

# 8장 프로세스

## 8.1 쉘과 프로세스

# 셸(Shell)이란 무엇인가?

- 셸의 역할

- 셸은 사용자와 운영체제 사이에 창구 역할을 하는 소프트웨어 (사용자와 운영체제 사이의 인터페이스 )
- 명령어 처리기(command processor)
- 사용자로부터 명령어를 입력받아 이를 처리한다

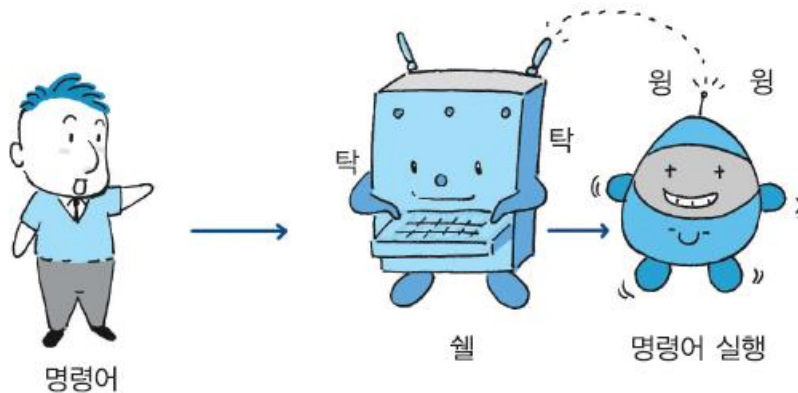


그림 6.1 셸의 역할

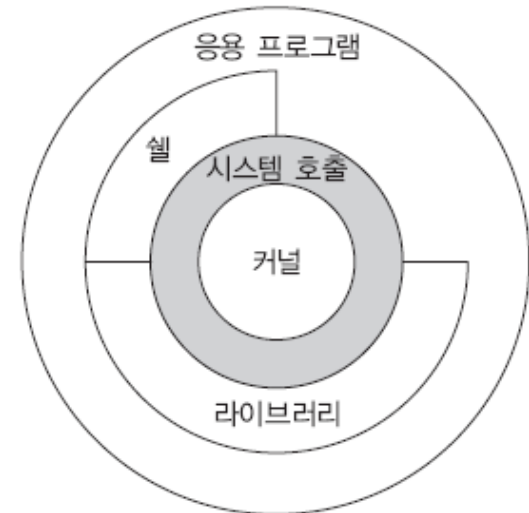
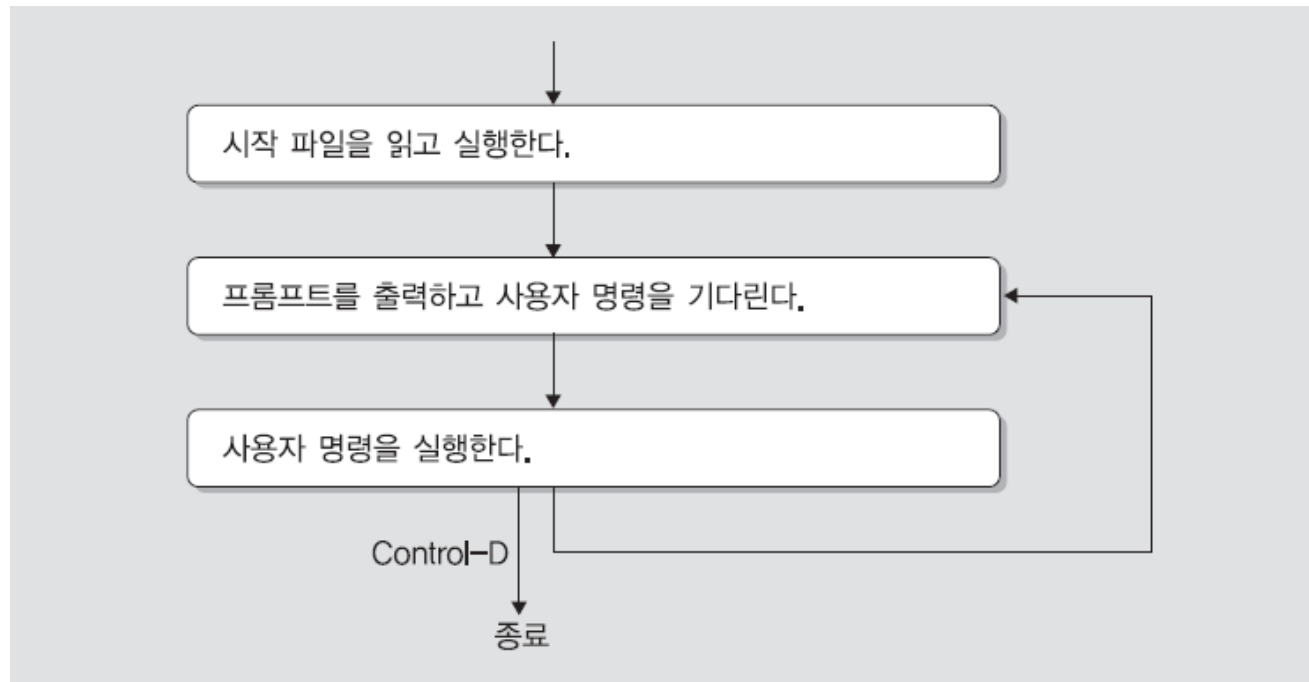


그림 1.3 유닉스 운영체제 구조

# 셸의 실행 절차

---



# 복합 명령어

---

- 명령어 열(command sequence)

\$ 명령어1; ... ; 명령어n

\$ date; who; pwd

- 명령어 그룹(command group)

\$ (명령어1; ... ; 명령어n)

\$ date; who; pwd > out1.txt

\$ (date; who; pwd) > out2.txt

# 전면 처리 vs 후면처리

---

- 전면 처리

- 명령어를 입력하면 명령어가 전면에서 실행되며 명령어 실행이 끝날 때까지 셸이 기다려 준다.

- 후면 처리

- 명령어들을 후면에서 처리하고 전면에서는 다른 작업을 할 수 있으면 동시에 여러 작업을 수행할 수 있다.
- \$ 명령어 &

# 후면 처리 예

---

- `$ (sleep 100; echo done) &`  
`[1] 8320`
- `$ find . -name test.c -print &`  
`[2] 8325`
- `$ jobs`  
`[1] + Running ( sleep 100; echo done )`  
`[2] - Running find . -name test.c -print`
- `$ fg %작업번호`  
`$ fg %1`  
`( sleep 100; echo done )`
- 후면처리 입출력  
`$ find . -name test.c -print > find.txt &`  
`$ find . -name test.c -print | mail chang &`  
`$ wc < inputfile &`

# 프로세스(process)

---

- 실행중인 프로그램을 **프로세스**(process)라고 부른다.
- 각 프로세스는 유일한 프로세스 번호 PID를 갖는다.
- ps 명령어를 사용하여 나의 프로세스들을 볼 수 있다.

```
$ ps
```

```
PID TTY TIME CMD
```

```
8695 pts/3 00:00:00 csh
```

```
8720 pts/3 00:00:00 ps
```

```
$ ps u
```

```
USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND
```

```
chang 8695 0.0 0.0 5252 1728 pts/3 Ss 11:12 0:00 -csh
```

```
chang 8793 0.0 0.0 4252 940 pts/3 R+ 11:15 0:00 ps u
```

# ps (process status)

---

- ps [-옵션] 명령어

- 현재 존재하는 프로세스들의 실행 상태를 요약해서 출력

```
$ ps
```

```
PID TTY TIME CMD
```

```
25435 pts/3 00:00:00 csh
```

```
25461 pts/3 00:00:00 ps
```

- \$ ps -aux (BSD 유닉스)

- - a: 모든 사용자의 프로세스를 출력
- - u: 프로세스에 대한 좀 더 자세한 정보를 출력
- - x: 더 이상 제어 터미널을 갖지 않은 프로세스들도 함께 출력

- \$ ps -ef (시스템 V)

- - e: 모든 사용자 프로세스 정보를 출력
- - f: 프로세스에 대한 좀 더 자세한 정보를 출력



# sleep

---

- sleep 명령어
  - 지정된 시간만큼 실행을 중지한다.

\$ sleep 초

\$ (echo 시작; sleep 5; echo 끝)

# kill

---

- kill 명령어
    - 현재 실행중인 프로세스를 강제로 종료
- \$ kill [-시그널] 프로세스번호
- \$ (echo 시작; sleep 5; echo 끝) &
- 1230
- \$ kill 1230

# wait

---

\$ wait [프로세스번호]

- 해당 프로세스 번호를 갖는 자식 프로세스가 종료될 때까지 기다린다.
- 프로세스 번호를 지정하지 않으면 모든 자식 프로세스를 기다린다.

\$ (sleep 10; echo 1번 끝) &

1231

\$ echo 2번 끝; wait 1231; echo 3번 끝

2번 끝

1번 끝

3번 끝

\$ (sleep 10; echo 1번 끝) &

\$ (sleep 10; echo 2번 끝) &

\$ echo 3번 끝; wait; echo 4번 끝  
3번 끝

1번 끝

2번 끝

4번 끝

# exit

---

- exit
  - 셸을 종료하고 종료코드(exit code)을 부모 프로세스에 전달  
\$ exit [종료코드]

## 8.2 프로그램 실행

# 프로그램 실행 시작

---

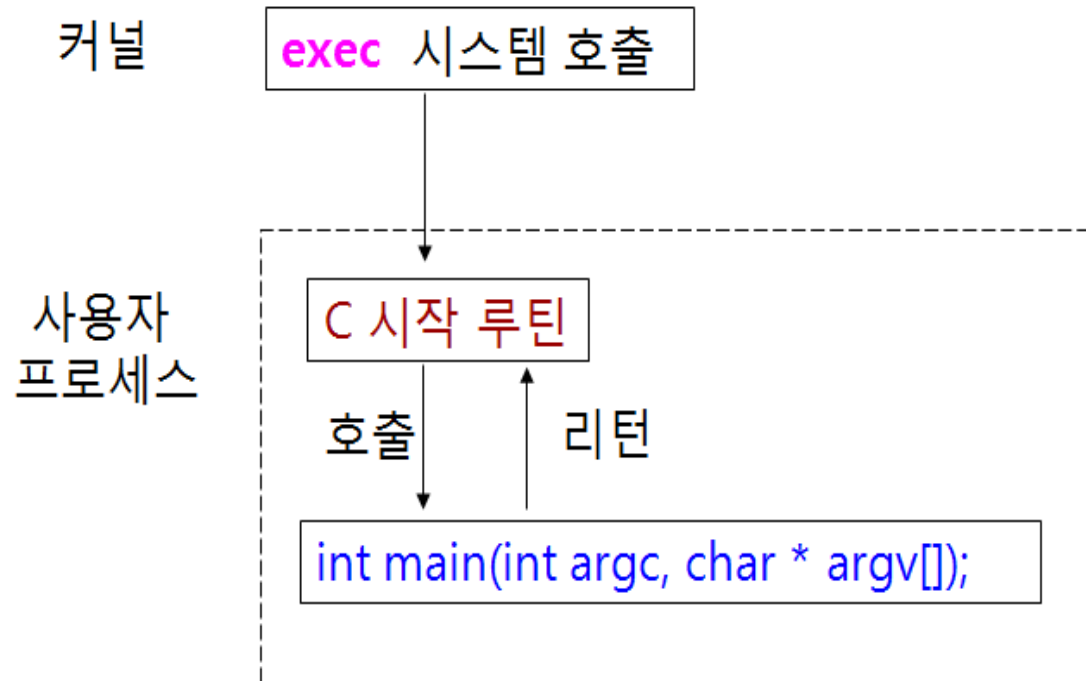
- exec 시스템 호출
  - C 시작 루틴에 명령줄 인수와 환경 변수를 전달하고
  - 프로그램을 실행시킨다.
- C 시작 루틴(start-up routine)
  - main 함수를 호출하면서 명령줄 인수, 환경 변수를 전달

**exit( main( argc, argv) );**

- 실행이 끝나면 반환값을 받아 exit 한다.

# 프로그램 실행 시작

---

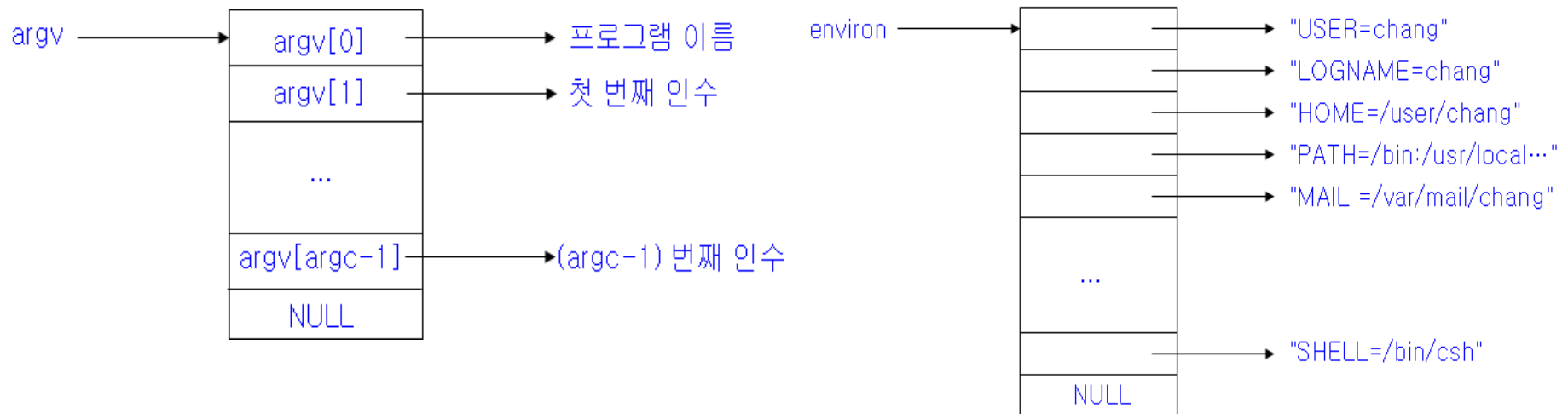


# 명령줄 인수/환경 변수

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

argc : 명령줄 인수의 개수

argv[] : 명령줄 인수 리스트를 나타내는 포인터 배열





# args.c

---

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

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

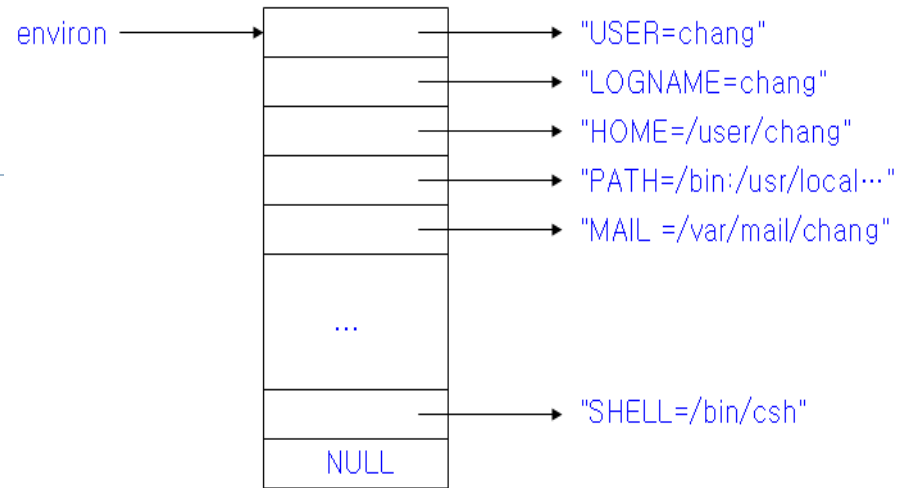
    exit(0);
}
```

# environ.c

```
#include <stdio.h>
/* 모든 환경 변수를 출력한다. */
int main(int argc, char *argv[])
{
    char **ptr;
    extern char **environ;

    for (ptr = environ; *ptr != 0; ptr++) /* 모든 환경 변수 값 출력*/
        printf("%s \n", *ptr);

    exit(0);
}
```



# 환경 변수 접근

---

- getenv() 시스템 호출을 사용하여 환경 변수를 하나씩 접근하는 것도 가능하다.

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

```
char *getenv(const char *name);
```

환경 변수 name의 값을 반환한다. 해당 변수가 없으면 NULL을 반환한다.

# printenv.c

---

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

/* 환경 변수를 3개 프린트한다. */
int main(int argc, char *argv[])
{
    char    *ptr;

    ptr = getenv("HOME");
    printf("HOME = %s \n", ptr);

    ptr = getenv("SHELL");
    printf("SHELL = %s \n", ptr);

    ptr = getenv("PATH");
    printf("PATH = %s \n", ptr);

    exit(0);
```

# 환경 변수 설정

- putenv(), setenv()를 사용하여 특정 환경 변수를 설정한다.

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

```
int putenv(const char *name);
```

name=value 형태의 스트링을 받아서 이를 환경 변수 리스트에 넣어준다.  
name이 이미 존재하면 원래 값을 새로운 값으로 대체한다.

```
int setenv(const char *name, const char *value, int rewrite);
```

환경 변수 name의 값을 value로 설정한다. name이 이미 존재하는 경우에는  
rewrite 값이 0이 아니면 원래 값을 새로운 값으로 대체하고 rewrite 값이 0이  
면 그대로 둔다.

```
int unsetenv(const char *name);
```

환경 변수 name의 값을 지운다.

## 8.3 프로그램 종료

# 프로그램 종료

---

- 정상 종료(normal termination)
  - `main()` 함수에서 `return ret;`  
C 시작 루틴은 이 리턴 값을 가지고 `exit()`을 호출
  - 프로그램 내에서 직접 `exit()`을 호출
  - 프로그램 내에서 직접 `_exit()`을 호출
- 비정상 종료(abnormal termination)
  - `abort()`
    - 프로세스에 `SIGABRT` 시그널을 보내어 프로세스를 비정상적으로 종료
  - 시그널에 의한 종료

# 프로그램 종료

---

- `exit()`

- 모든 열려진 스트림을 닫고(`fclose`), 출력 버퍼의 내용을 디스크에 쓰는(`fflush`) 등의 뒷정리 후 프로세스를 정상적으로 종료
- **종료 코드**(exit code)를 부모 프로세스에게 전달한다.

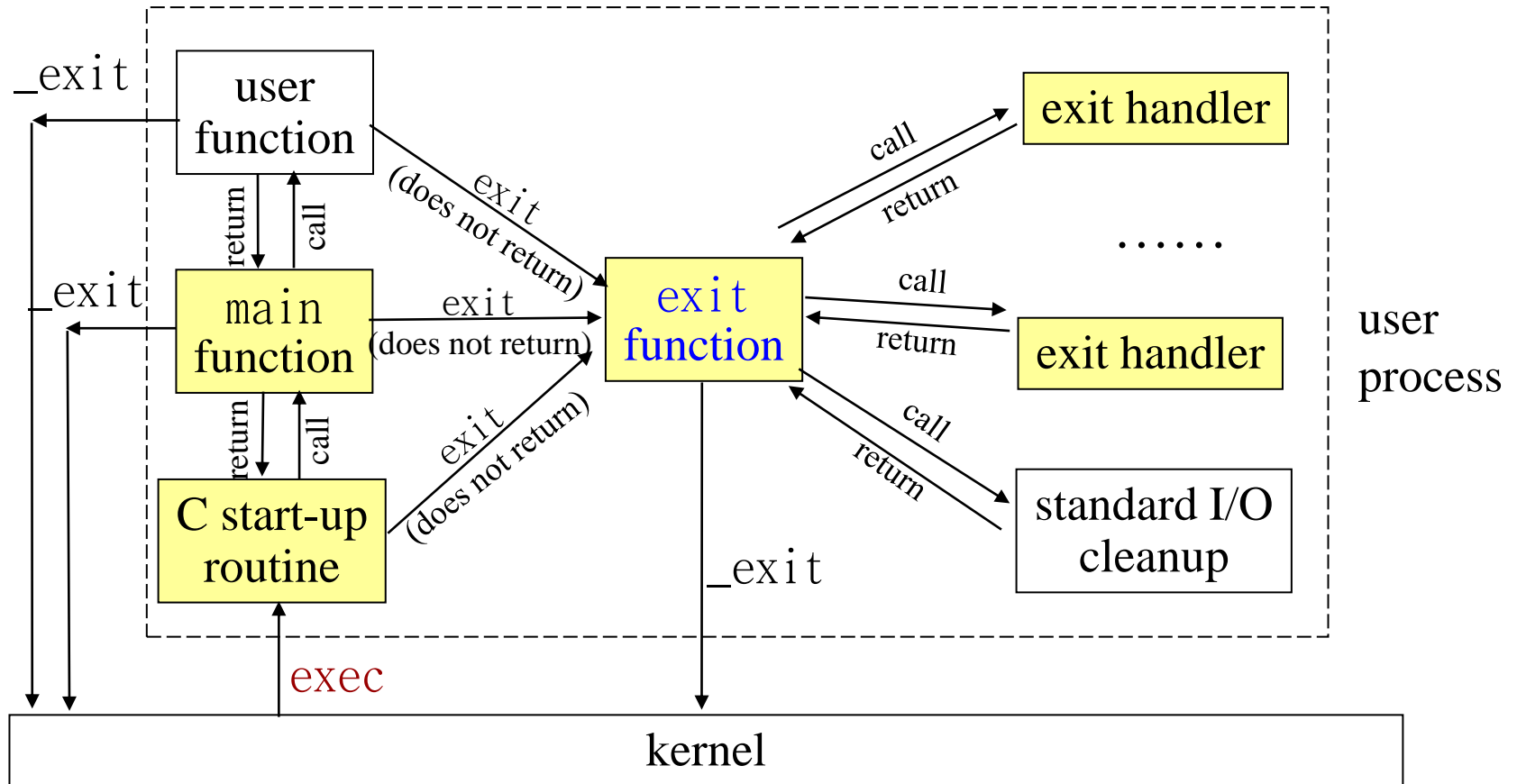
```
#include <stdlib.h>
void exit(int status);
뒷정리를 한 후 프로세스를 정상적으로 종료시킨다.
```

- `_exit()`

```
#include <stdlib.h>
void _exit(int status);
뒷정리를 하지 않고 프로세스를 즉시 종료시킨다.
```



# C 프로그램 시작 및 종료



# atexit()

---

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

```
int atexit(void (*func)(void));
```

returns: 0 if OK, nonzero on error

- **exit 처리기를 등록한다**
  - 프로세스당 32개까지
- **func**
  - exit 처리기
  - 함수 포인터(이름)
- **exit() 는 exit handler 들을 등록된 역순으로 호출한다**

# exit 처리기 예

```
#include <stdio.h>
#include <stdlib.h>
void exit_handler1(void) {
    printf("첫 번째 exit 처리기\n");
}

void exit_handler2(void) {
    printf("두 번째 exit 처리기\n");
}

int main(void) {
    if (atexit(exit_handler1) != 0)
        perror("exit_handler1 등록할 수 없음");
    if (atexit(exit_handler2) != 0)
        perror("exit_handler2 등록할 수 없음");
    printf("main 끝\n");
    exit(0);    // return 0;
}
```

```
$ ./a.out
main 끝
두 번째 exit 처리기
첫 번째 exit 처리기
```

## 8.4 프로세스 ID

# 프로세스 ID

---

- 각 프로세스는 프로세스를 구별하는 번호인 프로세스 ID를 갖는다.
- 각 프로세스는 자신을 생성해준 부모 프로세스가 있다.

`int getpid( );`    프로세스의 ID를 리턴한다.

`int getppid( );`    부모 프로세스의 ID를 리턴한다.

# pid.c

---

```
#include <stdio.h>
/* 프로세스 번호를 출력한다. */
int main()
{
    int pid;
    printf("나의 프로세스 번호 : [%d] \n", getpid());
    printf("내 부모 프로세스 번호 : [%d] \n", getppid());
}
```

```
$ ./a.out
나의 프로세스 번호 : [8414]
내 부모 프로세스 번호 : [2942]
$ ps
  PID TTY          TIME CMD
 2942 pts/0    00:00:00 bash
 8421 pts/0    00:00:00 ps
```

# 프로세스의 사용자 ID

- 프로세스는 프로세스 ID 외에
- 프로세스의 사용자 ID와 그룹 ID를 갖는다.
  - 그 프로세스를 실행시킨 사용자의 ID와 사용자의 그룹 ID
  - 프로세스가 수행할 수 있는 권한을 결정하는 데 사용된다.

/usr/bin/passwd 명령어는 /etc/passwd 파일에 있는 내용을 변경한다.

```
$ ls -l /etc/passwd
```

```
-rw-r--r--. 1 root root
```

```
$ ls -l /usr/bin/passwd
```

```
-rwSr-xr-x. 1 root root
```

root가 아닌 사용자는 /etc/passwd 파일을 변경할 권한이 없다.

passwd 명령어는 set-user-id 이 설정되어 있다.

이런 파일을 실행하면 프로세스의 **유효 사용자 ID**는 root가 되고 root 권한을 가진다. root가 수정할 수 있는 파일을 수정 가능해진다.

# 프로세스의 사용자 ID

---

- 프로세스의 **실제 사용자 ID(real user ID)**
  - 그 프로세스를 실행한 원래 사용자의 사용자 ID로 설정된다.
  - 예를 들어 chang이라는 사용자 ID로 로그인하여 어떤 프로그램을 실행시키면 그 프로세스의 실제 사용자 ID는 chang이 된다.
- 프로세스의 **유효 사용자 ID(effective user ID)**
  - 현재 유효한 사용자 ID로 파일의 접근 권한을 검사할 때 주로 사용
  - 유효 사용자 ID와 실제 사용자 ID는 **특별한 실행파일**을 실행할 때를 제외하고는 동일하다.



# 프로세스의 실제/유효 사용자/그룹 ID 반환 함수

---

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

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

`uid_t getuid( );`      프로세스의 실제 사용자 ID를 반환한다.

`uid_t geteuid( );`      프로세스의 유효 사용자 ID를 반환한다.

`uid_t getgid( );`      프로세스의 실제 그룹 ID를 반환한다.

`uid_t getegid( );`      프로세스의 유효 그룹 ID를 반환한다.

# uid.c

```
#include <stdio.h>
#include <pwd.h>
#include <grp.h>
```

```
/* 사용자 ID를 출력한다. */
```

```
int main()
{
```

```
    int pid;
```

```
    printf("나의 실제 사용자 ID : %d(%s) \n", getuid\(\), getpwuid(getuid())->pw_name);
    printf("나의 유효 사용자 ID : %d(%s) \n", geteuid\(\), getpwuid(geteuid())->pw_name);
    printf("나의 실제 그룹 ID : %d(%s) \n", getgid\(\), getgrgid(getgid())->gr_name);
    printf("나의 유효 그룹 ID : %d(%s) \n", getegid\(\), getgrgid(getegid())->gr_name);
```

```
}
```

```
struct passwd {
    char *pw_name;      /* username */
    char *pw_passwd;    /* user password */
    uid_t pw_uid;       /* user ID */
    gid_t pw_gid;       /* group ID */
    char *pw_gecos;     /* user information */
    char *pw_dir;       /* home directory */
    char *pw_shell;     /* shell program */
};

struct group {
    char *gr_name;      /* group name */
    char *gr_passwd;    /* group password */
    gid_t gr_gid;       /* group ID */
    char **gr_mem;      /* group members */
};
```

# 프로세스의 사용자 ID

---

- 프로세스의 실제/유효 사용자 ID 변경
- 프로세스의 실제/유효 그룹 ID 변경

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

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

```
int setuid(uid_t uid);
```

프로세스의 실제 사용자 ID를 uid로 변경한다.

```
int seteuid(uid_t uid);
```

프로세스의 유효 사용자 ID를 uid로 변경한다.

```
int setgid(gid_t gid);
```

프로세스의 실제 그룹 ID를 gid로 변경한다.

```
int setegid(gid_t gid);
```

프로세스의 유효 그룹 ID를 gid로 변경한다.

# set-user-id 실행파일

---

- 특별한 실행권한 set-user-id(set user ID upon execution)
  - set-user-id 설정된 실행파일을 실행하면
  - 이 프로세스의 유효 사용자 ID는 그 실행파일의 소유자로 바뀜.
  - 이 프로세스는 실행되는 동안 그 파일의 소유자 권한을 갖게 됨.
- 예 : /usr/bin/passwd 명령어
  - set-user-id 실행권한이 설정된 실행파일이며 소유자는 root
  - 일반 사용자가 이 파일을 실행하게 되면 이 프로세스의 유효 사용자 ID는 root가 됨.
  - /etc/passwd처럼 root만 수정할 수 있는 파일의 접근 및 수정 가능

# set-user-id 실행파일

---

- set-user-id 실행권한은 심볼릭 모드로 's'로 표시

```
$ ls -asl /bin/su /usr/bin/passwd
```

```
32 -rwsr-xr-x. 1 root root 32396 2011-05-31 01:50 /bin/su
```

```
28 -rwsr-xr-x. 1 root root 27000 2010-08-22 12:00 /usr/bin/passwd
```

- set-user-id 실행권한 설정

```
$ ls -l a.out
```

```
-rwxrwxr-x. 1 yunis yunis 8464 4월 18 12:56 a.out
```

```
$ chmod 4775 a.out
```

```
$ ls -l a.out
```

```
-rwsrwxr-x. 1 yunis yunis 8464 4월 18 12:56 a.out
```

- set-group-id 비트 설정

```
$ chmod 2755 file1
```

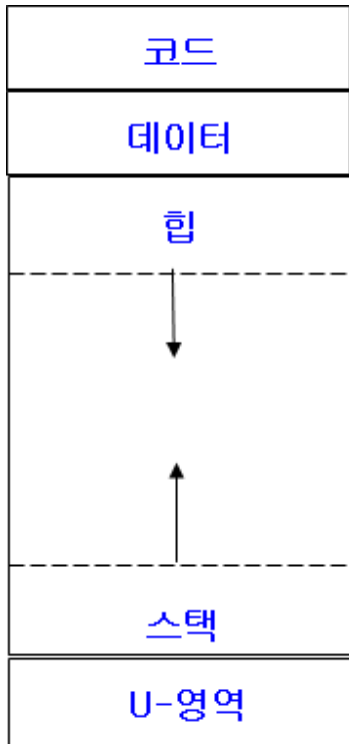
## 8.5 프로세스 구조

# 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 구조



- 텍스트(text)
  - 프로세스가 실행하는 실행코드를 저장하는 영역
- 데이터 (data)
  - 전역 변수(global variable) 및 정적 변수(static variable)를 위한 메모리 영역
- 힙(heap)
  - 동적 메모리 할당을 위한 영역이다. C 언어의 malloc 함수를 호출하면 이 영역에서 동적으로 메모리를 할당해준다.
- 스택(stack area)
  - 함수 호출을 구현하기 위한 실행시간 스택(runtime stack)을 위한 영역으로 지역변수, 매개변수, 반환주소 등이 저장된다.
- U-영역(user-area)
  - 열린 파일 디스크립터, 현재 작업 디렉터리 등과 같은 프로세스의 정보를 저장하는 영역이다.