리눅스시스템 및 응용

Week 10

C Programming Basic in Directory Management

학습목표

- *지난주(9주차) 이어서 ...*
- GNU C Compiler
- 프로그램 오류시 에러 출력
- Memory Allocation Programming 연습
- 명령어행 인자
- 리눅스와 유닉스의 파일

GNU C 컴파일러 : gcc

- GNU C 컴파일러
  - gcc
  - 기능 : C 프로그램을 컴파일해 실행 파일을 생성
  - 형식 : gcc [옵션][파일명]
  - 옵션 -c : 오브젝트 파일(.o)만 생성
  - -o 실행 파일명 : 지정한 이름으로 실행 파일을 생성, 기본 실행 파일명은 a.out
  - 사용 예:
    - **$ gcc test.c**
    - **$ gcc -c test.c**
    - **$ gcc -o test test.c**

- 기본 :
  - **% gcc -o hello.c**
  - **% ./a.out**
- Object file option
  - **% gcc -o hello hello.c**
  - **% ./hello**
- 도움말
  - **% man gcc**
  - **% info gcc** : 추가정보

Header file

- Header file 위치
  - /usr/include/ 아래에 위치
    - **/usr/include/[sys, linux, X11, g++-2, …]**
  - 컴파일 타임 헤더파일
    - **% gcc –I/usr/openwin/include fred.c**
  - 프로그램 내 Header file including
    - **#include <stdio.h>**

Library Linking

- 재사용 가능, 공통사용, 미리 컴파일된 함수
- 라이브러리
  - 정적 라이브러리 static library
  - 동적 라이브러리 dynamic library

- 위치 : /lib /usr/lib
- 분류 :
  - 정적 라이브러리 : .a
  - 공유 라이브러리 : .so, .sa

- Linking방법(default lib directory)
  - **% gcc –o fred fred.c /usr/lib/libm.a**
  - **% gcc –o fred fred.c –lm**

- Linking방법(other lib directory)
  - **% gcc –o xfred –L/usr/openwin/lib xfred.c -Lx11**

- 컴파일 과정
  - 텍스트로 작성한 프로그램을 시스템이 이해할 수 있는 기계어로 변환하는 과정
  - '컴파일을 한다' = 보통 컴파일 과정과 라이브러리 링크 과정을 하나로 묶어서 수행하는 것

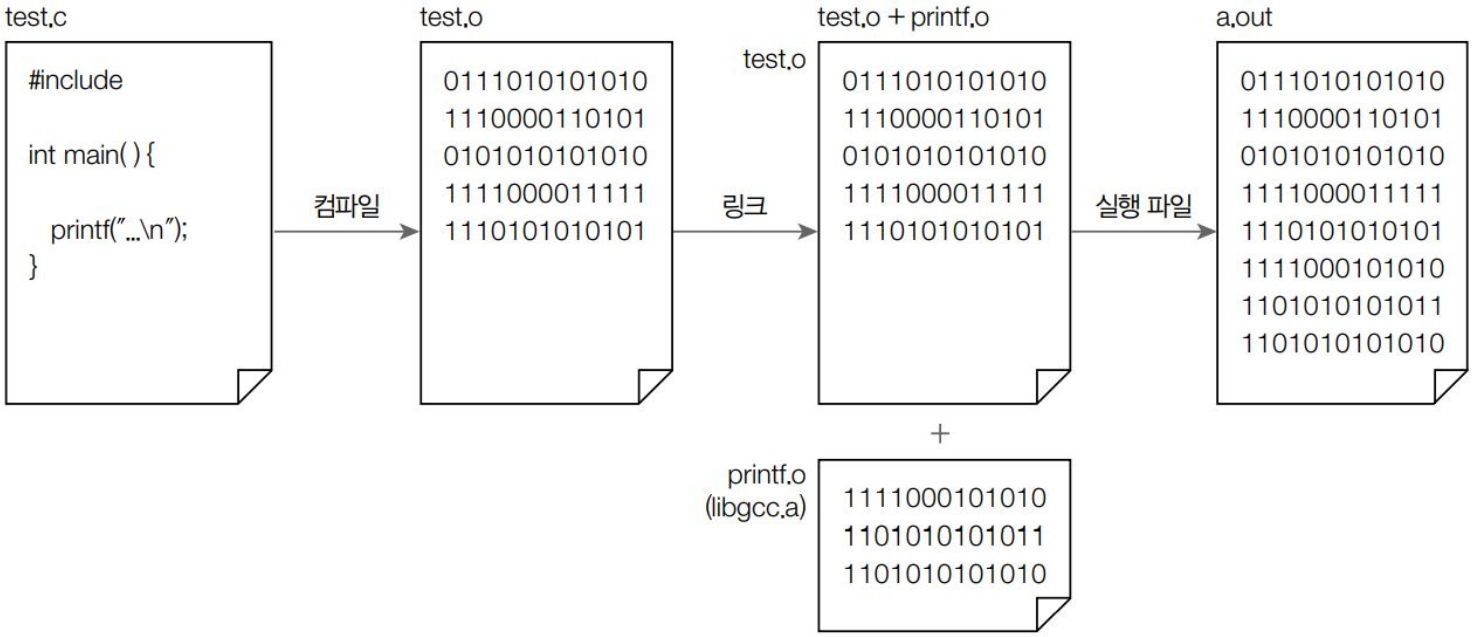

그림 1-3 C 프로그램의 컴파일 과정 예

- gcc
  - 파일명을 별도로 지정하지 않았으므로 a.out이라는 이름으로 실행 파일이 생성

```
$ gcc ch1_2.c
$ ls
a.out ch1_2.c
```

  - 실행 파일명을 ch1_2.out이라고 하려면 다음과 같이 -o 옵션을 사용

```
$ gcc –o ch1_2.out ch1_2.c
$ ls
a.out ch1_2.out ch1_2.c
```

  - 실행 파일명을 입력하면 프로그램이 실행
  - 현재 디렉터리(.)가 경로에 설정되어 있지 않 다면 현재 디렉터리를 지정해 실행
  - 이후에는 현재 디렉터리가 경로에 있다고 가정하여 ./를 표시하지 않음

```
$ gcc –o ch1_2.out ch1_2.c
$ ls
a.out ch1_2.out ch1_2.c
```

오류 메시지 출력

- perror(3)

```
#include <stdio.h>                                    [함수 원형]

void perror(const char *s);
```

  - s : 출력할 문자열

  - perror()함수의 특징
    - 실행 파일과 오브젝트 파일을 모두 삭제하려면 make clean을 수행
    - Perror( ) 함수는 errno에 저장된 값을 읽어 이에 해당하는 메시지를 표준 오류(파일 기술자 2번)로 출력
    - Perror(3) 함수의 인자로는 일반적으로 프로그램 이름을 지정하는 것이 좋음

– perror() 함수로 오류 메시지 출력하기

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <errno.h>
04 #include <stdlib.h>
05
06 int main() {
07     if(access("test.txt", R_OK) == -1) {
08         perror("test.txt");
09         exit(1);
10     }
11 }
```

실행

```
$ ch1_4.out
test.txt: No such file or directory
```

- **07~08행** access() 함수에서 오류가 발생하면 perror() 함수를 호출한다. 이때 perror() 함수의 인자로 "test.txt"를 지정
- **09행** perror() 함수는 오류 메시지 출력만 하므로 오류의 결과로 프로그램을 종료해야 한다면 exit() 함수를 호출해야 함
- **실행 결과** 인자로 전달한 문자열 "test.txt"와 콜론이 출력되고 한 칸 띄어서 메시지가 출력

- strerror(3)

```
#include <string.h>                                    [함수 원형]

char *strerror(int errnum);
```

  - errnum : errno에 저장된 값

  - strerror() 함수의 특징
    - strerror() 함수는 ANSI C에서 추가로 정의한 함수
    - 함수의 인자로 errno에 저장된 값을 받아 오류 메시지를 리턴
    - 리턴된 오류 메시지를 사용자가 적절하게 가공할 수 있다는 장점

– strerror() 함수로 오류 메시지 출력하기

```
01 #include <stdio.h>
02 #include <unistd.h>
03 #include <errno.h>
04 #include <stdlib.h>
05 #include <string.h>
06
07 extern int errno;
08
09 int main() {
10       char *err;
11
12       if(access("test.txt", R_OK) == -1) {
13            err = strerror(errno);
14            printf("오류: %s(test.txt)₩n", err);
15            exit(1);
16       }
17 }
```

실행

```
$ ch1_5.out
오류: No such file or directory(test.txt)
```

- **3행** strerror() 함수는 인자로 받은 errno 변수에 저장된 오류 번호에 따라 오류 메시지가 저장된 문자열을 가리키는 포인터를 리턴
- **14행** 13행에서 리턴한 문자열을 적절한 형태로 가공해 오류 메시지를 출력
- **실행 결과** 14행에서 지정한 대로 출력

Memory Allocation

- malloc(3)

```
#include <stdlib.h>                                    [함수 원형]

void *malloc(size_t size);
```

  - size : 할당받을 메모리 크기
  - malloc() 함수의 특징
    - 인자로 지정한 크기의 메모리를 할당하는 데 성공하면 메모리의 시작 주소를 리턴
    - 만약 메모리 할당에 실패하면 NULL 포인터를 리턴
    - 인자로 지정하는 메모리 크기는 바이트 단위
    - 할당된 메모리에는 어떤 형태의 데이터도 저장할 수 있음
    - malloc() 함수는 할당된 메모리를 초기화하지 않는다는 데 주의
    - 요소가 10개이고 각 요소의 크기가 20바이트인 배열을 저장할 수 있는 메모리를 할당 하는 예

```
char *ptr
ptr = calloc(10*20);
```

- calloc(3)

```
#include <stdlib.h>                                    [함수 원형]

void *calloc(size_t nmemb, size_t size);
```

  - nmemb : 배열 요소의 개수
  - size : 할당받을 메모리 크기

  - calloc() 함수의 특징
    - calloc() 함수는 nmemb×size바이트 크기의 배열을 저장할 메모리를 할당
    - calloc() 함수는 할당된 메모리를 0으로 초기화

    - 요소가 10개이고 각 요소의 크기가 20바이트인 배열을 저장할 수 있는 메모리를 할당 하는 예

```
char *ptr
ptr = calloc(10, 20);
```

- ralloc(3)

```
#include <stdlib.h>                                    [함수 원형]

void *realloc(void *ptr, size_t size);
```

  - ptr : 할당받은 메모리를 가리키는 포인터
  - size : 할당받을 메모리 크기

  - ralloc() 함수의 특징
    - realloc() 함수는 이미 할당받은 메모리에 추가로 메모리를 할당할 때 사용
    - 이전에 할당받은 메모리와 추가할 메모리를 합한 크기의 메모리를 새롭게 할당하고 주소를 리턴
    - 이때 이전 메모리의 내용을 새로 할당된 메모리로 복사
    - malloc() 함수로 할당받은 메모리에 추가로 100바이트를 할당하는 예

```
char *ptr, *new;
ptr = malloc(sizeof(char) * 100);
new = realloc(ptr, 100);
```

- free(3)

```
#include <stdlib.h>                                    [함수 원형]

void free(void *ptr);
```

  - ptr : 해제할 메모리 주소

  - free() 함수의 특징
    - free() 함수는 사용을 마친 메모리를 해제하고 반납
    - free() 함수가 성공하면 ptr이 가리키던 메모리는 더 이상 의미가 없음

Memory Allocation Examples

- 간단한 메모리 할당 (복습)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define A_MEGABYTE (1024 * 1024)

int main() {
   char *some_memory;
   int megabyte = A_MEGABYTE;
   some_memory = (char *)malloc(megabyte);
   if (some_memory != NULL) {
         sprintf(some_memory, "Hello World\n");
         printf("%s", some_memory);
   }
}
```

- 모든 물리 메모리 할당
  - 리눅스에 배정된 모든 메인메모리 할당. 실제 프로그램은 그 이전에 강제 종료 가능성

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#define A_MEGABYTE (1024 * 1024)
int main() {
   char *some_memory;
   size_t size_to_allocate = A_MEGABYTE;
   int megs_obtained = 0;
   while (megs_obtained < 10000) {
      some_memory = (char *)malloc(size_to_allocate);
      if (some_memory != NULL) {
           megs_obtained = megs_obtained + size_to_allocate;
           sprintf(some_memory, "Hello World");
           printf("%s-now allocated %d Megabytes\n“,some_memory,megs_obtained);
      }
      else { printf(“\n Memory allocation failure\n”); exit(1)}
   }
}
```

- 유효한 메모리 크기 측정 (Segmentation Fault 가능성 있음)

```
#include <unistd.h>
 #include <stdlib.h>
 #include <stdio.h>
 #define ONE_K (1024)
 int main() {
    char *some_memory;
    int size_to_allocate = ONE_K;
    int megs_obtained = 0;
    int ks_obtained = 0;
    while (1) {
       for (ks_obtained = 0; ks_obtained < 1024; ks_obtained++) {
       some_memory = (char *)malloc(size_to_allocate);
       if (some_memory == NULL) printf("allocation failure\n");
          sprintf(some_memory, "Hello World");
       }
       megs_obtained = megs_obtained + ONE_K;
       printf("Now allocated %d Megabytes\n", megs_obtained);
    }
    printf("allocation success\n");
}
```

- 메모리 남용해 보기
  - Character String Array에서 NULL은 End of String 의미

```
#include <unistd.h>
#include <stdlib.h>
#define ONE_K (1024)

int main() {
   char *some_memory;
   char *scan_ptr;
   some_memory = (char *)malloc(ONE_K);
   if (some_memory == NULL)
      printf(“allocation failure\n”);
   scan_ptr = some_memory;
   while(1) {
      *scan_ptr = '\0’;
      scan_ptr++;
   }
   printf(“allocation success\n”);
}
```

- Null Pointer 접근
  - memory 공간에 자리 잡지 않은(즉 할당이 안된) 데이터
  - Unix/Linux에서는 허용하지 않음

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

int main() {
   char *some_memory = (char *)0;
   printf("A read from null %s\n", some_memory);
   sprintf(some_memory, "A write to null\n");
}
```

  - 결과:
    - A read from null (null) : GNU C library에서는 null 문자 제공
    - segmentation fault(core dumped) : null string에 쓰기는 error

- 다른 null pointer 접근 예

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main() {
   char z = (const char *)0;
   printf("I read from location zero\n");
}
```

  - segmentation fault(core dumped)
  - memory 위치 0로 부터 직접 읽기, 이는 허용되지 않음

- free()

```
#include <stdlib.h>
#define ONE_K (1024)
int main(){
   char *some_memory;
   some_memory = (char *)malloc(ONE_K);
   if (some_memory != NULL) {
        free(some_memory);
   }
}
```

명령행 인자

- 명령행
  - 리눅스 시스템에서 사용자가 명령을 입력하는 행
  - 프롬프트가 나타나고 커서가 사용자 입력을 기다리고 있는 행
  - option parameter(argv)에 따라 프로그램의 제어는 달라진다

  - 명령행 인자 (CLA)
    - 사용자가 명령행에서 명령을 실행할 때 해당 명령(실행 파일명)과 함께 지정하는 인자
    - 명령행 인자는 명령의 옵션, 옵션의 인자, 명령의 인자로 구성
  - 명령행 인자의 전달
    - 보통 main() 함수는 다음과 같이 정의

```
int main() {...}
(또는)
int main(void) {...}
```

    - main() 함수에서 명령행 인자를 전달받으려면 다음과 같이 정의

```
int main(int argc, char *argv[]) {...}
```

– 명령행 인자 출력하기

```
01 #include <stdio.h>
02
03 int main(int argc, char *argv[]) {
04        int n;
05
06        printf("argc = %d₩n", argc);
07        for (n = 0; n < argc; n++)
08              printf("argv[%d] = %s₩n", n, argv[n]);
09 }
```

실행

```
$ ch1_6.out -h 2000
argc = 3
argv[0] = ch1_6.out
argv[1] = -h
argv[2] = 2000
```

- **03행** 명령행 인자를 받기 위해 main() 함수에 argc와 argv를 선언한다.
- **06행** 인자의 개수를 저장한 argc 값을 출력한다.
- **07~08행** 각 인자를 담은 argv의 내용을 출력한다.
- **실행 결과** 명령행에서 실행 파일명인 ch1_6.out 외에 -h와 2000을 인자로 입력
  따라서 main() 함수에 전달된 총 개수를 나타내는 argc 값은 3
  argv[0]에 실행 파일명이 저장되고, 차례로 인자가 저장됨을 알 수 있음
  argv로 전달되는 값은 문자열이므로 printf() 함수로 출력하려면 형식 지정자 %s를 사용해야 함

File in Linux and Unix

- Everything is FILE in Linux & Unix

- File property
  - 이름,생성/변경날짜, 허용권한,길이,inode정보

- Low level file handling system call
  - /dev 아래의 모든 device에 동일한 방법 적용

- 주요 system calls
  - open(), read(), write() : close(), ioctl()

- File I/O descriptor
  - 0 (표준입력), 1 (표준출력), 3 (표준에러)

- 리눅스와 디렉터리
  - 리눅스의 파일 구분
    - 리눅스에서는 파일을 일반 파일과 특수 파일, 디렉터리로 구분
    - 디렉터리는 해당 디렉터리에 속한 파일을 관리하는 특별한 파일

  - 리눅스의 파일 구성
    - 파일명: 사용자가 파일에 접근할 때 사용
    - Inode: 파일의 소유자나 크기 등의 정보와 실제 데이터를 저장하고 있는 데이터 블록의 위치를 나타내는 주소들이 저장
    - 데이터 블록: 실제로 데이터가 저장되는 하드디스크의 공간

- 리눅스와 디렉터리
  - 리눅스의 함수

표 2-1 디렉터리 생성과 삭제 함수

기능	함수
디렉터리 생성	`int mkdir(const char *pathname, mode_t mode);`
디렉터리 삭제	`int rmdir(const char *pathname);`

표 2-2 디렉터리 관리 함수

기능	함수
현재 위치 확인	`char *getcwd(char *buf, size_t size);`
	`char *get_current_dir_name(void);`
디렉터리명 변경	`int rename(const char *oldpath, const char *newpath);`
디렉터리 이동	`int chdir(const char *path);`
	`int fchdir(int fd);`

- 파일의 종류
  - 크게 일반 파일, 특수 파일, 디렉터리로 구분

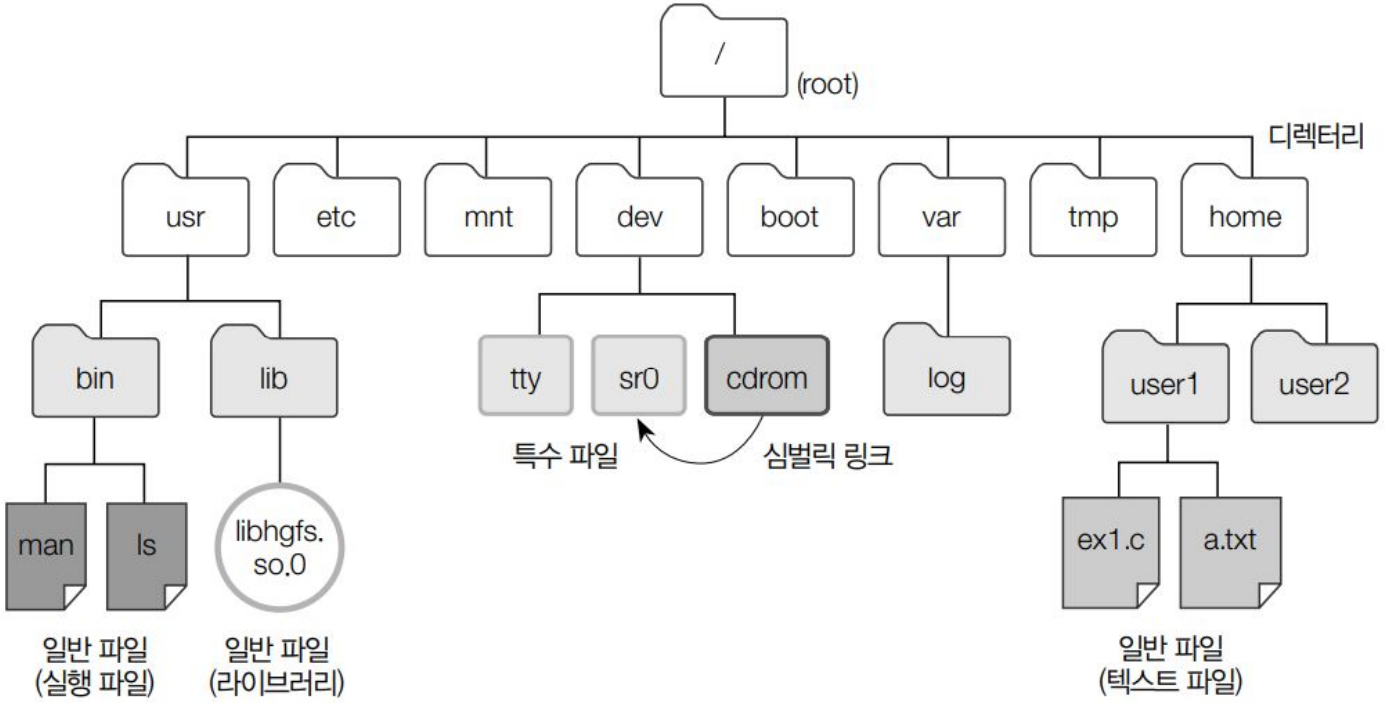

그림 2-1 리눅스 파일의 종류

- 파일의 종류
  - 일반 파일
    - 텍스트 파일, 실행 파일, 라이브러리, 이미지 등 리눅스에서 사용하는 대부분의 파일이 일반 파일에 해당
    - 데이터 블록에 텍스트나 바이너리 형태의 데이터를 저장하고 있음
    - vi 같은 편집기를 사용해 만들기도 하고 컴파일러나 다른 응용 프로그램에서 생성할 수 도 있음

```
$ ls -l /usr/bin
합계 177456
lrwxrwxrwx 1 root root        11 1월 27 16:12 GET -> lwp-request
lrwxrwxrwx 1 root root        11 1월 27 16:12 HEAD -> lwp-request
lrwxrwxrwx 1 root root        11 1월 27 16:12 POST -> lwp-request
-rwxr-xr-x 1 root root    141856 6월 22 2020 VGAuthService
lrwxrwxrwx 1 root root         4 1월 17 18:13 X -> Xorg
lrwxrwxrwx 1 root root         1 1월 27 16:12 X11 -> .
-rwxr-xr-x 1 root root   2434568 1월 17 18:13 Xephyr
-rwxr-xr-x 1 root root       274 1월 17 18:13 Xorg
-rwxr-xr-x 1 root root   2324456 1월 17 18:13 Xwayland
(생략)
```

- 파일의 종류
  - 특수 파일
    - 리눅스에서 통신을 하거나 터미널 또는 디스크 등의 장치를 사용할 때 연관된 특수 파일을 이용
    - 장치 사용하지 않는 대신 장치의 종류를 나타내는 장치 번호를 inode에 저장
    - 장치 관련 특수 파일을 다른 파일과 구분해 장치 파일이라고도 함
    - 데이터 블록을 일에는 블록 장치 파일과 문자 장치 파일이 있음
      - » 블록 장치 파일은 블록 단위로 데이터를 읽고 씀
      - » 문자 장치 파일은 하드 디스크인 경우 섹터 단위로 읽고 씀

```
$ ls -l /dev
합계 0
crw------- 1 root root      10, 175   2월 14 20:19 agpgart
crw-r--r-- 1 root root       10,235   2월 14 20:19 autofs
drwxr-xr-x 2 root root         360   2월 14 20:19 block
drwxr-xr-x 2 root root          80   2월 14 20:19 bsg
crw------- 1 root root      10, 234   2월 14 20:19 btrfs-control
drwxr-xr-x 3 root root          60   2월 14 20:19 bus
lrwxrwxrwx 1 root root           3   2월 14 20:19 cdrom -> sr0
lrwxrwxrwx 1 root root           3   2월 14 20:19 cdrw -> sr0
drwxr-xr-x 2 root root        3720   2월 14 20:19 char
crw--w---- 1 root tty         5, 1   2월 14 20:20 console
(생략)
```

- 파일의 종류
  - 디렉터리
    - 리눅스에서는 디렉터리도 파일로 취급
    - 디렉터리와 연관된 데이터 블록은 해당 디렉터 리에 속한 파일의 목록과 inode를 저장
    - 디렉터리를 생성하려면 mkdir, 삭제하려면 rmdir 또는 rm –r, 복사하려면 cp -r 명령을 사용

```
$ ls -l /
합계 1190428
lrwxrwxrwx   1  root  root          7 1월 27 16:12 bin -> usr/bin
drwxr-xr-x   4  root  root       4096 2월 11 17:02 boot
drwxrwxr-x   2  root  root       4096 1월 27 16:15 cdrom
drwxr-xr-x  19  root  root       4180 2월 14 20:19 dev
drwxr-xr-x 130  root  root      12288 2월 20 09:26 etc
drwxr-xr-x   3  root  root       4096 1월 27 16:18 home
(생략)
```

- 파일의 종류 구분
  - ls -l 명령을 사용하면 파일의 종류를 알 수 있음
  - 명령의 결과 중 파일의 권한을 표시하는 부분인 -rwxr-x-x에서 맨 앞의 하이픈(-)이 파일의 종류를 나타냄

표 2-4 파일의 종류 식별 문자

문자	파일의 종류
-	일반 파일
d	디렉터리
b	블록 장치 특수 파일
c	문자 장치 특수 파일
l	심벌릭 링크

```
$ ls -l /dev
합계 0
(생략)
drwxr-xr-x   2 root   root         60  2월 14 20:19 lightnvm
lrwxrwxrwx   1 root   root         28  2월 14 20:19 log -> /run/systemd/journal/dev-log
crw-rw----   1 root   disk    10, 237  2월 14 20:19 loop-control
brw-rw----   1 root   disk       7, 0  2월 14 20:19 loop0
(생략)
```

- 파일의 구성 요소
  - 파일명
    - 사용자가 파일에 접근할 때 사용하며 파일명과 관련된 inode가 반드시 있어야 함
    - 유닉스에서는 예전에는 시스템 파일명으로 최대 14자까지 사용할 수 있었지만, 현재는 255바이트까지 사용할 수 있음
    - 리눅스에서는 255바이트보다 긴 파 일명도 사용할 수 있음
    - 파일명이나 디렉터리명은 /와 null 문자를 제외하고 아무 문자나 사용할 수 있음
    - 그러나 출력이 가능 한 문자를 사용하고 혼동을 줄 수 있는 특수문자는 사용을 자제하는 것이 관례

    - 파일명 지정 주의 사항
      - » 파일명과 디렉터리명에 사용하는 알파벳은 대소문자를 구분
      - » 파일명과 디렉터리명이 '.'으로 시작하면 숨김 파일로 간주

- inode
  - 파일에 대한 정보를 저장하고 있는 객체로, 실제로 디스크에 저장되어 있음
  - 리눅스 커널의 입장에서는 파일의 정보를 관리하는 자료 구조로 사용
  - inode는 외부적으로는 번호로 표현하며, 내부적으로는 두 부분으로 나누어 정보를 저장

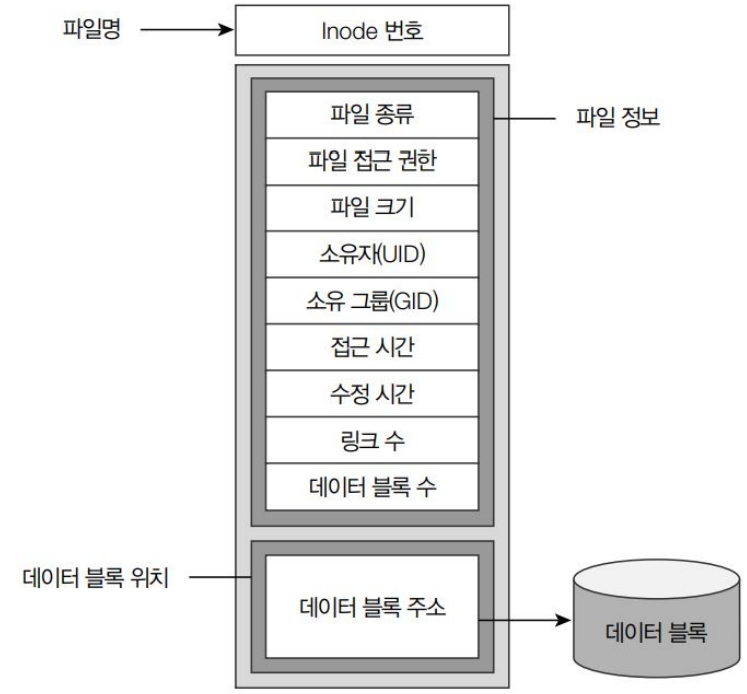

그림 2-2 inode의 구조

- inode
  - inode의 첫 번째 부분에는 파일에 관한 정보가 저장
    - » (파일 종류, 파일 접근 권한, 파일 크기, 소유자, 소유 그룹, 파일 변경 시각, 하드 링크 수, 데이터 블록 수 등)
  - ls –l 명령은 inode의 정보를 읽어서 출력
  - 두 번째 부분에는 파일의 실제 데이터가 저장되어 있는 데이터 블록의 위치를 나타내는 주소들이 저장
  - 파일의 inode 번호는 ls –i 명령으로 알 수 있음

```
$ ls -i
1048595 src     1073157 다운로드    1073156 바탕화면    1073162 사진    1073158 템플릿
1073159 공개    1073160 문서        1073163 비디오      1073161 음악
```

- 데이터 블록
  - 실제로 데이터가 저장되는 하드 디스크의 공간
  - 일반 파일이나 디렉터리, 심벌릭 링크는 데이터 블록에 관련 내용을 직접 저장
  - 장치 파일은 데이터 블록을 사용하지 않고 장치에 관한 정보를 inode에 저장

03. 디렉터리 생성과 삭제

■ 디렉터리 생성 : mkdir(2)

```
#include <sys/stat.h>                                    [함수 원형]
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
```

- pathname : 디렉터리가 포함된 경로
- mode : 접근 권한

- mkdir()의 특징
  - mkdir() 함수는 생성하려는 디렉터리명을 포함한 경로를 받고, 생성하는 디렉터리의 기본 접근 권한을 지정
  - mkdir() 함수는 수행에 성공하면 0을, 실패하면 –1을 리턴

03. 디렉터리 생성과 삭제

■ [예제 2-1] 디렉터리 생성하기

```
01 #include <sys/stat.h>
02 #include <stdlib.h>
03 #include <stdio.h>
04
05 int main() {
06     if (mkdir("LINUX", 0755) == -1) {
07         perror("Making LINUX Directory");
08         exit(1);
09     }
10 }
```

- **06행** 접근 권한을 755로 지정해 LINUX 디렉터리를 생성
- **실행 결과** LINUX 디렉터리가 생성

03. 디렉터리 생성과 삭제

■ [예제 2-2] 디렉터리 삭제하기

```
01 #include <unistd.h>
02 #include <stdlib.h>
03 #include <stdio.h>
04
05 int main() {
06      if (rmdir("LINUX") == -1) {
07          perror("LINUX");
08          exit(1);
09      }
10 }
```

- **06행** 현재 디렉터리에서 LINUX 디렉터리를 찾아 삭제
- **실행 결과** LINUX 디렉터리가 삭제

03. 디렉터리 생성과 삭제

■ 현재 작업 디렉터리의 위치 검색 1 : getcwd(3)

```
#include <sys/stat.h>                                    [함수 원형]

char *getcwd(char *buf, size_t size);
```

- buf : 현재 디렉터리의 절대 경로를 저장할 버퍼 주소
- size : 버퍼의 크기

- getcwd()의 특징
  - getcwd() 함수는 경로를 저장할 버퍼의 주소와 버퍼의 크기를 인자로 받음
  - 인자를 지정하는 방법: 2 가지 방법
    ① buf에 경로를 저장할 만큼 충분한 메모리를 할당하고 그 크기를 size에 지정
    ② buf에 NULL을 지정하고 할당이 필요한 메모리 크기를 size에 지정
    ③ buf에 NULL을 지정하고 size는 0으로 지정

■ [예제 2-3] 현재 디렉터리의 위치 검색하기 1

```
01 #include <unistd.h>
02 #include <stdlib.h>
03 #include <stdio.h>
04
05 int main() {
06     char *cwd;
07     char wd1[BUFSIZ];
08     char wd2[10];
09
10     getcwd(wd1, BUFSIZ);
11     printf("wd1 = %s₩n", wd1);
12
13     cwd = getcwd(NULL, BUFSIZ);
14     printf("cwd1 = %s₩n", cwd);
15     free(cwd);
16
17     cwd = getcwd(NULL, 0);
18     printf("cwd2 = %s₩n", cwd);
19     free(cwd);
20
21     if(getcwd(wd2, 10) == NULL) {
22         perror("getcwd");
23         exit(1);
24     }
25 }
```

■ [예제 2-3] 현재 디렉터리의 위치 검색하기 1

```
01 #include <unistd.h>
02 #include <stdlib.h>
03 #include <stdio.h>
04
05 int main() {
06     char *cwd;
07     char wd1[BUFSIZ];
08     char wd2[10];
09
10     getcwd(wd1, BUFSIZ);
11     printf("wd1 = %s₩n", wd1);
12
13     cwd = getcwd(NULL, BUFSIZ);
14     printf("cwd1 = %s₩n", cwd);
15     free(cwd);
16
17     cwd = getcwd(NULL, 0);
18     printf("cwd2 = %s₩n", cwd);
19     free(cwd);
20
21     if(getcwd(wd2, 10) == NULL) {
22         perror("getcwd");
23         exit(1);
24     }
25 }
```

- **06행** 시스템이 할당하는 버퍼의 주소를 저장하기 위한 포인터 변수
- **07행** 경로를 저장하기 위한 배열로, 크기는 BUFSIZ
  BUFSIZ는 stdio.h 파일에 8192로 정의되어 있음
  따라서 wd1은 8KB 크기의 배열
- **08행** 경로를 저장하기 위한 배열로, 크기는 10바이트
- **10~11행** wd1 배열에 현재 경로를 저장하고 이를 출력
- **13~15행** cwd 포인터에 BUFSIZ만큼 메모리를 할당하고 이 메모리에 경로를 저장
  15행에서는 free() 함수를 사용해 메모리를 해제
- **17~19행** getcwd() 함수의 인자로 NULL과 0을 지정
  이 경우 시스템이 자동으로 경로에 필요한 메모리를 할당하고 주소를 리턴
  19행에서 사용이 끝난 메모리를 해제
- **21~24행** getcwd() 함수의 인자로 저장할 경로보다 크기가 작은 버퍼를 지정
  getcwd()함수 처리에서 오류가 발생하면 NULL을 리턴
- **실행 결과** 11행, 14행, 18행에서는 경로를 정상적으로 출력
  22행에서 오류 메시지를 출력
  오류 메시지는 결과가 메모리의 범위를 벗어났다는 뜻

03. 디렉터리 생성과 삭제

■ 현재 작업 디렉터리 위치 검색 2 : get_current_dir_name(3)

```
#include <unistd.h>                                    [함수 원형]

char * get_current_dir_name();
```

- void : 함수로 전달할 인자가 없다는 뜻

- get_current_dir_name() 의 특징
  - 현제 디렉터리의 절대 경로를 리턴(스트링으로 반환값)
  - 인자로 아무것도 전달하지 않으며, 시스템이 메모리를 자동으로 할당해 경로를 저장하고 리턴

03. 디렉터리 생성과 삭제

■ [예제 2-4] 현재 디렉터리의 위치 검색하기 2

```
01 #define _GNU_SOURCE
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06
07 int main() {
08     char *cwd;
09
10     cwd = get_current_dir_name();
11     printf("cwd = %s₩n", cwd);
12     free(cwd);
13 }
```

- **01행** #define문을 사용해 _GNU_SOURCE를 정의
- **10행** get_current_dir_name() 함수를 사용해 경로를 검색
  시스템이 메모리를 자동으로 할당하고 경로를 저장해 리턴한다.
- **11행** 10행에서 리턴한 경로를 출력한다.
- **12행** 메모리 사용이 끝났으므로 free() 함수로 메모리를 해제

03. 디렉터리 생성과 삭제

■ 디렉터리명 변경 : rename(2)

```
#include <stdio.h>                                    [함수 원형]

int rename(const char *oldpath, const char *newpath);
```

- oldpath : 변경할 파일/디렉터리명
- newpath : 새 파일/디렉터리명

- rename()의 특징
  - 만약 두 번째 인자로 지정한 이름이 이미 있으면 해당 디렉터리를 삭제
  - 실행 도중 오류가 발생하면 원본과 새로운 디렉터리명이 모두 남음
  - rename() 함수는 수행에 성공하면 0을, 실패하면 -1을 리턴함
  - rename() 함수는 파일명을 변경하는 데도 사용할 수 있음
  - rename() 함수의 매뉴얼을 보려면 man –s 2 rename을 이용해야 함

03. 디렉터리 생성과 삭제

■ [예제 2-5] 디렉터리명 변경하기

```
01 #include <sys/stat.h>
02 #include <stdlib.h>
03 #include <stdio.h>
04
05 int main() {
06       if (rename( " LINUX", "LINUXforever") == -1) {
07           perror("rename");
08           exit(1);
09       }
10 }
```

- **06행** 디렉터리명을 LINUX에서 LINUXforever로 변경
- **실행 결과** 디렉터리명이 LINUX에서 LINUXforever로 바뀐 것을 알 수 있음
- 만약 에러메시지가 출력된다면 LINUX 디렉토리가 현재 없는 경우임. 따라서 LINUX 디렉토리를 만들고 난 뒤 프로그램을 테스트해야 함 !

03. 디렉터리 생성과 삭제

■ 디렉터리 이동 1 : chdir(2)

```
#include <unistd.h>                                    [함수 원형]

int chdir(const char *path);
```

- path: 이동하려는 디렉터리 경로

- chdir()의 특징
  - 이동하려는 디렉터리의 경로를 인자로 받으며, 절대 경로와 상대 경로 모두 사용할 수 있음
  - chdir() 함수는 수행에 성공하면 0을, 실패하면 -1을 리턴

- chdir()과 다음에 나오는 fchdir()은 프로그램 속에서 워킹디렉토리 위치를 변경할 때 사용하는 함수. 따라서 shell에서는 변화가 없다

03. 디렉터리 생성과 삭제

■ [예제 2-6] 디렉터리 이동하기 1

```
01 #include <unistd.h>
02 #include <stdio.h>
03 #include <stdlib.h>
04
05 int main() {
06 char *cwd;
07
08 cwd = getcwd(NULL, BUFSIZ);
09 printf("1.Current Directory: %s₩n", cwd);
10
11 chdir("bit");
12
13 cwd = getcwd(NULL, BUFSIZ);
14 printf("2.Current Directory: %s₩n", cwd);
15
16 free(cwd);
17 }
```

- **08~09행** 현재 디렉터리의 경로를 getcwd() 함수로 읽어 출력
- **11행** bit 디렉터리로 이동
- **13~14행** 현재 디렉터리의 경로를 getcwd() 함수로 읽어 출력
- **실행 결과** 디렉터리가 이동, 그런데 디렉터리 이동은 프로그램 내부에서만 진행된 것으로 pwd 명령으로 확인하면 현재 디렉터리가 바뀐 것은 아님을 알 수 있음

03. 디렉터리 생성과 삭제

■ 디렉터리 이동 2 : fchdir(2)

```
#include <unistd.h>                                    [함수 원형]

int fchdir(int fd);
```

- fd : 이동하려는 디렉터리의 파일 디스크립터

- chdir()의 특징
  - 파일 디스크립터를 인자로 받음 파일
    - » 파일 디스크립터: open()함수로 디렉터리를 열고 돌려받는 것
  - fchdir() 함수를 사용하려면 open() 함수로 해당디렉터리를 먼저 열어야 함
  - fchdir() 함수는 수행에 성공하면 0을, 실패하면 –1을 리턴

03. 디렉터리 생성과 삭제

■ [예제 2-7] 디렉터리 이동하기 2

```
01 #include <fcntl.h>
02 #include <unistd.h>
03 #include <stdio.h>
04 #include <stdlib.h>
05
06 int main() {
07     char *cwd;
08     int fd;
09
10     cwd = getcwd(NULL, BUFSIZ);
11     printf("1.Current Directory: %s₩n", cwd);
12
13     fd = open("bit", O_RDONLY);
14     fchdir(fd);
15
16     cwd = getcwd(NULL, BUFSIZ);
17     printf("2.Current Directory: %s₩n", cwd);
18
19     close(fd);
20     free(cwd);
21 }
```

- **01행** O_RDONLY는 fcntl.h 파일에 정의되어 있음
- **10~11행** 현재 디렉터리의 경로를 getcwd() 함수로 읽어 출력
- **13행** open() 함수로 bit 디렉터리를 실행
- **14행** fchdir() 함수를 사용해 디렉터리를 이동
  open() 함수가 리턴한 파일 디스크립터를 인자로 지정
- **16~17행** 현재 디렉터리의 경로를 getcwd() 함수로 읽어 출력
- **19~20행** close() 함수로 열린 디렉터리를 닫고 free() 함수로 메모리를 해제
- **실행 결과** 디렉터리가 이동, pwd 명령으로 확인해보면 프로그램 내부에서만 디렉터리가 바뀐 것을 알 수 있음

Question?