8장 프로세스

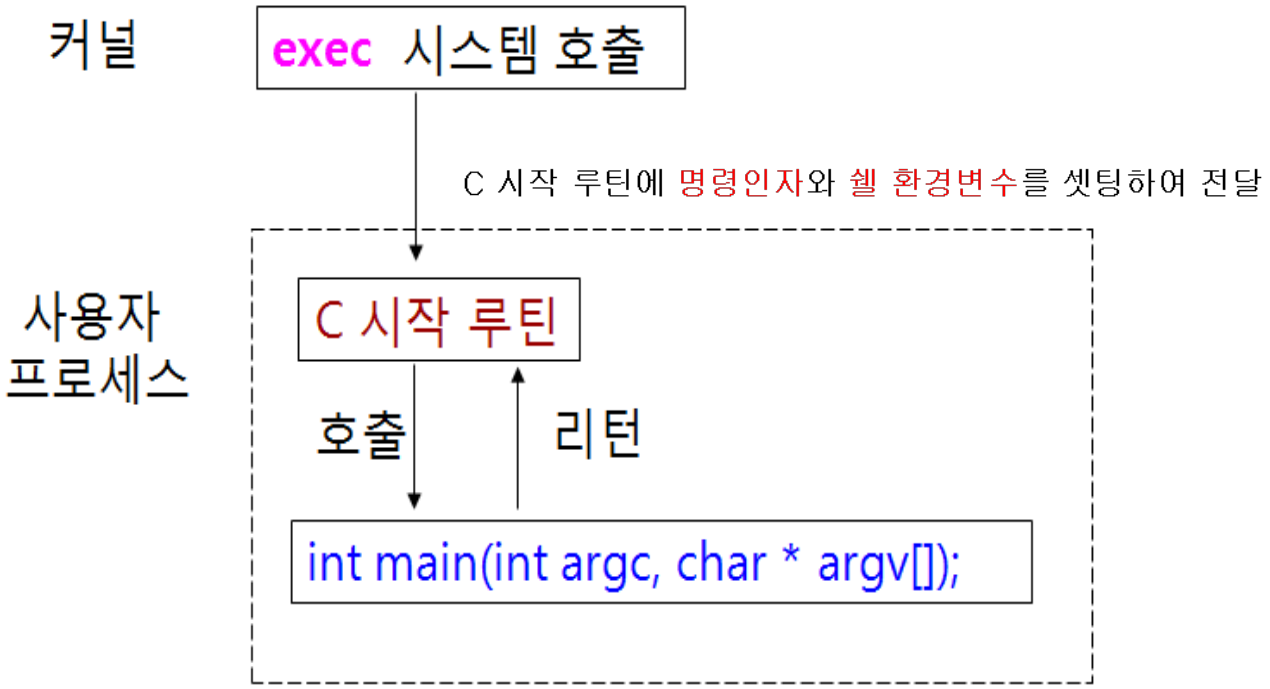
8.1 프로그램 시작

프로그램 실행 과정

- 리눅스에서 프로그램은 어떻게 실행이 시작되고 종료될까?
  - 사용자가 프로그램을 쉘 프롬프트에서 지정하여 실행
  - 이는 내부적으로, 현재 실행 중인 프로그램(즉, shell) 내에서 fork() & exec( ) 시스템 호출을 통해 새로운 프로그램이 실행 됨

- C 시작 루틴(C start-up routine)
  - 컴파일러가 실행 파일에 C 시작 루틴을 포함시킴
    - 실행 파일 포맷에 맞게 생성
  - 이 루틴은 exec로부터 전달받은 명령줄 인수, 환경 변수를 main 함수로 전달하는 역할을 수행

프로그램 실행 시작


커널
exec 시스템 호출
C 시작 루틴에 명령인자와 쉘 환경변수를 셋팅하여 전달
사용자
프로세스
C 시작 루틴
호출
리턴
int main(int argc, char * argv[]);

(참고) 주요 시스템 호출 요약

주요 자원	시스템 호출
파일	open(), close(), read(), write(), dup(), lseek() 등
프로세스	fork(), exec(), exit(), wait(), getpid(), getppid() 등
메모리	malloc(), calloc(), free() 등
시그널	signal(), alarm(), kill(), sleep() 등
프로세스 간 통신	pipe(), socket() 등

프로그램 실행 종료

- main 함수의 실행이 끝나면 다음을 호출

  exit(main(argc, argv[]));

  - C 시작 루틴은 프로그램의 실행이 끝나면 main 함수로 부터 반환 값을 받아 exit 한다.

  - 일반적으로 main 함수는 정상 종료 시, 절차는 다음과 같다.

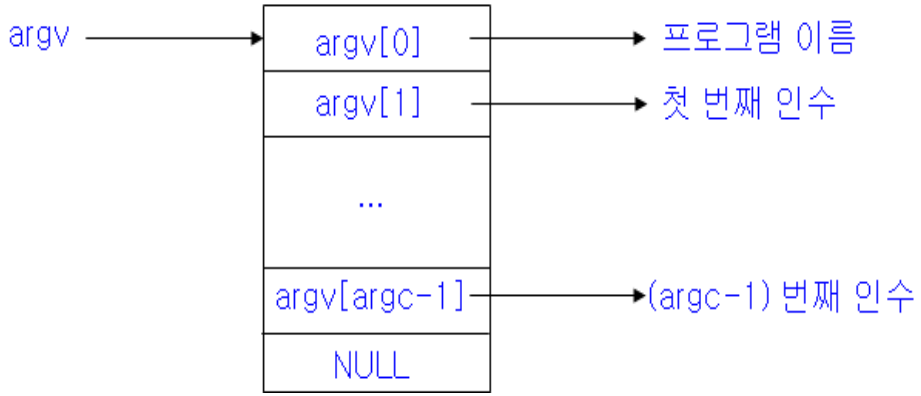
  main 함수 return 0 → C 시작 루틴이 Exit(0) 를 호출 → 커널 실행 → (scheduling) → Shell 로 복귀

명령줄 인수

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 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]);
                                %s는 넘겨받은 주소부터 널 이전까지 출력하라는 의미
    exit(0);
}
```

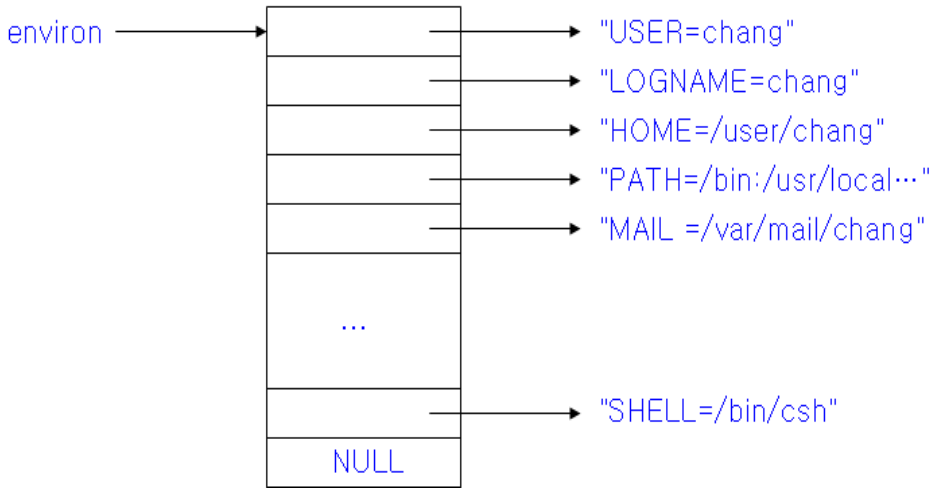
실행 결과 사례

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

환경 변수

새롭게 실행되는 프로그램에게 환경 변수 값 전달

- 쉘이 가지고 있던 환경 변수를 쉘이 새로운 프로그램을 실행시킬 때, 실행되는 그 프로그램에게 넘겨 준다.
- 전역변수 environ을 통해 포인터 배열 (char * environ[]) 형태로 전달 받음.



environ.c

```
#include <stdio.h>
/* 모든 명령줄 인수와 환경 변수를 프린트한다. */
int main(int argc, char *argv[])
{
    char **ptr;                    //char* ptr[] 즉, 포인터 배열
    extern char **environ;     //외부 변수

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

    exit(0);
}
```

environ (포인터 배열의 시작 번지를 가지는 변수)
*environ (배열 한칸 한칸, 이것도 주소이며 포인터)
**environ("USER=chang" 과 같은 값들)

environ 시작 위치에서 1씩 증가하면서 각 환경변수 출력

특정한 환경 변수 접근

- 외부 변수 environ은 환경 변수 전체를 접근
- getenv() 시스템 호출을 사용하여 특정 환경 변수를 하나씩 접근하는 것도 가능하다.

```
#include <stdlib.h>

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

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

C-shell의 setenv를 통해 설정된 환경 변수에 접근하기 위한 함수 : getenv( )

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.

8.2 프로그램 종료

프로그램 종료

- 정상 종료(normal termination)
  - main() 함수가 실행을 마치고 리턴하면 (return(0);)
  - C 시작 루틴(C start-up routine)은 이 리턴값을 가지고 exit()을 호출
  - (혹은) 프로그램 내에서 그 프로그램이 직접 exit() 혹은 _exit()을 호출할 수 있다.
- 비정상 종료(abnormal termination)
  - abort() 시스템 호출을 통한 종료
    - 커널이 프로세스에 SIGABRT 시그널을 보내어 해당 프로세스를 비정상적으로 종료 시킴
    - 발생 이유 : 잘못된 메모리 영역 참조, 잘못된 권한 설정 등

프로그램 종료

- exit()
  - 모든 열려진 파일 스트림을 닫고(fclose), 출력 버퍼의 내용을 디스크에 쓰고 (fflush), 프로세스를 정상적으로 종료
  - 종료되는 프로세스의 종료 코드가 status에 저장
    - (참고) status 변수의 사용 : wait() 시스템 호출 (9장 슬라이드 12 참조)
  - 종료 코드를 부모 프로세스에게 전달

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

프로그램 종료

- _exit()

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

자원 및 데이터 관리성 vs. 빠른 종료 보장

atexit()

```
#include <stdlib.h>

void atexit(void (*func)(void));
            반환값: zero if OK / nonzero on error
```

- exit 처리기를 등록한다
  - 기본적인 exit() 함수의 일(fclose, fflush) 이외에, 추가적으로 사용자가 원하는 후처리 작업 등록 가능
  - 프로세스당 32개까지

- func
  - 추가작업을 위한 exit 처리기
  - exit 수행 시 추가로 처리할 함수 이름 (함수 포인터)

선언 : void (*fptr) (char);
형식 : 반환형 / 함수 이름 (반드시 괄호) / 인수

- exit() 는 exit handler 들을 등록된 역순으로 호출한다

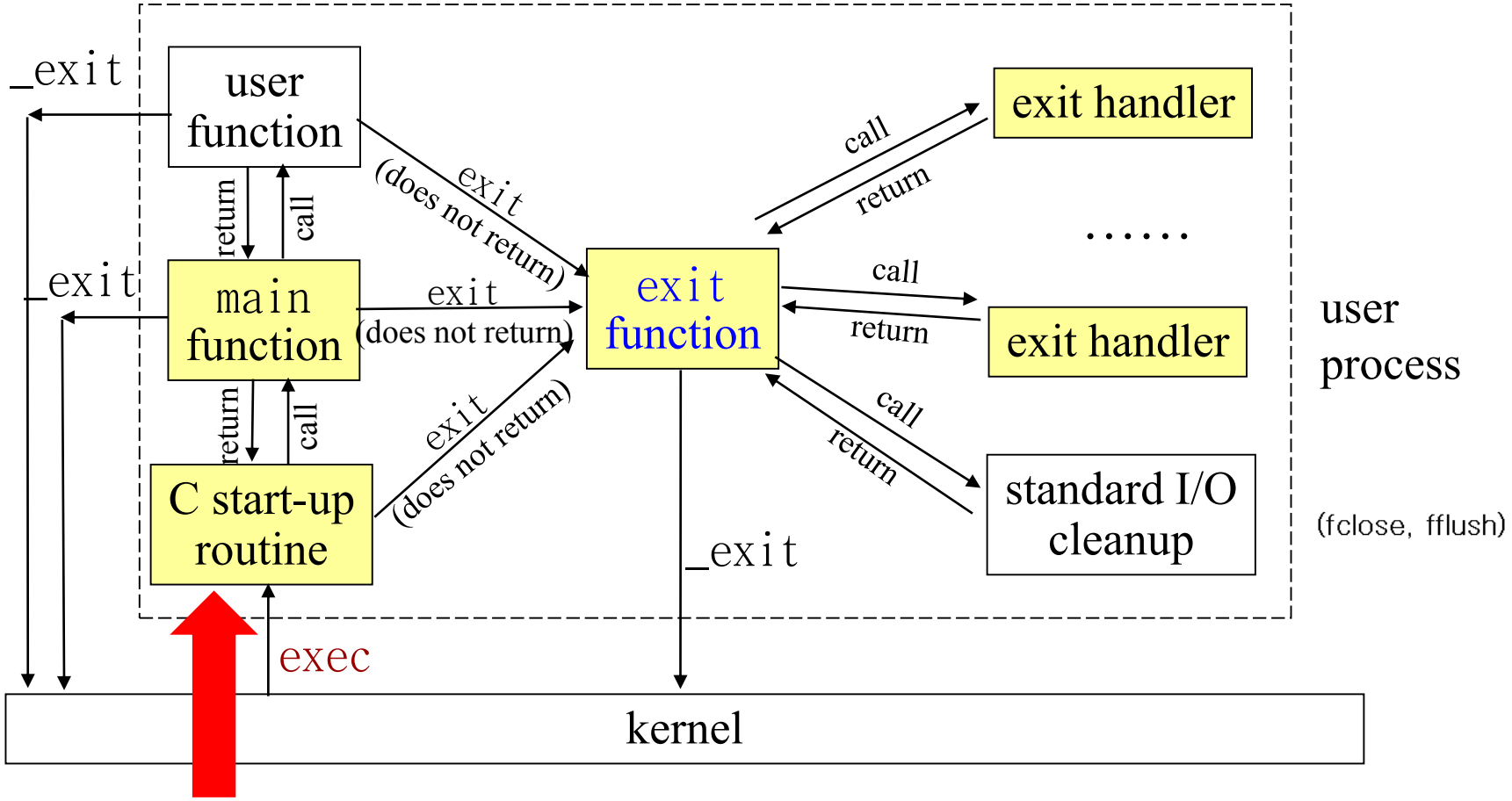
atexit.c : exit 처리기 예

```
1  #include <stdio.h>
2  static void exit_handler1(void),
3                  exit_handler2(void);
4  int main(void) {
5  if (atexit(exit_handler1) != 0) //처리기 등록
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);   // exit 함수 호출 시, atexit에서 등록된
              // 두 exit_handler 가 역순으로 호출 됨
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 }
```

실행 결과

```
$atexit
main 끝
두번째 exit 처리기
첫번째 exit 처리기
```

C 프로그램 시작 및 종료



그림, 좌측 하단에서 상단으로 이동

프로그램 종료

- 정상적인 경우, C start-up routine에서 exit 을 수행하며,
  - 별도의 exit handler 없이 standard I/O cleanup 루틴을 수행으로 정상 종료 됨
- Exit 함수가 추가 작업으로 exit handler를 호출 할 수 있음
  - 등록된 역순으로 호출 됨
- 그 후, exit의 원래 역할인 cleanup 작업 수행 (fclose, fflush)
- Does not return ? 특수한 경우로 돌려 받지 않는다
  - Exit 함수 호출 후, 종료 되었기 때문에 사실 돌려 줄 대상이 없다.
- user function, main function 각각에서 exit 될 수 있음

8.3 프로세스 ID

프로세스 ID

- 각 프로세스는 프로세스를 구별하는 고유 번호인 프로세스 ID를 갖는다.
- 각 프로세스는 자신을 생성해 준 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 }
```

여기서 부모 프로세스는 누구일까? (next slide)

프로세스 ID

- 실행 결과

```
$ hello &
Hello !
나의 프로세스 번호 : [16165]
내 부모 프로세스 번호 : [9045]   // 부모 프로세스는 bash

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
```

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 연산(사용 권한)을 결정하는 데 사용

- 프로세스 사용자 ID와 그룹 ID는 다시 다음과 같이 두 가지로 구분된다.
  - 실제 사용자 (혹은 그룹) ID (Real UID)
  - 유효 사용자 (혹은 그룹) ID (Effective UID)

프로세스의 사용자 ID

- 프로세스 사용자 ID
  - 실제 사용자 ID (real user ID)
  - 유효 사용자 ID (effective user ID)

- 프로세스의 실제 사용자 ID(real user ID)
  - 해당 프로세스를 실행한 실제 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)
  - 새로 파일을 만들 때 그 파일의 소유자 결정 용도 혹은
  - 파일에 대한 접근 권한을 검사하는 용도로 사용된다.

- 유효 사용자 ID 사용법
  - 일반적으로 유효 사용자 ID와 실제 사용자 ID는 특별한 실행파일을 실행할 때를 제외하고는 동일하다.

  - Setuid가 지정되어 있으면, 해당 프로그램 실행 시, 파일을 실행한 사용자가 아닌, 파일 소
  - 유주의 uid를 파일 실행자의 effective uid에 부여 → 즉, 프로그램 실행 시 현재 사용자의 EUID(Effective UID)를 프로그램 소유자의 RUID(Real UID)로 설정

프로세스의 사용자 ID 확인

- 프로세스의 실제/유효 사용자 ID 반환
  - getuid( ) vs. geteuid( )

- 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.
```

프로세스의 사용자 ID 변경

- 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

- set-user-id(set user ID upon execution) 실행권한
  - set-user-id가 설정된 실행파일을 실행하면
  - 이 프로세스의 유효 사용자 ID는 그 실행파일의 소유자로 바뀜.
  - 이 프로세스는 실행되는 동안 그 파일의 소유자 권한을 갖게 됨.

- 예

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

  - set-user-id 실행권한이 설정된 실행파일이며 소유자는 root
  - 일반 사용자가 이 파일을 실행하게 되면, 파일의 실행 동안, 이 프로세스의 유효 사용자 ID는 실제 소유주인 root가 됨
  - /etc/passwd처럼 root만 수정할 수 있는 파일의 접근 시 활용
  - 즉, 일반 사용자가 root 만 접근, 수정 할 수 있는 파일을 접근, 수정 할 수 있게 됨

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

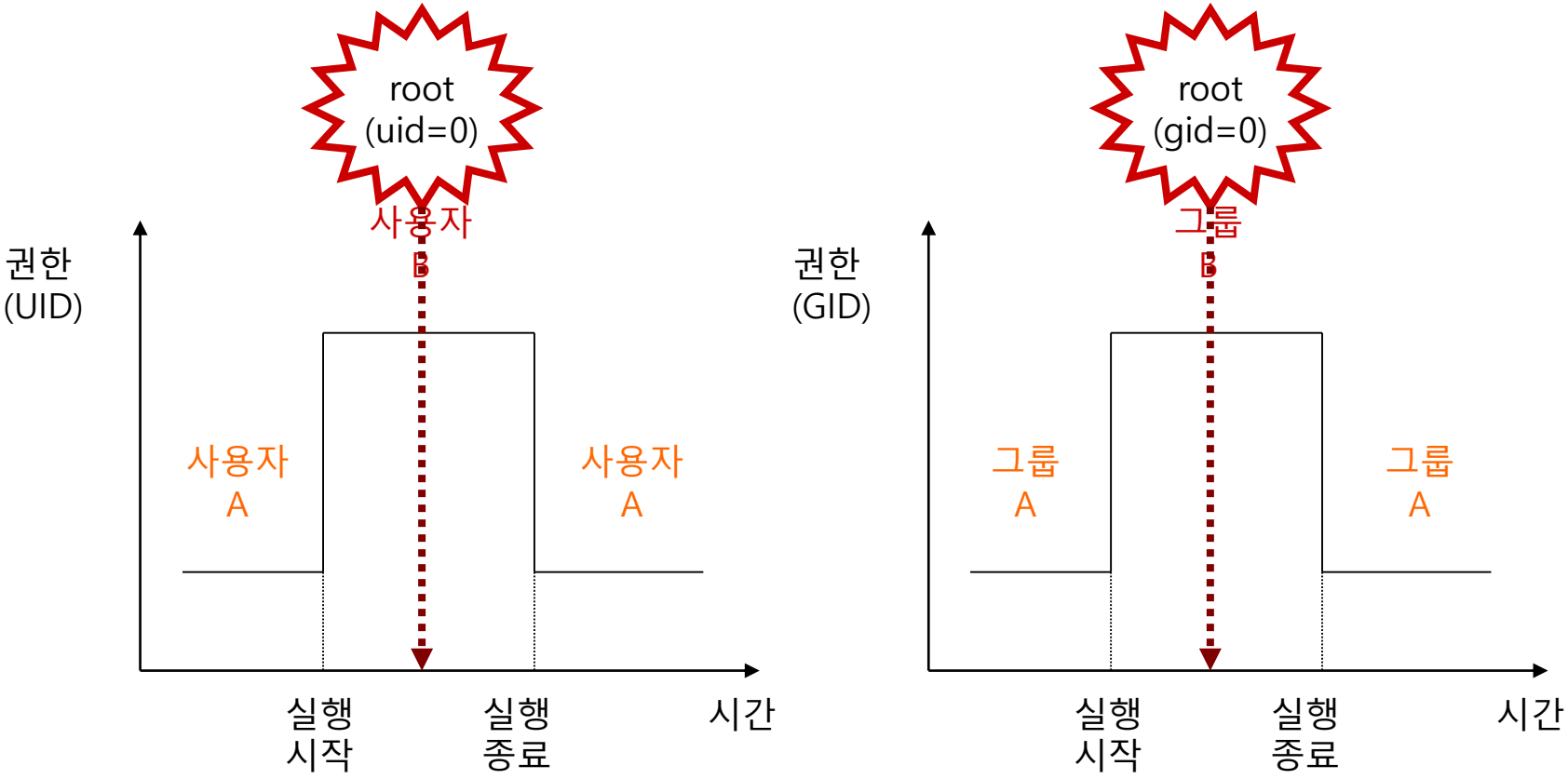
- set-uid 실행권한 설정 (8진수 모드 4000)

  $ chmod 4755 file1
  rwsr-xr-x

- set-gid 비트 설정 (8진수 모드 2000)

  $ chmod 2755 file1
  rwxr-sr-x

set-user-id 실행파일과 권한 상승



(참고) Sticky Bit

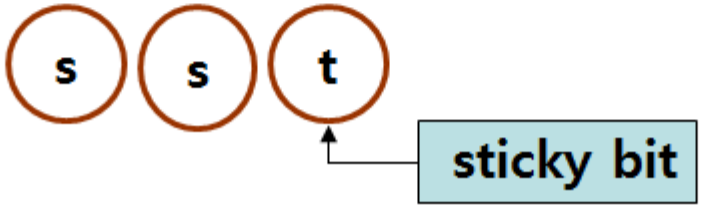
Sticky Bit

- 디렉터리에 sticky bit이 붙으면 디렉터리 내의 파일 삭제에 제한
  - 해당 디렉터리의 쓰기 권한이 있는자
  - 해당 파일의 소유자
  - 해당 디렉터리의 소유자
  - 슈퍼유저

  [예] /tmp

디렉토리 접근 권한에 ----wx-wx 가 포함될 경우
타인이 해당 디렉토리를 수정 및 삭제를 할 수 있음.
디렉토리 생성은 가능하나, 삭제는 루트만 가능하도록 설정

- 정규 파일에 sticky bit이 붙으면,
  - 가상 메모리에 로드된 프로그램을 가상 메모리에 존속하게 함



디렉토리 소유자, 파일의 소유자만 삭제 가능
다른 사용자는 자신이 소유하지 않은 파일을 삭제할 수 없음

프로그램 종료 후에도,
SWAP 영역에서 제거되지 않음
다음 실행 시, 메모리에 빠르게 로딩
빈번히 쓰이는 프로그램에 유용
수퍼 유저만 설정 가능

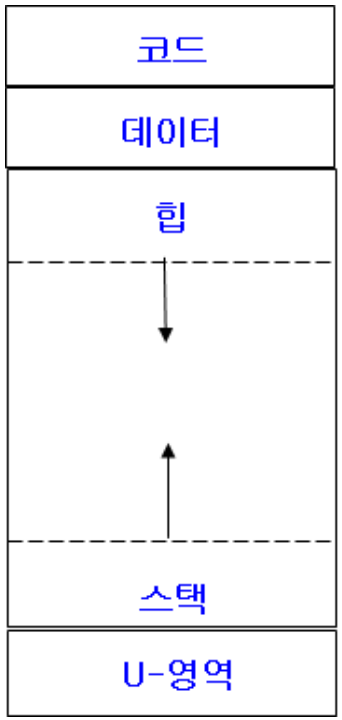
8.4 프로세스 구조

프로세스

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

프로세스 구조

- 프로세스 구조



- size 명령어
  - 실행 파일, 즉 해당 프로세스의 각 영역별 크기를 알려 줌
  - $size /bin/ls

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
```

dec / hex : 10진수/16진수로의 크기 합산

프로세스 구조

- 코드 혹은 텍스트(text)
  - 기계어로 작성된 명령어
  - 프로세스가 실행하는 실행코드, 리터럴 상수를 저장하는 영역이다.

Symbolic constant (const double PI=3.14;)
Macro constant (#define MAX 100)
string constant (char*)

- 데이터 (data)
  - 전역 변수(global variable) 및 정적 변수(static variable)를 위한 메모리 영역
  - 심볼릭 상수, 매크로 상수, 문자열 상수 저장 영역
  - 프로그램의 시작~종료까지 유지되는 데이터 저장
  - 초기화 되지 않은 데이터를 저장하는 영역을 구분하여 bss 영역이라 함
    - bss : Block start by symbol

E.g. int maxcount = 99; (initialized)
E.g. long sum[1000]; (uninitialized)
→ 초기화 되지 전역변수 공간 : bss

프로세스 구조

- 스택(stack area)
  - 함수 호출을 구현하기 위한 영역
  - 활성 레코드(activation record)가 저장된다.
    - 지역 변수, 매개 변수, 리턴 주소, 반환 값 등
  - 시스템이 자동으로 할당하고 해제하는 영역

- 힙(heap area)
  - 동적 메모리 할당을 위한 영역
  - malloc 함수를 호출하면 이 영역에서 동적으로 메모리를 할당 해 줌
  - 개발자가 할당과 해제를 직접 수행하여야 하는 영역

프로세스 구조

- U-영역(user-area) :
  - 메타데이터 저장 공간(PCB)
    - Process Control Block
  - 열린 파일 디스크립터(fd Table), 현재 작업 디렉터리 등과 같은 프로세스의 정보를 저장하는 영역
  - struct proc, struct user 등의 자료가 저장되는 커널 내의 영역
  - 프로세스의 실행을 위해 운영체제가 필요로 하는 자료 가운데 swap out 가능한 정보들

핵심 개념

- 프로세스는 실행중인 프로그램이다.
- 쉘은 사용자와 운영체제 사이에 창구 역할을 하는 소프트웨어로 사용자로부터 명령어를 입력받아 이를 처리하는 명령어 처리기 역할을 한다.
- 프로그램이 실행되면 프로그램의 시작 루틴에게 명령줄 인수와 환경 변수가 전달된다.
- exit()는 뒷정리를 한 후 프로세스를 정상적으로 종료시키고 _exit()는 뒷정리를 하지 않고 프로세스를 즉시 종료시킨다.
- exit 처리기는 exit()에 의한 프로세스 종료 과정에서 자동으로 수행된다.
- 각 프로세스는 프로세스 ID를 갖는다. 각 프로세스는 자신을 생성해준 부모 프로세스가 있다.
- 각 프로세스는 실제 사용자 ID와 유효 사용자 ID를 가지며 실제 그룹 ID와 유효 그룹 ID를 갖는다.
- 프로세스 이미지는 텍스트(코드), 데이터, 힙, 스택 등으로 구성된다.