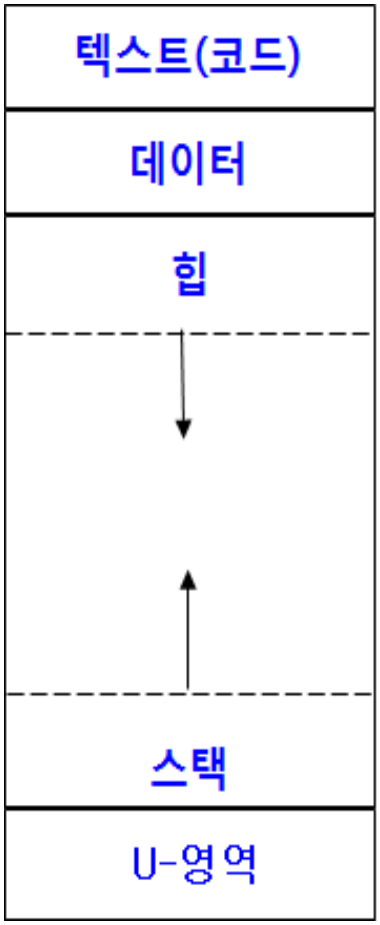
오픈소스소프트웨어

Open-Source Software

ICT융합학부 조용우

프로세스 생성

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

프로세스 크기 : size

$ size [실행파일]

실행파일의 각 영역의 크기를 알려준다.

실행파일을 지정하지 않으면 a.out을 대상으로 한다.

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

프로세스 ID

#inlcude <unistd.h>

int getpid();

프로세스의 ID를 반환한다.

Int getppid();

부모 프로세스의 ID를 반환한다.

- 각 프로세스는 프로세스를 구별하는 번호인 프로세스 ID를 갖는다.

프로세스 우선순위

- 리눅스에서 0~139까지 우선순위를 가지고 있으며, 0~ 99까지는 realtime 100~139까지는 사용자를 위함.
- Nice value range는 -20부터 +19정도가 제일 좋으며 기본값 0을 기준으로 -20이 제일 높고 +19가 제일 낮다.
- Nice value range 와 우선순위의 관계 : PR = 20 + NI
  - PR = 20 + (-20 to +19)일 것이며 0~39까지의 범위이기 때문에 사용자 범위인 100~139와 적합.

프로세스 우선순위

- 리눅스에서 0~139까지 우선순위를 가지고 있으며, 0~ 99까지는 realtime 100~139까지는 사용자를 위함.
- Nice value range는 -20부터 +19정도가 제일 좋으며 기본값 0을 기준으로 -20이 제일 높고 +19가 제일 낮다.
- Nice value range 와 우선순위의 관계 : PR = 20 + NI
  - PR = 20 + (-20 to +19)일 것이며 0~39까지의 범위이기 때문에 사용자 범위인 100~139와 적합.

프로세스 생성

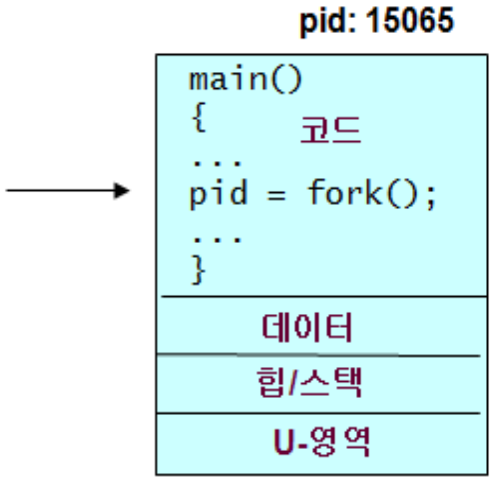
```
#include <unistd.h>
pid_t fork(void);
```

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

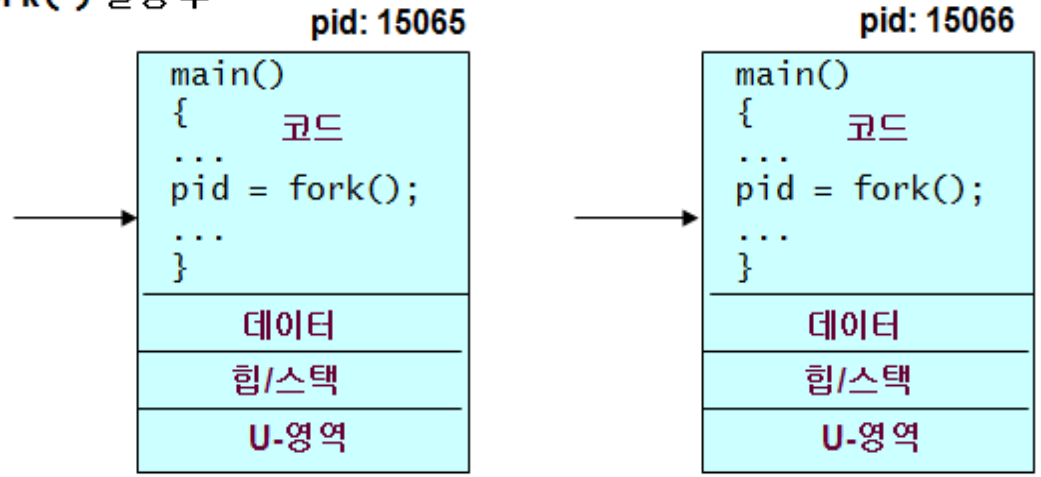
- fork() 시스템 호출
  - 부모 프로세스를 똑같이 복제하여 새로운 자식 프로세스를 생성
  - 자기복제(自己複製)

프로세스 생성

fork( ) 실행 전



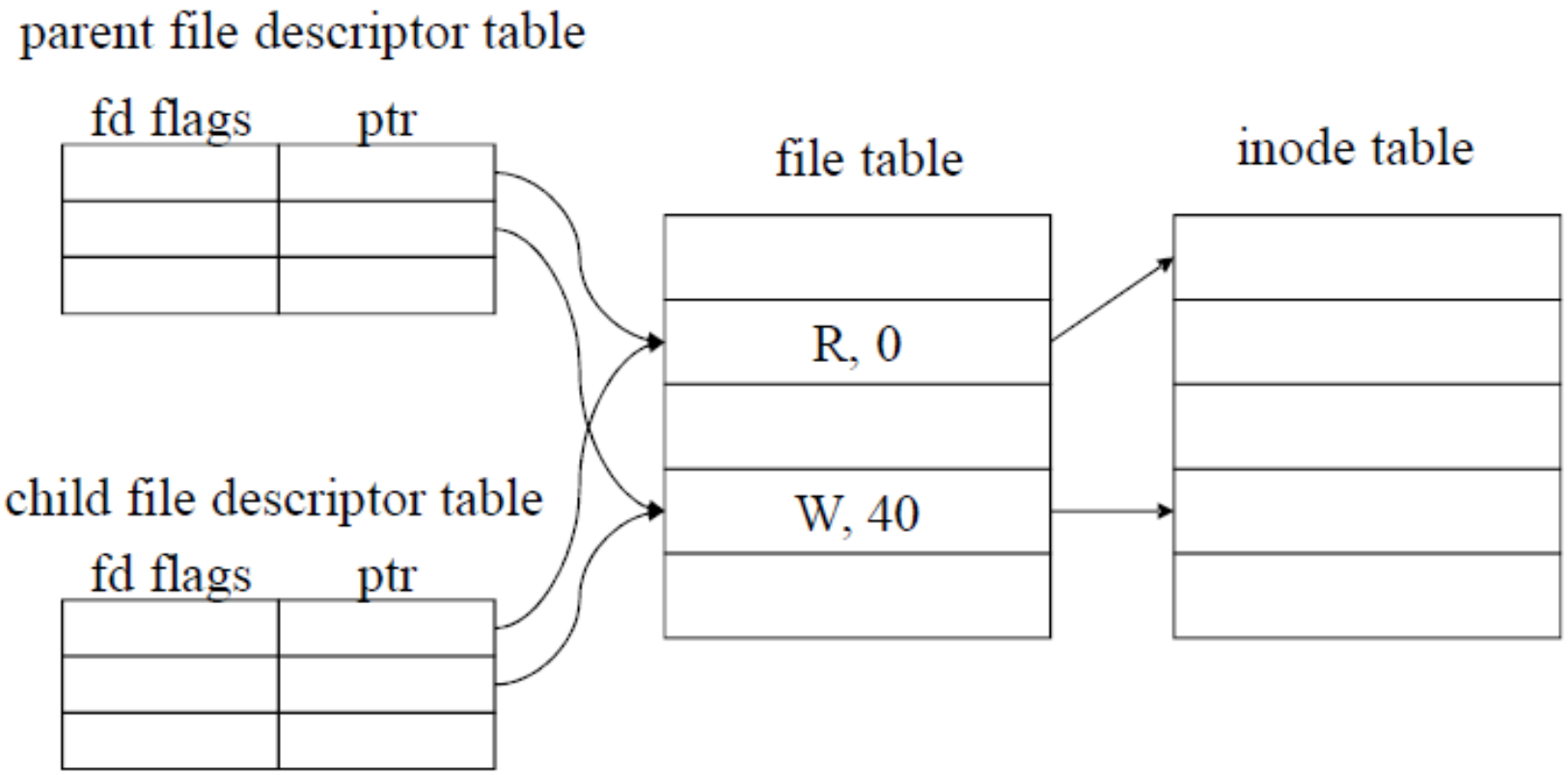
fork( ) 실행 후



프로세스 생성

- fork()는 한 번 호출되면 두 번 리턴한다.
- 자식 프로세스에게는 0을 리턴하고 부모 프로세스에게는 자식 프로세스 ID를 리턴한다.
- 자식 프로세스는 부모 프로세스로부터 상속받는다.
- 부모 프로세스와 자식 프로세스는 병행적으로 각각 실행을 계속한다.
- 부모 프로세스가 먼저 시작할지 자식 프로세스가 먼저 시작할지 절대로 모른다.

프로세스 생성


parent file descriptor table
fd flags
ptr
child file descriptor table
fd flags
ptr
file table
R, 0
W, 40
inode table

프로세스 생성

```
#include <stdio.h>
main(int argc, char* argv[])
{
  int getpid(), getppid();

  printf("[%d] parent process id: %d\n",
      getpid(), getppid());

  fork();

  printf("\n\t[%d] parent process id: %d\n",
      getpid(), getppid());
}
```

```
$ fork
[18607] parent process id: 18606

        [18608] parent process id: 18607

        [18607] parent process id: 18606
```

부모-자식 프로세스

- fork() 호출 후에 리턴값이 다르므로 이를 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 }
```

프로세스 기다리기 : wait()

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

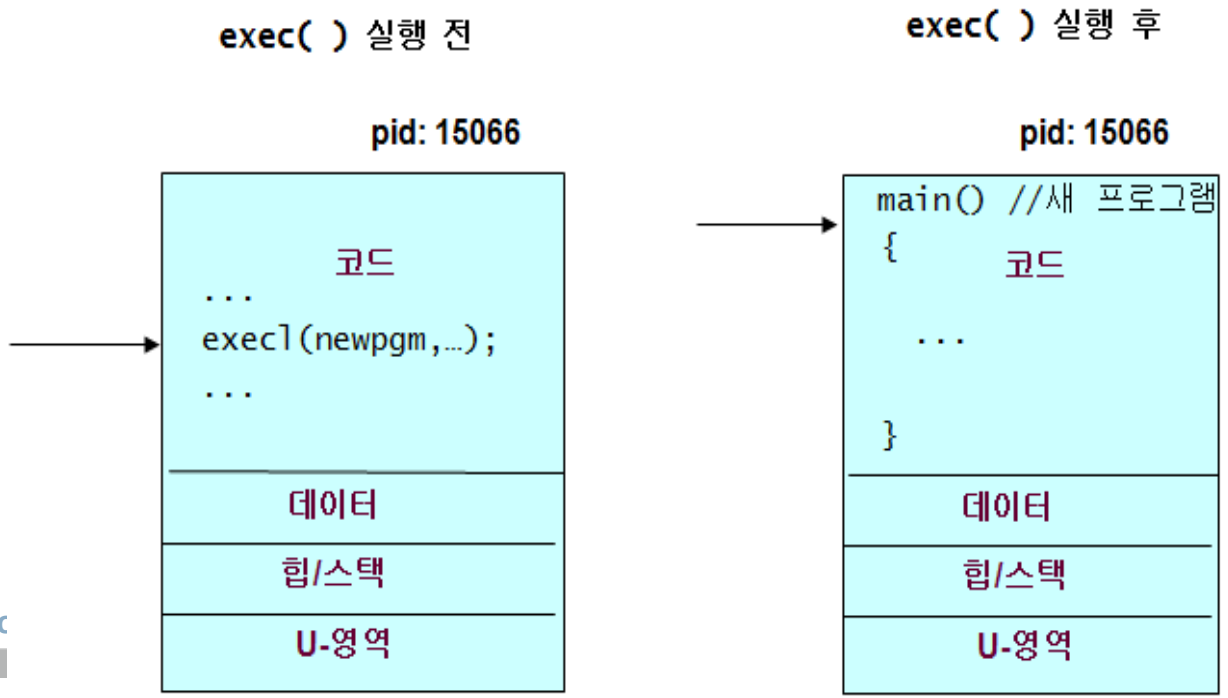
자식 프로세스 중의 하나가 종료할 때까지 기다린다. 자식 프로세스가 종료하면 종료코드가 *status에 저장된다. 종료한 자식 프로세스의 ID를 반환한다

프로그램 실행

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

프로그램 실행 : exec()

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



프로그램 실행 : exec()

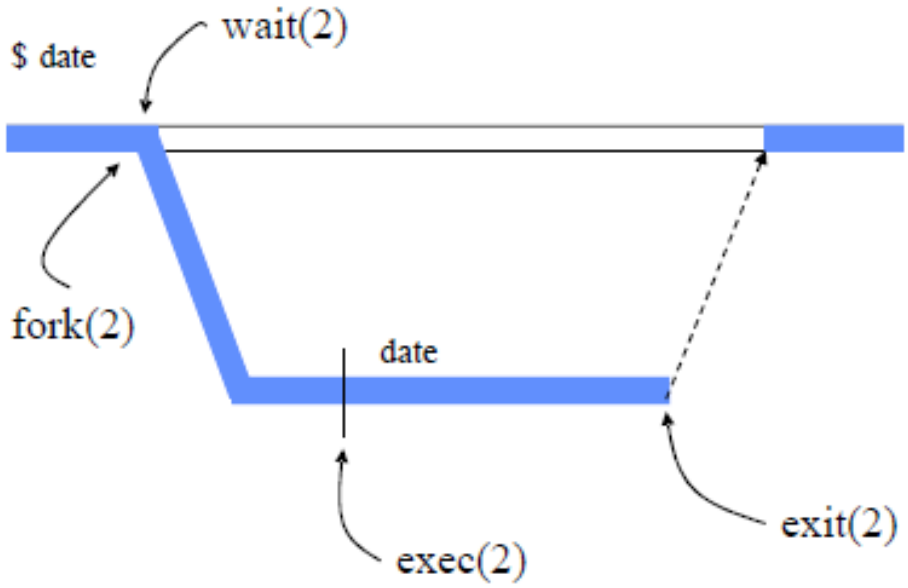
```
#include <unistd.h>
int execl (char* path, char* arg0, char* arg1, … , char* argn, NULL)
int execv (char* path, char* argv[])
int execlp (char* file, char* arg0, char* arg1, … , char* argn, NULL)
int execvp (char* file, char* argv[])
```

프로그램 실행 : exec()

- 호출한 프로세스의 코드, 데이터, 힙, 스택 등을 path가 나타내는 새로운 프로그램으로 대치한 후 새 프로그램을 실행한다.
- 성공한 exec( ) 호출은 리턴하지 않으며 실패하면 -1을 리턴한다.
- exec() 호출이 성공하면 리턴 할 곳이 없어진다.

쉘의 명령어 처리 원리

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



프로그램 실행

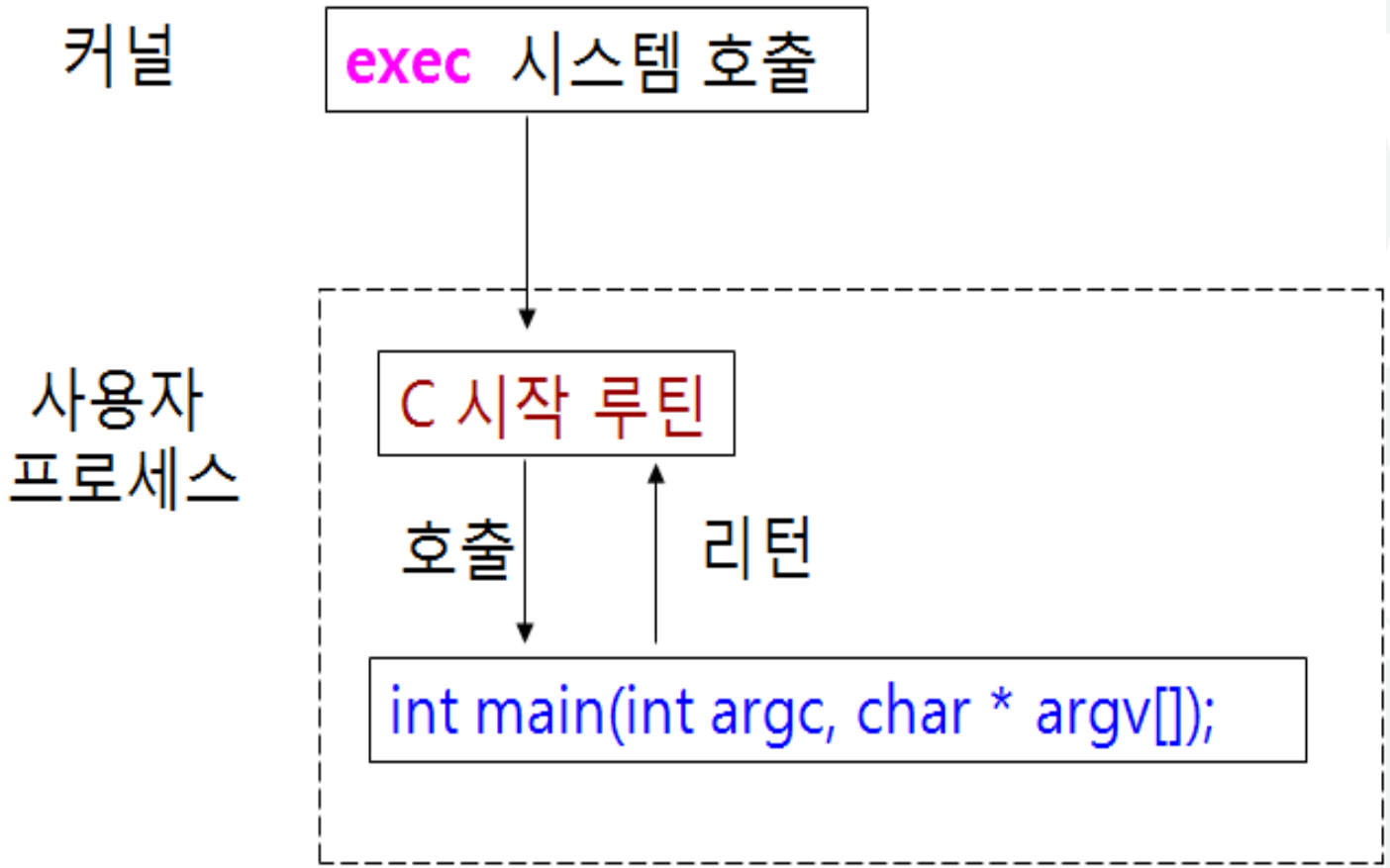
프로그램 실행 시작

- exec 시스템 호출
  - C 시작 루틴에 명령줄 인수와 환경변수를 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

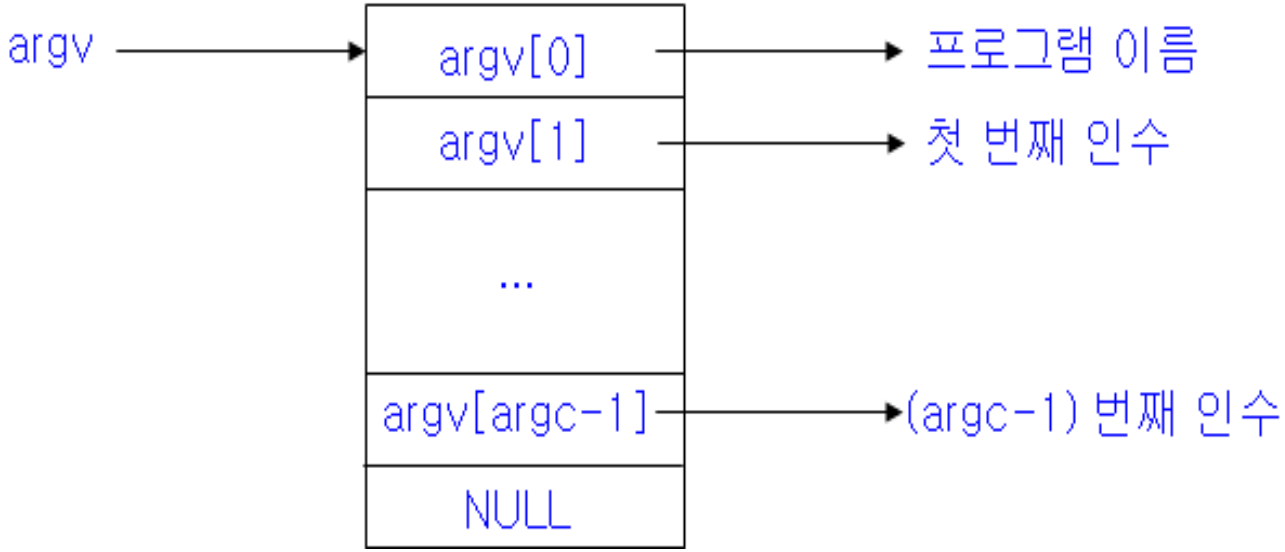


명령줄 인수/환경 변수

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

argc : 명령줄 인수의 수

argv[] : 명령줄 인수 리스트를 나타내는 포인터 배열



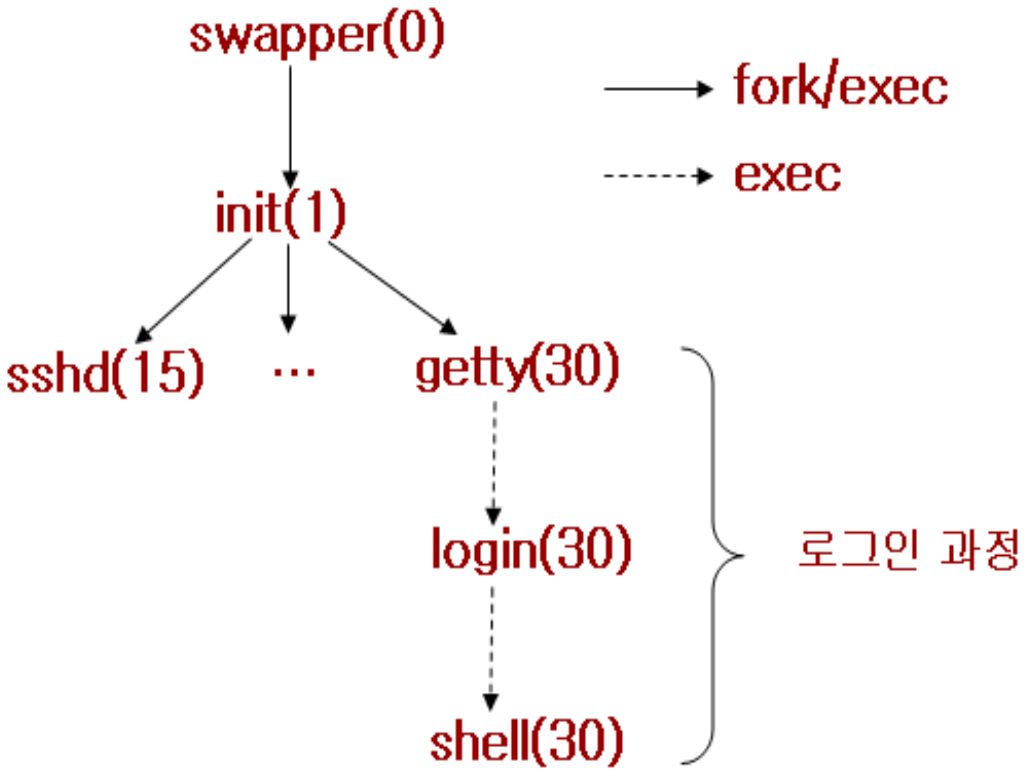
시스템 부팅

시스템 부팅

```
$ ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 Apr16 ? 00:00:04 /sbin/init
root 2 0 0 Apr16 ? 00:00:00 [kthreadd]
root 3 2 0 Apr16 ? 00:00:00 [migration/0]
root 4 2 0 Apr16 ? 00:00:00 [ksoftirqd/0]
root 5 2 0 Apr16 ? 00:00:00 [watchdog/0]
...
root 120 1 0 Apr16 ? 00:00:00 /usr/sbin/sshd
...
root 350 1 0 Apr16 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

시스템 부팅

- 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

시스템 부팅

$ pstree

실행중인 프로세스들의 부모, 자식관계를 트리 형태로 출력한다.

```
[1111222333@node1 ~]$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─5*[a]
        ├─24*[a.out]
        ├─abrt-dbus───3*[{abrt-dbus}]
        ├─2*[abrt-watch-log]
        ├─abrtd
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─10*[al]
        ├─20*[alarm]
        ├─21*[alarm.out]
        ├─alsactl
        ├─at-spi-bus-laun─┬─dbus-daemon───{dbus-daemon}
        │                 └─3*[{at-spi-bus-laun}]
        ├─at-spi2-registr───2*[{at-spi2-registr}]
        ├─atd
```