

# Linux Programming

## 8장. 프로세스

---

sisong@ut.ac.kr

한국교통대학교 컴퓨터공학전공

송석일

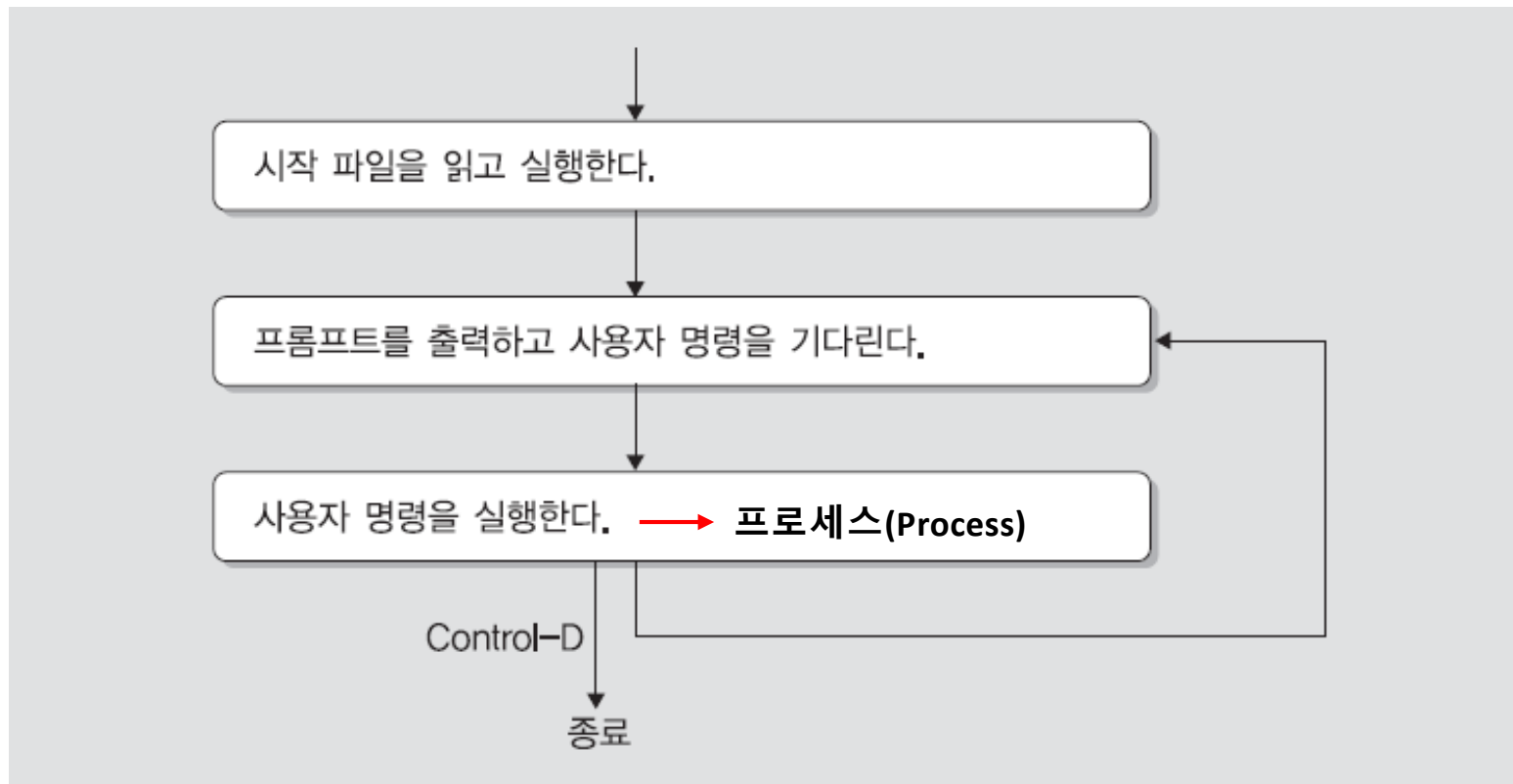


## 8.1 쉘과 프로세스

# 셸(Shell)이란 무엇인가?

## ■ 셸의 역할

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



# 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 전면에서 프로세스가 실행되며 실행이 종료될때 까지 쉘이 기다림

- 후면 처리

- 명령어를 입력하면 후면에서 프로세스가 실행되며 전면에서는 프로세스가 종료되기 전에 다른 명령어 실행이 가능함
- \$ 명령어 &

# 후면 처리 예

---

- 후면처리 명령

```
$ (sleep 100; echo done) &  
[1] 8320
```

```
$ find . -name test.c -print &  
[2] 8325
```

```
$ jobs  
[1]- 실행중 ( sleep 100; echo done )  
[2]+ 완료 find . -name test.c -print
```

- fg %작업번호 // 해당 후면 작업을 전면 작업으로 전환한다.

```
$ fg %1  
( sleep 100; echo done )
```

- 후면처리 입출력

```
$ find . -name test.c -print > find.txt &  
$ find . -name test.c -print | mail lect &  
$ wc < inputfile &
```

# 프로세스(process)

## ■ 개요

- 실행중인 프로그램을 프로세스(process)라고 함
- 각 프로세스는 유일한 프로세스 번호 PID를 갖는다.
- ps 명령어를 사용하여 현재 쉘 및 리눅스 운영체제 전체의 프로세스 확인 가능

\$ ps

```
PID    TTY  TIME  CMD
31745 pts/0 00:00:00 bash
59887 pts/0 00:00:00 ps
```

\$ ps u

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
lect	1517	0.0	0.2	174124	6748	tty2	Ssl+	10월12	0:00	gdm-x-session
lect	1521	0.0	1.8	266776	55496	tty2	Sl+	10월12	0:57	Xorg ...
lect	1557	0.0	0.4	198524	13976	tty2	Sl+	10월12	0:00	gnome-session
lect	31745	0.0	0.1	21380	5572	pts/0	Ss	10월13	0:01	bash
lect	59455	0.0	0.1	21436	3348	pts/0	R+	01:44	0:00	ps u

# 프로세스 상태: ps

---

- ps [-옵션] 명령어

- 현재 존재하는 프로세스들의 실행 상태를 요약해서 출력

```
$ ps
```

```
PID      TTY  TIME  CMD
```

```
31745 pts/0 00:00:00 bash
```

```
59887 pts/0 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



# 프로세스 상태: ps

---

```
$ ps -ef | more
```

```
UID PID PPID C STIME TTY TIME  CMD
root 1  0  0 9월30 ? 00:00:23 /sbin/init auto noprompt
root 2  0  0 9월30 ? 00:00:00 [kthreadd]
root 3  2  0 9월30 ? 00:00:00 [rcu_gp]
root 4  2  0 9월30 ? 00:00:00 [rcu_par_gp]
root 6  2  0 9월30 ? 00:00:00 [kworker/0:0H-events_highpri]
root 9  2  0 9월30 ? 00:00:00 [mm_percpu_wq]
root 10 2  0 9월30 ? 00:00:00 [rcu_tasks_rude_]
root 11 2  0 9월30 ? 00:00:00 [rcu_tasks_trace]
root 12 2  0 9월30 ? 00:00:03 [ksoftirqd/0]
root 13 2  0 9월30 ? 00:01:07 [rcu_sched]
root 14 2  0 9월30 ? 00:00:02 [migration/0]
...
root 24 2  0 9월30 ? 00:00:00 [netns]
root 25 2  0 9월30 ? 00:00:00 [inet_frag_wq]
root 26 2  0 9월30 ? 00:00:00 [kauditd]
--More--
```

# 프로세스 제어

---

- 프로세스들을 제어하기 위한 명령어들
  - sleep, kill, wait, exit 명령어 등
- sleep 명령어
  - 지정된 시간만큼 실행을 중지

\$ sleep 초

\$ (echo 시작; sleep 5; echo 끝)

# kill

---

- kill

- 현재 실행중인 프로세스를 강제로 종료

```
$ kill [-시그널] 프로세스번호  
$ kill %작업번호
```

```
$ (sleep 100; echo done) &  
[1] 8320
```

```
$ kill 8320                    혹은                    $ kill %1  
[1]+ 종료됨 ( sleep 100; echo done )
```

# wait

---

- wait [프로세스번호]

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

```
$ (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

---

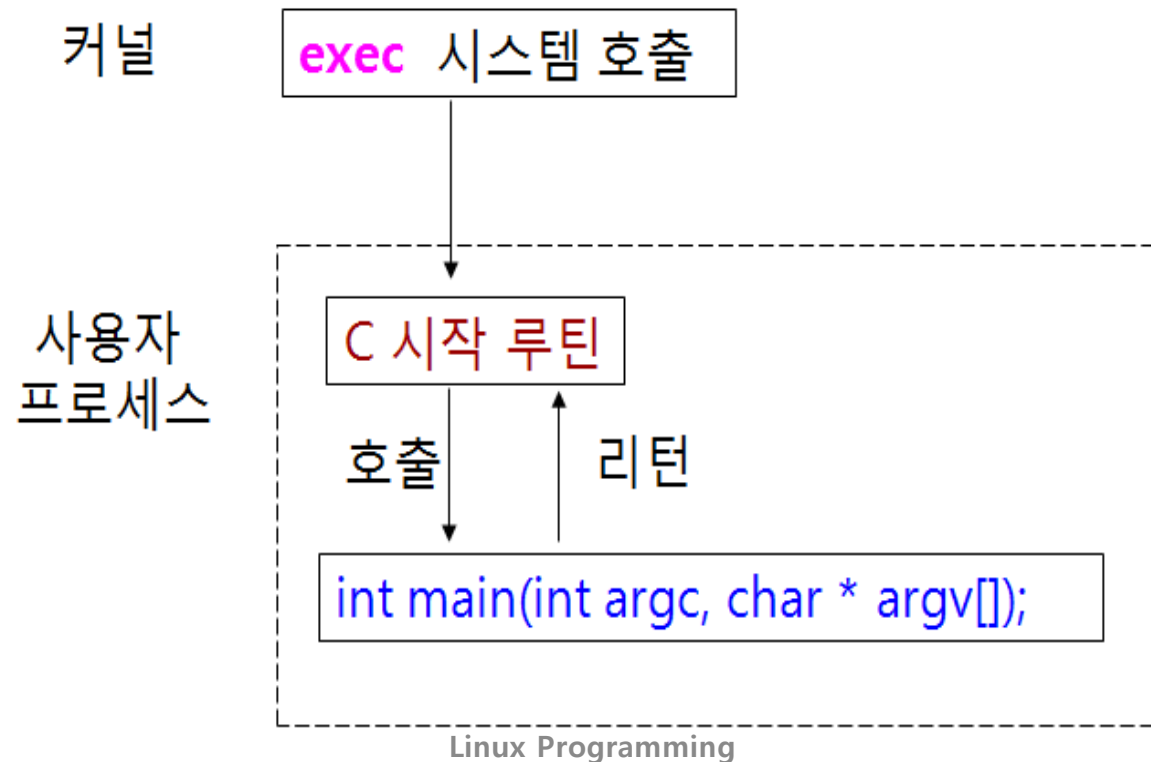
- exit
  - 셸을 종료하고 종료코드(exit code)을 부모 프로세스에 전달

\$ exit [종료코드]

## 8.2 프로그램 실행

# 프로그램 실행 시작

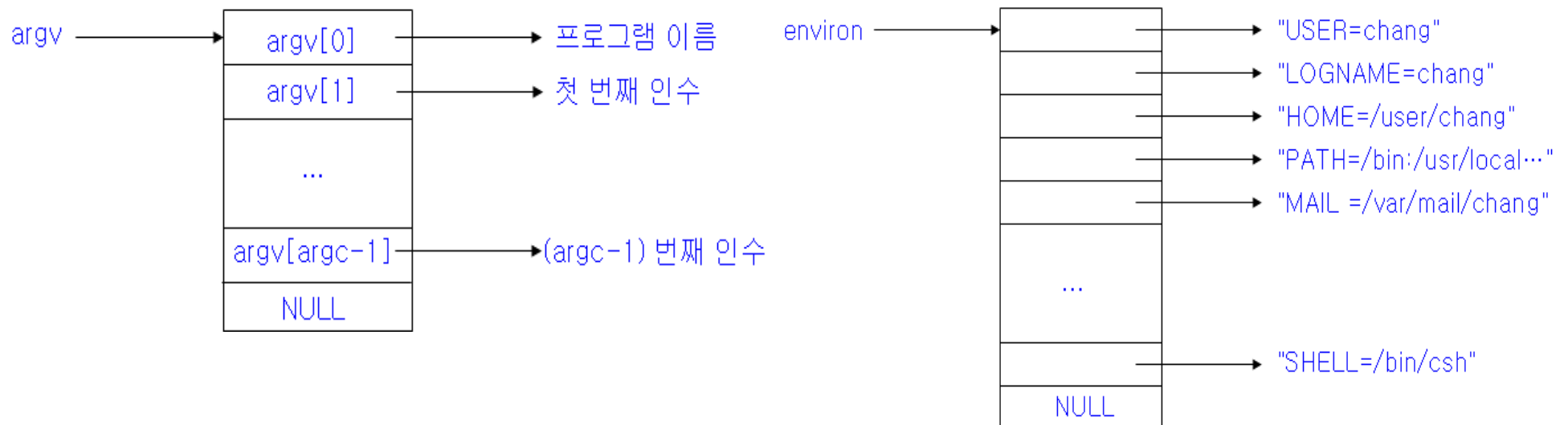
- exec 시스템 호출
  - C 시작 루틴에 명령줄 인수와 환경 변수를 전달하고 프로그램 실행
- C 시작 루틴(start-up routine)
  - main 함수를 호출하면서 명령줄 인수, 환경 변수를 전달
  - 실행이 끝나면 main()에서 반환값을 받아 exit



# 명령줄 인수/환경 변수

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

- argc : 명령줄 인수의 개수
- argv[] : 명령줄 인수 리스트를 나타내는 포인터 배열





# 명령줄 인수 출력: args.c

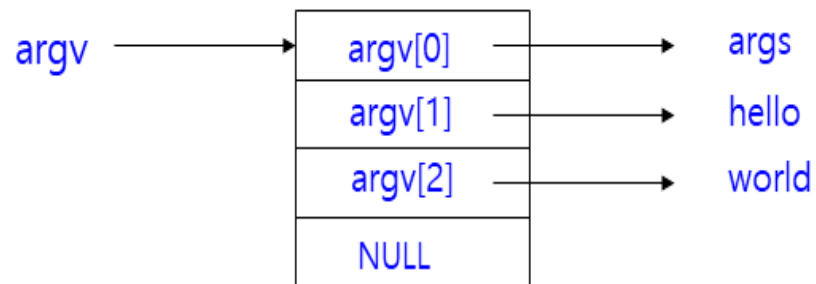
```
1. #include <stdio.h>
2. #include <stdlib.h>

3. /* 모든 명령줄 인수를 출력한다. */
4. int main(int argc, char *argv[])
5. {
6.     int i;

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

9.     exit(0);
10. }
```

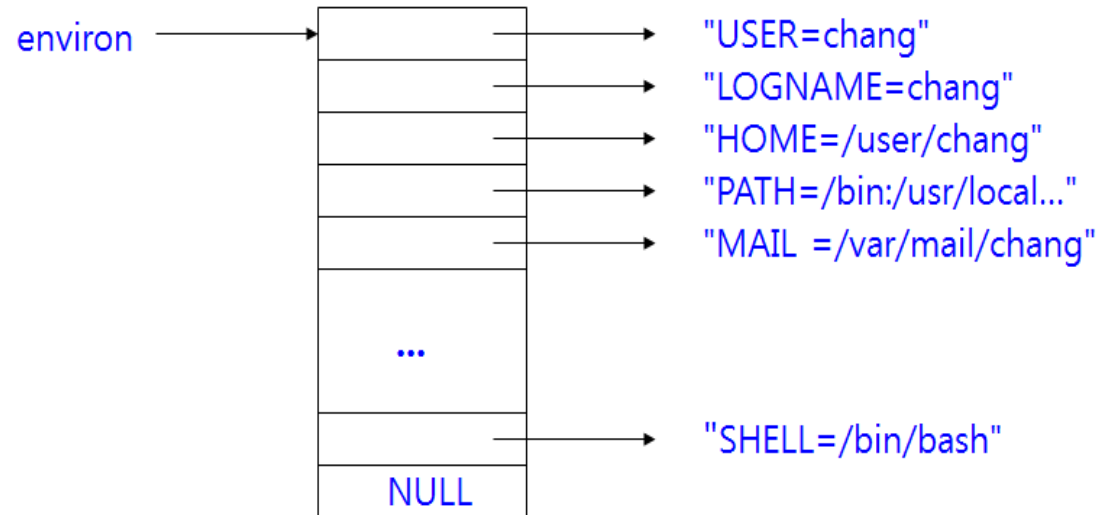
```
$ args hello world
argv[0]: args
argv[1]: hello
argv[2]: world
```



# 환경 변수 출력: environ.c

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. /* 모든 환경 변수를 출력한다. */
4. int main(int argc, char *argv[])
5. {
6.     char **ptr;
7.     extern char **environ;

8.     for (ptr = environ; *ptr != 0; ptr++) /* 모든 환경 변수 값 출력*/
9.         printf("%s \n", *ptr);
10.
11.     exit(0);
12. }
```



```
$ environ
```

```
HOME=/user/faculty/lect
```

```
PATH=./usr/local/bin:/bin:/sbin:/usr/bin:/usr/ucb:/etc:/usr/sbin:/usr/ccs/bin
```

```
...
```

# 환경 변수 접근

---

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

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

- 환경 변수 name의 값을 반환
- 해당 변수가 없으면 NULL을 반환

- 환경 변수를 하나씩 접근

# myenv.c

---

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

3. /* 환경 변수를 3개 프린트한다. */
4. int main(int argc, char *argv[])
5. {
6.     char *ptr;

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

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

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

13.    exit(0);
14. }
```

# 환경 변수 설정 : putenv(), setenv()

---

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

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

- name=value 형태의 스트링을 받아서 이를 환경 변수 리스트에 넣어줌
- name이 이미 존재하면 원래 값을 새로운 값으로 대체

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

- 환경 변수 name의 값을 value로 설정
- name이 이미 존재하는 경우, rewrite 값이 0이 아니면 원래 값을 새로운 값으로 대체하고 rewrite 값이 0이면 그대로 둠

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

- 환경 변수 name의 값을 지움

## 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 → 프로세스간 통신, `kill -9 pid`

# 프로그램 종료

---

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

```
void exit(int status);
```

- 뒷정리를 한 후 프로세스를 정상적으로 종료

- 모든 열린 스트림을 닫고(fclose), 출력 버퍼의 내용을 디스크에 쓰는(fflush) 등의 뒷정리 후 프로세스를 정상적으로 종료
- 종료 코드(exit code)를 부모 프로세스에게 전달

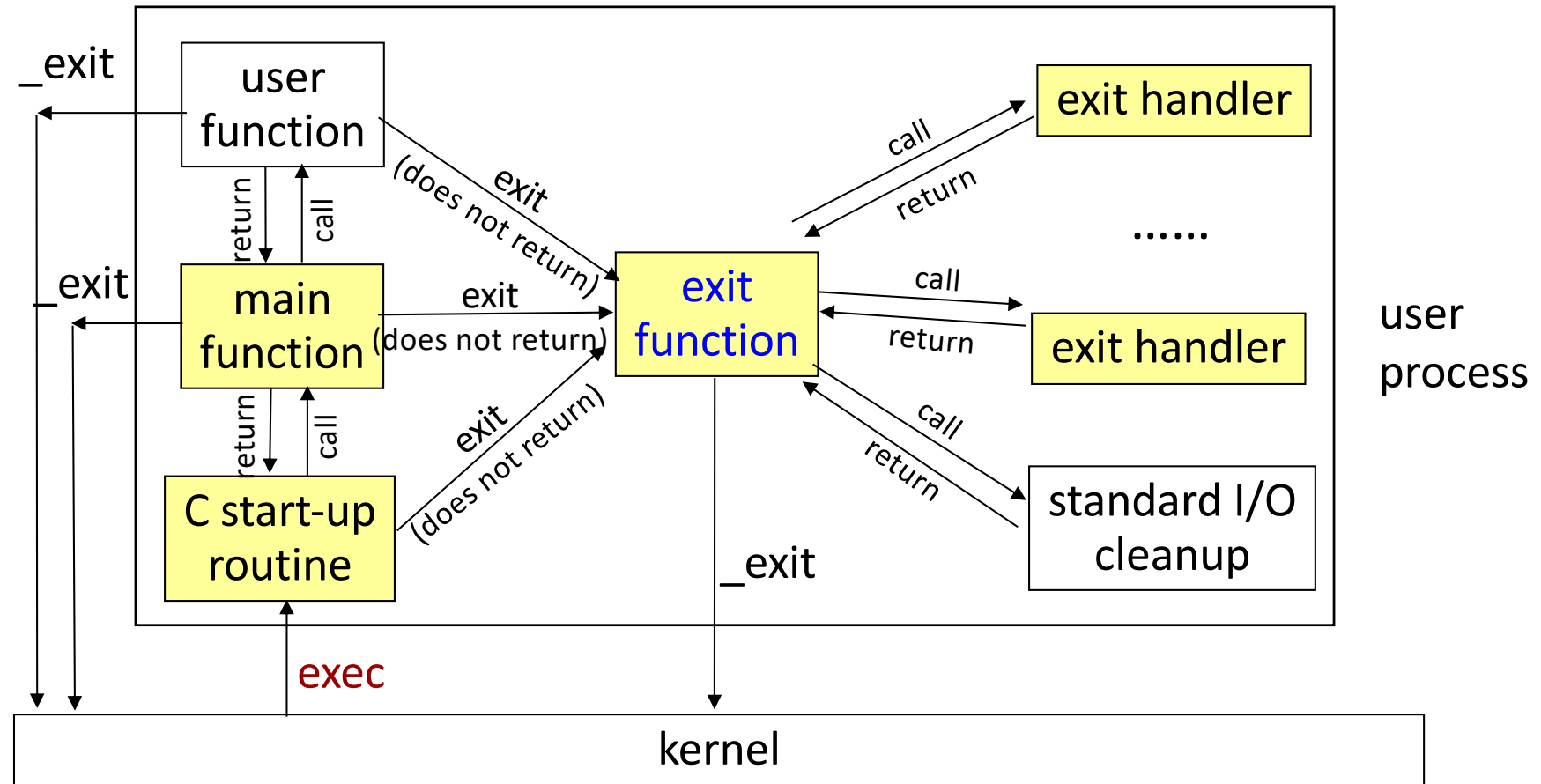
```
#include <stdlib.h>
```

```
void _exit(int status);
```

뒷정리를 하지 않고 프로세스를 즉시 종료



# C 프로그램 시작 및 종료



# exit 처리기 atexit()

---

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

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

- 문제 없으면 0 반환
- 에러가 발생하면 0이 아닌 값 반환

- exit 처리기를 등록 : 프로세스당 32개
- func
  - exit 처리기
  - 함수 포인터(이름)
- exit()는 프로세스 종료 전에 exit handler 들을 등록된 역순으로 호출

# exit 처리기 예

```
1. #include <stdio.h>
2. #include <stdio.h>
3. static void exit_handler1(void), exit_handler2(void);
4. int main(void)
5. {
6.     if (atexit(exit_handler1) != 0)
7.         perror("exit_handler1 등록할 수 없음");
8.     if (atexit(exit_handler2) != 0)
9.         perror("exit_handler2 등록할 수 없음");
10.    printf("main 끝 \n");
11.    exit(0);
12. }

13. static void exit_handler1(void)
14. {
15.    printf("첫 번째 exit 처리기\n");
16. }
17. static void exit_handler2(void)
18. {
19.    printf("두 번째 exit 처리기\n");
20. }
```

```
$ atexit
main 끝
두 번째 exit 처리기
첫 번째 exit 처리기
```

## 8.4 프로세스 ID와 프로세스의 사용자 ID

# 프로세스 ID

---

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

```
int getpid( );
```

- 프로세스의 ID를 반환

```
int getppid( );
```

- 부모 프로세스의 ID를 반환

- 각 프로세스는 프로세스를 구별하는 번호인 프로세스 ID를 가짐
- 각 프로세스는 자신을 생성해준 부모 프로세스가 있음

# 프로세스 번호 출력: pid.c

---

```
1. #include <stdio.h>
2. #include <unistd.h>

3. /* 프로세스 번호를 출력한다. */
4. int main()
5. {
6.     printf("나의 프로세스 번호 : %d \n", getpid());
7.     printf("내 부모 프로세스 번호 : %d \n", getppid());
8. }
```

```
$ pid
나의 프로세스 번호 : [23502]
내 부모 프로세스 번호 : [22692]
```

# 프로세스의 사용자 ID

---

- 프로세스는 프로세스 ID와 프로세스를 실행시킨 사용자 ID, 그룹 ID 속성을 추가로 가짐
  - 프로세스가 수행할 수 있는 연산을 결정하는 데 필요
  - 프로세스의 실제 사용자 ID(real user ID)
    - 프로세스를 실행한 원래 사용자의 사용자 ID
  - 프로세스의 유효 사용자 ID(effective user ID)
    - 현재 유효한 사용자 ID로 새로 파일을 만들 때나 파일에 대한 접근 권한을 검사할 때 주로 사용
    - 보통 유효 사용자 ID와 실제 사용자 ID는 특별한 실행파일을 실행할 때를 제외하고는 동일

# 프로세스의 사용자 ID

---

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

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

```
uid_t getuid( );
```

- 프로세스의 실제 사용자 ID를 반환

```
uid_t geteuid( );
```

- 프로세스의 유효 사용자 ID를 반환

```
uid_t getgid( );
```

- 프로세스의 실제 그룹 ID를 반환

```
uid_t getegid( );
```

- 프로세스의 유효 그룹 ID를 반환



# 프로세스의 사용자 ID

---

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

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

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

- 프로세스의 실제 사용자 ID를 uid로 변경

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

- 프로세스의 유효 사용자 ID를 uid로 변경

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

- 프로세스의 실제 그룹 ID를 gid로 변경

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

- 프로세스의 유효 그룹 ID를 gid로 변경

# 프로세스의 사용자 ID: uid.c

```
1. #include <stdio.h>
2. #include <pwd.h>
3. #include <grp.h>
4. #include <unistd.h>

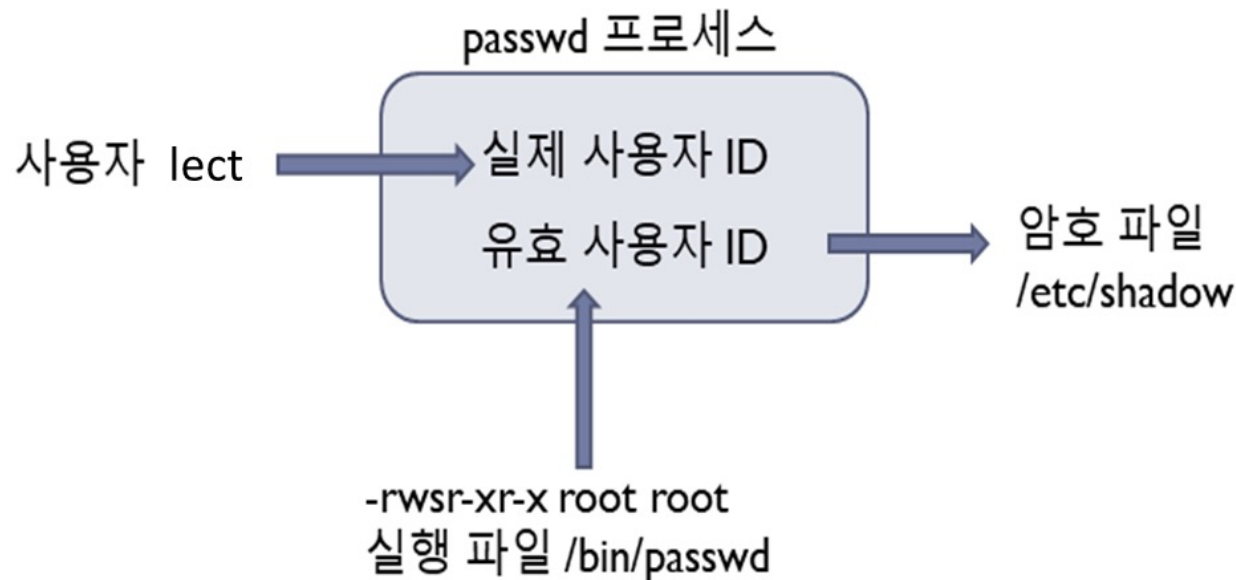
5. /* 사용자 ID를 출력한다. */
6. int main()
7. {
8.     printf("나의 실제 사용자 ID : %d(%s) \n", getuid(), getpwuid(getuid())->pw_name);
9.     printf("나의 유효 사용자 ID : %d(%s) \n", geteuid(), getpwuid(geteuid())->pw_name);
10.    printf("나의 실제 그룹 ID : %d(%s) \n", getgid(), getgrgid(getgid())->gr_name);
11.    printf("나의 유효 그룹 ID : %d(%s) \n", getegid(), getgrgid(getegid())->gr_name);
12. }
```

```
$ uid
나의 실제 사용자 ID : 1000(lect)
나의 유효 사용자 ID : 1000(lect)
나의 실제 그룹 ID : 1000(lect)
나의 유효 그룹 ID : 1000(lect)
```

# set-user-id 실행권한

- set-user-id(set user ID upon execution) : 특별 실행권한
  - 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

```
$ ls -l /bin/passwd  
-rwsr-xr-x 1 root root 59976 Nov 24 21:05 /bin/passwd
```



# set-user-id 실행권한 설정

---

- 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

```
$ chmod 4755 file1
```

- set-group-id 실행권한 설정

```
$ chmod 2755 file1
```

# 실행 예

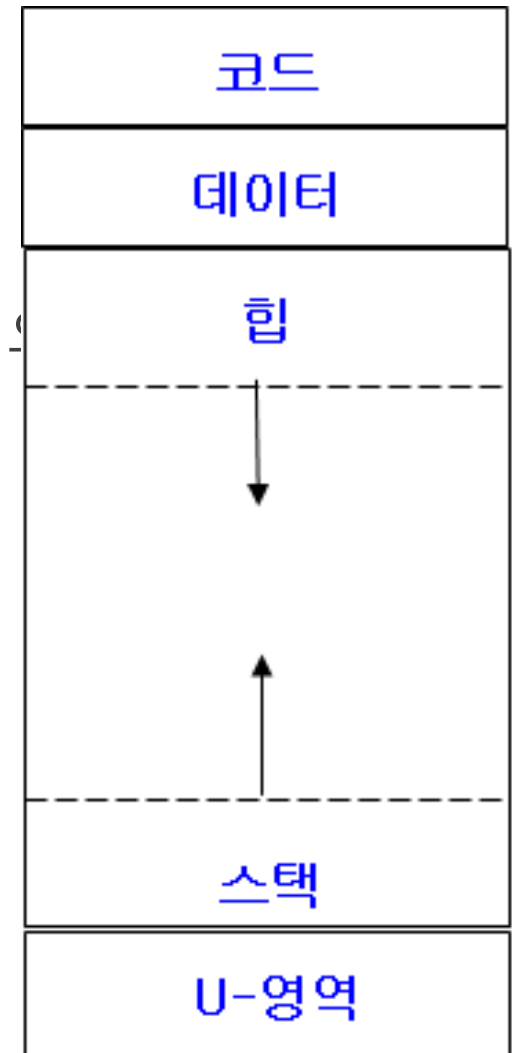
---

```
$ su
# chown root uid
# chmod 4755 uid
# exit
$ uid
나의 실제 사용자 ID : 1000(lect)
나의 유효 사용자 ID : 0(root)
나의 실제 그룹 ID : 1000(lect)
나의 유효 그룹 ID : 1000(lect)
```

## 8.5 프로세스 이미지

# 프로세스

- 프로세스는 실행중인 프로그램
- 프로그램 실행을 위해서는
  - 프로그램의 코드, 데이터, 스택, 힙, U-영역 등이 필요
- 프로세스 이미지(구조)
  - 메모리 내의 프로세스 레이아웃
- 프로그램 != 프로세스



# 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 영역
  - C 언어의 malloc 함수를 호출해서 동적으로 메모리 할당
- 스택(stack area)
  - 함수 호출을 구현하기 위한 실행시간 스택(runtime stack)을 위한 영역으로 활성 레코드(activation record)가 저장
- U-영역(user-area)
  - 열린 파일 디스크립터, 현재 작업 디렉터리 등과 같은 프로세스의 정보를 저장하는 영역