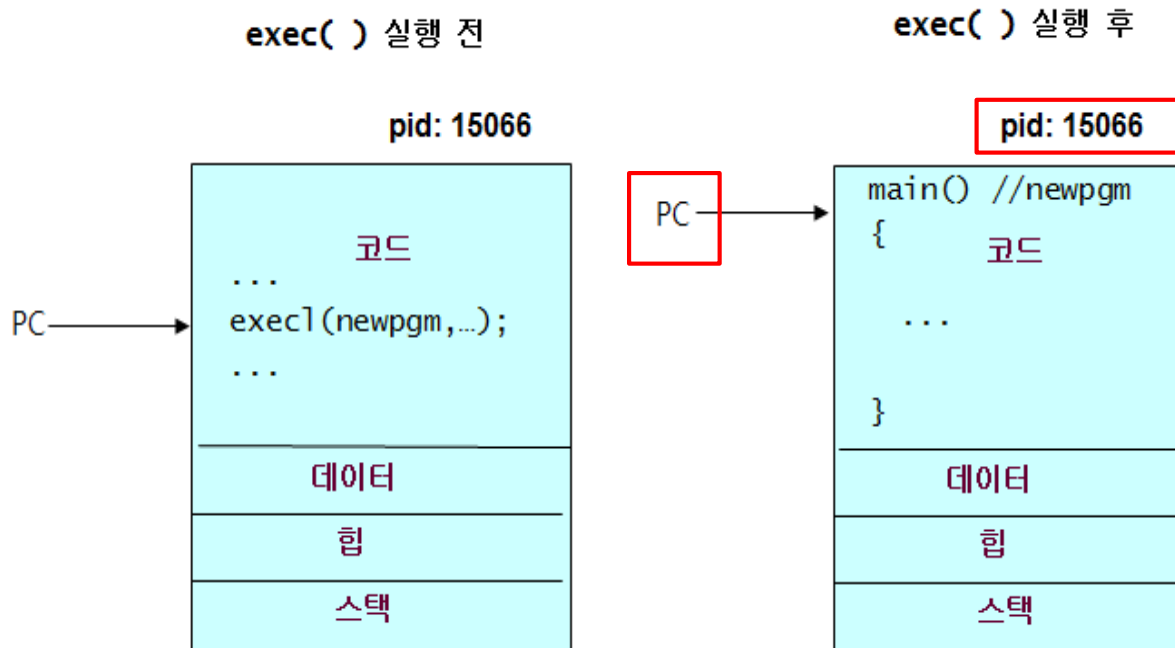
fork를 안한 후 exec 실행

-> 현재 프로세스가 새로운 프로그램으로 완전히 대체

현재 실행 중인 프로그램은 종료, 그 자리를 exec를 호출된 프로그램이 차지

# 프로그램 실행: exec()

- 프로세스가 `exec()` 호출을 하면,
  - 그 프로세스 내의 프로그램은 완전히 새로운 프로그램으로 대체
  - 자기 대체 (self-substitution)
  - 새 프로그램의 `main()`부터 (즉, 처음부터) 실행이 시작된다.



# 프로그램 실행: exec()

- exec() 호출이 성공하면 리턴할 곳이 없어진다. 그 이유는?
  - 즉, 성공한 exec() 호출은 절대 리턴하지 않는다.
  - 실패한 호출만 반환한다.

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

```
int execl(char* path, char* arg0, char* arg1, ... , char* argn, NULL)
```

```
int execv(char* path, char* argv[ ])
```

```
int execlp(char* file, char* arg0, char* arg1, ... , char* argn, NULL)
```

```
int execvp(char* file, char* argv[ ])
```

호출한 프로세스의 코드, 데이터, 힙, 스택 등을 path(혹은 file)가 나타내는 새로운 프로그램으로 대치한 후 새 프로그램을 실행한다.

성공한 exec( ) 호출은 리턴하지 않으며 실패하면 -1을 리턴한다.

Execl : 명령어 경로 + 명령어 + 명령줄 모든 인자를 하나씩 나열 (arg0: 명령어, + arg1 ... 인자) + NULL

Execv : 명령어 경로 + 명령줄 인자를 포인터 배열로

Execlp/execvp : path 대신, 실행할 프로그램 이름을 주고, 환경 변수 PATH에서 찾음

## 프로그램 13.4: 프로그램 실행

---

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 /* echo 명령어를 실행한다. */
5 int main( )
6 {
7     printf("시작\n");
8     execl("/bin/echo", "echo", "hello", NULL);
9     printf("exec 실패!\n");
10 }
```

execl("/bin/echo", "echo", "hello", "hi", NULL)  
-> hello hi

### 실행결과

시작

hello

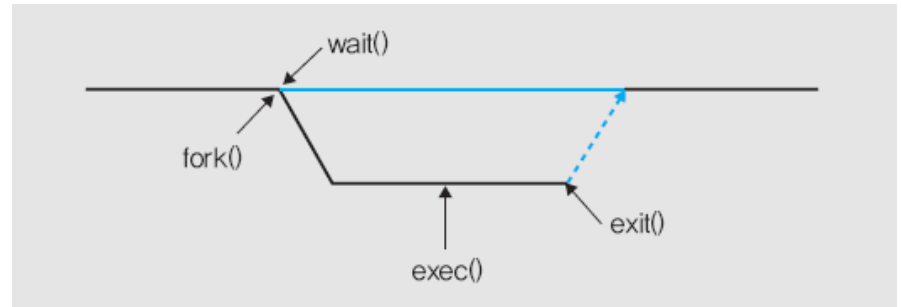
순차 실행에 있어서

마지막 printf 문장은 왜 출력되지 않나?

# fork/exec

- 보통 fork() 호출 후에 exec() 호출
  - 새로 실행할 프로그램에 대한 정보를 exec( )의 arguments로 전달한다
- exec() 호출이 성공하면
  - 자식 프로세스는 새로운 프로그램을 실행하게 되고
  - 부모는 계속해서 다음 코드를 실행하게 된다.

```
int pid, child, status;  
pid = fork();  
if (pid == 0 ) {  
    exec(arguments);  
    exit(1);  
} else {  
    child = wait(&status);  
}
```



# execute1.c

---

```
#include <stdio.h>
/* 자식 프로세스를 생성하여 echo 명령어를 실행한다. */
int main( )
{
    printf("부모 프로세스 시작\n");
    if (fork( ) == 0) {
        execl("/bin/echo", "echo", "Hello", NULL);
        fprintf(stderr,"첫 번째 실패");    //exec이 성공하면 이 줄은 결코 실행되지 않음
        exit(1);
    }
    printf("부모 프로세스 끝\n");    // 부모는 wait() 수행 없이 자식 생성 후, 바로 종료
}
```

```
$ execute1
부모 프로세스 시작
Hello
부모 프로세스 끝
```



# execute2.c

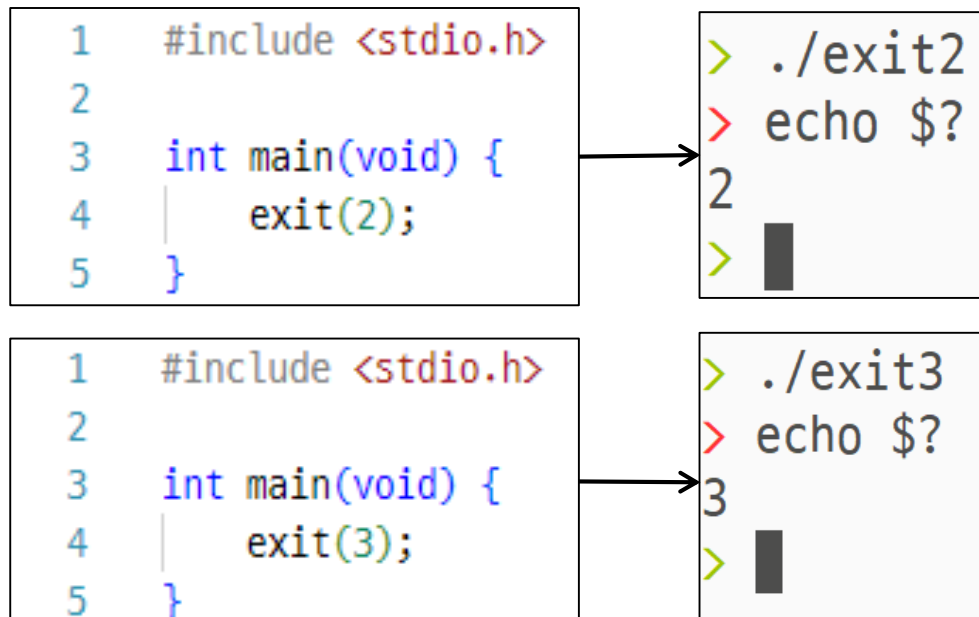
```
#include <stdio.h> ...
/* 세 개의 자식 프로세스를 생성하여 각각
   다른 명령어를 실행한다.*/
int main( )
{
    printf("부모 프로세스 시작\n");
    if (fork( ) == 0) {
        execl("/bin/echo", "echo", "hello", NULL);
        fprintf(stderr,"첫 번째 실패");
        exit(1);
    }
    if (fork( ) == 0) {
        execl("/bin/date", "date", NULL);
        fprintf(stderr,"두 번째 실패");
        exit(2);
    }
    if (fork( ) == 0) {
        execl("/bin/ls", "ls", "-l", NULL);
        fprintf(stderr,"세 번째 실패");
        exit(3);
    }
    printf("부모 프로세스 끝\n");
}
```

← 명령어가 인자가 없을 경우,  
Execl의 세번째 이후 해당 인자 생략 가능 (가변인자)

execl("/bin/date", NULL);  
-> error

# (참고) exit() 함수의 반환 값 확인 방법

- exit() 코드 앞에 적절한 출력 메시지 사용
  - e.g., printf("exit 1");
- \$? 변수 사용
  - \$? 변수 가장 최근에 종료한 프로세스의 종료 상태 저장하는 변수



## (응용) execute3.c

이전의 코드는 코드내에 정해진 특정 명령만 수행  
명령줄 인수로부터 받은 임의의 명령어가 실행가능한 코드로 변경

```
#include <stdio.h> ...
/* 명령줄 인수로 받은 새로운 임의의 명령을 실행시킨다. */
int main(int argc, char *argv[])
{
    int child, pid, status;
    pid = fork( );
    if (pid == 0) { // 자식 프로세스
        execvp(argv[1], &argv[1]);
        fprintf(stderr, "%s:실행 불가\n",argv[1]); //성공하면 이 문장은 결코 실행되지 않음
    } else { // 부모 프로세스
        child = wait(&status); //종료 코드를 status에 넣고, status와 자식 pid 반환
        printf("[%d] 자식 프로세스 %d 종료 \n", getpid(), pid);
        printf("\t종료 코드 %d \n", status>>8);
    }
}
```

실행 사례>

execute3 wc you.txt

wc 실행 결과...

[26470] 자식 프로세스 26471 종료

종료코드 0

# fork, exec, wait 일반적 구조

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4
5 /* 자식 프로세스를 생성하여 echo 명령어를 실행한다. */
6 int main( )
7 {
8     int pid, child, status;
9
10    printf("부모 프로세스 시작\n");
11    pid = fork();
12    if (pid == 0) {
13        exec1("/bin/echo", "echo", "hello", NULL);
14        fprintf(stderr, "첫 번째 실패");
15        exit(1);
16    }
17    else {
18        child = wait(&status);
19        printf("자식 프로세스 %d 끝\n", child);
20        printf("부모 프로세스 끝\n");
21    }
22 }
```

## 실행결과

부모 프로세스 시작

hello

자식 프로세스 15066 끝

부모 프로세스 끝

# system()

---

- Fork( ) 시스템 호출을 수행하고, exec( ) 시스템 호출을 연속하여 명령어를 실행 시키는 것은 상당히 번거로운 절차
- 이러한 과정을 자동으로 수행하는 C 라이브러리 함수 : system( )

# system() (fork/exec 과정을 수행하는 C 라이브러리 함수)

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

```
int system(const char *cmdstring);
```

이 함수는 /bin/sh -c cmdstring를 호출하여 **cmdstring**에 지정된 명령어를 실행하며, 명령어가 끝난 후, 명령어의 종료코드를 반환한다.

- 예시 : 자식 프로세스를 생성하고 /bin/sh로 하여금 **지정된 명령어**를 실행
  - system("**ls -asl**"); → 셸이 ls -asl을 수행토록 지시
- system( ) 함수 구현
  - fork( ), exec( ), waitpid( ) 시스템 호출을 순차적으로 이용
- 반환값
  - 명령어의 종료코드
  - -1 with errno: fork() 혹은 waitpid() 실패
  - 127 : exec() 실패 → \_exit(127);과 동일

# syscall.c (system ( ) 사용 사례)

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

- (1) Date 명령어 실행
- (2) 존재하지 않는 명령어 실행
- (3) Who 명령어 실행하고, 특정 종료코드 반환

```
int main()
{
```

WEXITSTATUS( ) : <sys/wait.h>에 정의된 매크로 함수  
내부적으로 표현된 종료코드 값을 찾아 반환

```
    int status;
    if ((status = system("date")) < 0)
        perror("system() 오류"); //오류 메시지 출력 함수
    printf("종료코드 %d\n", WEXITSTATUS(status));
```

```
    if ((status = system("hello")) < 0)           //fork() is ok, but no exec(hello) 해당 명령어 없음
        perror("system() 오류");                 //i.e., exec 실패, 종료 코드 127
    printf("종료코드 %d\n", WEXITSTATUS(status));
```

```
    if ((status = system("who; exit 44")) < 0)    // 명령 실행 후, 특정 종료 코드 지정 가능
        perror("system() 오류");
    printf("종료코드 %d\n", WEXITSTATUS(status));
```

```
}
```

# (참고) system() 함수의 내부 구현 모습

```
#include <sys/types.h> /* system.c */
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>
int system(const char *cmdstring)
{
    pid_t pid; int status;

    if (cmdstring == NULL) /* 명령어가 NULL인 경우 */
        return(1);
    if ( (pid = fork()) < 0) {
        status = -1; /* 프로세스 생성 실패 */
    } else if (pid == 0) { /* 자식 */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127); /* execl 실패 */      (_exit 뒷정리 없이 즉시 종료) : 오류 코드 127
    } else { /* 부모 */
        while (waitpid(pid, &status, 0) < 0) /* 종료코드를 status에, 자식 pid 수신)
            if (errno != EINTR) {
                status = -1; /* waitpid()로부터 EINTR외의 오류 */
                break;
            }
        }
    }
    return(status);
}
```



## 9.3 입출력 재지정

# 입출력 재지정

- 명령어의 표준 출력이 파일에 저장

\$ 명령어 > 파일

- 출력 재지정 기능 구현

```
fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
```

`dup2(fd, 1);` //파일을 open 후 그 fd를 1(stdout)에 덮어쓰는 → 덮어써서 1은 없어지고 fd만 남음

즉, 파일 디스크립터 fd를 표준출력(1)에 dup2()로 시스템 호출

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

```
int dup(int oldfd);
```

oldfd에 대한 복제본인 새로운 파일 디스크립터를 생성하여 반환하다.

```
int dup2(int oldfd, int newfd);
```

oldfd을 newfd에 복제하고 복제된 새로운 파일 디스크립터를 반환한다.

# redirect1.c

셸 상의 명령어 > 에 대한 내부적 구현 사례

```
1 #include <stdio.h>
2 #include <fcntl.h>
3 ...
4 /* 표준 출력을 파일에 재지정하는 프로그램 */
5 int main(int argc, char* argv[])
6 {
7     int fd, status;
8     fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0600);
9     dup2(fd, 1); /* oldfd fd를 newfd 표준출력에 복제, 즉 fd를 표준 출력 1번에 덮어 씌. 이제 표준 출력은 fd 파일에만 저장 */
10    close(fd);
11    printf("Hello stdout !\n");
12    fprintf(stderr, "Hello stderr !\n");
13 }
```

O\_TRUNC : 파일이 이미 있는 경우 내용을 지운다

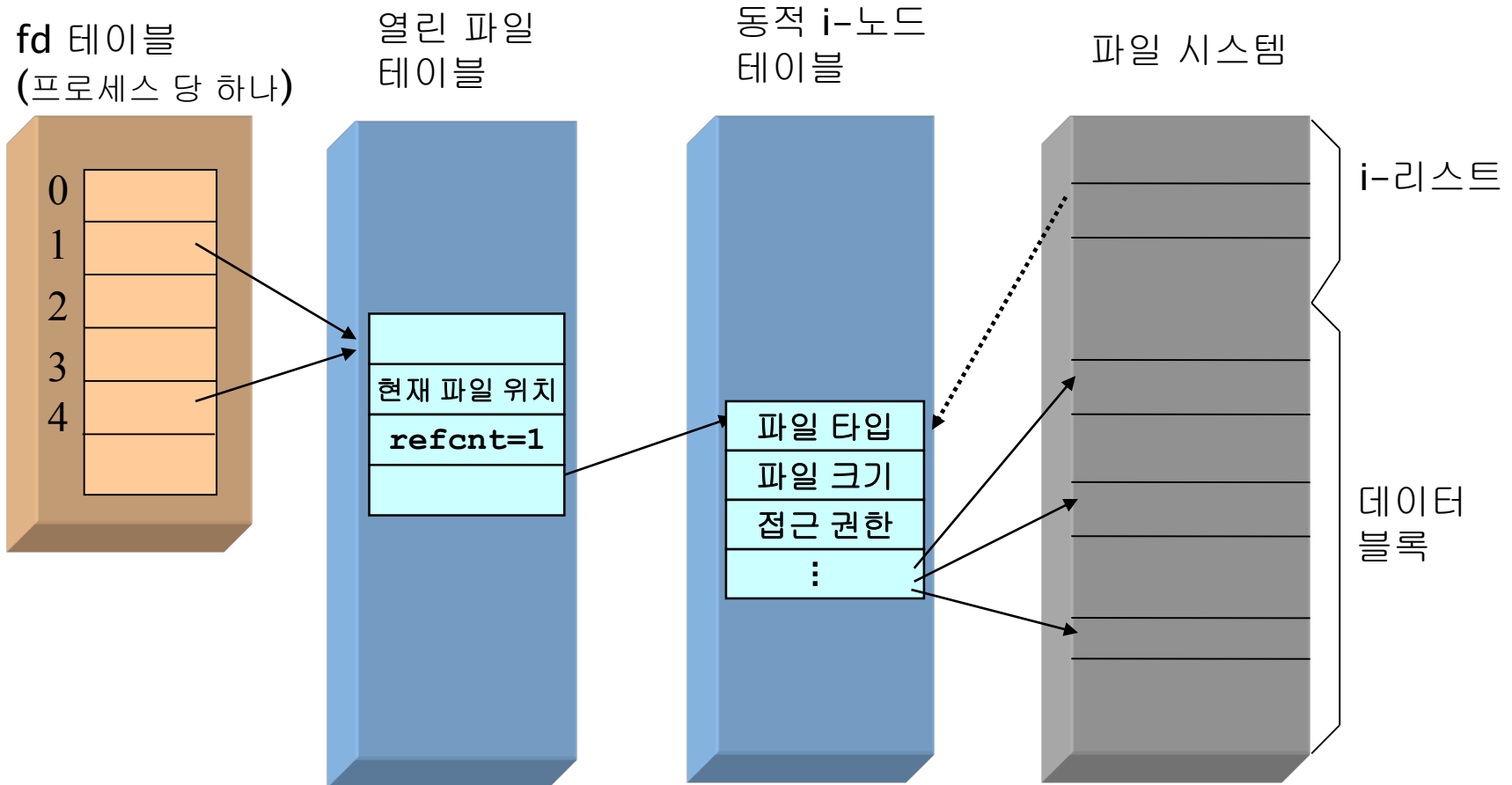
에러를 파일에 출력하고 싶으면  
dup2(fd, 2);

//fd 파일로 저장

//표준 오류는 따로 정의하지 않음  
표준 에러가 재지정되지 않았으므로 기본 출력 대상은 여전히 화면  
//따라서 모니터로 출력

표준 오류 출력  
fprintf : 파일로 출력,  
stderr 표준 오류 파일, 표준 오류 파일은 모니터 따라서 모니터로 출력

# dup2(fd, 1) 실행 결과

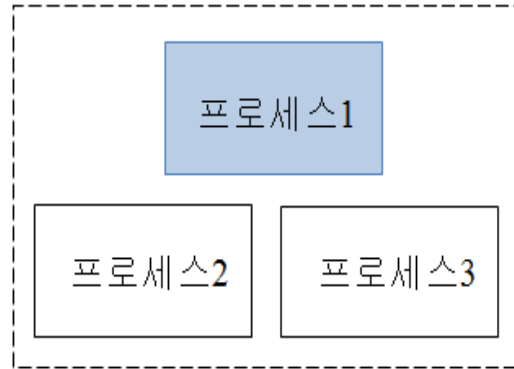


## 9.4 프로세스 그룹

# 프로세스 그룹

---

- 프로세스 그룹은 여러 프로세스들의 집합이다.
- 보통 부모 프로세스(그룹 리더)가 생성하는 자손 프로세스들은 부모와 같은 프로세스 그룹에 속한다.
- 프로세스 그룹 리더는 Process GID와 PID가 동일하다.



프로세스 그룹

# 프로세스 그룹 사용

- 프로세스 그룹 내의 모든 프로세스에 시그널을 보낼 때 사용
  - `$ kill -9 pid` 프로세스 ID(Process ID) 셸이 생성한 명령어는 같은 그룹임. 쉘도 같이 종료됨.
  - `$ kill -9 0` (pid 대신 **0**을 입력하면 현재 속한 **그룹내의 모든 프로세스** 종료)
  - `$ kill -9 -pid` (**음수로 표시된 pid는 gid**, 특정 그룹 모두에게 전달)
  -
- `pid_t waitpid(pid_t pid, int *status, int options);`
  - `pid == -1` : 임의의(any process) 아무 자식 프로세스가 종료하기를 기다린다
  - `pid > 0` : 특정 자식 프로세스 pid가 종료하기를 기다린다 (**일반적**).
  - `pid == 0` : 호출자와 **같은 그룹 내의 임의의** 자식 프로세스가 종료하기를 기다린다.
  - `pid < -1` : 음수는 gid, 즉 **특정 그룹 내의 임의의** 자식 프로세스가 종료하기를 기다린다. (참고, 그룹 리더의: `pid==gid`)

# 프로세스 그룹

---

- 프로세스가 가지는 두가지 ID
  - 프로세스 ID (PID)
  - 프로세스 그룹 ID (GID)
- 각 프로세스는 하나의 프로세스 그룹에 속함.
- 각 프로세스는 자신이 속한 프로세스 그룹 ID를 가지며 fork 시 같은 그룹을 물려받는다.

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpgrp(void); //get process group
호출한 프로세스의 프로세스 그룹 ID를 반환한다.
```



# 프로세스 그룹: pgrp1.c

---

```
#include <sys/types.h>
#include <unistd.h>
main()
{
    int pid, gid;
    printf("PARENT: PID = %d GID = %d\n", getpid(), getpgrp());
    pid = fork();
    if (pid == 0) { // 자식 프로세스
        printf("CHILD: PID = %d GID = %d\n", getpid(), getpgrp());
    }
}
```

```
PARENT:PID = 2130168 GID = 2130168
CHILD:PID = 2130169 GID = 2130168
```

PID는 다르고, GID는 부모 자식 사이에 동일하게 생성

# 프로세스 그룹

---

- 새로운 프로세스 그룹 생성
  - 새로운 프로세스 그룹을 생성하거나, 다른 그룹에 멤버로 참여
  - `int setpgid(pid_t pid, pid_t pgid);`
    - 다음 슬라이드 참조
- 프로세스 그룹 소멸
  - 그룹에 속한 마지막 프로세스가 종료하는 경우
  - 마지막 프로세스가 다른 그룹으로 조인하는 경우

# 프로세스 그룹

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

main()
{
    int pid, gid;
    printf("PARENT: PID = %d GID = %d \n", getpid(), getpgid());
    pid = fork();
    if (pid == 0){
        setpgid(getpid(), 0);
        printf("CHILD: PID = %d GID = %d\n", getpid(), getpgid());
    }
}
```

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

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

```
int setpgid(pid_t pid, pid_t pgid);
```

프로세스 pid의 프로세스 그룹 ID를 pgid로 설정한다.

성공하면 0을 실패하면 -1를 반환한다.

```
y@22312072@acslab-146:~/11$ ./pgrp2
PARENT: PID = 2157537 GID = 2157537
CHILD: PID = 2157538 GID = 2157538

main()
{
    int pid, gid;
    printf("PARENT: PID = %d GID = %d \n", getpid(), getpgid());
    pid = fork();
    if (pid == 0){
        setpgid(0, getpgid());
        printf("CHILD: PID = %d GID = %d\n", getpid(), getpgid());
    }
}
```

```
y@22312072@acslab-146:~/11$ ./pgrp2
PARENT: PID = 2165880 GID = 2165880
CHILD: PID = 2165881 GID = 2165880
```

- 새로운 프로세스 그룹을 생성하거나 다른 그룹에 멤버로 참여
  - `pid == pgid` → 새로운 그룹 생성 후, 그 그룹의 리더가 됨
  - `pid != pgid` → 다른 그룹의 멤버가 됨 (그룹 이동, **일반적** 경우)
  - `pid == 0 = getpid()` → 이 함수 호출자의 PID값이 그대로 사용됨
  - `pgid == 0` → pid가 새로운 그룹 리더가 됨
- 호출자가 새로운 프로세스 그룹을 생성하고 그룹의 리더가 되는 방법
  - `setpgid(getpid(), getpid());` 혹은
  - `setpgid(0,0);` → 다음 슬라이드 사용 사례 참조

# 프로세스 그룹: pgrp2.c

---

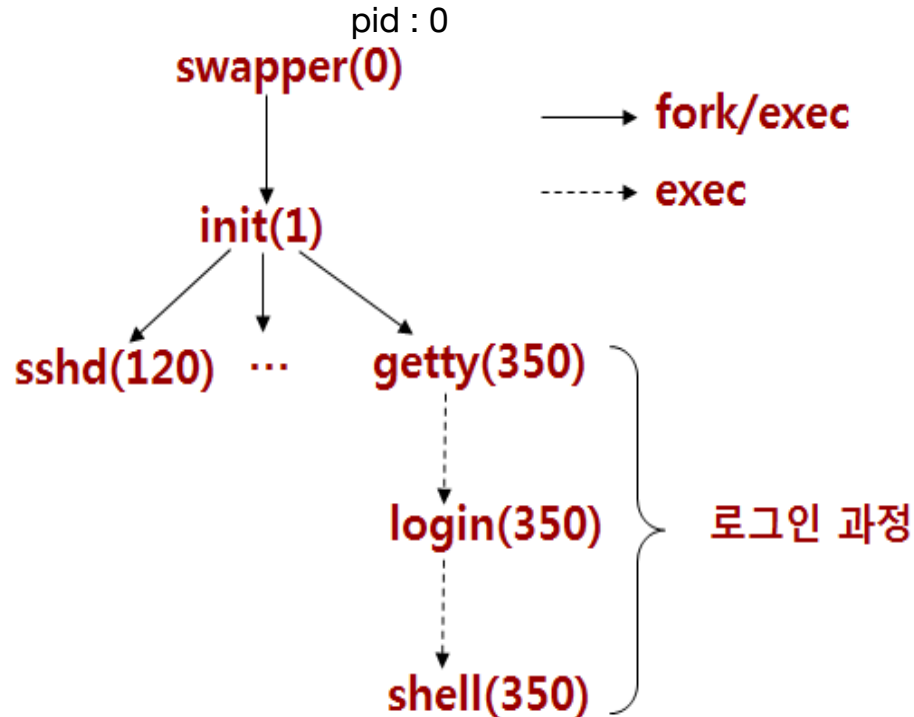
```
#include <sys/types.h>
#include <unistd.h>
main()
{
    int pid, gid;
    printf("PARENT: PID = %d  GID = %d \n", getpid(), getpgrp());
    pid = fork();
    if (pid == 0) {
        setpgid(0, 0); //자식 프로세스가 새로운 그룹 생성 후, 리더가 됨
        printf("CHILD: PID = %d  GID = %d \n", getpid(), getpgrp());
    }
}
```

자식 프로세스가 새로운 프로세스 그룹을 형성하고 리더가 되는 사례

## 9.5 시스템 부팅

# 시스템 부팅

- 시스템 부팅은 fork/exec 시스템 호출을 통해 이루어진다.
  - getty에서 shell까지는 exec만 호출 (후면에서 계속 프로세스 유지될 필요 없다)
  - shell 종료 후, init이 handling (init이 처음부터 터미널/로긴/셸 과정 다시 진행)
  - shell 이하는 다시 fork/exec로 진행 (fork 없이 exec만 하면 shell이 사라져서 안됨)



# 시스템 부팅

---

- swapper(스케줄러 프로세스) : pid 0
  - 커널 내부에서 만들어진 프로세스로 프로세스를 스케줄링 한다
- init(초기화 프로세스) : pid 1
  - /etc/init 혹은 /sbin/init에 존재
  - /etc/inittab 파일에 기술된 대로 시스템을 초기화
  - /etc/rc\*로 시작하는 스크립트 실행
  - 모든 프로세스의 조상
- 참고) pid 2: 페이지 데몬 (메모리 관리 전용 프로세스)

# 시스템 부팅

---

- **getty 프로세스**
  - 이 프로세스로부터 로그인 과정이 진행
  - 로그인 프롬프트를 내고 키보드 입력을 감지한다.
  - /bin/login 프로세스 실행
- **login 프로세스**
  - /etc/passwd 참조하여 사용자의 아이디 및 패스워드를 검사
  - /bin/sh 프로세스 실행
- **shell 프로세스**
  - 시작 파일을 실행한 후에 프롬프트를 내고 명령어 입력 대기

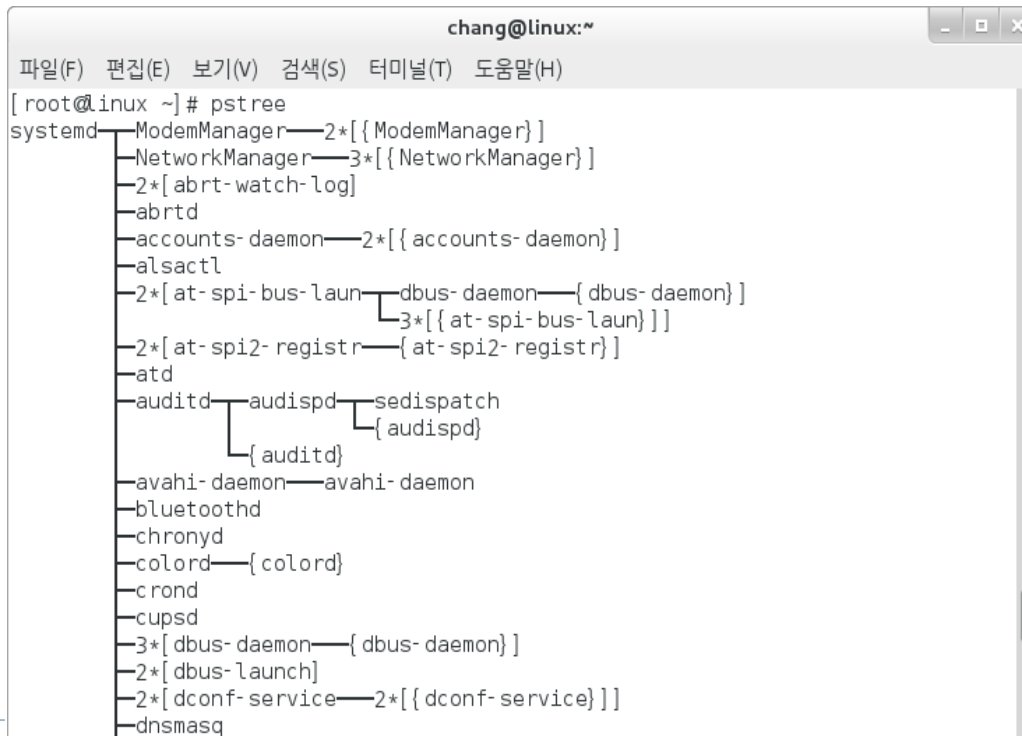


# 프로세스 트리 확인 및 출력

- 사용법

`$ pstree`

실행중인 프로세스들의 부모, 자식 관계를 트리 형태로 출력한다.



```
chang@linux:~  
파일(F) 편집(E) 보기(V) 검색(S) 터미널(T) 도움말(H)  
[root@linux ~]# pstree  
systemd--ModemManager--2*[ { ModemManager} ]  
      |--NetworkManager--3*[ { NetworkManager} ]  
      |--2*[ abrt-watch-log]  
      |--abrt  
      |--accounts-daemon--2*[ { accounts-daemon} ]  
      |--alsactl  
      |--2*[ at-spi-bus-laun--dbus-daemon-- { dbus-daemon} ]  
      |                               |  
      |                               3*[ { at-spi-bus-laun} ] ]  
      |--2*[ at-spi2-registr-- { at-spi2-registr} ]  
      |--atd  
      |--auditd--audispd--sedispatch  
      |               |  
      |               { audispd}  
      |               { auditd}  
      |--avahi-daemon--avahi-daemon  
      |--bluetoothd  
      |--chronyd  
      |--colord-- { colord}  
      |--crond  
      |--cupsd  
      |--3*[ dbus-daemon-- { dbus-daemon} ]  
      |--2*[ dbus-launch]  
      |--2*[ dconf-service--2*[ { dconf-service} ] ]  
      |--dnsmasq
```

# 핵심 개념

---

- 프로세스는 실행중인 프로그램이다.
- `fork()` 시스템 호출은 부모 프로세스를 똑같이 복제하여 새로운 자식 프로세스를 생성한다.
- `exec()` 시스템 호출은 프로세스 내의 프로그램을 새로운 프로그램으로 대체하여 새로운 프로그램을 실행시킨다.
- 시스템 부팅은 `fork/exec` 시스템 호출을 통해 이루어진다.