파일 접근권한 변경

- 파일명으로 접근권한 변경 : umask – test1.c

```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t newmask);
```

  - 각 프로세스에는 파일 생성 마스크(file creation mask)가 설정
  - 파일이 생성될 때 어떤 모드가 주어지든지 자동적으로 허가 비트를 0으로 만들기 위해 사용 (지정된 허가가 우연히 켜지는 것을 방지)

fd = open(filename, O_CREAT, mode) 는

fd = open(filename, O_CREAT, (~mask)&mode) 와 동일

예) fd = open(filename, O_CREAT, 0644)에 umask = 007을 적용하면

기타의 100 & 000 (~111) = 000

=> fd = open(filename, O_CREAT, 0640)

fd = open(filename, O_CREAT, 0666)에 umask = 022를 적용하면

그룹의 110 & 101 (~010) = 100

기타의 110 & 101 (~010) = 100

=> fd = open(filename, O_CREAT, 0644)

04. 링크 파일 생성

■ 링크

- 링크의 정의
  - 이미 있는 파일이나 디렉터리에 접근할 수 있는 새로운 이름을 의미
  - 같은 파일이나 디렉터리지만 여러 이름으로 접근할 수 있게 하는 것
  - 심벌릭 링크와 하드 링크로 나뉨
  - 링크 기능을 이용하면 사용자 편의성을 높일 수 있음
    - ex) 이전 시스템과의 호환성을 유지, 경로가 복잡한 파일에 간단한 경로를 제공

- 하드 링크
  - 파일에 접근할 수 있는 파일명을 새로 생성하는 것으로, 기존 파일과 동일한 inode를 사용
  - 하드 링크를 생성하면 inode에 저장된 링크 개수가 증가

- 심벌릭 링크
  - 기존 파일에 접근할 수 있는 다른 파일을 만듬
  - 기존 파일과 다 른 inode를 사용하며, 기존 파일의 경로를 저장

04. 링크 파일 생성

■ 하드 링크 생성 : link(2)

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

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

- oldpath : 기존 파일의 경로
- newpath : 새로 생성할 링크의 경로

- link() 함수의 특징
  - 하드 링크를 생성할 때 사용
  - 기존 파일의 경로를 첫 번째 인자인 oldpath로 , 새로 생성할 하드 링크 파일의 경로를 두 번째 인자인 newpath로 받음
  - 하드 링크는 같은 파일 시스템에 있어야 하므로 두 경로를 반드시 같은 파일 시스템으로 지정해야 함
  - 수행에 성공하면 0을, 실패하면 –1을 리턴

04. 링크 파일 생성

■ [예제 3-8] link() 함수로 하드 링크 생성하기(test2.c)

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <unistd.h>
04 #include <stdio.h>
05
06 int main() {
07     struct stat statbuf;
08
09     stat("linux.txt", &statbuf);
10     printf("Before Link Count = %d₩n", (int)statbuf.st_nlink);
11
12     link("linux.txt", "linux.ln");
13
14     stat("linux.txt", &statbuf);
15     printf("After Link Count = %d₩n", (int)statbuf.st_nlink);
16 }
```

실행

```
$ ls -l linux.*
-rwxrwx--- 1 jw jw 219  2월 28 17:54 linux.txt
$ ch3_8.out
Before Link Count = 1
After Link Count = 2
$ ls -l linux.*
-rwxrwx--- 2 jw jw 219  2월 28 17:54 linux.ln
-rwxrwx--- 2 jw jw 219  2월 28 17:54 linux.txt
$ ls -i linux.*
1048600 linux.ln  1048600 linux.txt
```

- **09~10행** 하드 링크를 생성하기 이전의 링크 개수를 확인
- **12행** linux.txt 파일의 하드 링크로 linux.ln을 생성
- **14~15행** 하드 링크를 생성한 이후의 링크 개수를 확인
- **실행 결과** linux.ln 파일이 생성되었고, 링크 개수가 변경
  linux.txt와 linux.ln 파일의 inode가 1048600으로 동일하다는 것도 알 수 있음

04. 링크 파일 생성

■ 심벌릭 링크 생성 : symlink(2)

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

int symlink(const char *target, const char *linkpath);
```

- target : 기존 파일의 경로
- linkpath : 새로 생성할 심벌릭 링크의 경로

- symlink() 함수의 특징
  - 기존 파일의 경로를 첫 번째 인자인 target으로 받고 새로 생성할 심벌릭 링크의 경로를 두 번째 인자인 linkpath로 받음
  - 심벌릭 링크는 기존 파일과 다른 파일 시스템에도 생성할 수 있음
  - symlink( ) 함수는 수행에 성공하면 0을, 실패하면 –1을 리턴

04. 링크 파일 생성

■ [예제 3-9] symlink() 함수로 심벌릭 링크 생성하기(test3.c)

```
01 #include <unistd.h>
02
03 int main() {
04        symlink("linux.txt", "linux.sym");
05 }
```

실행

```
$ ls -l linux.*
-rwxrwx--- 2 jw jw 219  2월 28 17:54 linux.ln
-rwxrwx--- 2 jw jw 219  2월 28 17:54 linux.txt
$ ch3_9.out
$ ls -l linux.*
-rwxrwx--- 2 jw jw 219  2월 28 17:54 linux.ln
lrwxrwxrwx 1 jw jw   9  3월  4 20:12 linux.sym -> linux.txt
-rwxrwx--- 2 jw jw 219  2월 28 17:54 linux.txt
```

- **04행** linux.txt 파일의 심벌릭 링크인 linux.sym을 생성
- **실행 결과** linux.sym 파일이 생성
  심벌릭 링크를 생성했지만 기존 파일에는 아무 변화가 없음을 알 수 있음

04. 링크 파일 생성

■ 심벌릭 링크의 정보 검색 : lstat(2)

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

int lstat(const char *pathname, struct stat *statbuf);
```

- pathname : 심벌릭 링크의 경로
- statbuf : 새로 생성할 링크의 경로

- lstat() 함수의 특징
  - lstat() 함수는 심벌릭 링크 자체의 파일 정보를 검색
  - 심벌릭 링크를 stat() 함수로 검색 하면 원본 파일에 대한 정보가 검색된다는 점에 주의
  - 첫 번째 인자로 심벌릭 링크가 온다는 것을 제외하면 stat() 함수와 동일한 방식으로 사용

04. 링크 파일 생성

■ [예제 3-10] 심벌릭 링크의 정보 검색하기(test4.c)

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <unistd.h>
04 #include <stdio.h>
05
06 int main() {
07      struct stat statbuf;
08
09      printf("1. stat : linux.txt ---₩n");
10      stat("linux.txt", &statbuf);
11      printf("linux.txt : Link Count = %d₩n", (int)statbuf.st_nlink);
12      printf("linux.txt : Inode = %d₩n", (int)statbuf.st_ino);
13
14      printf("2. stat : linux.sym ---₩n");
15      stat("linux.sym", &statbuf);
16      printf("linux.sym : Link Count = %d₩n", (int)statbuf.st_nlink);
17      printf("linux.sym : Inode = %d₩n", (int)statbuf.st_ino);
18
19      printf("3. lstat : linux.sym ---₩n");
20      lstat("linux.sym", &statbuf);
21      printf("linux.sym : Link Count = %d₩n", (int)statbuf.st_nlink);
22      printf("linux.sym : Inode = %d₩n", (int)statbuf.st_ino);
23 }
```

실행

```
$ ls -li linux.*
1048600 -rwxrwx--- 2 jw jw 219  2월 28 17:54 linux.ln
1048710 lrwxrwxrwx 1 jw jw   9  3월  4 20:12 linux.sym -> linux.txt
1048600 -rwxrwx--- 2 jw jw 219  2월 28 17:54 linux.txt
$ ch3_10.out
1. stat : linux.txt ---
linux.txt : Link Count = 2
linux.txt : Inode = 1048600
2. stat : linux.sym ---
linux.sym : Link Count = 2
linux.sym : Inode = 1048600
3. lstat : linux.sym ---
linux.sym : Link Count = 1
linux.sym : Inode = 1048710
```

- **10~12행** 기존 linux.txt 파일의 정보를 stat() 함수로 검색해 링크 개수와 inode 번호를 출력
- **15~17행** stat() 함수를 사용해 심벌릭 링크 파일인 linux.sym의 정보를 검색하고 링크 개수와 inode 번호를 출력
- **20~22행** lstat() 함수를 사용해 심벌릭 링크 파일인 linux.sym의 정보를 검색하고 링크 개수와 inode 번호를 출력
- **실행 결과** 10행과 15행에서 검색한 정보가 동일함 즉, 심벌릭 링크의 정보를 stat() 함수로 검색하면 원본 파일의 정보가 검색되는 것을 알 수 있음
  20행에서 lstat() 함수로 검색한 심벌릭 링크의 정보는 링크 파일 자체에 대한 정보로, 15행과 다른 값이 출력

04. 링크 파일 생성

■ 심벌릭 링크의 내용 읽기 : readlink(2)

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

ssize_t readlink(const char *pathname, char *buf, size_t bufsiz);
```

- pathname : 심벌릭 링크의 경로
- buf : 읽어온 내용을 저장할 버퍼
- bufsiz : 버퍼의 크기

- readlink() 함수의 특징
  - 심벌릭 링크의 경로를 받아 해당 파일의 내용을 읽음
  - 호출시 읽은 내용을 저장할 버퍼와 해당 버퍼의 크기를 인자로 지정
  - 수행에 성공 하면 읽어온 데이터의 크기(바이트 수)를, 실패하면 –1을 리턴

04. 링크 파일 생성

■ [예제 3-11] readlink() 함수로 심벌릭 링크의 내용 읽기(test5.c)

```
01 #include <sys/stat.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 int main() {
07     char buf[BUFSIZ];
08     int n;
09
10     n = readlink("linux.sym", buf, BUFSIZ);
11     if (n == -1) {
12         perror("readlink");
13         exit(1);
14     }
15
16     buf[n] = '₩0';
17     printf("linux.sym : READLINK = %s₩n", buf);
18 }
```

실행

```
$ ch3_11.out
linux.sym : READLINK = linux.txt
$ ls -l linux.sym
lrwxrwxrwx 1 jw jw 9  3월  4 20:12 linux.sym -> linux.txt
```

- **10행** linux.sym 파일의 내용을 readlink() 함수로 읽어 buf에 저장
- **16~17행** buf의 끝에 널을 추가하고 저장된 내용을 출력한
- **실행 결과** 심벌릭 링크를 ls -l 명령으로 확인했을 때 '->' 다음에 오는 원본 파일의 경로가 심벌릭 링크의 데이터 블록에 저장된 내용
  심벌릭 링크의 크기는 'linux.txt'의 바이트 수인 9임을 알 수 있음

04. 링크 파일 생성

■ 심벌릭 링크 원본 파일의 경로 읽기 : realpath(3)

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

char *realpath(const char *path, char *resolved_path);
```

- path : 심벌릭 링크의 경로명
- resolved_path : 경로명을 저장할 버퍼 주소

- realpath() 함수의 특징
  - 심벌릭 링크명을 받아 실제 경로명을 resolved_name에 저장
  - 성공하면 실제 경로명이 저장된 곳의 주소를, 실패하면 널 포인터를 리턴

04. 링크 파일 생성

■ [예제 3-12] realpath() 함수로 원본 파일의 경로 읽기

```
01 #include <sys/stat.h>
02 #include <stdlib.h>
03 #include <stdio.h>
04
05 int main() {
06     char buf[BUFSIZ];
07
08     realpath("linux.sym", buf);
09     printf("linux.sym : REALPATH = %s₩n", buf);
10 }
```

실행

```
$ ch3_12.out
linux.sym : REALPATH = /home/jw/src/ch3/linux.txt
```

- **08~09행** realpath() 함수를 사용해 심벌릭 링크인 linux.sym의 원본 파일 경로를 얻고 출력

04. 링크 파일 생성

■ 링크 끊기 : unlink(2)

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

int unlink(const char *pathname);
```

- pathname : 삭제할 링크의 경로

- unlink() 함수의 특징
  - unlink() 함수에서 연결을 끊은 경로명이 그 파일을 참조하는 마지막 링크라면 파일은 삭제
  - 만약 인자로 지정한 경로명이 심벌릭 링크이면 링크가 가리키는 원본 파일이 아니라 심벌릭 링크 파일이 삭제

04. 링크 파일 생성

■ [예제 3-12] realpath() 함수로 원본 파일의 경로 읽기

```
01 #include <sys/stat.h>
02 #include <unistd.h>
03 #include <stdio.h>
04
05 int main() {
06     struct stat statbuf;
07
08     stat("linux.ln", &statbuf);
09     printf("1.linux.ln : Link Count = %d₩n", (int)statbuf.st_nlink);
10
11     unlink("linux.ln");
12
13     stat("linux.txt", &statbuf);
14     printf("2.linux.txt: Link Count = %d₩n", (int)statbuf.st_nlink);
15
16     unlink("linux.sym");
17 }
```

실행

```
$ ls -l linux.*
-rwxrwx--- 2 jw jw 219 2월 28 17:54 linux.ln
lrwxrwxrwx 1 jw jw   9 3월  4 20:12 linux.sym -> linux.txt
-rwxrwx--- 2 jw jw 219 2월 28 17:54 linux.txt
$ ch3_13.out
1.linux.ln: Link Count = 2
2.linux.txt: Link Count = 1
$ ls -l linux.*
-rwxrwx--- 1 jw jw 219 2월 28 17:54 linux.txt
```

- **08~09행** stat() 함수로 linux.ln 파일의 정보를 검색해 링크 개수를 출력
- **11행** unlink() 함수를 사용해 하드 링크 파일인 linux.ln의 링크를 끊어 삭제
- **13~14행** stat() 함수로 원본 파일인 linux.txt 파일의 정보를 검색해 링크 개수를 출력
- **16행** unlink() 함수를 사용해 심벌릭 링크 파일인 linux.sym의 링크를 끊어 삭제
- **실행 결과** 09행에서 검색한 하드 링크 파일의 링크 개수는 2
  11행에서 unlink() 함수로 하드 링크를 삭제한 뒤 원본 파일의 링크 개수를 확인한 결과 1로 변경됨
  심벌릭 링크 파일이 삭제된 것을 알 수 있음

■ 실습(1)

IT@COOKBOOK

시스템 프로그래밍

리눅스&유닉스

Chapter 04 파일 입출력

목차

학습목표

- 파일 입출력의 특징을 이해한다.
- 저수준의 파일 입출력 함수를 사용할 수 있다.
- 고수준의 파일 입출력 함수를 사용할 수 있다.
- 임시 파일을 생성해 파일 입출력을 할 수 있다.

01. 개요

■ 파일

▪ 파일

- 관련 있는 데이터의 집합으로, 저장 장치에 일정한 형태로 저장
- 데이터를 저장하는 데는 물론 데이터를 전송하거나 장치에 접근하는 데도 사용
- 리눅스에서 파일은 크게 일반 파일과 특수 파일로 구분
- 특수 파일의 생성과 삭제 및 입출력은 특수 파일별로 약간씩 차이가 있음

표 4-1 파일의 종류

종류	용도
일반 파일	텍스트나 바이너리 형태의 자료를 저장하는 파일
특수 파일	데이터 전송 또는 장치 접근에 사용하는 파일

01. 개요

■ 파일

▪ 파일 읽고 쓰는 방법

- 저수준 파일 입출력
  - 리눅스 커널의 시스템 호출을 이용해 파일 입출력을 수행
  - 시스템 호출을 이용하므로 파일에 좀 더 빠르게 접근할 수 있는 장점
  - 또한 바이트 단위로 파일의 내용을 다루므로 일반 파일뿐만 아니라 특수 파일도 읽고 쓸 수 있음
  - 바이트 단위로만 입출력을 수행 가능 하므로 응용프로그램 작성시 다른 추가기능을 함수로 추가 구현 해야함
  - 열린 파일을 참조할 때 파일 기술자 사용

- 고수준 파일 입출력
  - 저수준 파일 입출력의 불편함을 해결하기 위해 제공
  - C 언어의 표준 함수로 제공
  - 데이터를 바이트 단위로 한정하지 않고 버퍼를 이용해 한꺼번에 읽기와 쓰기를 수행
  - 다양한 입출력 데이터 변환 기능도 이미 구현되어 있어 자료형에 따라 편리하게 이용할 수 있음
  - 열린 파일을 참조할 때 파일 포인터 사용

01. 개요

■ 파일

- 파일 읽고 쓰는 방법

표 4-2 저수준 파일 입출력과 고수준 파일 입출력 비교

	저수준 파일 입출력	고수준 파일 입출력
파일 지시자	int fd	FILE *fp;
특징	– 훨씬 빠름 – 바이트 단위로 읽고 쓰기 – 특수 파일에 대한 접근 가능	– 사용하기 쉬움 – 버퍼 단위로 읽고 쓰기 – 데이터의 입출력 동기화가 쉬움 – 여러 가지 형식을 지원
주요 함수	open(), close(), read(), write(), dup(), dup2(), fcntl(), lseek(), fsync()	fopen(), fclose(), fread(), fwrite(), fputs(), fgets(), fprintf(), fscanf(), freopen(), fseek()

02. 저수준 파일 입출력

■ 파일 기술자

▪ 파일 기술자

- 현재 열려 있는 파일을 구분할 목적으로 시스템에서 붙여놓은 번호
- 저수준 파일 입출력에서는 열 린 파일을 참조하는 데 사용하는 지시자 역할 수행
- 파일 기술자는 정숫값으로, open() 함수를 사용해 파일을 열 때 부여
- 프로세스가 파일을 열 때 파일 기술자에는 0번부터 순서대로 가장 작은 번호가 자동 할당
- 0번부터 시작하여 0번, 1번, 2번 파일 기술자에는 기본적인 용도가 지정되어 있음
  - 0번: 표준 입력
  - 1번: 표준 출력
  - 2번: 표준 오류 출력

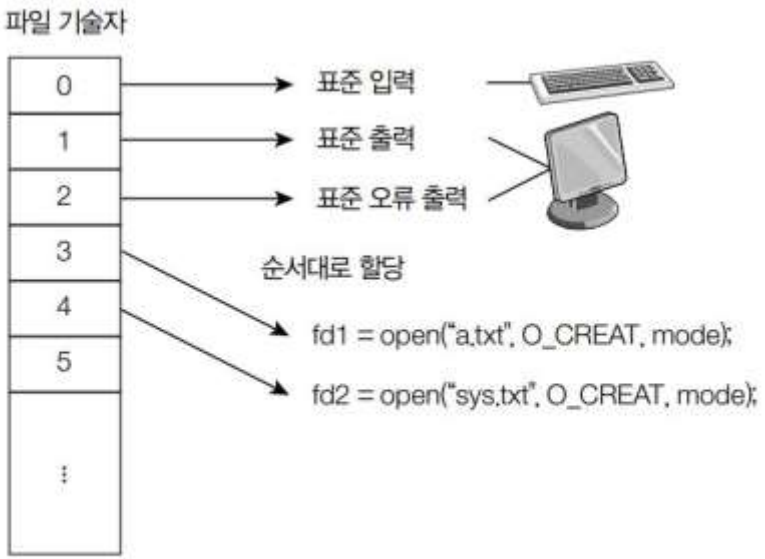

그림 4-1 파일 기술자 할당

02. 저수준 파일 입출력

■ 파일 열기 : open(2)

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

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

- pathname : 열려는 파일이 있는 경로
- flags : 파일 상태 플래그
- mode : 접근 권한

- open() 함수의 특징
  - pathname에 지정한 파일을 lags에 지정한 상태 플래그의 값에 따라 열고 파일 기술자를 반환
  - 파일의 상태를 조정 하는 flags 값은 man 명령으로 확인할 수 있음

02. 저수준 파일 입출력

■ 파일 열기 : open(2)

- flags의 주요 플래그

표 4-3 flags에 사용하는 주요 플래그

종류	기능
O_RDONLY	파일을 읽기 전용으로 연다.
O_WRONLY	파일을 쓰기 전용으로 연다.
O_RDWR	파일을 읽기/쓰기용으로 연다.
O_CREAT	파일이 없으면 생성한다. 파일을 생성할 권한은 당연히 있어야 한다. 파일이 이미 있다면 아무 의미 없는 옵션이다.
O_EXCL	O_CREAT 옵션과 함께 사용할 경우 기존에 없는 파일이면 파일을 생성하고, 이미 있으면 파일을 생성하지 않고 오류 메시지를 출력한다.
O_APPEND	이 옵션을 지정하면 파일의 맨 끝에 내용을 추가한다.
O_TRUNC	파일을 생성할 때 이미 있는 파일이고 쓰기 옵션으로 열었으면 내용을 모두 지우고 파일 길이를 0으로 변경한다.
O_SYNC/ O_DSYNC	파일에 쓰기 동작을 할 때는 보통 버퍼에만 쓰고 나중에 디스크와 같은 저장 장치로 옮겨 쓰는데, 이 옵션이 설정되어 있으면 저장 장치에 쓰기를 마쳐야 쓰기 동작을 완료한다. O_SYNC 플래그는 파일의 수정 시각 속성도 수정할 때까지 기다린다. 이 옵션을 설정하면 프로그램 실행 속도는 느려질 수 있지만 디스크에 확실하게 저장됨을 보장한다.

02. 저수준 파일 입출력

■ 파일 열기 : open(2)

- 플래그 지정
  - 쓰기 전용으로 열 때(이미 파일이 있는 경우)

```
O_WRONLY | O_TRUNC
```

  - 쓰기 전용으로 열 때(파일이 없는 경우)

```
O_WRONLY | O_CREAT | O_TRUNC
```

  - 읽기/쓰기/추가용으로 열 때

```
O_RDWR | O_APPEND
```

  - 상수를 이용해 0644 권한을 지정

```
mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
```

02. 저수준 파일 입출력

■ 파일 생성 : creat(2)

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

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

- pathname : 파일을 생성할 경로
- mode : 접근 권한

- creat() 함수의 특징
  - open() 함수에 파일 생성 기능이 없던 구 버전 유닉스에서 사용하던 것
  - open() 함수와 달리 플래그를 지정하는 부분이 없음
  - creat() 함수로 파일을 생성하면 파일 기술자가 리턴되므로 별도로 open() 함수를 호출해 파일을 열 필요가 없음

  - 다음에서 creat() 함수와 open() 함수는 같은 의미

```
creat(pathname, mode);
open(pathname, O_CREAT | O_WRONLY | O_TRUNC, mode);
```

02. 저수준 파일 입출력

■ 파일 닫기 : close(2)

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

int close(int fd);
```

- fd : 파일 기술자

- close() 함수의 특징
  - 한 프로세스가 열 수 있는 파일 개수에 제한이 있으므로 파일 입출력 작업을 모두 완료하면 반드시 파일 을 닫아야 함
  - 파일을 성공적으로 닫으면 0을 리턴
  - 파일 닫기에 실패하면 -1을 리턴하고 오류 코드를 외부 변수 errno에 저장

02. 저수준 파일 입출력

■ [예제 4-1] 새 파일 생성하기 (test6.c)

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <fcntl.h>
04 #include <unistd.h>
05 #include <stdlib.h>
06 #include <stdio.h>
07
08 int main() {
09     int fd;
10     mode_t mode;
11
12     mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
13
14     fd = open("test.txt", O_CREAT, mode);
15     if (fd == -1) {
16         perror("Creat");
17         exit(1);
18     }
19     close(fd);
20 }
```

실행

```
$ ls test.txt
ls: 'test.txt'에 접근할 수 없습니다: 그런 파일이나 디렉터리가 없습니다
$ ch4_1.out
$ ls -l test.txt
-rw-r--r-- 1 jw jw 0  3월  6 22:57 test.txt
```

- **12행** 파일 접근 권한을 OR 연산으로 지정
- **14행** test.txt 파일을 12행에서 지정한 권한(0644)으로 생성
- **15~18행** 파일 생성 도중 오류가 발생하면 리턴값이 -1이 되므로 16행에서 perror() 함수를 사용해 오류 메시지를 출력 17행에서는 exit( ) 함수를 사용해 프로그램의 실행을 종료
- **실행 결과** test.txt 파일이 생성되었고 접근 권한이 0644임을 알 수 있음

02. 저수준 파일 입출력

■ [예제 4-2] O_EXCL 플래그 사용하기

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <fcntl.h>
04 #include <unistd.h>
05 #include <stdlib.h>
06 #include <stdio.h>
07
08 int main() {
09     int fd;
10
11     fd = open("test.txt", O_CREAT | O_EXCL, 0644);
12     if (fd == -1) {
13         perror("Excl");
14         exit(1);
15     }
16     close(fd);
17 }
```

실행

```
$ ls test.txt
test.txt
$ ch4_2.out
Excl: File exists
```

```
$ rm test.txt
$ ch4_2.out
$ ls test.txt
test.txt
```

- **11행** 모드를 0644와 같이 숫자로 지정하고 O_EXCL 플래그를 같이 지정
  이 경우 test.txt 파일이 이미 있으면 오류 메시지를 출력
- **12~15행** 파일 생성 도중 오류가 발생하면 오류 메시지를 출력하고 프로그램의 실행을 종료
- **첫 번째 실행 결과** 파일이 있을 경우 "Excl: File exists"라는 오류 메시지를 출력
  test.txt파일은 [예제 4-1]에서 이미 생성했으므로 오류가 발생한 것
- **두 번째 실행 결과** 파일을 지운 후 다시 실행하면 파일을 생성

02. 저수준 파일 입출력

■ [예제 4-2] O_EXCL 플래그 사용하기

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <fcntl.h>
04 #include <unistd.h>
05 #include <stdlib.h>
06 #include <stdio.h>
07
08 int main() {
09     int fd;
10
11     fd = open("test.txt", O_CREAT | O_EXCL, 0644);
12     if (fd == -1) {
13         perror("Excl");
14         exit(1);
15     }
16     close(fd);
17 }
```

- 16행에서 파일을 닫기 전에 다음과 같은 printf 문을 추가해 파일 기술자를 출력

```
printf("test.txt : fd = %d₩n", fd);
```

실행

```
$ rm test.txt
$ ch4_2.out
test.txt: fd = 3
```

02. 저수준 파일 입출력

■ [예제 4-3] 파일 기술자 할당하기

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 #include <fcntl.h>
04 #include <unistd.h>
05 #include <stdlib.h>
06 #include <stdio.h>
07
08 int main() {
09     int fd;
10
11     close(0);
12
13     fd = open("test.txt", O_RDWR);
14     if (fd == -1) {
15         perror("Open");
16         exit(1);
17     }
18
19     printf("test.txt : fd = %d₩n", fd);
20     close(fd);
21 }
```

실행

```
$ ch4_3.out
test.txt : fd = 0
```

- **11행** 0번 파일 기술자를 닫음
- **13행** 남아 있는 파일 기술자 중 가장 작은 숫자가 0이므로 test.txt 파일을 열고, 할당되는 파일 기술자는 실행 결과와 같이 0이 됨

02. 저수준 파일 입출력

■ 파일 읽기 : read(2)

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

ssize_t read(int fd, void *buf, size_t count);
```

- fd : 파일 기술자
- buf : 파일에 기록할 데이터를 저장한 메모리 영역
- count : buf의 크기(기록할 데이터의 크기)

- read() 함수의 특징
  - 파일 기술자가 가리키는 파일에서 count에 지정한 크기만큼 바이트를 읽어 buf로 지정한 메모리 영역에 저장
  - 실제로 읽은 바이트 수를 리턴하며, 오류가 발생하면 -1을 리턴
  - 만일 리턴값이 0이면 파일의 끝에 도달해 더 이상 읽을 내용이 없음을 의미
  - 파일에 저장된 데이터가 텍스트든 이미지든 상관없이 무조건 바이트 단위로 읽음

02. 저수준 파일 입출력

■ [예제 4-4] 파일 읽기

```
01 #include <fcntl.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 int main() {
07     int fd, n;
08     char buf[10];
09
10     fd = open("linux.txt", O_RDONLY);
11     if (fd == -1) {
12         perror("Open");
13         exit(1);
14     }
15
16     n = read(fd, buf, 5);
17     if (n == -1) {
18         perror("Read");
19         exit(1);
20     }
21
22     buf[n] = '₩0';
23     printf("n=%d, buf=%s₩n", n, buf);
24     close(fd);
25 }
```

실행

```
$ cat linux.txt
Linux System Programming
$ ch4_4.out
n=5, buf=Linux
```

- **10행** linux.txt 파일을 읽기 전용으로 실행
- **16행** 5바이트를 읽어 buf에 저장하고 read() 함수의 리턴값을 저장
  출력 결과를 보면 5임을 알 수 있음
- **22행** read() 함수는 읽어온 데이터의 끝에 자동으로 널을 추가하지 않기 때문에 buf에 널 문자('₩0')를 추가
  buf를 문자열로 출력하려면 널 문자를 추가해야 함
- **23행** linux.txt 파일에서 읽어온 바이트 수와 buf의 내용을 출력
  실행 결과를 보면 linux.txt 파일의 처음 5바이트를 읽어왔음을 알 수 있음)

- 처음 파일의 오프셋은 'L'을 나타내지만 (오프셋 0)
  read() 함수를 실행한 다음에는 5바이트 이동해 S 앞의 공백 문자(' ')를 가리킴 (오프셋 5)
  따라서 계속 읽어오면 ' '부터 읽음

02. 저수준 파일 입출력

■ 파일 쓰기 : write(2)

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

ssize_t write(int fd, const void *buf, size_t count);
```

- fd : 파일 기술자
- buf : 파일에 기록할 데이터를 저장한 메모리 영역
- count : buf의 크기(기록할 데이터의 크기)

- write() 함수의 특징
  - read() 함수와 인자의 구조는 같지만 의미가 다름
  - 파일 기술자는 쓰기를 수행할 파일을 가리키고, buf는 파일에 기록할 데이터를 저장하고 있는 메모리 영역을 가리킴
  - buf가 가리키는 메모리 영역에서 count로 지정한 크기만큼 읽어 파일에 쓰기를 수행
  - write() 함수도 실제로 쓰기를 수행한 바이트 수를 리턴하며, 오류가 발생하면 –1을 리턴

02. 저수준 파일 입출력

■ [예제 4-5] 파일 쓰기(test7.c)

```
01 #include <fcntl.h>
02 #include <unistd.h>
03 #include <stdlib.h>
04 #include <stdio.h>
05
06 int main() {
07     int rfd, wfd, n;
08     char buf[10];
09
10     rfd = open("linux.txt", O_RDONLY);
11     if(rfd == -1) {
12         perror("Open linux.txt");
13         exit(1);
14     }
15
16     wfd = open("linux.bak", O_CREAT | O_WRONLY | O_TRUNC, 0644);
17     if (wfd == -1) {
18         perror("Open linux.bak");
19         exit(1);
20     }
21
22     while ((n = read(rfd, buf, 6)) > 0) {
23         if (write(wfd, buf, n) != n) perror("Write");
24     }
25
26     if (n == -1) perror("Read");
27
28     close(rfd);
29     close(wfd);
30 }
```

실행

```
$ ls linux.bak
ls: 'linux.bak'에 접근할 수 없습니다: 그런 파일이나 디렉터리가 없습니다
$ ch4_5.out
$ cat linux.bak
Linux System Programming
```

- **10행** linux.txt 파일을 읽기 전용으로 열고, 파일 기술자를 rfd에 저장
- **16행** linux.bak 파일을 쓰기 전용으로 생성하고, 기존 파일이 있으면 내용을 비움
- **22~24행** while 문을 사용해 linux.txt 파일을 6바이트씩 읽고 출력
  22행에서 read() 함수가 리턴한 값이 buf에 저장된 데이터의 크기이므로
  이를 23행에서 write() 함수의 세 번째 인자로 사용할 수 있다.
  write() 함수의 리턴값이 출력할 데이터의 크기인 n과 다르면 쓰기 동작에
  문제가 있다는 의미이므로 오류 메시지를 출력
- **실행 결과** linux.bak 파일이 생성되고 데이터가 저장
  즉 linux.txt 파일이 linux.bak 파일로 복사

02. 저수준 파일 입출력

■ 파일 오프셋 위치 지정 : lseek(2)

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

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

- fd : 파일 기술자
- offset : 이동할 오프셋 위치
- whence : 오프셋의 기준 위치

- lseek() 함수의 특징
  - lseek() 함수는 파일 기술자가 가리키는 파일에서 offset으로 지정한 크기만큼 오프셋을 이동
  - 이때 offset의 값은 whence의 값을 기준으로 해석

표 4-4 오프셋의 기준 위치를 나타내는 whence 값

값	설명
SEEK_SET	파일의 시작을 기준으로 계산
SEEK_CUR	현재 위치를 기준으로 계산
SEEK_END	파일의 끝을 기준으로 계산

02. 저수준 파일 입출력

■ 파일 오프셋 위치 지정 : lseek(2)

- lseek() 함수의 특징
  - 파일의 시작에서 다섯 번째 위치로 이동

```
lseek(fd, 5, SEEK_SET);
```

  - 파일의 끝에서 0번째이므로 파일 끝으로 이동

```
lseek(fd, 0, SEEK_END);
```

  - 현재 위치를 기준으로 0만큼 이동한 값을 구해 파일 오프셋 위치를 확인

```
cur_offset = lseek(fd, 0, SEEK_CUR);
```

02. 저수준 파일 입출력

■ [예제 4-6] 파일 오프셋 사용하기(test8.c)

```
01 #include <sys/types.h>
02 #include <fcntl.h>
03 #include <unistd.h>
04 #include <stdlib.h>
05 #include <stdio.h>
06
07 int main() {
08     int fd, n;
09     off_t start, cur;
10     char buf[256];
11
12     fd = open("linux.txt", O_RDONLY);
13     if (fd == -1) {
14         perror("Open linux.txt");
15         exit(1);
16     }
17
18     start = lseek(fd, 0, SEEK_CUR);
19     n = read(fd, buf, 255);
20     buf[n] = '₩0';
21     printf("Offset start=%d, n=%d, Read Str=%s", (int)start, n, buf);
22     cur = lseek(fd, 0, SEEK_CUR);
23     printf("Offset cur=%d₩n", (int)cur);
24
25     start = lseek(fd, 6, SEEK_SET);
26     n = read(fd, buf, 255);
27     buf[n] = '₩0';
28     printf("Offset start=%d, Read Str=%s", (int)start, buf);
29
30     close(fd);
31 }
```

실행

```
$ ch2_6.out
Offset start=0, n=25, Read Str=Linux System Programming
Offset cur=25
Offset start=6, Read Str=System Programming
```

- **12행** linux.txt 파일을 읽기 전용으로 실행
- **18행** 파일의 현재 오프셋 위치를 파악해보면 0임을 알 수 있음
- **19~21행** 파일을 읽고 마지막에 널 값을 넣어 출력
  n의 값이 25로 출력되었는데, 이는 읽은 글자 수 + 널 문자 개수'
- **22행** 파일에서 데이터를 읽은 후 현재 위치를 확인
  (이 예제에서는 오프셋이 start + n 위치로 이동)
- **25~28행** 파일의 시작을 기준으로 오프셋이 6인 위치로 이동한 후 데이터를 읽고 널 값을 채워 출력
- **실행 결과** 오프셋의 위치에 따라 읽어온 데이터가 다름을 알 수 있음

파일입출력

- O_APPEND
  - open이 성공하면 파일의 마지막 바이트 바로 뒤에 위치
  - 그 이후의 write는 전부 파일의 끝에 자료를 추가하게 됨
  - 파일의 끝에 자료를 추가하는 방법

    - lseek 사용

      lseek(fd, 0, SEEK_END)
      write(fd, buf, BUFSIZE)

    - O_APPEND

      open("filename", O_WRONLY|O_APPEND)
      write(fd, buf, BUFSIZE)

표준 입력, 표준 출력 및 표준 오류

- 표준 입력(0), 표준 출력(1), 표준 오류(2)
  - redirection < >
    - prog_name < infile
      - 파일기술자 0로부터 읽어 들일 때, infile로 부터 자료를 읽음
    - prog_name > outputfile
      - 출력을 outputfile로 변경
    - prog_name < infile > outputfile
  - pipe
    - prog1 | prog2

표준 입출력 (test9.c)

```
01  #include <fcntl.h>
02  #include <unistd.h>
03  #include <stdlib.h>
04  #include <stdio.h>
05  #define SIZE 512

06  int main(void) {
07      ssize_t nread;
08      char buf[SIZE]
09
10      while ((nread = read(0, buf, SIZE)) > 0
11              write(1, buf, nread);
12
13      return 0;
14  }
```

■ 실습(2)