8장 프로세스

리눅스 시스템 프로그래밍

청주대학교 전자공학과
한철수

목차

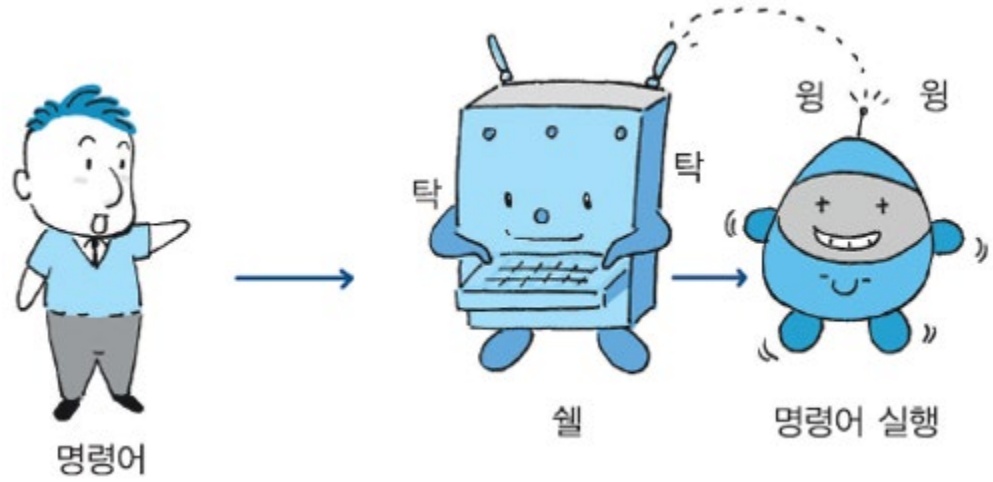
- 쉘과 프로세스
- 프로그램 실행
- 프로그램 종료
- 프로세스 ID와 프로세스의 사용자 ID
- 프로세스 이미지

프로세스

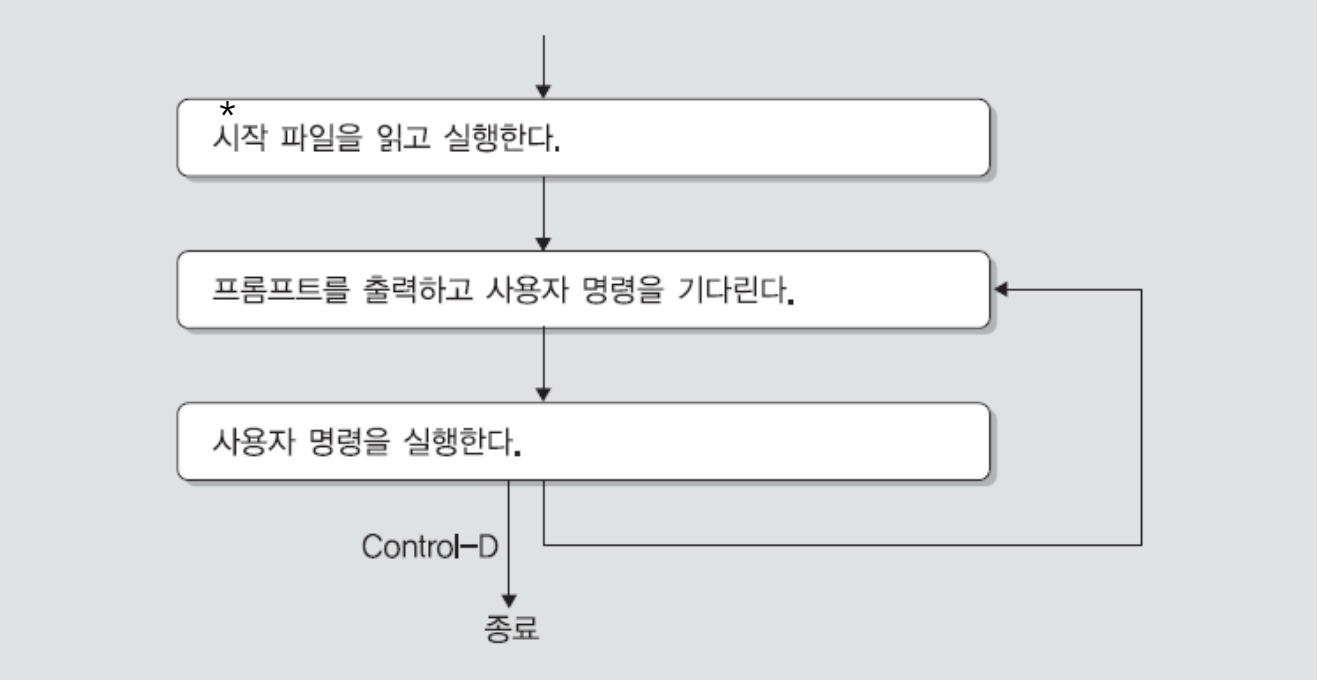
- 프로세스(process)는 파일과 더불어 리눅스 운영체제의 핵심 개념 중 하나임.
- 리눅스 시스템을 깊이 있게 이해하기 위해서는 프로세스에 대하여 정확히 이해해야 함.
- 프로세스는 **실행중인 프로그램**이라고 간단히 말할 수 있음.
  - 프로그램이 실행되면 프로세스가 됨.
- 이장의 내용
  - 쉘이 제공하는 프로세스 관련 기능
  - 프로그램이 실행되고 종료되는 과정
  - 프로세스의 ID
  - 프로세스의 내부 구조

쉘

- 쉘(shell)은 사용자와 운영체제 사이에 창구 역할을 하는 소프트웨어임.
  - 사용자로부터 명령어를 입력 받아 이를 처리하는 명령어 처리기(command processor)의 역할을 수행함.
- Ubuntu 터미널의 기본 쉘
  - bash(배시 쉘, bash shell)



쉘의 실행 절차



*사용자 시작 파일은 사용자의 사용 환경을 설정하는 데 사용됨.
사용자 시작 파일: ~/.bashrc

쉘의 명령어 실행과 자식 프로세스

- 쉘은 사용자가 입력한 명령어를 실행하기 위해 새로운 자식 프로세스를 생성하고, 이 자식 프로세스로 하여금 입력된 명령어를 실행하게 함.
- 예

```
$ date
2023. 05. 04. (목) 11:20:40 KST
$ ./hello
hello world!
$
```

- 쉘이 새로운 자식 프로세스를 생성하고, 이 자식 프로세스가 입력된 명령어를 실행함.
- 쉘은 자식 프로세스의 실행이 끝날 때까지 기다림.
- 자식 프로세스의 실행이 끝나면 다시 쉘 프롬프트를 출력하고 다음 명령어를 기다림.

명령어 열

- 명령어 열은 여러 명령어를 순차적으로 실행함.
  - 쉘은 각 명령어의 실행을 위한 새로운 자식 프로세스를 순차적으로 생성하면서, 각 프로세스로 하여금 한 명령어 씩 순차적으로 실행하게 함.
  - 명령어 열의 형태

    $ 명령어1;명령어2;…;명령어n

- 예

  $ date;who;pwd

세 개의 명령어를 순차적으로 실행시키기 위해 3개의 새로운 자식 프로세스를 순차적으로 생성하면서 각 명령어를 순차적으로 실행함.

명령어 그룹

- 명령어 그룹은 여러 명령어를 순차적으로 실행하는 점은 명령어 열과 같음.
  - 명령어 그룹의 형태

    $ (명령어1;명령어2;…;명령어n)
- 하지만 명령어 그룹은 모든 명령어가 표준입력(stdin), 표준출력(stdout), 표준에러(stderr)를 공유한다는 점이 다름.
  - 따라서 입출력 재지정과 파이프를 사용할 때, 마치 하나의 명령어 처럼 입출력 재지정 및 파이프 처리를 할 수 있음.
- 예
  - 명령어 열

    $ date; who; pwd > out1.txt ← pwd의 결과만 out1.txt에 저장됨.
  - 명령어 그룹

    $ (date; who; pwd) > out2.txt ← (date;who;pwd)의 모든 결과가 out2.txt에 저장됨.

전면 처리

- 전면 처리란 명령어를 입력하면 명령어가 전면에서 실행되며, 명령어 실행이 끝날 때까지 쉘이 기다려 주는 명령어 처리 방식을 말함.
  - 전면에서 실행되고 있는 명령어는 필요에 따라 키보드와 모니터로 입출력을 할 수 있음.
  - 명령어를 전면 처리하면 한 순간에 하나의 명령어만 실행할 수 있음.

명령어의 강제 종료, 정지, 계속

- 전면 처리 실행중인 명령어를 **강제 종료**.
  - ^C // 컨트롤 키와 c키를 동시에 누름.
- 전면 처리 실행중인 명령어를 **정지시킨 후, 후면으로 보냄**.
  - ^Z // 컨트롤 키와 z키를 동시에 누름.
- 후면에서 정지된 명령어를 **전면**(**f**ore**g**round)**에서 계속 실행**.
  - $ fg
- 후면에서 정지된 명령어를 **후면**(**b**ack**g**round)**에서 계속 실행**.
  - $ bg
- 예

```
$ (sleep 100; echo done)
^C              // 강제 종료됨.
$  (sleep 100; echo done)
^Z              // 정지시킨 후, 후면으로 보냄.
$ fg            // 후면에서 정지된 명령어를 전면에서 계속 실행.
```

후면 처리

- 후면 처리를 이용하면 특정 명령어를 후면에서 처리하고, 전면에서는 다른 작업을 수행함으로써 동시에 여러 작업을 수행할 수 있음.
  - 사용법
    - $ 명령어 & // 후면 처리할 명령어 뒤에 & 기호를 붙여 실행함.
  - 시간이 오래 걸리는 작업이나 동시에 여러 작업을 수행하고자 할 때 후면 처리를 이용할 수 있음.

후면 처리 예제

- 두 명령어를 후면 처리 실행

```
$ (sleep 100; echo done) &          // 100초 기다린 후, 문자열 “done”을 출력함.
[1] 8320
$ find . -name test.c -print &      // 현재 작업 디렉터리에서 test.c라는 이름의
[2] 8325                               파일을 찾아 있으면 파일이름을 출력함.
```

작업 번호 PID

- 후면 처리되고 있는 작업들의 표시

```
$ jobs
```

- 후면 처리에서 전면 처리로 전환

```
$ fg %작업번호
$ fg %1
```

- 후면처리에서의 표준 입출력
  - 후면 처리의 출력이 전면 처리의 출력과 뒤섞이지 않도록 하기 위한 조치

```
$ find . -name test.c -print > find.txt &        // 후면 처리의 출력을 파일에 저장함.
$ find . -name test.c -print | mail chang &      // 후면 처리의 출력을 메일로 전송함.
```

  - 후면 처리는 키보드로부터 입력을 받을 수 없으므로, 텍스트 파일로부터 입력 받아야 함.

```
$ wc < inputfile &                               // 텍스트 파일로부터 입력 받음.
```

프로세스

- 실행중인 프로그램을 프로세스(process) 혹은 작업(job)이라고 함.
  - 시스템 내에는 여러 개의 프로세스가 동시에 수행되고 있음.
  - 각각의 프로세스는 유일한 프로세스 번호(**PID**)를 가짐.
- 프로세스를 확인하고 제어하기 위한 다양한 명령어들
  - ps
  - sleep
  - kill
  - wait
  - exit

ps 명령어

- ps 명령어는 현재 존재하는 프로세스들의 실행 상태를 요약해서 출력함.
- 옵션을 사용하지 않으면, 나의 프로세스에 대한 정보만을 출력함.

```
$ ps
PID    TTY   TIME     CMD
25435 pts/3 00:00:00 csh
25461 pts/3 00:00:00 ps
```

- $ ps aux (BSD 유닉스 형식)
  - a: 모든 사용자의 프로세스 정보를 출력
  - u: 프로세스에 대한 정보를 자세히 출력
  - x: 더 이상 제어 터미널을 갖지 않은 프로세스들도 함께 출력
- $ ps -ef (시스템 V 형식)
  - e: 모든 사용자의 프로세스 정보를 출력
  - f: 프로세스에 대한 정보를 자세히 출력

sleep 명령어

- 지정된 시간만큼 프로세스의 실행을 멈춤.
- 사용법

  $ sleep 초
- 예

  $ (echo 시작; sleep 5; echo 끝)

- "시작"을 출력한 후, 5초 후에 "끝"을 출력함.
- sleep 명령어는 여러 명령어를 수행할 때, 사용자의 의도에 따라 시간적인 간격을 두기 위하여 유용하게 사용됨.

kill 명령어

- kill 명령어는 프로세스를 강제적으로 종료 시키는 명령어임.
  - **프로세스 번호(PID)** 또는 **작업 번호**를 명령줄 인수로 적어 실행하면 해당 프로세스를 종료 시킴.
- 사용법

  $ kill [-시그널] 프로세스번호
  - kill 명령어는 수행중인 특정 프로세스에 원하는 시그널을 보내는 기능을 수행함.
  - 시그널 번호를 생략하면 종료 시그널을 보내기 때문에 해당 프로세스가 종료됨.
- 예

  $ (echo 시작; sleep 60; echo 끝) & ← 후면 처리 시작

  [1] 1230

  $ kill 1230 혹은 $ kill %1 ← 프로세스 번호(PID) 혹은 작업 번호를 이용해 강제 종료 시킬 수 있음.

wait 명령어

- 해당 프로세스 번호를 갖는 자식 프로세스가 종료될 때까지 기다리게 하는 명령어임.
  - 프로세스 번호를 지정하지 않으면 모든 자식 프로세스가 끝나기를 기다림.
- 사용법

  $ wait [프로세스번호]
- 예

```
$ (sleep 60; echo 1번 끝) &
[1] 1231
$ echo 2번 끝; wait 1231; echo 3번 끝
2번 끝
1번 끝
3번 끝
$ (sleep 60; echo 1번 끝) &
$ (sleep 60; echo 2번 끝) &
$ echo 3번 끝; wait; echo 4번 끝
3번 끝
1번 끝
2번 끝
4번 끝
```

후면 실행되는 지정된 자식 프로세스가 끝날 때까지 기다림.

후면 실행되는 두 개의 자식 프로세스가 모두 끝날 때까지 기다림.

exit 명령어

- 쉘을 종료하고 종료코드(exit code)를 부모 프로세스에 전달함.
- 사용법

  $ exit [종료코드]
- 예

  exit

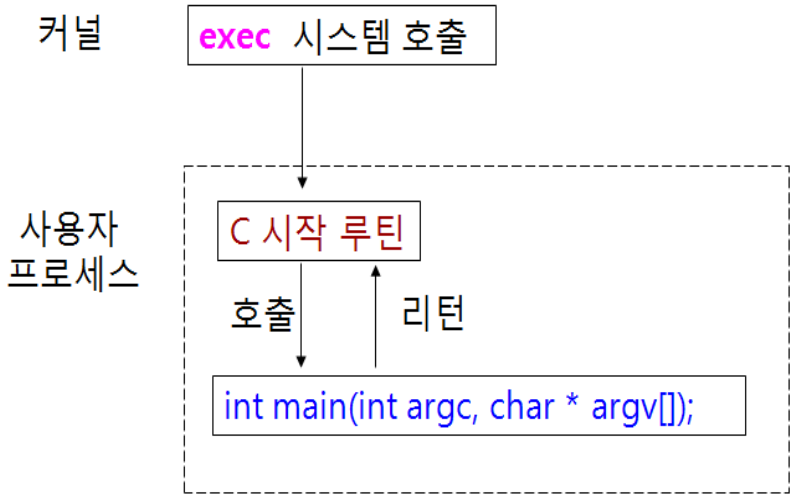
  exit 1

프로그램의 실행

- 사용자가 프로그램을 실행시키는 방법은 크게 두가지임.
  - 쉘 프롬프트에서 프로그램을 지정하여 실행시키는 것
  - 실행중인 프로그램(사용자 프로세스) 내에서 exec() 시스템 호출을 이용하여 다른 프로그램을 실행시키는 것
- 위 두 방법은 사실 동일함.
  - 쉘도 실행중인 프로세스이며, 사용자로부터 입력 받은 실행할 프로그램을 쉘도 exec() 시스템 호출을 이용하여 실행시키기 때문.
- 결국 모든 프로그램은 exec() 시스템 호출에 의해서 실행됨.

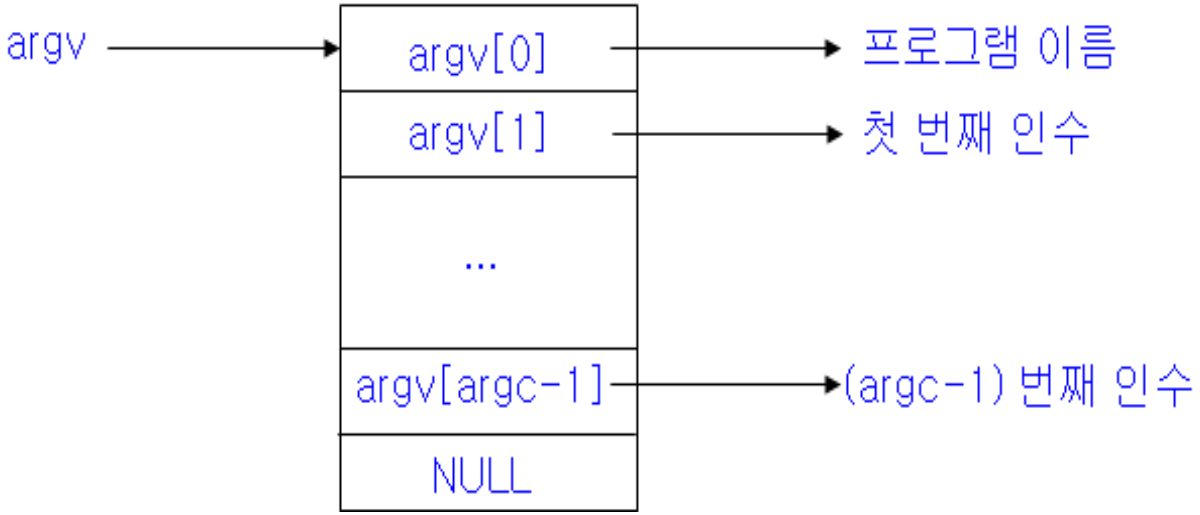
프로그램 실행 시작

- C 프로그램을 컴파일하면 실행파일에는 C 프로그램 코드와 C 시작 루틴이 포함됨.
- exec() 시스템 호출
  - 실행될 프로그램의 C 시작 루틴에게 명령줄 인수와 환경 변수를 전달함.
- C 시작 루틴
  - 전달 받은 명령줄 인수와 환경 변수를 main 함수에 전달함.
  - main 함수의 실행이 끝나면 반환 값을 받아 exit함.
    - **exit(main(argc, argv));**



명령줄 인수

- exec() 시스템 호출은 실행되는 프로그램에게 명령줄 인수를 전달함.
  - 실행되는 프로그램의 main() 함수는 argc와 argv를 통해 명령줄 인수의 개수와 명령줄 인수에 대한 포인터 배열을 전달 받음.
- int main(int argc, char *argv[]);
  - argc: 명령줄 인수의 개수
  - argv[]: 명령줄 인수의 리스트를 나타내는 포인터 배열



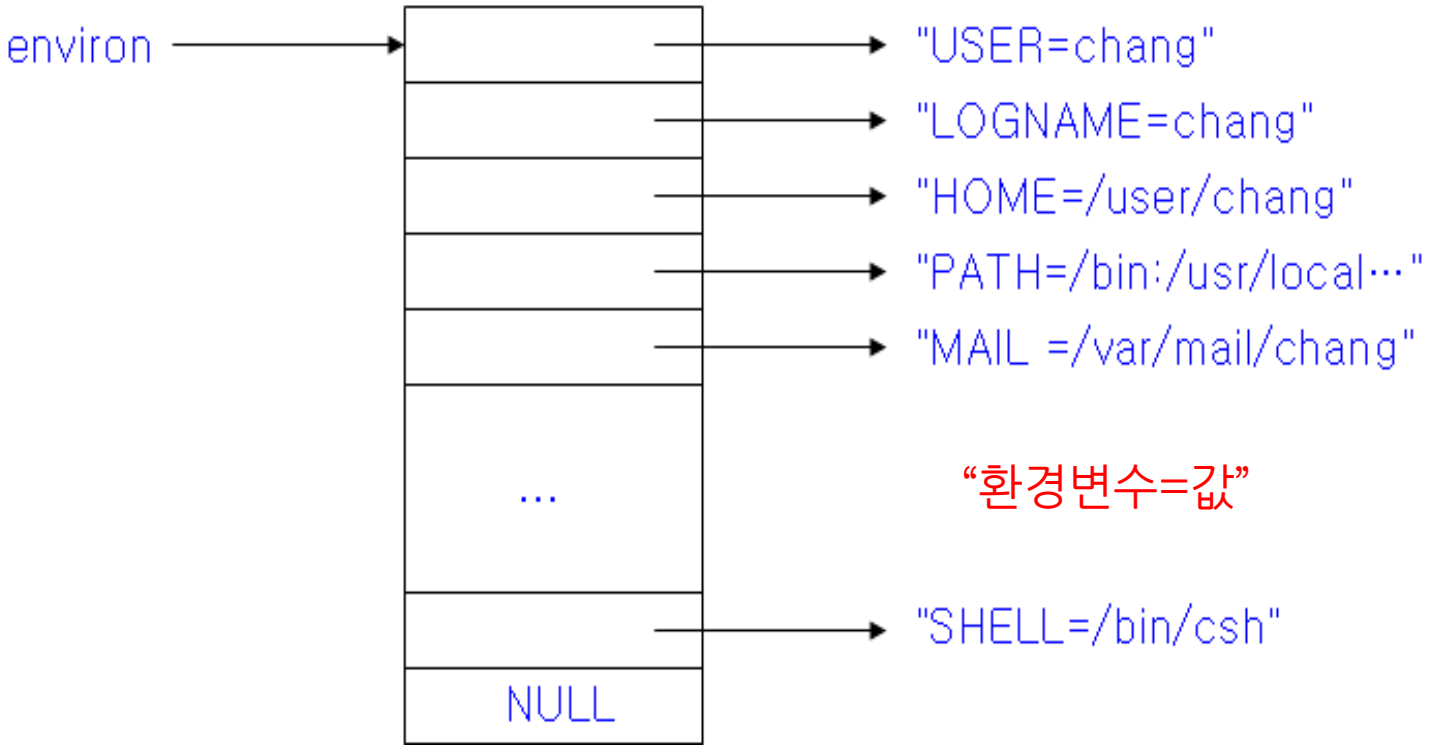
args.c

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

int main(int argc, char *argv[])
{
    for (int i = 0; i < argc; i++)
        printf("argv[%d]: %s₩n", i, argv[i]);

    exit(0);
}
```

```
cju@cju-VirtualBox:~/chap8$ gcc -o args args.c
cju@cju-VirtualBox:~/chap8$ ./args hello world
argv[0]: ./args
argv[1]: hello
argv[2]: world
cju@cju-VirtualBox:~/chap8$
```

환경 변수

- 환경 변수는 쉘이 원래 가지고 있던 것으로, 쉘이 프로그램을 실행시킬 때 실행되는 프로세스에게 넘겨줌.
  - 전역 변수 environ을 통해 환경 변수와 값의 리스트를 문자열의 배열 형태로 전달함.



environ.c

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

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

    for (char **ptr = environ; *ptr != 0; ptr++)
        printf("%s₩n", *ptr);

    exit(0);
}
```

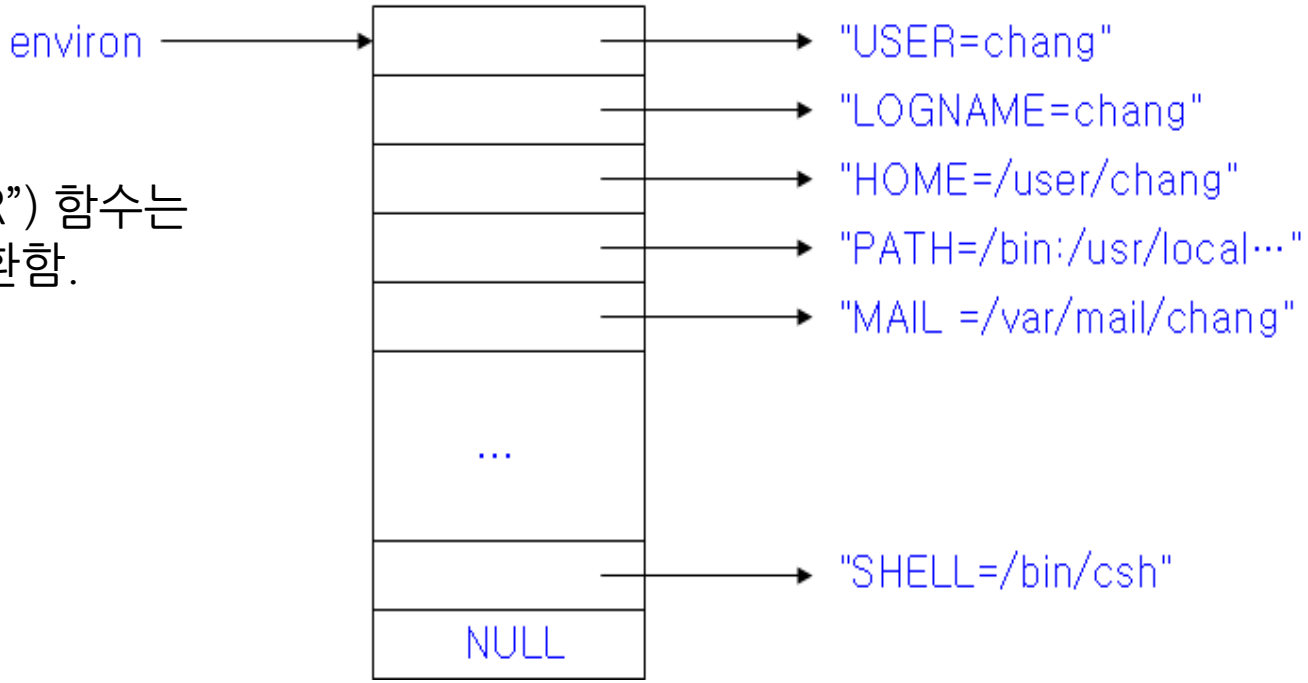
```
cju@cju-VirtualBox:~/chap8$ gcc -o environ environ.c
cju@cju-VirtualBox:~/chap8$ ./environ
SHELL=/bin/bash
SESSION_MANAGER=local/cju-VirtualBox:@/tmp/.ICE-unix/1760,unix/cju-VirtualBox:/t
mp/.ICE-unix/1760
QT_ACCESSIBILITY=1
```

환경 변수 접근

- getenv() 함수를 사용하면 특정 환경 변수의 값을 가져올 수 있음.

```
#include <stdlib.h>

char *getenv(const char *name);

환경 변수 name의 값을 반환함. 해당 이름의 변수가 없으면 NULL을 반환함.
```

- 예
getenv("USER") 함수는 "chang"을 반환함.



printenv.c

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

int main(int argc, char *argv[])
{
    char *ptr=NULL;
    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);
}
```

```
cju@cju-VirtualBox:~/chap8$ ./printenv
HOME = /home/cju
SHELL = /bin/bash
PATH = /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
usr/local/games:/snap/bin:/snap/bin
cju@cju-VirtualBox:~/chap8$
```

환경 변수 설정, 삭제

- 아래의 함수를 이용하면 특정 환경 변수의 값을 설정하거나 지우는 것도 가능함.

#include <stdlib.h>

- int putenv(const char *string);

"name=value" 형태의 문자열을 입력 받아서, 이를 환경 변수 리스트에 넣어줌. name이 이미 존재하면 원래 값을 새로운 값으로 대체함.

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

환경 변수 name의 값을 value로 설정함. name이 이미 존재하는 경우에는 rewrite 값이 0이 아니면 원래 값을 새로운 값으로 대체하고, rewrite 값이 0이면 그대로 둠.

- int unsetenv(const char *name);

환경 변수 name을 삭제함.

질문

Q&A

프로그램의 종료 방법

- 정상 종료(normal termination)
  1. main() 함수의 실행을 마치고 반환 값을 반환하면, C 시작 루틴은 이 반환 값을 가지고 exit() 함수를 호출함.
  2. 프로그램 내에서 직접 exit() 함수를 호출함.
  3. 프로그램 내에서 직접 _exit() 함수를 호출함.
- 비정상 종료(abnormal termination)
  1. abort() 함수에 의한 종료
     - 프로세스에 SIGABRT 시그널을 보내어 프로세스를 비정상적으로 종료 시킴.
  2. 시그널에 의한 종료
     - 실행 중인 프로세스가 시그널을 받으면 갑자기 비정상적으로 종료하게 됨.

exit(), _exit() 함수

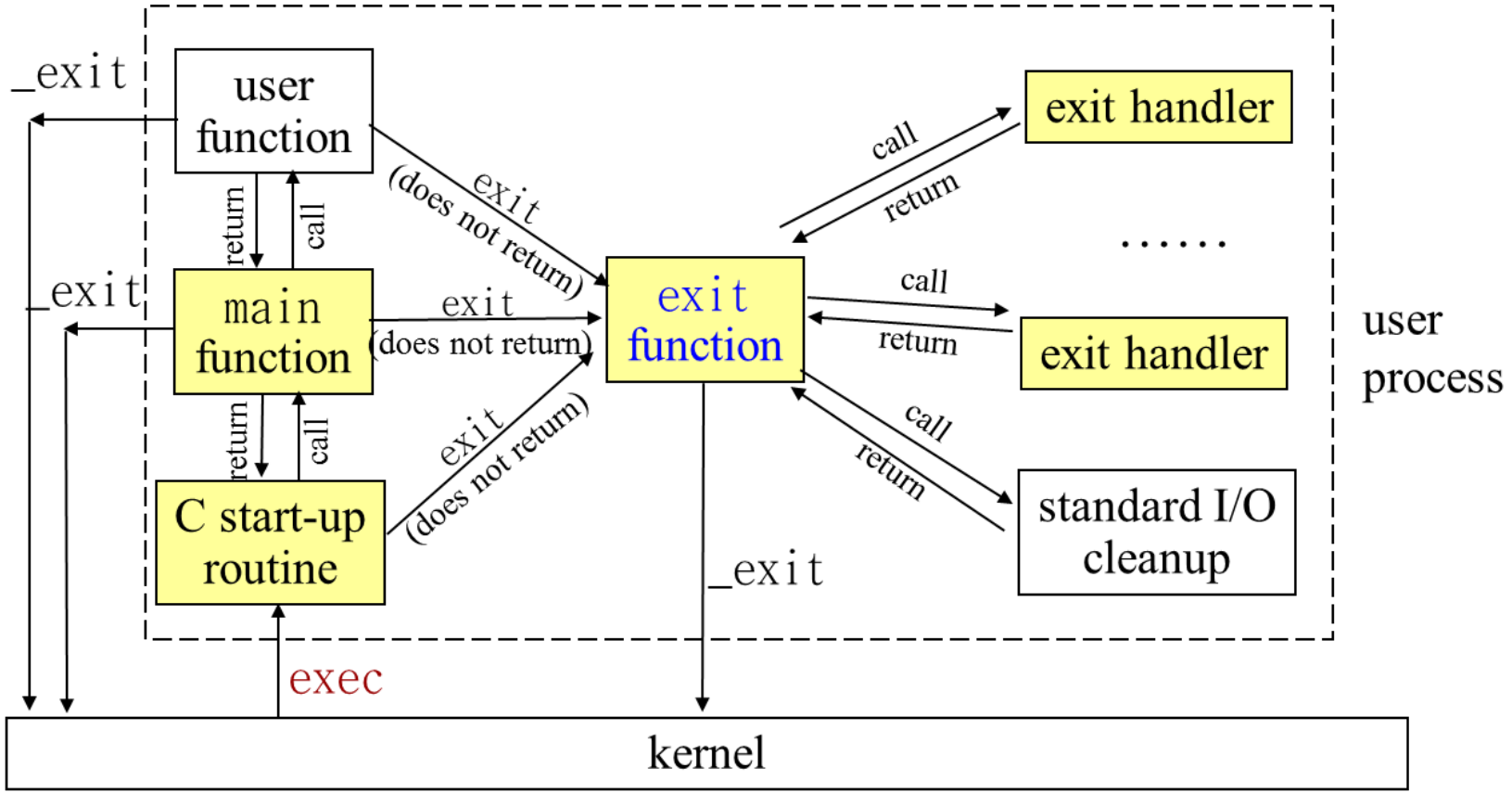
- exit()
  - 모든 스트림을 닫고(fclose), 출력 버퍼의 내용을 디스크에 쓰는 등의 뒷정리(fflush) 후, 프로세스를 종료 시킴.
  - 종료 코드(exit code)를 부모 프로세스에게 전달함.

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

- _exit()
  - 뒷정리를 하지 않고 종료 시킴.

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

프로그램의 시작과 종료 과정



exit 처리기

- exit() 시스템 호출 함수에 의해 프로세스가 종료될 때 표준 입출력의 뒷정리가 기본적으로 수행됨.
- 사용자는 기본적인 뒷정리 외에 별도의 뒷정리 작업을 하기 위하여 atexit() 시스템 호출 함수를 이용하여 exit 처리기 함수를 등록할 수 있음.

```
#include <stdlib.h>
int atexit(void (*func)(void));
exit 처리기로 함수 func를 등록함.
등록이 성공하면 0을, 실패하면 0이 아닌 숫자를 반환함.
```

- exit 처리기는 프로세스당 32개까지 등록이 가능함.
  - 등록된 exit 처리기는 프로그램이 종료될 때 등록된 역순으로 호출되어 실행됨.

atexit.c

```c
#include <stdio.h>
#include <stdlib.h>
void exit_handler1();
void exit_handler2();
int main() {
    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);
}
```

```c
void exit_handler1() {
    printf("첫 번째 exit 처리기₩n");
}
void exit_handler2() {
    printf("두 번째 exit 처리기₩n");
}
```

```
cju@cju-VirtualBox:~/chap8$ gcc -o atexit atexit.c
cju@cju-VirtualBox:~/chap8$ ./atexit
main 끝
두 번째 exit 처리기
첫 번째 exit 처리기
cju@cju-VirtualBox:~/chap8$
```

프로세스 ID

- 프로그램이 실행되면 프로세스가 됨.
  - 한 프로그램은 여러 번 실행될 수 있으므로, 한 프로그램으로부터 여러 개의 프로세스가 생성될 수 있음.
- 각 프로세스는 프로세스를 구별하는 번호인 **프로세스 ID (PID)**를 가짐.
- 모든 프로세스는 다른 프로세스에 의해 생성됨.
  - 자신을 생성해준 프로세스를 부모 프로세스라고 함.
- PID 관련 시스템 호출

int getpid( );	프로세스 ID를 반환함.
Int getppid( );	부모 프로세스의 프로세스 ID를 반환함.

pid.c

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    printf("나의 프로세스 번호 : [%d]₩n", getpid());
    printf("내 부모 프로세스 번호 : [%d]₩n", getppid());
    return 0;
}
```

```
cju@cju-VirtualBox:~/chap8$ gcc -o pid pid.c
cju@cju-VirtualBox:~/chap8$ ./pid
나의 프로세스 번호 : [4378]
내 부모 프로세스 번호 : [2372]
cju@cju-VirtualBox:~/chap8$ ps
    PID TTY          TIME CMD
   2372 pts/0    00:00:00 bash
   4380 pts/0    00:00:00 ps
cju@cju-VirtualBox:~/chap8$
```

chdir() 시스템 호출

- 각 프로세스는 현재 작업 디렉터리를 가짐.

```
int chdir(char* pathname);
```

- chdir() 시스템 호출은 현재 작업 디렉터리를 pathname으로 변경함.
- chdir() 시스템 호출이 성공하려면 그 디렉터리에 대한 실행 권한이 있어야 함.

프로세스의 사용자 ID와 그룹 ID

- 프로세스는 프로세스 ID 외에도 다양한 ID를 가짐
  - 프로세스 ID (PID)
  - 프로세스 사용자 ID
    - 프로세스 실제 사용자 ID ← 프로세스를 실행시킨 사용자 ID
    - 프로세스 유효 사용자 ID ← 현재 유효한 사용자 ID
  - 프로세스 그룹 ID
    - 프로세스 실제 그룹 ID ← 프로세스를 실행시킨 사용자의 그룹 ID
    - 프로세스 유효 그룹 ID ← 현재 유효한 사용자의 그룹 ID
- 이러한 여러 가지 ID는 프로세스가 수행할 수 있는 권한을 검사하는 데 사용됨.
  - 실제 ID와 유효 ID는 일반적으로 같지만, 특별한 경우에는 달라질 수 있음.
    - **프로세스 실제 사용자 ID가 cju이고 프로세스 유효 사용자 ID가 root이면, root 소유의 파일에 접근이 가능함**.

다양한 ID를 읽어오는 시스템 호출

```
#include <sys/types.h>
#include <unistd.h>
uid_t getuid( );     프로세스의 실제 사용자 ID를 반환함.
uid_t geteuid( );    프로세스의 유효 사용자 ID를 반환함.
uid_t getgid( );     프로세스의 실제 그룹 ID를 반환함.
uid_t getegid( );    프로세스의 유효 그룹 ID를 반환함.
```

다양한 ID를 변경하는 시스템 호출

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);        프로세스의 실제 사용자 ID를 uid로 변경함.
int seteuid(uid_t uid);       프로세스의 유효 사용자 ID를 uid로 변경함.
int setgid(gid_t gid);        프로세스의 실제 그룹 ID를 gid로 변경함.
int setegid(gid_t gid);       프로세스의 유효 그룹 ID를 gid로 변경함.
```

uid.c

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

int main()
{
    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);
}
```

사용자 ID를 반환함.

사용자 ID(숫자)에 해당하는 사용자 정보를 구조체 변수 형태로 읽어와, 그 속에서 변수 pw_name에 접근함.

```
cju@cju-VirtualBox:~/chap8$ gcc -o uid uid.c
cju@cju-VirtualBox:~/chap8$ ./uid
나의 실제 사용자 ID : 1000(cju)
나의 유효 사용자 ID : 1000(cju)
나의 실제 그룹 ID : 1000(cju)
나의 유효 그룹 ID : 1000(cju)
cju@cju-VirtualBox:~/chap8$
```

set-user-id 실행권한

- 실행 파일의 set-user-id라는 특별한 실행권한을 설정하면, 실제 사용자 ID와 유효 사용자 ID를 다르게 만들 수 있음.
  - 실제 사용자 ID는 프로세스를 실행시킨 사용자 ID이고, 유효 사용자 ID는 실행파일의 소유자가 됨.
  - 유효 사용자 ID가 실행파일의 소유자이기 때문에, 프로세스가 실행되는 동안 그 파일의 소유자 권한을 갖게 됨.
- 예
  - /usr/bin/passwd 명령어는 set-user-id 실행권한이 설정된 실행파일로서, 소유자는 root임.
  - 일반 사용자가 이 파일을 실행하게 되면 이 프로세스의 실제 사용자 ID는 사용자이지만, 프로세스의 유효 사용자 ID는 root임.
  - 따라서 이 프로세스는 /etc/passwd처럼 root만 수정할 수 있는 파일에 접근하여 파일을 수정하는 것이 가능함.

set-user-id 권한의 실행파일

- set-user-id 실행권한은 's'로 표시됨.

  $ ls -l /bin/su /bin/passwd

  -rwsr-xr-x. 1 root root 59976 11월 24 21:05 /bin/su

  -rwsr-xr-x. 1 root root 55672 2월 21 2022 /bin/passwd

- set-user-id 실행권한의 설정 방법

  $ chmod 4755 file명

- set-group-id 실행권한의 설정 방법

  $ chmod 2755 file명

예

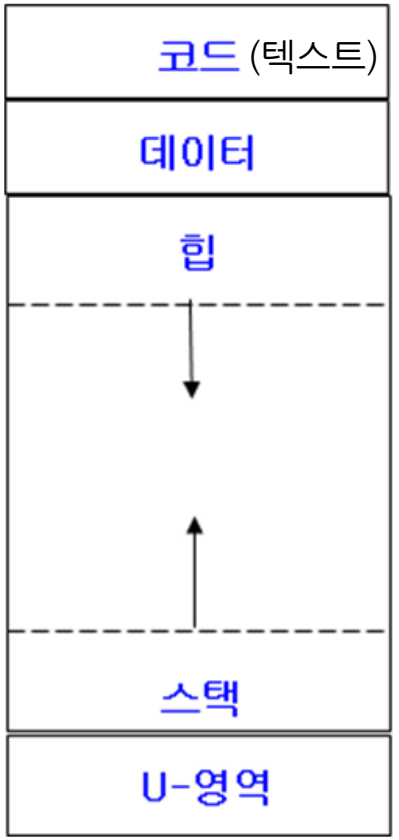
```
cju@cju-VirtualBox:~/chap8$ gcc -o uid uid.c
cju@cju-VirtualBox:~/chap8$ ls -l uid
-rwxrwxr-x 1 cju cju 16224  5월 10 22:24 uid
cju@cju-VirtualBox:~/chap8$ sudo chown root uid
[sudo] cju 암호:
cju@cju-VirtualBox:~/chap8$ ls -l uid
-rwxrwxr-x 1 root cju 16224  5월 10 22:24 uid
cju@cju-VirtualBox:~/chap8$ sudo chmod 4775 uid
cju@cju-VirtualBox:~/chap8$ ls -l uid
-rwsrwxr-x 1 root cju 16224  5월 10 22:24 uid
cju@cju-VirtualBox:~/chap8$ ./uid
나의 실제 사용자 ID : 1000(cju)
나의 유효 사용자 ID : 0(root)
나의 실제 그룹 ID : 1000(cju)
나의 유효 그룹 ID : 1000(cju)
cju@cju-VirtualBox:~/chap8$
```

질문

Q&A

프로세스 이미지

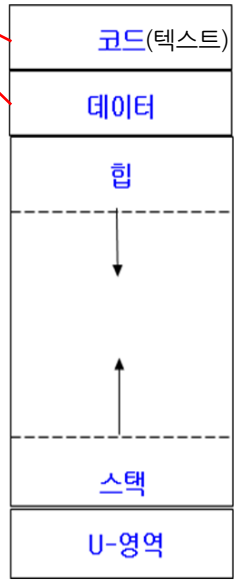
- 프로그램을 실행하기 위해서는 코드(텍스트), 데이터, 힙, 스택 등의 영역을 위한 메모리가 필요하고, 이러한 메모리 배치를 프로세스 이미지라고 함.
- 프로세스 이미지를 구성하는 각 영역의 역할
  - 텍스트(text)
    - 프로세스가 실행하는 실행 코드를 저장함.
  - 데이터(data)
    - 전역 변수 및 정적 변수를 저장함.
  - 힙(heap)
    - 동적 메모리 할당을 위한 영역으로, malloc() 함수를 호출하면 이 영역에서 메모리를 할당해줌.
  - 스택(stack)
    - 함수 호출의 구현에 이용되는 영역으로, 함수가 호출될 때마다 해당 함수의 지역 변수, 매개변수, 반환주소, 반환 값 등을 포함하는 활성 레코드가 저장됨.
  - U-영역(user-area)
    - 열린 파일의 디스크립터, 현재 작업 디렉터리 등과 같은 프로세스의 정보를 저장하는 영역임.



size 명령어

- 실행 파일의 텍스트, 데이터 등의 크기를 알려줌.

```
cju@cju-VirtualBox:~/chap8$ size /bin/ls
   text    data     bss     dec     hex filename
 120464    4720    4800  129984   1fbc0 /bin/ls
cju@cju-VirtualBox:~/chap8$ size uid
   text    data     bss     dec     hex filename
   2240     648       8    2896     b50 uid
cju@cju-VirtualBox:~/chap8$
```



질문

Q&A