

13장 프로세스 원리

숙명여대 창병모

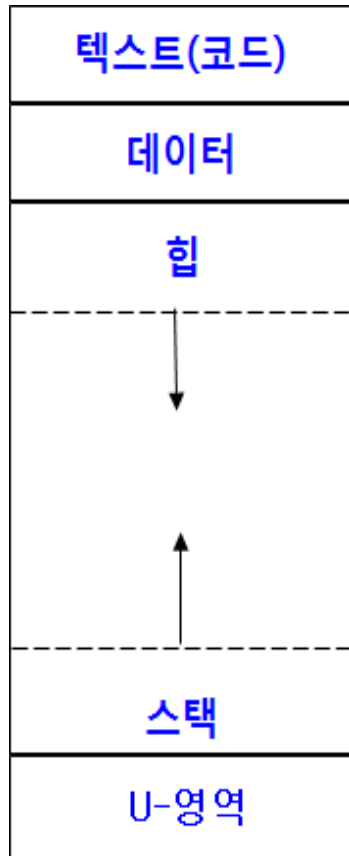
13.1 프로세스 이미지

프로세스

- 프로세스는 실행중인 프로그램이다.
- 프로그램 실행을 위해서는
 - 프로그램의 코드, 데이터, 스택, 힙, U-영역 등이 필요하다.
- 프로세스 이미지(구조)는 메모리 내의 프로세스 레이아웃
- 프로그램 자체가 프로세스는 아니다 !

프로세스 이미지

- 프로세스 구조



- 텍스트(코드)
 - 프로세스가 실행하는 실행 코드를 저장하는 영역
- 데이터
 - 프로그램 내에 선언된 전역 변수(global variable) 및 정적 변수(static variable) 등을 위한 영역
- 힙
 - 동적 메모리 할당을 위한 영역
- 스택
 - 함수 호출을 구현하기 위한 실행시간 스택(runtime stack)을 위한 영역
- U-영역
 - 열린 파일의 파일 디스크립터, 현재 작업 디렉터리 등과 같은 프로세스의 내부 정보

size 명령어

- 사용법

```
$ size [실행파일]
```

실행파일의 각 영역의 크기를 알려준다. 실행파일을 지정하지 않으면 a.out를 대상으로 한다.

- 예

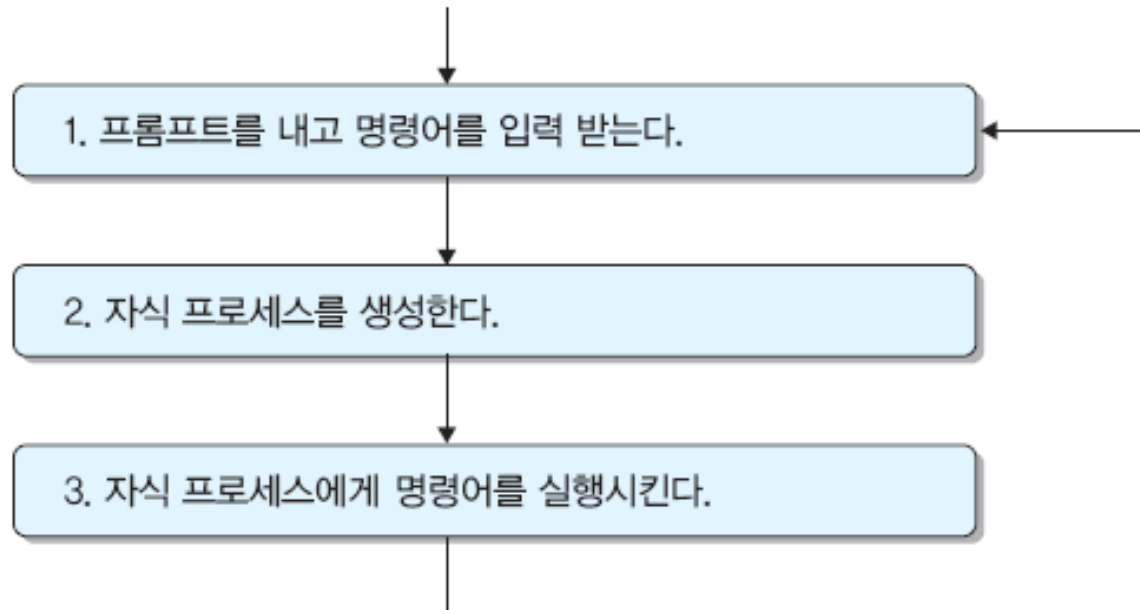
```
$ size /bin/ls
```

text	data	bss	dec	hex	filename
109479	5456	0	114935	1c0f7	/bin/ls

13.2 프로세스 ID

셸의 명령어 처리과정

\$ 명령어 &
[1] 프로세스번호



프로세스 ID

- 각 프로세스는 프로세스를 구별하는 번호인 프로세스 ID를 갖는다.

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

```
int getpid( );
```

프로세스의 ID를 반환한다.

```
int getppid( );
```

부모 프로세스의 ID를 반환한다.

프로세스 ID

- 프로그램 13.1 프로세스 ID

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 int main()
5 {
6     printf("Hello !\n");
7     printf("나의 프로세스 번호 : [%d] \n", getpid());
8     printf("내 부모 프로세스 번호 : [%d] \n", getppid());
9     system("ps");
10 }
```

프로세스 ID

- 실행 결과

```
$ hello &
```

```
Hello !
```

```
나의 프로세스 번호 : [16165]
```

```
내 부모 프로세스 번호 : [9045]
```

```
PID TTY TIME CMD
```

```
9045 pts/3 00:00:00 bash
```

```
16165 pts/3 00:00:00 hello
```

```
16169 pts/3 00:00:00 ps
```

13.3 프로세스 생성

프로세스 생성

- fork() 시스템 호출
 - 부모 프로세스를 똑같이 복제하여 새로운 자식 프로세스를 생성
 - 자기복제(自己複製)

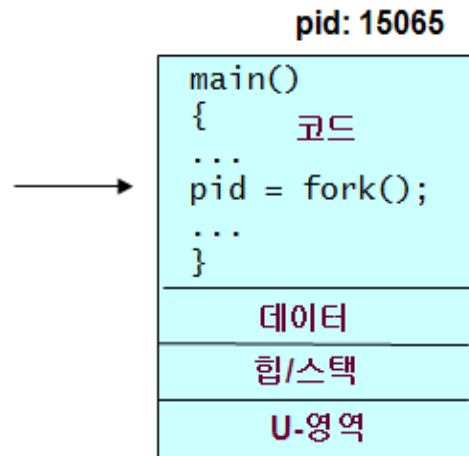
```
#include <unistd.h>
```

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

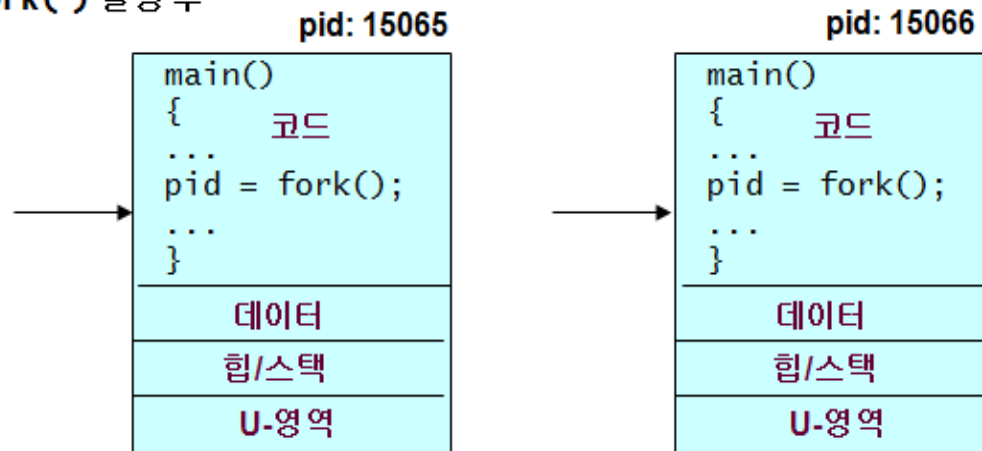
새로운 자식 프로세스를 생성한다. 자식 프로세스에게는 0을 반환하고
부모 프로세스에게는 자식 프로세스 ID를 반환한다.

프로세스 생성

fork() 실행 전



fork() 실행 후



프로세스 생성

- `fork()`는 한 번 호출되면 두 번 리턴한다.
 - 자식 프로세스에게는 0을 리턴하고
 - 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.
- 부모 프로세스와 자식 프로세스는 병행적으로 각각 실행을 계속한다.

프로그램 13.2: 프로세스 생성

```
1 #include <stdio.h>
2 #include <unistd.h>
3
4 /* 자식 프로세스를 생성한다. */
5 int main()
6 {
7     int pid;
8     printf("[%d] 프로세스 시작 \n", getpid());
9     pid = fork();
10    printf("[%d] 프로세스 : 반환값 %d\n", getpid(), pid);
11 }
```

실행결과

[15065] 프로세스 시작

[15065] 프로세스 : 반환값 15066

▶ 1[15066] 프로세스 : 반환값 0

부모-자식 프로세스

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

```
pid = fork();  
if ( pid == 0 )  
{ 자식 프로세스의 실행 코드 }  
else  
{ 부모 프로세스의 실행 코드 }
```


프로그램 13.3: 자식 프로세스 생성

```
#include <stdlib.h>
#include <stdio.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());
    }
}
```

프로그램 13.3: 자식 프로세스 생성

실행결과

[Parent] : Hello, world ! pid=15065

[Child] : Hello, world ! pid=15066

프로세스 기다리기: wait()

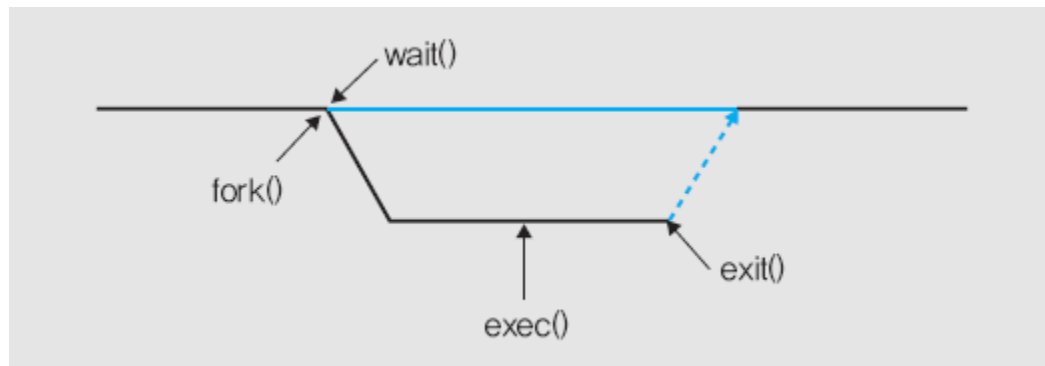
- 사용법

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

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

```
pid_t wait(int *status);
```

자식 프로세스 중의 하나가 종료할 때까지 기다린다. 자식 프로세스가 종료하면 종료코드가 *status에 저장된다. 종료한 자식 프로세스의 ID를 반환한다.



13.4 프로그램 실행

프로그램 실행의 원리

- 셸의 예

```
$ hello &
```

```
[1] 16165
```

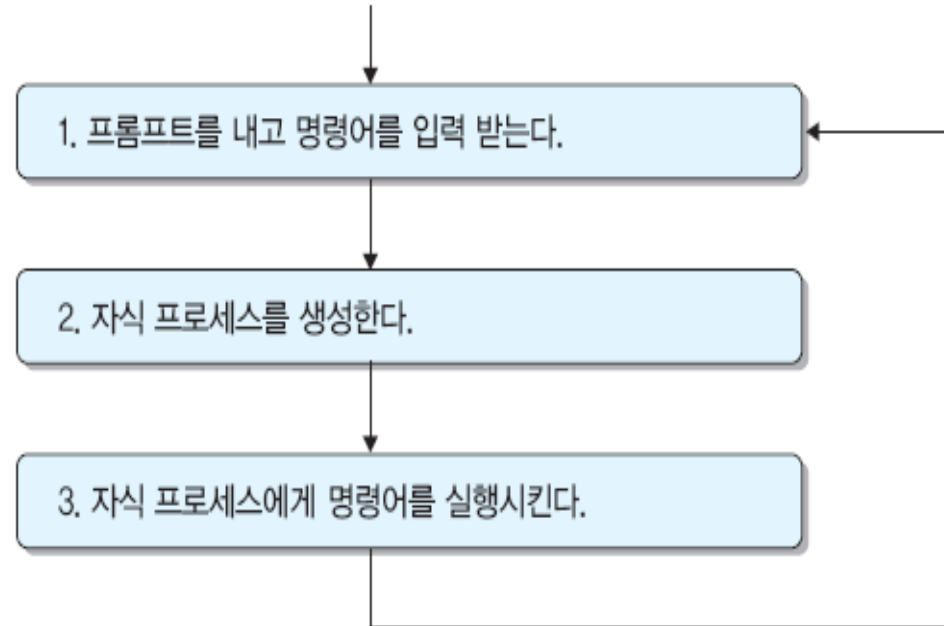
```
$ ps
```

```
PID TTY TIME CMD
```

```
9045 pts/3 00:00:00 bash
```

```
16165 pts/3 00:00:00 hello
```

```
16169 pts/3 00:00:00 ps
```

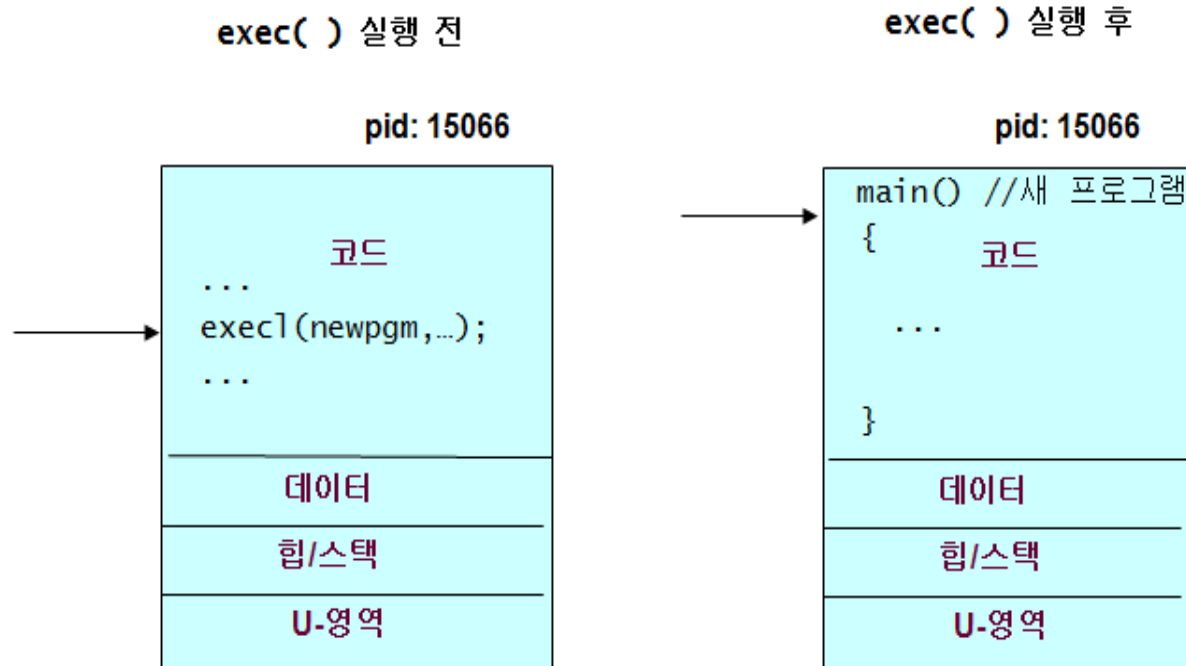


프로그램 실행

- fork() 후
 - 자식 프로세스는 부모 프로세스와 똑같은 코드 실행
- 자식 프로세스에게 새로운 프로그램을 시키려면 어떻게 하여야 할까?
 - 프로세스 내의 프로그램을 새 프로그램으로 대체
 - exec() 시스템 호출 사용
- 보통 fork() 후에 exec()

프로그램 실행: exec()

- 프로세스가 exec() 호출을 하면,
 - 그 프로세스 내의 프로그램은 완전히 새로운 프로그램으로 대체
 - 자기대치(自己代置)
- 새 프로그램의 main()부터 실행이 시작한다.



프로그램 실행: exec()

- exec() 호출이 성공하면 리턴할 곳이 없어진다.
- 성공한 exec() 호출은 절대 리턴하지 않는다.

```
#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 대체한 후 새 프로그램을 실행한다. 성공한 exec() 호출은 반환하지 않으며 실패하면 -1을 반환한다.

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 }
```

실행결과

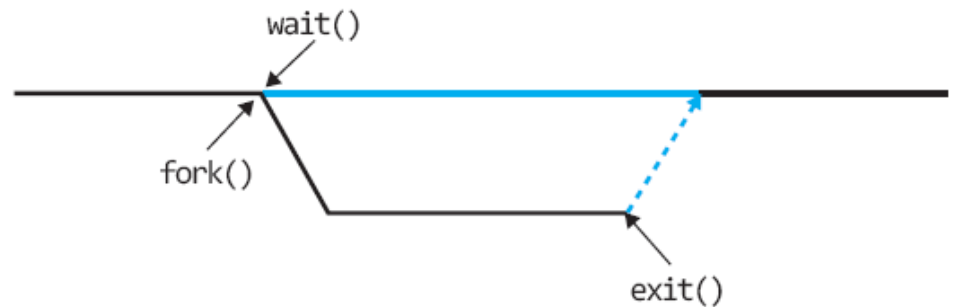
시작

hello

셸의 명령어 처리 원리

- 보통 `fork()` 호출 후에 `exec()` 호출
 - 새로 실행할 프로그램에 대한 정보를 `arguments`로 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);  
}
```



프로그램 13.5: 프로그램 실행

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
5 /* 자식 프로세스를 생성하여 echo 명령어를 실행한다. */
6 int main( )
7 {
8     int pid, child, status;
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 }

프로그램 13.5: 프로그램 실행

● 실행결과

부모 프로세스 시작

hello

자식 프로세스 15066 끝

부모 프로세스 끝

13.5 프로그램 실행 과정

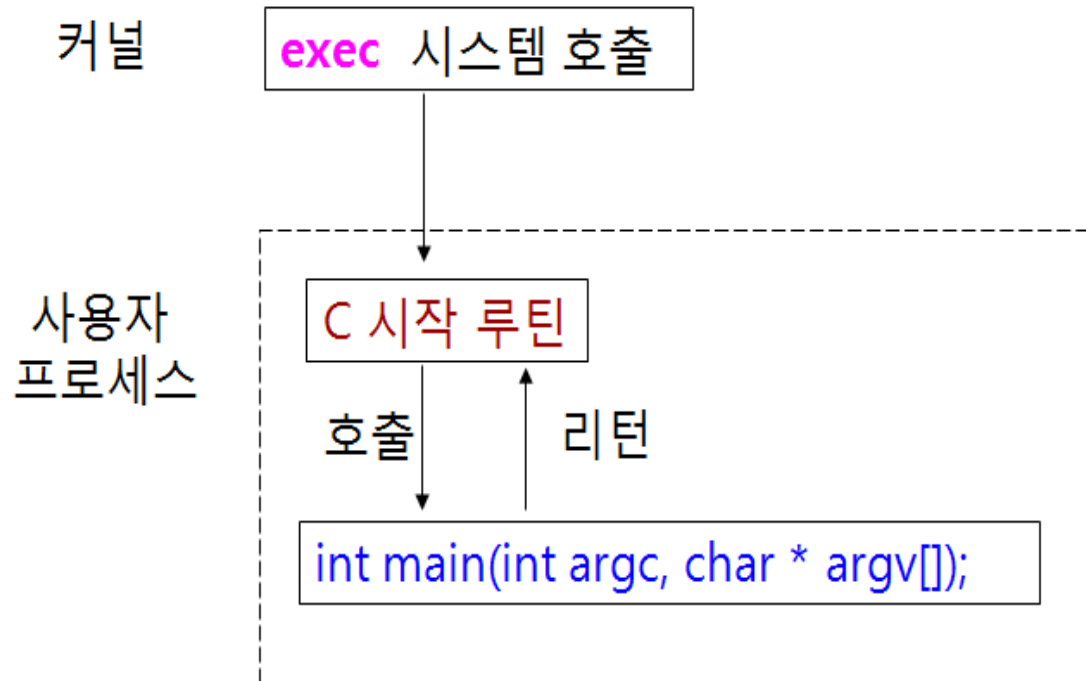
프로그램 실행 시작

- exec 시스템 호출
 - C 시작 루틴에 명령줄 인수와 환경 변수를 전달하고
 - 프로그램을 실행시킨다.
- C 시작 루틴(start-up routine)
 - main 함수를 호출하면서 명령줄 인수, 환경 변수를 전달

exit(main(argc, argv));

- 실행이 끝나면 반환값을 받아 exit 한다.

프로그램 실행 시작

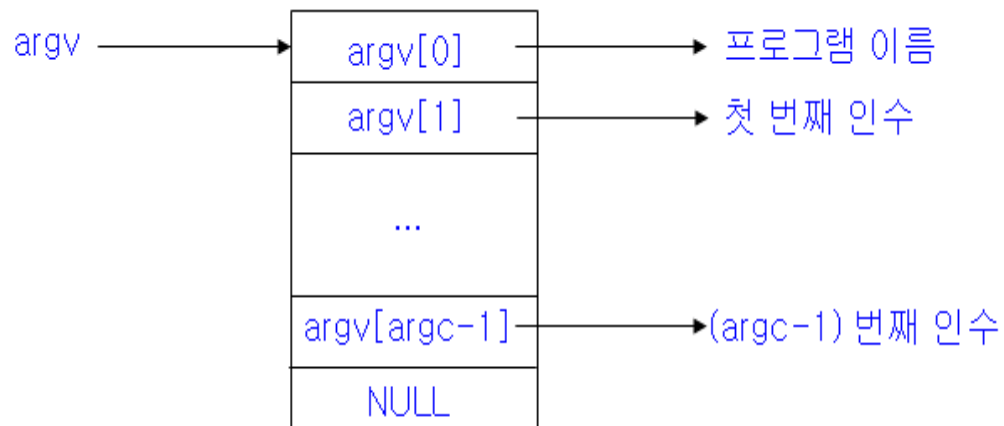


명령줄 인수/환경 변수

```
int main(int argc, char *argv[]);
```

argc : 명령줄 인수의 수

argv[] : 명령줄 인수 리스트를 나타내는 포인터 배열



프로그램 13.6 명령줄 인수

```
#include <stdio.h>
/* 모든 명령줄 인수를 프린트한다. */
int main(int argc, char *argv[])
{
    int i;

    for (i = 0; i < argc; i++) /* 모든 명령줄 인수 프린트 */
        printf("argv[%d]: %s \n", i, argv[i]);

    return 0;
}
```

- 실행 결과

```
$ printargv hello world
argv[0]: printargv
argv[1]: hello
argv[2]: world
```

13.5 시스템 부팅

시스템 부팅

```
$ ps -ef
```

```
UID PID PPID C STIME TTY TIME CMD
```

```
root 1 0 0 May21 ? 00:00:04 /sbin/init
```

```
root 2 0 0 May21 ? 00:00:00 [kthreadd]
```

```
root 3 2 0 May21 ? 00:00:00 [migration/0]
```

```
root 4 2 0 May21 ? 00:00:00 [ksoftirqd/0]
```

```
root 5 2 0 May21 ? 00:00:00 [watchdog/0]
```

```
...
```

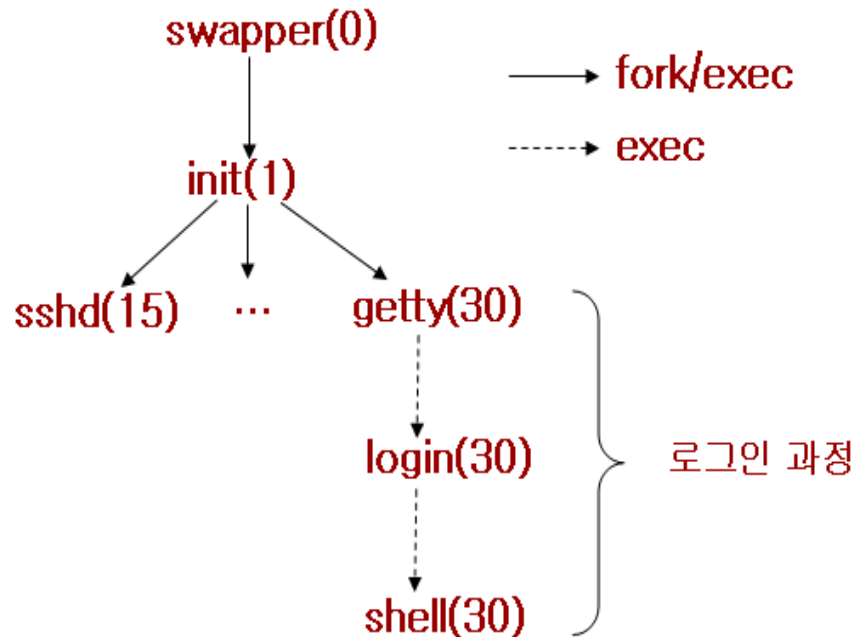
```
root 120 1 0 May21 ? 00:00:00 /usr/sbin/sshd
```

```
...
```

```
root 350 1 0 May21 tty2 00:00:00 /sbin/mingetty /dev/tty2
```

시스템 부팅

- 시스템 부팅은 fork/exec 시스템 호출을 통해 이루어진다.



시스템 부팅

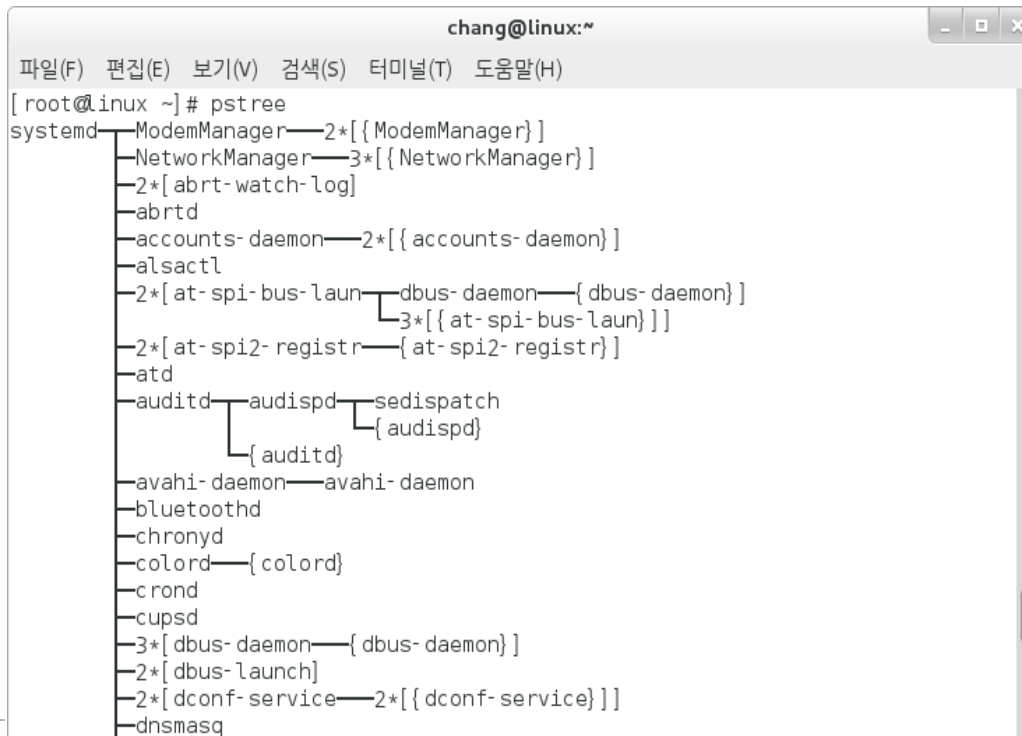
- swapper(스케줄러 프로세스)
 - 커널 내부에서 만들어진 프로세스로 프로세스 스케줄링을 한다
- init(초기화 프로세스)
 - /etc/inittab 파일에 기술된 대로 시스템을 초기화
- 서비스 데몬 프로세스
 - 서비스들을 위한 데몬 프로세스들이 생성된다. 예: ftpd
- getty 프로세스
 - 로그인 프롬프트를 내고 키보드 입력을 감지한다.
- login 프로세스
 - 사용자의 로그인 아이디 및 패스워드를 검사
- shell 프로세스
 - 시작 파일을 실행한 후에 쉘 프롬프트를 내고 사용자로부터 명령어를 기다린다

프로세스 트리 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-d  
      |--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.
- 시그널은 예기치 않은 사건이 발생할 때 이를 알리는 소프트웨어 인터럽트이다.