<File system structure>

파일 타입(3가지)

➔ 일반 file, directory file, device file (모니터, 키보드, 하드디스크...)

파일 안에 있는 것을 stream file(collection of bytes)이라고 함

1) 일반 파일 : 데이터를 포함하고 있는 텍스트 또는 이진 파일

2) 디렉터리 파일 : 자기 자신도 하나의 파일, 상위 디렉터리에 등록되어 있음, 파일의 이름들과 파일 정보에 대한 포인터들을 포함한다.

3) 디바이스 파일 : 시스템에 접속하는 주변 기기나 메모리 등의 장치 각각에 대응하는 특수 파일, /dev 디렉터리에서 관리, stream file이다

-**character special file** : 문자 단위로 데이터 처리가 가능한 장치에 해당하는 파일(버퍼X)

Ex) 프린터, 터미널

-**block special file** : 블록 단위로 데이터를 처리하는 장치에 해당하는 파일 (버퍼 O)

Ex) 디스크, 자기테이프

디렉터리 구조

/bin : binary file, 기본 실행 명령

/boot : 시스템이 부팅될 적에 관여하는 파일들

/dev : device에 관련된 파일

/etc : 시스템 구성에 관련된 파일

/home : 사용자에 대한 파일

/mnt : 마운트에 관한

/sbin : 시스템 관리자용 명령어 저장 (ex.ifconfig, shutdown) + 네트워크 관련 명령어

/tmp : 임시저장되어 있는 파일

/usr : 응용프로그램과 관련된 실행 파일, 라이브러리 기타 파일들이 존재

/usr/bin : 응용 프로그램의 실행 파일

/var : 시스템 운용중에 생성 되었다가 삭제되는 데이터를 일시적으로 저장하는 디렉터리

/var/spool : 스풀링 관련(프린터 스풀링, E-mail 스풀링 등등)

/var/log : 로그 파일

d: directory file, l : symbolic link, b: block device file, c : char device file, - : 일반 file

Owner : User, group, other

권한 바꾸기

1) Chmod ugo(+)(-)rwx filename : user , group , other, +(추가), - (제거)

2) Chown : 소유자 바꾸기

3) Chgrp : 그룹을 바꾸기

Ls -al /dev > device : /dev 내용을 device라는 파일에 저장

운영체제에

1) process management

2) memory management

3) file management : device file 형태로 관리 (하드웨어 제어)

- block형 device : ex) 하드디스크

서로 다른 하드디스크 원판이 수직형태로 같은 track number에 있으면 동일하게 취급할 수가 있다.

(마치 cylinder 처럼 보인다.) -> read/write 시간을 줄일 수가 있다.

그러나, 각 디스크마다 track number가 같더라도 partition을 보통 다르게 준다.

<Device file name> :

Logical disk, 그리고 logical disk안에 파일시스템 존재 ->ppt 4 ~ 참고

Ex) /dev/dsk/0s1, /dev/dsk/0s2, ...

디바이스 자체는 내부적으로 driver number로 구분된다.

-Drive number : **major number**(위에서는 0번을 부여함) , **minor number** 로 구분된다.

-각각의 파티션마다 파일시스템(boot, supe block r, i-node list block , data block이 존재한다)

<2.2 file 소개 4번 ppt 그림에서 >

<logical Disk> : 4개로 구성됨

- **Root file system** : OS가 동작하는데 필요한 최소한의 file, directory 존재 ,
  Ex) 커널, /bin, /etc, /lib, /dev
- **Swap file system** : 하드디스크 일부를 메인메모리(DRAM)로 쓴다
- **Usr file system** : utility files (사용자들을 위한 공간)
- **Home file system** : user area, login directory

각각의 파일시스템에

1) boot block : 모든 파일시스템의 1번째 블록

2) super block : File system을 관리하는 정보가 저장(**메타데이터**) **-파일 시스템 총괄 관리**

-메인메모리에 있는 super block과 하드디스크 해당 파티션에 있는 super block이 항상 같아야 한다

-비정상적으로 종료되면, **fsck()**를 사용해서 메인메모리와 하드디스크의 내용이 같은지 확인한다

-사용자가 **'sync'** system call을 통해 수행 할 수도 있다.

3) inode list block : **File 관리에 필요한 정보 저장**

-파일 하나당 대응되는 inode가 1개 있음

-Data block을 가리키는 포인터 주소값이 inode list block에 있다.

File type, number of links, File owner, file protection mode, time stamp, file size, pointer to data block

4) Data block : **실제 데이터**, **directory files**가 있다. **(ppt 11쪽 그림 보기)**

-Directory file에는 inode-no, file name entry가 있다.

-Device file은 data가 없다.

-그러나 inode list block에 device file에 대한 device number가 저장된다.

-Device number은 major no + minor no로 구성된다. Device number은 major no + minor no로 커널 내의 **device 드라이버로 연결해준다**

(디바이스 드라이버는 리컴파일해서 커널 시스템에 녹아져 있거나 run-time 모듈로도 만들 수 있다.)

-> 해당되는 device에 write 하거나 read 하는 역할

Device file 만들기 : - mknod <device file name> <file type> <device no>

File type : c or b

Ex) **sudo mknod /dev/mydevice c 4 150**

Device file을 만들면 major number나 minor number에 대응되는 device driver가 있다.
따라서, 해당 device driver가 특정 하드웨어 physical device를 제어를 하게 된다.

Device switch : 하드웨어 디바이스에 대한 시스템 콜(open, close..) 을 device drivers로 연결한다.

Ex)

Hdopen , hdclose (하드디스크)

bdevsw (block device switch)

cdevsw (character device switch)

예로들어

1) ls -l /dev/ttyS10 하면

Crw-rw-rw- 1 root dialout 4, 74 Aug 31 16 : 25 /dev /ttyS10 이 나오는데

여기서 4 가 major driver number, 74가 minor driver number

C가 character device

<마운트>

/etc/mout /dev/dsk/1s4/ /usr1 ->/dev/dsk/1s4를 /usr1밑에 마운트

- char형 device

<파일 입출력>

-r w x (111)

<Setuid>

Setuid가 세팅(1)되어 있으면 : 파일을 실행하는 동안에 파일을 생성한 사람이 세팅한 범위 안에서 자유롭게 사용할 수 있다. (ex. Passwd파일) -> 자기 자신의 유저 아이디에 속한 패스워드만 바꿀 수 있다

<Setgid>

Setgid : 그룹으로 묶어놨을 때의 경우로 setuid와 똑같음

<Sticky bit>

sticky비트 (공유모드) : /tmp 디렉터리에 대해 유저들끼리 파일 공유 가능
(하지만, 파일을 올린 사람만 삭제 가능하다)

Ex) 0755로 오픈된 testfile이 있다 하면

Chmod 4755 testfile – set uid 되어서 rws r-x r-x

Chmod 6755 testfile – set uid, set gid가 되어 rwsr-s r-x

Chmod 2755 testfile – set gid 되어서 rwxr_s_r_x

Chmod 1755 testfile – set sticky bit가 되어서 rwxr_x_r_t

(ppt 참고)

Umask : 로그인한 유저에 대해 제한된 값 (User mask)

Ex) root user의 umask 의 값 : 0022 이면

->setuid, setgid, sticky는 자유롭고, root user입장에서 본인에 대해 r,w,x 자유롭게 지정할 수 있으나, 파일을 만들어 내더라도 그룹, others에 대해서 w권한을 줄 수는 없다.

-일반 user의 mask 값 : 0022

->setuid, setgid, sticky, 본인은 자유롭지만, 속한 group, other에 대해서는 w권한을 줄 수 없다

-일반 파일인 경우 대부분 텍스트 파일이라 실행이 없다.
초기 u=rwx, g=rwx, o= 의 의미는
Rw- rw- ---(660) umask는 007 이다
디렉토리 파일은 rwx rwx --- (770) umask는 007이다
여기서 7->mask를 다 쒸웠다. R w x 아무것도 줄 수가 없다

-파일의 실제 허가(protection) mode
->AND(mode, NOT(umask))
Ex) 시스템 상에서 umask 0 0 0 2 가 일 때
fileA 0 6 6 6 이면 0 6 6 4 가 된다.

<system call 함수 >

1) int open(const char *pathname, int flags, mode_t mode)

Flag : #include <fcntl.h>
O_APPEND : 파일의 끝에서부터 데이터를 추가하여 쓴다
O_TRUNC : open하면서 파일의 크기를 0으로 만든다
O_CREAT : open하면서 파일이 존재하지 않다면 파일을 만든다.

Mode : #include <sys/stat.h>
S_IRUSR : 소유자에게 읽기 권한
S_IWUSR : 소유자에게 쓰기 권한
S_IXUSR : 소유자에게 실행 권한
S_IROTH : other에게 읽기 권한
S_ISVTX : sticky 비트를 설정
S_ISGID : 셋 그룹 ID 비트를 설정
S_ISUID : 셋 사용자 ID 비트를 설정

2) int close(int fd);

3) ssize_t read(int fd, void *buf, size_t nbytes);

->파일에서 원하는 크기의 데이터를 buf에 저장한다
->**리턴 값 성공 시 읽은 byte수**, **파일의 끝을 만나면 0**, 실패하면 -1 임
->ssize_t : signed integer
Ex) while((ssize_t nread = read(fd, buffer, BUFSIZE)) > 0) {
Total += nread;
}

4) ssize_t write(int fd, void *buf, size_t nbytes)

->**파일에 데이터를 쓴다**

->**리턴 값 성공 시 파일에 쓰여진 데이터의 바이트 수** , 실패하면 -1

->**buf : 쓸 데이터를 저장하고 있는 메모리 공간**

Ex) if((write(1, "system programming", 20)) != 20) {

Write(2, "error", sizeof(error));

}

Ex2)

Char a[30];

Int N = Read(0, a, 10);

Write(1, a, n);

}

5) off_t lseek (int fd, off_t offset, int whence);

->lseek은 파일에 대한 임의접근을 가능하게 한다

->리턴 값 성공하면 현재의 읽기/쓰기 포인터의 위치를 리턴, 실패하면 -1

Offset : **기준점에 더해질 바이트의 수** (양수, 음수 둘 다 가능)

Whence : 기준점

SEEK_SET	0	offset이 파일의 맨 앞에서부터 더해져서 이동
SEEK_CUR	1	현재위치에 offset이 더해져 이동
SEEK_END	2	파일의 마지막 바이트 번호에 더해 이동

Ex) if(lseek(STDIN_FILENO, 0, SEEK_CUR) == -1) {

Printf("cannot seek₩n");

} ->lseek은 표준 입력 모든 형태를 지원하지 않는다

-파일 끝에서부터 쓰는 방법

1) fd = open("filename", **O_RDWR**);

Lseek(fd, (off_t)0, SEEK_END);

Write(fd, outbuf, BUFSIZE);

2) fd = **open(**"filename", O_WRONLY | **O_APPEND**);

Write(fd, outbuf, BUFSIZE);

EX 3)

```
Int fd;
Char buf[20];
Fd = open("seekdata", O_RDWR); // 12345abcde67890fghij
Read(fd, buf, 5);
Write(1, buf, 5);                                    //12345fghij
Lseek(fd, -6, SEEK_END); // 맨 뒤에 NULL 부터해서 0까지 옮겨야함
Read(fd, buf , 5);
Write(1, buf, 5);
```

6) 1. int dup2(int fildes, int fildes2); // 파일 디스크립터, **내가 부여하고 싶은 파일 디스크립터**
2. int Dup(); // **시스템이 알아서 자동으로 부여** (파일 디스크립터 중에서 **가장 작은 번호**)

7) int stat(const char *pathname, struct stat *buf);
Int fstat(int fd, struct stat *buf);
Int lstat(const char *pathname, struct stat *buf);
->**파일 상태에 대한 정보를 stat 구조체에 가져온다**
->리턴값 **성공 시 0**, 실패 시 -1
->buf : stat 구조체(파일정보)를 저장하는 구조체
->lstat()는 대상 파일이 심볼릭 링크인 경우에 링크가 나타내는 파일이 아니라 링크 자신의 상태

```
Struct stat {
        Mode_t      st_mode // 파일 타입,모드에 대한 정보        %o
        Nlink_t     st_nlink // 하드 링크 개수                  %lo
        Ino_t       st_ino  // 파일의 i-node 번호               %ld
        Dev_t       st_dev // i-node가 있는 디바이스의 디바이스 번호     %ld
        Uid_t       st_uid  // 파일 소유자 ID                   %d
        Gid_t       st_gid  // 파일 소유 그룹 ID                %d
        Off_t       st_size //파일의 크기                       %ld
        Time_t      st_atime;       //최종 엑세스 시각          %d
        Time_t      st_mtime        // 최종 변경 시각
        Time_t      st_ctime        //최종 상태 변경 시각 (내용이 아니라 속성)
        Blkcnt_t    st_blocks
}
```

-st_dev : inode가 저장되어 있는 장치의 장치 번호를 저장

8) int access(const char *pathname, int mode)
->**파일**에 대하여 **mode로 전해준 액세스 권한이 있는지 없는지 확인한다**

-> 리턴값 **성공시 0, 실패시 -1**
Mode : **R_OK** : 읽기 //**W_OK :**쓰기 //**X_OK** : 실행 //**F_OK** :파일이 존재?

<File Descriptor table>
-열려진 모든 파일들은 각각 _IO_FILE 이라는 구조체 (/usr/include/libio.h)에 정보가 저장됨
-한 프로세스에 의해 열려진 모든 파일들의 _IO_FILE 타입의 구조체들은 연결리스트로 결합되어지며 이것을 파일 디스크립터 테이블이라고 한다.
-File descriptor table에서 특정 파일의 위치를 가리키는 index 값(int 값)을 file descriptor라고 한다
-File descriptor 값들 중 0,1,2는 특별한 파일들을 위하여 사용된다
0번 : stdin (키보드 - 입력)
1번 : stdout (모니터 – 출력)
2번 : stderr (에러에 관련)

<close () 시스템 콜>
-한 프로세스에서 열 수 있는 최대 파일 수는 limits.h내의 상수 OPEN_MAX에서 정의됨 (/usr/include/bits/posix1_lim.h)

STDIN_FILENO : 파일디스크립터 0 (키보드)
운영체제가 지원하는 파일 시스템의 I/O 단위는 8192 bytes
8192byte의 배수가 아닐 경우는 I/O횟수가 많아져서 성능이 하락할 수가 있다.

-디렉터리에 inode 번호 + 파일 이름이 있는데, inode 번호에 대응되는 것이 inode 리스트의 같은 번호에 링크에 연결된다 (하드링크)
Unix command : ln
-같은 파일시스템에서만 하드링크를 걸 수 있다.

-lrwxrwxrwx 1 root 7 sep ~ lib->/usr/lib 일 때
File name : lib
실제 데이터 : /usr/lib

UNIX command : ln -s /usr/lib lib (심볼릭 링크)
-특정 아이노드에 심볼릭 링크가 걸려있는 개수 : st_link

-같은 아이노드를 가리키고 있으면 파일이름이 달라도 실제 들어가 있는 데이터 내용은 같아서 상관 없다
-link count가 0 이 되면 파일의 inode와 data block이 삭제된다.
(실제는 현재 파일을 사용하는 프로세스가 없다는 것을 확인 한 후 삭제 작업을 한다)

<파일 이동 : mv>
-inode와 data block이 바뀌는게 아니라 directory entry가 바뀌는 것이다
결과적으로 파일을 이동시킨다는 것은 다른 디렉터리로 이동하는 것.
파일 이름 변경도 디렉터리 블록에서 같은 디렉터리 또는 다른 디렉터리로 이동하는 것
(directory block에 있는 file entry를 다른 이름으로 다시 하나 만들고 기존에 있는 것을 delete)

-현재 디렉터리는 . 으로 되어있다. 상위는 . . <점 두개
현재 디렉터리의 이름은 그 디렉터리의 상위 디렉터리에 가야만 현재 디렉터리의 이름이 나온다.

디렉터리를 만들면 그에 연결된 link counter는 최소 3개 (루트만 제외하고)

<명령어 모음>
1) df -il / ~ : 현재 컴퓨터에 올라와 있는 디스크의 파일시스템 정보
2) ls -il : 내 파일들에 대한 아이노드 정보(link counter도 있음)
3) ln f1 f2 : f1에 대한 다른 이름 f2로 하드링크를 건다 (같은 아이노드를 가지지만 다른 이름임)
4) ls -il f*
5)unlink f2 : link를 끊는다
6)ln -s f1 f3 : f1에 f3라는 이름으로 소프트링크 건다
(소프트 링크 타입은 l 로 되어 있다.)

일반 file max 값 : 0666
Directory max 값 : 0777
-directory에 set uid는 일반적으로 주지 않는다. 따라서 최대가 3

<명령어>
1) stat 파일이름 : 해당 파일에 대한 stat 정보(i-node)
Device : 1,5 inode: ~
Access : mon sep 7 : 16 ~ //제일 마지막으로 읽어낸 시간

Modify : ~ //제일 마지막으로 내용을 변경한 시간
Change ~ // 제일 마지막으로 파일의 메타데이터가 바뀐 시간

Ex) link f1 f2 하면 change 타임이 바뀐다 (메타데이터)
-echo "h1" > f1 하면 f1파일에 내용이 써진다

Ex) chmod u-r f1 하면 change 타임만 바뀐다

<Process level>

Level 0 : swapper(scheduler) process : 스스로 실행되면서 스스로 프로세스가 된다
-커널 프로그램의 일부다.
-fork() 시스템 콜을 통해 자식 프로세스를 만들어 낸다 (프로세스마다 프로세스Id가 있다)
-프로세스Id가 0 이면 special kernel proccess이다.
-process 0 번이 자식 프로세스인 프로세스ID를 1번을 가진 자식을 만든다.(Init process임)
-init process가 그 이후로 생성된 모든 프로세스의 조상이 된다.
-/etc/init.d (shell prompt)

프로세스 보기 명령어 : ps -el
<signal> : 실행중인 프로세스를 외부에서 제어하는 수단 : kill() system call
Signal 정보 보기 : kill -l

Level 1 : Kernel process

<시스템 콜 함수 모음>

1)char *getcwd(char *buf, size_t size); == pwd
호출에 성공하면 크기가 size 바이트이고 buf 포인터가 가리키는 버퍼에 현재 작업 디렉터리를 절대 경로로 복사한 다음 buf 포인터를 반환한다. 호출에 실패하면 NULL 반환
<코드>

```
Char buf[100];
DIR *dir;
```

```
Struct dirent *dr;
Dir = opendir(".");
While((dr = readdir(dir)) != NULL) {
        Memset(buf, 0, sizeof(buf));
        Printf("file name is %s₩n", dr->d_name);
        If(!getcwd(buf, 100) {
                Perror("getcwd error");
        }
        Printf("path name : %s₩n", buf);
}
Return 0;
```

```
Char *cwd;
If(Cwd = getcwd(NULL, 0)) == NULL) 로 해도 결과는 똑같음. Buf가 NULL이고 SIZE가 0 이면 C라이브러리는 현재 작업 디렉터리를 저장할 만큼 충분한 버퍼를 할당하고 저장한다.
```

2)현재 작업 디렉터리 바꾸기

-사용자가 시스템에 처음 로그인 하면 login 프로세스는 사용자의 현재 작업 디렉터리를 /etc/passwd 파일에 명시된 홈 디렉터리로 설정한다.

Int chdir(const char *path) // int fchdir(int fd);

->현재 작업 디렉터리를 path가 가리키는 경로로 변경한다.

->호출에 성공하면 0을 반환하고 실패하면 -1을 반환한다.

Ex)

```
If(chdir("./suf") < 0 ) {
        Perror("chdir error");
}
Else {
        If(getcwd(path, MAX) == NULL) {
                Perror("getcwd error");
        }
        Else {
                Printf("current working directory is %s₩n", path);
        }
```

3) int mkdir(const char *path, mode_t mode)

->디렉터리를 생성한다.
성공시 return 0 실패시 -1

4) int symlink(const char *realname, const char *symname)
->성공시 return 0 에러 r-1

5) int readlink(const char *pathname, char *buf, int bufsize);
->**심볼릭 링크 파일의 내용을 읽는다**
->**sumlink()에서 저장된 actualpath가 buf에 채워진다**
->성공 시 읽은 string의 길이 실패 시 -1
->심볼릭 링크를 open으로 열면 링크된 대상 파일이 열린다.

***stricky bit** : 기본적으로 유닉스 운영체제는 파일의 소유자 아니면 해당 디렉터리의 파일을 지우거나 수정하지 못하도록 umask를 설정한다. 그러나, /tmp, /var/tmp는 모든 사용자가 파일을 만들고 수정, 삭제할 수 있다. (permission 777)
문제 : permission이 777이다 보니, 파일들을 아무나 지워버리는 문제가 발생한다(사용중 인데..)
해결 : sticky bit을 설정
1)permission이 777인 파일에 대해서 파일의 소유자만이 삭제할 수 있다(수정, 읽기, 실행은 허용)
2) sticky bit가 설정된 디렉터리 자체도 소유자만이 삭제할 수 있다.

-sticky bit 지정하는 법
->**chmod 1777** stickytest (맨 앞이 sticky bit임)

<프로세스> : 프로그램이 실행이 되려면 커널의 도움(자원)을 받아서 프로세스가 된다.
-프로그램이 실행이 되기 위해서는 code, heap(메모리), stack, value of register 등등 필요하다.
-이 환경들을 커널이 유지 관리한다.
-프로세스도 user process / kernel process(system call 로 변경) 로 나뉜다.
-소프트웨어적 인터럽트(트랩)으로 변경된다. 하드웨어적인 인터럽트는 아님. 인터럽트 발생 내용을 처리하고 다시 user process로 되돌려 준다.

Process level
1)Level 0 : swapper (scheduler)

->하나의 커널에 있는 프로그램인데, 스스로 실행되고 스스로 프로세스가 된다.(유일함)
->fork() 라는 시스템 콜을 통해 child process를 만듬
->process가 만들어지면 유일한 process ID가 생성된다.
->process0이 fork()해서 자식을 만드는데 process1이 되는데 그것을 init process라고 한다.
->init process를 만든 후에 process0은 scheduler가 된다.
2)Level 1 : kernel process

3)Level 2: system process
4) Level 3 :c m d process

<user id>
1) real user id :
2) effective user id : resource 소유자가 아닌 process가 사용하려면 실행가능한 resource에 set uid가 설정되어 있어서 파일의 소유자가 아니더라도 effective user로 사용하도록 할 수 있다.

<signal>
->실행 중인 프로세스를 외부에서 제어하는 수단
->kill() system call을 이용하여 process 간에 signal을 보냄 (소프트웨어 인터럽트)

<프로세스>
-program code(text section)
-data section : 전역 변수 저장 / Stack : 지역 변수 저장

PCB :
1) process state : new, ready, running, waiting ? 상태 기록
2) program counter
3) CPU registers
4) CPU scheduling information
5) mamagement information : base and limit registers, page tables, segment tables
6) I/O status information

-인터럽트가 걸리면 해당 프로세스의 상태를 pcb에 기록하고, 인터럽트한 프로세스의 준비를함(PCB RUN상태로 바꾸고 , CPU reg도 맞추고...) 이 시간이 idle한 시간인데 이걸 **context switching**이라고 함

프로세스 생성

-한 개의 프로세스는 많은 프로세스를 만들 수 있다
-부모가 가지고 있던 자원의 일부를 공유한다 (자식 프로세스도 자신만의 독립된 공간이 있고 거기서 실행이 된다)

프로세스 실행
-특별한 경우가 아니면 자식 프로세스는 부모 프로세스의 program counter를 똑같이 실행한다.
(자식 프로세스에게 다른 일을 시키고 싶으면 자식 프로세스의 공간에 exec을 통해서 코드를 올림)
-자식 프로세스가 종료될 때까지 부모 프로세스는 기다린다. (wait() 시스템 콜)

Fork() : create a new process
Exec() : loads a new program in a process address space made by fork
Wait() : waits until the termination of the child

Exit()
-제일 마지막 명령을 실행하면 프로세스는 종료되는데, **내부적으로 운영체제에게 exit() 시스템 콜을 통해 프로세스를 삭제한다.**
-프로세스는 부모 프로세스에게 wait() system call을 통해 데이터를 리턴 할 수 있다.
-프로세스가 사용했던 메모리, 버퍼 등은 운영체제에 의해 정리된다.
Abort() : 어떤 프로세스가 다른 프로세스를 강제 종료 시킬 수 있다 (권한이 있다면)
Kill() : 어느 특정 프로세스가 자기 자신을 임의로 종료시킬 수 있다.

C-start-up-routine
-exec()을 통해 커널에서 시작된다.
-command-line-arguments(argc, argv[])을 받고, 부모 프로세스로부터 환경(메모리 공간, 레지스터)을 상속받는다. 이것을 기반으로 main()을 호출한다.

프로세스 종료
-정상종료
1)main 에서 return 만났을 때 2) 마지막 명령을 실행할 때
3) exit()을 호출 4)_exit()을 호출

-비정상종료
1) abort() 2)signal로 종료

<시스템 콜 >

1) **void exit(int status);** ->**#include <stdlib.h>**

->아래와 같은 **cleanup 하는 과정을 거친다**

->이 프로세스가 오픈했던 모든 stream을 닫는다

->결과물을 디스크에 옮긴다

2) **void _exit(int status);**

->exit() 처럼 **cleanup 과정을 거치지 않고 바로 커널로 넘기고 삭제한다.**

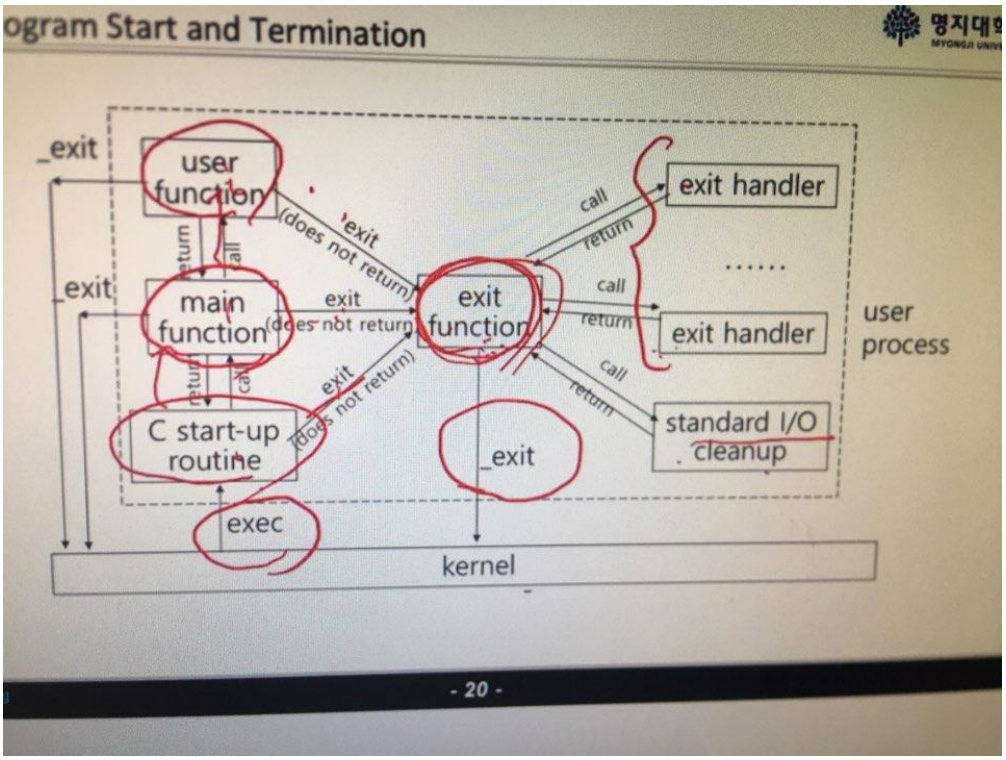
3) **void atexit(void(*func)(void));**

->exit() 하면서 func를 처리한다 (최대 32개 처리할 수 있다)

->func() : exit handler, 함수를 가리키는 포인터

->**exit() 시스템 콜을 하면 atexit()에 있는 exit handler를 call한다**. 실행은 제일 마지막에 등록된 것부터 실행한다.

->강제종료 (abort, signal)을 하면 atexit()에 등록된 함수가 있더라도 실행을 하지 않는다.



-**맨 처음**에 프로그램이 시작될 때 **커널에 exec()**을 통해 시스템 콜을 한다.
-**C start up routine이 가동**을 하고 **main()을 call해서 실행**시킨다.
(argv[argc] 는 NULL char 이다)
-어디서는 exit()은 가능하다. Exit()이 정상적이라면 exit handler에 등록되어 있는 순서 역으로 exit handler 함수를 실행시킨다. 입출력에 관해 등록되어 있는 것들을 cleanup 시키고 파일 디스크립터 등도 정리한다. 마지막으로 exit()도 _exit()을 한다.
-main, user 에서든 정상종료지만 _exit()을 항상 할 수 있다 (바로 커널로 넘어감)
->clean up 하는 일련의 과정은 거치지는 않음

참고)
Int fput(const char *str, FILE *stream);
->**str이 가리키는 문자열을 stream에 쓴다**. Str이 가리키는 문자열을 NULL 문자에 도달할 때 까지 스트림에 복사한다 (마지막 NULL문자는 stream에 복사되지 않는다)
->오류 : EOF
Int fputc(int c, FILE *pfile);
->**file에 문자 하나를 쓴다**
->
Error 함수 ex)

```
Void err_sys(const char *message) {
        Fputs(message, stderr);
        Fpuc('₩n', stderr);
        Exit(1);
}
```

<프로세스 명령어>
1) Ps

PID	**TTY**	**TIME**	**CMD**
8	tty1	00:00:00	bash

프로세스 id **어느 터미널에서 실행중인지? 얼마정도 시간이 걸렸는지? 동원된 명령어**

2) ps -l : 더 자세히

-UID : user id
-PID : process id
-PPID : parent process id
-

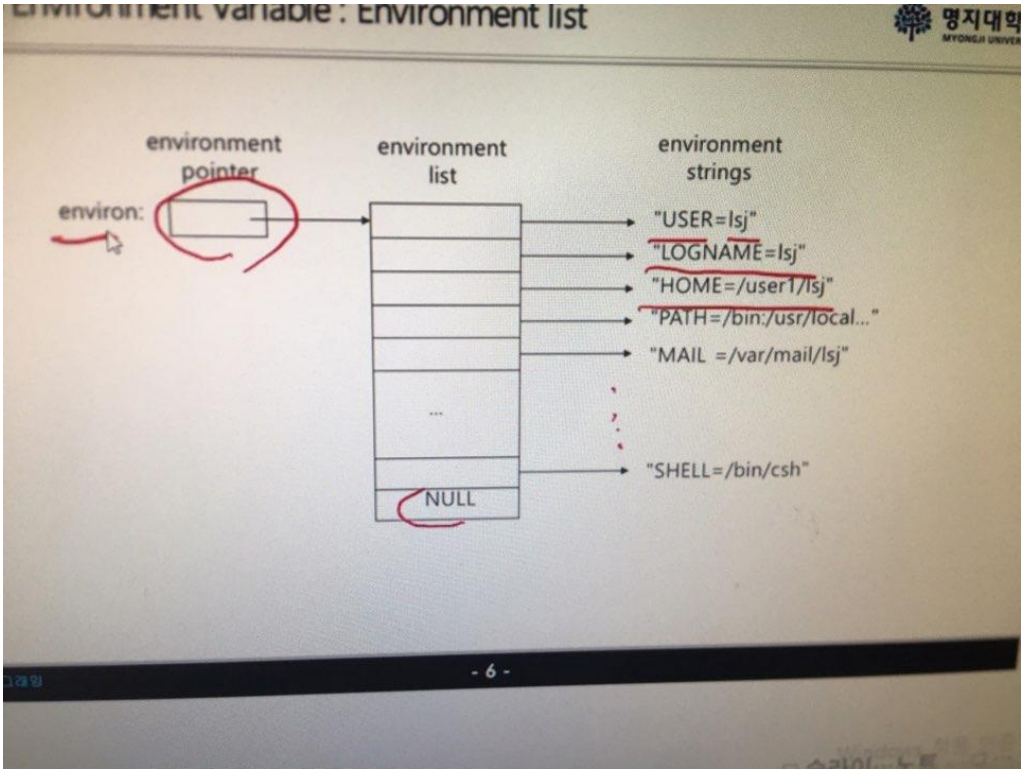
3) ps aux : 실행중인 모든 프로세스 정보 (굉장히 많음)
4**) ps aux | grep bash** : bash쉘과 관련되서 실행중인 프로세스

5) **top** : 온갖 프로세스의 작업 상황(CPU 얼마나 사용..? 등등)

<Environment Variables> : 부모 프로세스로부터 상속 받음 (기본적으로 세팅되어 있음)
->어떤 쉘 ? 어떤 리눅스 배포판을 사용하냐에 따라 환경변수에 들어가 있는 파일명이 다름
기본적으로 .bashrc 아니면 .bash profile에 들어가 있음)

기본 형태 : 환경 변수 이름(name) = 값(value)

명령어로 env 를 치면 환경 변수와 그에 설정된 값을 볼 수 있음
-이 환경 변수를 프로그램 상에서 수정하려면, #include <stdlib.h> 과 extern char **environ; 선언해야
-" 환경 변수 이름 = 값₩0" ←이게 하나의 string 임, 제일 끝은 NULL 포인터



Ex)
#include <stdlib.h>

```
#include <stdio.h>

Extern char **environ;
Int main() {
        Char **env = environ;                   // char* (*env) = environ;
        While(*env) {
                Printf("%s₩n", *env);
                Env++;
        }
        Exit(0);
}
```

<시스템 콜>

1) char *getenv(const char *name)

->주어진 이름의 string을 환경 변수 list 에서 찾는다

->성공적이면 변수 **이름에 해당되는 값이 리턴된다. 변수가 없으면 NULL이 리턴** 된다.

2) int putenv(const char *str);

-> 이미 있던 값이 바뀌거나 없다면 environment list에 이 string이 추가된다

->str : string의 형태로 name=value 를 집어 넣는다

->return -1 이면 에러

Ex)

```
Int main(int argc, char *argv[]) {
        Char *var, *value;
        If(argc == 1 || argc > 2) {
                Fprintf(stderr, "usage : environ var [var]₩n");
                Exit(1);
        }
        Var = argv[1];
        Value = getenv(var);
        If(value) {
                Printf("variable %s has value %s₩n", var, value);
        }
}
```

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

->rewrite : 기존에 어떤 값이 세팅 되어 있다면 그 값을 바꿀 건지 ?
(**rewrite 을 0으로 하면 바꾸지 않겠다**, 0이 아니면 바꾸겠다 의미)
->환경 변수 name = value를 등록한다

4) int unsetenv(const char *name);
->환경 변수 이름을 목록에서 제거한다.

-.bashrc 파일을 찾아서 안에 있는 것을 바꿔야 영구적으로 바뀐다.
-echo $name : 환경 변수에 대응되는 값을 보여준다
-setenv, putenv와 유사하게 외부 명령어로 환경 변수 값을 설정 할 수 있다.
->export "name=value"
->환경 변수 삭제 : unset name

-

<번 외>
환경 변수 : 프로세스가 컴퓨터에서 동작하는 방식에 영향을 미치는 동적인 값들의 모임으로 쉘에서 정의되고 실행하는 동안 프로그램에 필요한 변수를 나타냄

-export 변수 = 값 : 환경변수 등록
-echo$변수 -> 값 나옴
-$변수 → '값' 을 의미
-export PATH=$PATH:~/ : path 값 뒤에다가 home 폴더 경로를 넣는다
-모든 명령어는 /bin 밑에 실행파일로 저장이 되있는데 PATH 는 명령어의 위치를 찾는데 쓰임

동작 범위에 따른 환경 변수
1) 로컬 환경 변수 : 현재 세션에서만 동작하는 환경 변수
2) **사용자 환경 변수** : **특정 사용자에 대해서만 정의된 환경 변수로** 로컬 터미널 세션 또는 원격 로그인 세션을 사용하여 **로그인할 때마다 로드됨**
Ex) 특정 사용자의 홈 디렉터리에 존재하는 파일 : .bashrc, .bash_profile, bash_login, .profile
-.bashrc : 이 파일은 **전역적인 설정 파일인 /etc/bashrc 가 수행된 다음에 바로 수행됨**
****모든 사용자에게 영향을 주는 /etc/bashrc와 달리 .bashrc는 오직 bash를 실행하는 그 사용자에게만 영향을 준다**
-bash_profile : 특정 사용자의 원격 로그인 파일로 이 파일에 있는 환경 변수는 사용자가 원격 로그인

세션이 이루어질 시에 호출
**이 파일은 전역적인 설정 파일인 /etc/profile이 수행된 다음에 바로 수행됨

3) **시스템 전체 환경 변수** : 해당 시스템에 존재하는 모든 사용자가 사용할 수 있는 환경 변수
Ex) /etc/environment, /etc/profile, /etc/profile.d , /etc/bash.bashrc

-/etc/environment : 전반적인 시스템을 제어하는 파일로 필요한 환경 변수를 작성하거나 편집/제거함
이 파일에서 만든 환경 변수는 로컬 및 원격으로 접속한 모든 사용자가 액세스할 수 있음

-/etc/bash.bashrc : 시스템 전체의 bashrc 파일로 모든 사용자가 로컬 터미널 세션을 열 때마다 로드
이 파일에서 만든 환경 변수는 모든 사용자가 액세스할 수 있지만 로컬 터미널 세션에서만 가능

-/etc/profile : 시스템 전체의 profile 파일로 모든 사용자가 원격 로그인 세션이 이루어질 시 호출
이 파일에서 만든 환경 변수는 모든 사용자가 액세스할 수 있지만 원격 로그인 세션에서만 가능함

<리눅스 디렉터리 파일>

1) bin : 일반 사용자들을 위한 명령어
2) boot : 부트로더와 부팅을 위한 파일들이 있다.
3) dev : 장치 파일(시스템 디바이스 파일)들이 있다. 시스템의 모든 장치들이 파일로 표현되어 있음
Udev 라는 데몬이 이곳의 디바이스들을 관리한다.
/dev/had : Master IDE 하드디스크 /dev/hdb : Slave IDE 하드디스크
/dev/cdrom : cdrom 드라이브 /dev/tty0 : 첫번째 가상 콘솔

4) etc : 시스템 혹은 각종 프로그램들의 환경설정 파일들이 있다.
/etc/fstab : 파일 시스템 관리 /etc/group : 유저 그룹 관리
/etc/inittab : init 관리 /etc/passwd : 유저 관리
/etc/service: 포트 관리 /etc/sysconfig/iptables : 방화벽 설정
/etc/sysconfig/network-scripts/ifcfg-eth0 : 네트워크 카드 설정

5) home : 사용자들의 홈 디렉터리
6) lib : 시스템이 있는 프로그램들이 실행할 때 필요한 공유 라이브러리와 커널 모듈이 있다.
7) media : 플로피 디스크, CDROM, DVD, USB와 같은 이동식 디스크가 마운트 되는 곳
8) opt : 추가 패키지 설치
9) mnt : NFS와 같은 파일 시스템이 임시로 마운트 되는 곳

10) proc : 실행중인 프로세스의 정보와 CPU, 메모리 등의 시스템 정보가 가상의 파일로 저장됨
/proc/cpuinfo : cpu 정보 /proc/devices : 현재 커널에 설정되어 있는 장치목록
/proc/filesystems : 현재 커널에 설정되어 있는 파일시스템 목록
/proc/interrupts : 현재 사용중인 인터럽트에 대한 정보
/proc/ioports : 현재 사용중인 I/O 포트 정보 /proc/version : 현재 커널의 버전
/proc/loadavg : 시스템의 평균 부하량 /proc/stat : 시스템 상태

11) sbin : 시스템 관리 명령어들이 있다
Ifconfig : 네트워크 설정/확인 reboot : 시스템 리부팅
Shutdown : 시스템 종료 fsck : 디스크 점검....

12) tmp : 임시 파일들이 저장되는 곳
13) usr : 일반 사용자를 위한 응용 프로그램이 설치되는 곳
/usr/bin : 응용 프로그램의 실행 파일들이 있다
/usr/include : C언어의 헤더파일이 있다.
/usr/local : 대부분의 일반적으로 프로그램은 이곳에 설치한다

14) var : log파일 등 수시로 업데이트 되는 시스템 운영 중에 자주 변경되는 파일들이 있다.
/var/log : 각종 log 파일들이 저장된다. 시간이 지날수록 파일의 용량이 증가한다.

15) lost+found : 부팅시 파일 시스템에 문제가 생길 경우 fsck 명령어로 복구할 때 사용된다.