8장 프로세스

8.1 쉘과 프로세스

프로세스란?

- 프로세스
  - 실행 중인 상태의 프로그램
  - 동일한 프로그램으로 여러 개의 프로세스를 생성할 수 있다.
    - 각 프로세스를 프로그램의 인스턴스(instance)라고 한다.
- 프로그램
  - 파일 형태로 저장.
  - 명령어 코드

- 사용자가 프로세스를 생성하는 방법
  - 쉘(shell) 프롬프트 상에서 프로그램을 지정하여 실행
  - 실행 중인 사용자 프로세스를 통해서 프로그램을 실행

  - 위의 두 가지 방법은 사실 동일함

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 -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 -elf (시스템 V)
  - - e: 모든 사용자 프로세스 정보를 출력
  - - f: 프로세스에 대한 좀 더 자세한 정보를 출력

실행 중인 프로세스의 목록 확인하기

- "ps" 명령을 사용하여 실행 중인 프로세스의 목록을 확인

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
kimyh     2261  2260  0 Nov18 pts/2    00:00:00 -bash
kimyh     6319  2261  0 00:40 pts/2    00:00:00 ps -f
$
```

- 사용자가 소유주인 프로세스는 2개
  - "bash"는 쉘 프로세스이고, "ps –f"는 현재의 프로세스 목록을 보여주는 프로세스
  - "ps –f" 프로세스는 출력을 보여주는 시점에서는 존재했으나, 출력이 끝난 지금은 아마 종료되어 없을 것이다.

실행 중인 프로세스의 목록 확인하기

```
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
kimyh     2261  2260  0 Nov18 pts/2    00:00:00 -bash
kimyh     6319  2261  0 00:40 pts/2    00:00:00 ps -f
$
```

- UID (User ID) : 프로세스 소유주의 사용자 식별 번호이다.
- PID (Process ID) : 프로세스의 식별 번호이다.
- PPID (Parent Process ID) : 부모 프로세스의 식별 번호이다.
  - 모든 프로세스는 자신을 생성해준 부모 프로세스가 있다.
    - 단 한 개 예외가 있다. (init 프로세스)
  - 대부분의 사용자 프로세스는 쉘 프로세스가 부모 프로세스가 된다.

    위의 예에서 "ps –f"의 PPID는 "bash"의 PID와 같다.

프로세스의 가계도 확인해보기

- 모든 프로세스는 최초의 “init” 프로세스부터 시작된다.
- “ps” 명령과 “grep” 명령을 사용하여 init 프로세스까지 거슬러 가본다.

프로세스 확인

[예제] 동일한 계정을 사용하여 두 개의 터미널로 접속

```
$ tty
/dev/pts/2
$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
kimyh     2261  2260  0 Nov18 pts/2    00:00:00 -bash
kimyh     8528  2261  0 01:14 pts/2    00:00:00 ps -f
$ ps -ef | grep 'kimyh'
kimyh     2261  2260  0 Nov18 pts/2    00:00:00 -bash
kimyh     8251  8250  0 01:11 pts/3    00:00:00 -bash
kimyh     8535  2261  0 01:14 pts/2    00:00:00 ps -ef
kimyh     8536  2261  0 01:14 pts/2    00:00:00 grep kimyh
$ kill -9 8251
$ ps -ef | grep 'kimyh'
kimyh     2261  2260  0 Nov18 pts/2    00:00:00 -bash
kimyh     8946  2261  0 01:19 pts/2    00:00:00 ps -ef
kimyh     8947  2261  0 01:19 pts/2    00:00:00 grep kimyh
$
```

```
[2]$ tty
/dev/pts/3
[2]$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
kimyh     8251  8250  0 01:11 pts/3    00:00:00 -bash
kimyh     8295  8251  0 01:11 pts/3    00:00:00 ps -f
[2]$
```

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

exit

- exit
  - 쉘을 종료하고 종료코드(exit code)을 부모 프로세스에 전달

$ exit [종료코드]

8.2 프로그램 시작

프로그램 실행 시작

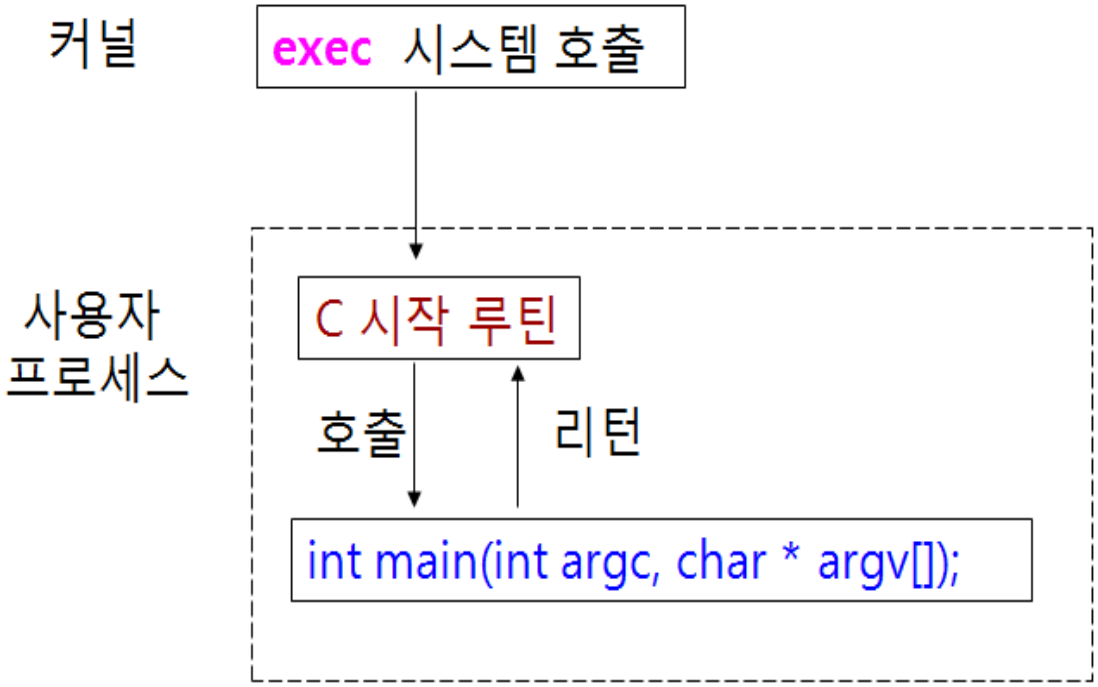
- exec 시스템 호출
  - C 시작 루틴에 명령줄 인수와 환경 변수를 전달하고
  - 프로그램을 실행시킨다.

- 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

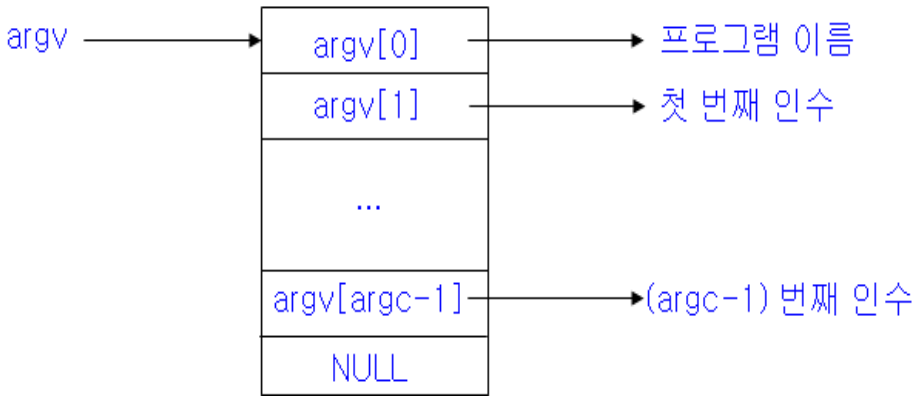


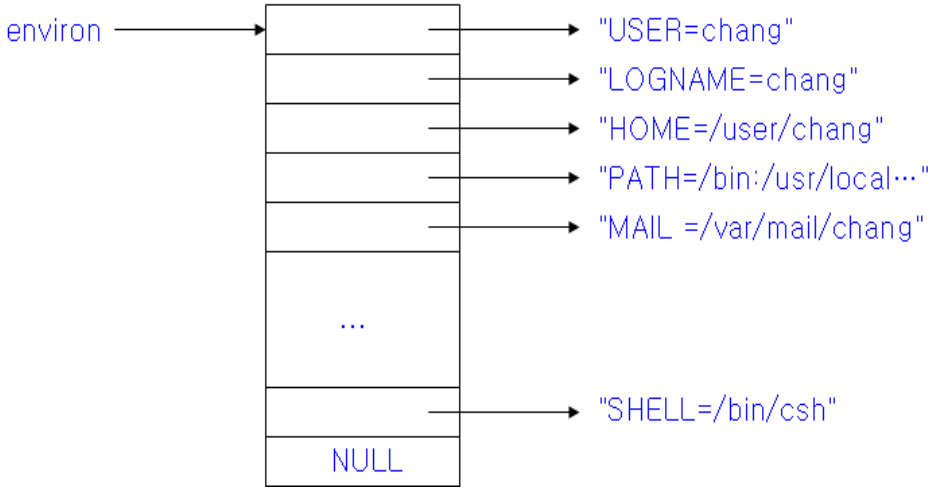
명령줄 인수/환경 변수

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 0이면 그대로 둔다.

int unsetenv(const char *name);

환경 변수 name의 값을 지운다.

8.3 프로그램 종료

프로그램 종료

- 정상 종료(normal termination)
  - main() 실행을 마치고 리턴하면 C 시작 루틴은 이 리턴값을 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.
```

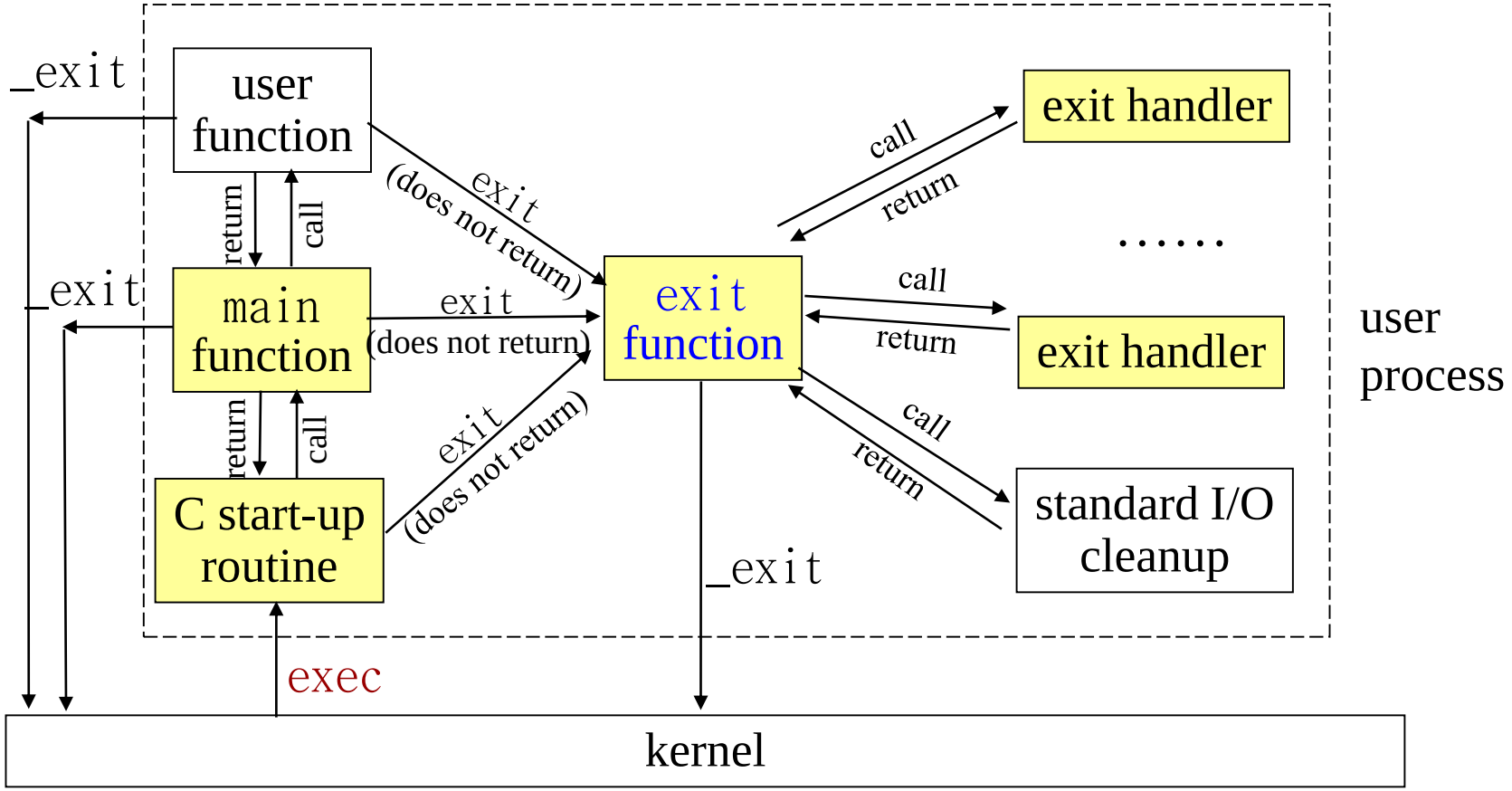
atexit()

```
#include <stdlib.h>

void 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

C 프로그램 시작 및 종료



exit 처리기 예

```
1  #include <stdio.h>
2  static void my_exit1(void),
3                  my_exit2(void);
4  int main(void) {
5  if (atexit(exit_handler1) != 0)
6     perror("exit_handler1 등록할 수 없음");
7  if (atexit(exit_handler2) != 0)
8     perror("exit_handler2 등록할 수 없음");
9   printf("main 끝 ₩n");
10 exit(0);
11 }
12
13 static void exit_handler1(void) {
14     printf("첫 번째 exit 처리기₩n");
15 }
16
17 static void exit_handler2(void) {
18     printf("두 번째 exit 처리기₩n");
19 }
```

```
$atexit
main 끝
첫 번째 exit 처리기
두 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

```
1 #include <stdio.h>
2
3 /* 프로세스 번호를 출력한다. */
4 int main()
5 {
6       int pid;
7       printf("나의 프로세스 번호 : [%d] ₩n", getpid());
8       printf("내 부모 프로세스 번호 : [%d] ₩n", getppid());
9 }
```

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 연산을 결정하는 데 사용된다.

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 새로 파일을 만들 때나 파일에 대한 접근 권한을 검사할 때 주로 사용된다.
  - 보통 유효 사용자 ID와 실제 사용자 ID는 특별한 실행파일을 실행할 때를 제외하고는 동일하다.

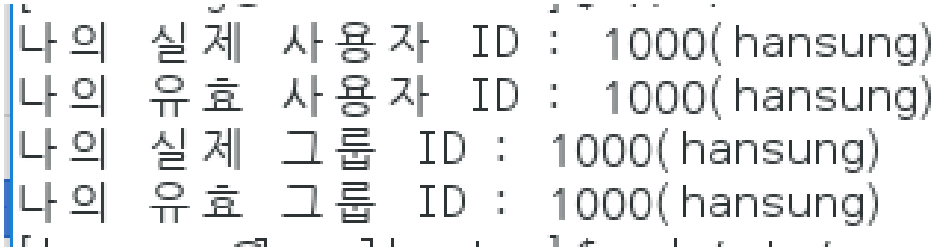
프로세스의 사용자 ID

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

```
#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);
}
```

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.
(단, 주로 root권한으로 실행할 때만 가능. ruid와 euid가 모두 바뀜)
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.
```

setuid, seteuid 예제

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

int main(int argc, char *argv[])
{
        uid_t uid, euid;
        int status;
        int fd;

        if(argc < 3)
        {
                printf("Usage: a.out pid fname\n");
                exit(1);
        }

        uid = getuid();
        euid = geteuid();

        printf("uid = %d, euid = %d\n", uid, euid);

        status = setuid(atoi(argv[1]));
        // try seteuid instead of setuid

        if (status < 0) {
            fprintf (stderr, "Couldn't set uid.\n");
            exit (status);
        }

        uid = getuid();
        euid = geteuid();

        printf("uid = %d, euid = %d\n", uid, euid);

        fd = creat(argv[2], 0644);
        if(fd <= -1){
                printf("File Open Error\n");
                exit(1);
        }

        return 0;
}
```

실행결과

- hansung(1000)으로 실행한 경우

```
[hansung@localhost uid]$ ./a.out 0 file1
uid = 1000, euid = 1000
Couldn't set uid.
[hansung@localhost uid]$ ./a.out 1001  file1
uid = 1000, euid = 1000
Couldn't set uid.
[hansung@localhost uid]$ ./a.out 1000  file1
uid = 1000, euid = 1000
uid = 1000, euid = 1000
[hansung@localhost uid]$ ls -al
total 24
drwxrwxr-x.  2 hansung hansung    61 Sep  3 00:00  .
drwx------. 17 hansung hansung  4096 Sep  2 23:48  ..
-rwxrwxr-x.  1 hansung hansung 13200 Sep  3 00:00  a.out
-rw-r--r--.  1 hansung hansung     0 Sep  2 23:59 'argv[2]'
-rw-r--r--.  1 hansung hansung     0 Sep  3 00:00  file1
-rw-rw-r--.  1 hansung hansung   687 Sep  3 00:00  test.c
[hansung@localhost uid]$
```

실행결과

- root(0)로 실행한 경우

```
[root@localhost uid]# ./a.out 0 file2
uid = 0, euid = 0
uid = 0, euid = 0
[root@localhost uid]# ls -al
total 24
drwxrwxr-x.  2 hansung hansung    59 Sep  3 00:04 .
drwx------. 17 hansung hansung  4096 Sep  3 00:04 ..
-rwxrwxr-x.  1 hansung hansung 13200 Sep  3 00:00 a.out
-rw-r--r--.  1 hansung hansung     0 Sep  3 00:00 file1
-rw-r--r--.  1 root    root        0 Sep  3 00:04 file2
-rw-rw-r--.  1 hansung hansung   687 Sep  3 00:00 test.c
[root@localhost uid]# ./a.out 1000  file3
uid = 0, euid = 0
uid = 1000, euid = 1000
[root@localhost uid]# ls -al
total 24
drwxrwxr-x.  2 hansung hansung    72 Sep  3 00:04 .
drwx------. 17 hansung hansung  4096 Sep  3 00:04 ..
-rwxrwxr-x.  1 hansung hansung 13200 Sep  3 00:00 a.out
-rw-r--r--.  1 hansung hansung     0 Sep  3 00:00 file1
-rw-r--r--.  1 root    root        0 Sep  3 00:04 file2
-rw-r--r--.  1 hansung root        0 Sep  3 00:04 file3
-rw-rw-r--.  1 hansung hansung   687 Sep  3 00:00 test.c
[root@localhost uid]# ./a.out 1001  file4
uid = 0, euid = 0
uid = 1001, euid = 1001
File Open Error
[root@localhost uid]#
```

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-uid 실행권한 설정

```
$ chmod 4755 file1
```

실행 예)

$ su
암호:
# chown root uid
# chmod 4755 uid
# exit
$ uid

```
[hansung@localhost ~]$ ./uid
나의 실제 사용자 ID : 1000(hansung)
나의 유효 사용자 ID : 0(root)
나의 실제 그룹 ID : 1000(hansung)
나의 유효 그룹 ID : 1000(hansung)
```

권한상승실습

```
#include <stdio.h>
#include <unistd.h>
int main()
{
        printf("uid :: %d euid :: %d\n", getuid(), geteuid());
        setuid(1001);
        printf("uid :: %d euid :: %d\n", getuid(), geteuid());
}
```

```
hansung:x:1000:1000:hansung:/home/hansung:/bin/bash
guest:x:1001:1001::/home/guest:/bin/bash
```

실행 예

- 실행 파일 소유주: root 실행 : root

```
[root@localhost hansung]# su root
[root@localhost hansung]# chown root setuid
[root@localhost hansung]# ls -al setuid
-rwxrwxr-x. 1 root hansung 8672 11월  1 22:23 setuid
[root@localhost hansung]# ./setuid
uid :: 0 euid :: 0
uid :: 1001 euid :: 1001
```

- 실행 파일 소유주: root 실행 : hansung(1000)

```
[hansung@localhost ~]$ ./setuid
uid :: 1000 euid :: 1000
uid :: 1000 euid :: 1000
```

  - uid와 euid가 바뀌지 않음(프로세스의 uid는 root권한으로만 바꿀 수 있음)

실행 예

- 실행 파일 소유주: root SetUIDbit: 설정 실행 : 1000(hansung)

```
[root@localhost hansung]# chmod 4777 setuid
[root@localhost hansung]# ls -al setuid
-rwsrwxrwx. 1 root hansung 8672 11월  1 22:23 setuid
[root@localhost hansung]# su hansung
[hansung@localhost ~]$ ./setuid
uid :: 1000 euid :: 0
uid :: 1001 euid :: 1001
```

  - euid가 root이므로 프로세스의 owner변경가능

실행 예

- 실행 파일 소유주: guest(1001) SetUIDbit: 실행 : hansung(1000)

```
[hansung@localhost ~]$ su guest
암호:
[guest@localhost hansung]$ cd ~
[guest@localhost ~]$ cp ~hansung/setuid setuid
[guest@localhost ~]$ ls -al setuid
-rwxrwxrwx. 1 guest guest 8672 11월  1 22:54 setuid
[guest@localhost ~]$ chmod 4777 setuid
[guest@localhost ~]$ ls -al setuid
-rwsrwxrwx. 1 guest guest 8672 11월  1 22:54 setuid
[guest@localhost ~]$ exit
exit
[hansung@localhost ~]$ ~guest/setuid
uid :: 1000 euid :: 1001
uid :: 1000 euid :: 1001
```

  - 일반 사용자이므로 setuid를 실행 못함

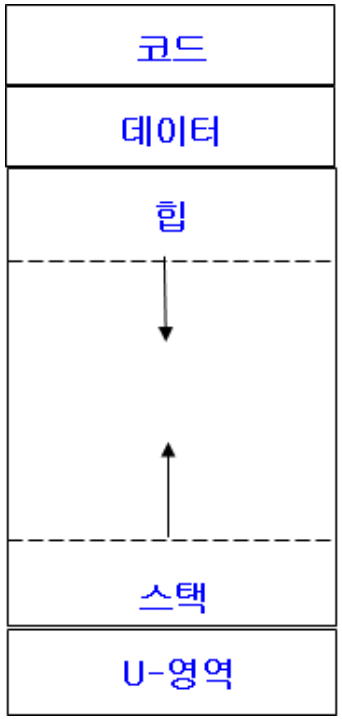
8.5 프로세스 구조

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

- 프로세스 구조



프로세스 구조

- 텍스트(text)
  - 프로세스가 실행하는 실행코드를 저장하는 영역이다.
- 데이터 (data)
  - 전역 변수(global variable) 및 정적 변수(static variable)를 위한 메모리 영역이다.
- 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 활성 레코드(activation record)가 저장된다.
- U-영역(user-area)
  - 열린 파일 디스크립터, 현재 작업 디렉터리 등과 같은 프로세스의 정보를 저장하는 영역이다.

U area

- 프로세스가 scheduler에 의해 CPU를 할당받아 실행될 때 필요한 정보
- 예
  - 프로세스 테이블에 대한 포인터
  - 실제 사용자 ID, 유효 사용자 ID
  - 현재 디렉토리와 루트디렉토리
  - 사용자 file descriptor table

Process table

- 실행중인 프로세스 리스트를 ㅋ터널에서 관리하는 테이블
- 엔트리 하나당 하나의 프로세스 정보가 관리
- 예
  - 상태정보, 프로세스의 크기
  - Region 테이블을 가리키는 포인터 정보와 U Area가 가리키는 포인터
  - 사용자의 ID또는 GID
  - 프로세스 번호 (PID
  - Signal 필드