Operating Systems LAB 6

Wonpyo Kim
skykwp@gmail.com

Outline

- Substance
  - File IO
  - File Information & Directory Management

File IO

- 프로그래밍에서는 vi와 같은 에디터를 사용하여 소스 파일을 생성, 수정, 삭제하고 컴파일 한 후 실행 파일을 만드는 등의 파일에 대한 동작이 필수적이다.
- 따라서, 모두 파일 단위로 작업이 이루어지는데 이러한 동작은 시스템에서 제공하는 파일 입출력 함수에 의해 수행된다.
- 실제 프로그램에서는 각 파일에 특정한 번호를 부여한다. 이 번호를 특별히 파일 식별자(file descriptor)라고 칭한다.

File IO

- 예제

```c
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

main()
{
    int fd1, fd2;

    fd1 = open("kim", O_RDONLY);
    fd2 = open("jung", O_WRONLY);

    printf("kim's file descriptor: %d\njung's file descriptor: %d\n", fd1, fd2);

    close(fd1);
    close(fd2);
}
```

```
s_guest@A1409-OSLAB-01:~/lab6$ ./a.out
kim's file descriptor: 3
jung's file descriptor: 4
```

- 실행 결과로 파일 식별자가 각각 3, 4로 부여되었다. 그 이유는 프로그램이 하나 실행될 때, 기본적으로 3개의 파일 식별자가 자동으로 부여되기 때문이다.
- 0: 키보드 표준입력, 1: 표준 출력, 2: 모니터에 나타나는 표준 오류

File IO

- 예제

```c
#include <fcntl.h>
#include <unistd.h>

#define BUFSIZE 1024

main()
{
    int fd1, fd2;
    ssize_t n;
    char buf[BUFSIZE];

    fd1 = open("kim", O_RDONLY);
    fd2 = open("jung", O_WRONLY);

    n = read(fd1, buf, BUFSIZE);

    write(fd2, buf, n);

    close(fd1);
    close(fd2);
}
```

```
s_guest@A1409-OSLAB-01:~/lab6$ ./a.out
s_guest@A1409-OSLAB-01:~/lab6$ cat kim
Operating system concepts
s_guest@A1409-OSLAB-01:~/lab6$ cat jung
Operating system concepts
```

- 실행에 앞서 "kim" 이라는 파일에는 "Operating system concepts" 라는 내용의 텍스트가 포함되어 있다.
- 프로그램은 버퍼에 파일의 내용을 읽은 후 fd2 라는 파일 식별자를 통해 버퍼의 내용을 파일로 쓰게된다.

File IO - open( )

- 특정한 파일을 프로그램 내에서 열기 위해서는 open 함수를 사용한다.

```
open 함수
기능
        파일을 연다.
기본형
        int open(const char *pathname, int flags[, mode_t mode]);
        pathname: 열고자하는 파일 이름
        flags: 파일에 대한 제어방법
        mode: 생성할 파일에 대한 접근권한
반환값
        성공: 양의 정수(파일 식별자)
        실패: -1
헤더파일
        <sys/types.h>
        <sys/stat.h>
        <fcntl.h>
```

- open 함수에서 반환되는 파일 식별자는 read, write, close 함수 등에서 활용할 수 있다.
- 첫 번째 매개 변수인 pathname 은 절대경로와 상대경로를 사용할 수 있다.

File IO - open( )

- open 함수의 두 번째 매개 변수인 flags 는 열고자 하는 파일에 대한 제어방법을 지정한다.
- 지정되는 값은 <fcntl.h> 헤더 파일에 정의된 상수의 OR 연산을 통해 형성된다. 다음 중 반드시 한 개를 선택해야 한다.

flags	의미
O_RDONLY	읽기 전용으로 연다
O_WRONLY	쓰기 전용으로 연다
O_RDWR	읽기와 쓰기용으로 연다

File IO - open( )

- 그런데, 추가적으로 다음과 같은 선택적인 옵션사항을 지정할 수 있다.

flags	의미
O_CREAT	해당 파일이 없으면 파일을 생성한다
O_EXCL	해당 파일이 존재하면 오류를 발생시키고 파일을 열지 않는다
O_TRUNC	해당 파일이 존재하면 파일의 길이를 0으로 만든다. 즉, 파일의 내용을 모두 지운다
O_APPEND	쓰기 동작 시 파일의 끝 부분에 추가한다
O_NOCTTY	만약 첫 번째 인수인 pathname이 터미널이라면 이 터미널을 프로그램의 제어 터미널로 할당하지 않는다
O_NONBLOCK	FIFO, 블록 특수 파일, 문자 특수 파일 등에서 입출력을 할 경우 읽거나 쓸 내용이 없더라도 장치가 준비 또는 사용 가능하게 되는 것을 기다리지 않고 바로 -1을 반환한다
O_SYNC	쓰기 동작 시 물리적인 쓰기 동작이 완료될 때까지 기다린다

File IO - open( )

- 예제

```c
#include <fcntl.h>

main()
{
    int fd;

    if ((fd = open("kim", O_RDONLY | O_CREAT, 0644)) == -1) {
        perror("open failed");
        exit(1);
    }
    close(fd);

    if ((fd = open("kim", O_WRONLY | O_CREAT | O_TRUNC, 0644)) == -1) {
        perror("open failed");
        exit(1);
    }
    close(fd);

    if ((fd = open("kim", O_RDONLY | O_CREAT | O_EXCL, 0644)) == -1) {
        perror("open failed");
        exit(1);
    }
    close(fd);

    exit(0);
}
```

- O_CREAT 옵션을 추가하면 파일을 생성할 수 있다. 첫 번째 if 문이 실행되면 파일이 생성된다.
- 두 번째 if 문에서 O_TRUNC 옵션을 사용하면, 파일이 이미 존재하는 경우, 파일의 길이를 0으로 하고 존재하지 않으면 새로 생성한다.
- 세 번째 if 문에서 O_EXCL 옵션을 사용하면, O_CREAT 옵션을 무시하고 읽기 전용으로 파일을 연다. 즉, 파일이 이미 존재할 때 open 함수의 실행을 실패(중지)시킬 때 사용할 수 있다.

File IO - close( )

- 열린 파일을 닫는 함수는 close 함수이다.

```
close 함수
기능
        파일을 닫는다.
기본형
        int close(int fd);
        fd: 닫고자 하는 파일의 파일 식별자
반환값
        성공: 0
        실패: -1
헤더파일
        <fcntl.h>
```

- 매개 변수인 fd는 닫고자 하는 파일 식별자로 open 또는 creat 함수에 의해 생성된다.

File IO - read( )

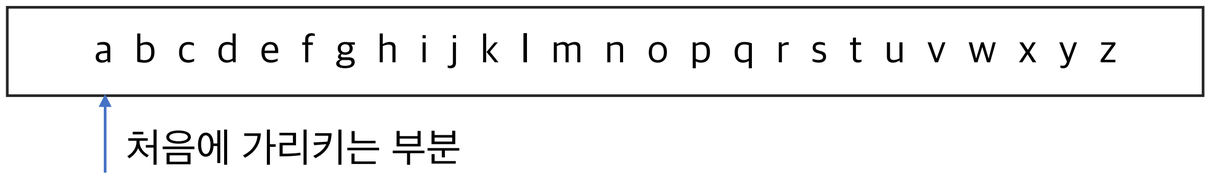
- 파일의 가장 기본적인 동작은 읽기와 쓰기이다. 파일은 특정 프로그램에 의해서 읽기, 쓰기 연산이 수행된다.
- 파일의 데이터를 읽는 함수는 read 함수이다.

read 함수
기능
	파일로부터 데이터를 읽는다.
기본형
	ssize_t read(int fd, void *buf, size_t nbytes);
	fd: 읽을 파일의 파일 식별자
	buf: 읽어온 데이터를 저장하는 변수
	nbytes: 읽어올 데이터의 바이트 수
반환값
	성공: 읽은 바이트 수
	실패: -1
헤더파일
	<unistd.h>

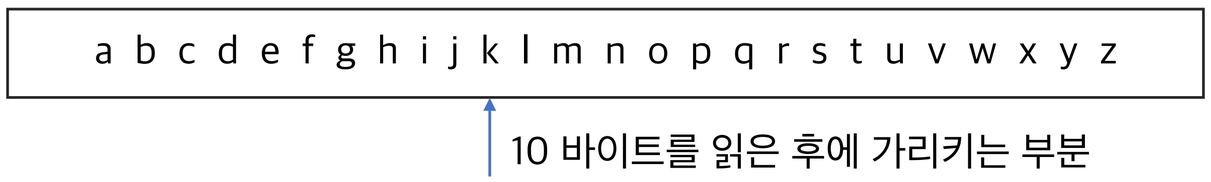
File IO - read( )

- 예제

alphabet 파일



alphabet 파일



- 예제로 수행할 코드는 위와 같은 동작을 한다. alphabet 파일을 하나 생성하고 알파벳을 채워준다. (공백없이)

File IO - read( )

- 예제

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

#define MAX 10

main()
{
    int fd;
    char buf1[MAX], buf2[MAX];

    if ((fd = open("alphabet", O_RDONLY)) == -1) {
        perror("open failed");
        exit(1);
    }

    read(fd, buf1, MAX);
    read(fd, buf2, MAX);

    printf("buf1[0]: %c\nbuf2[0]: %c\n", buf1[0], buf2[0]);

    close(fd);
    exit(0);
}
```

- 실행 결과는 'a' 와 'k' 가 출력된다.

- 코드는 읽기전용으로 파일을 연 후 MAX 값인 10(바이트)만큼 읽어서 buf1, buf2 에 저장한다.

- 이와 같이 파일의 내용을 가리키는 워커를 "읽기/쓰기 포인터"라 하며, 파일을 통해 읽거나 쓰여질 위치를 명시한다.

- 앞서 read 함수의 반환값은 읽어들인 바이트 수 라고 하였다. 따라서, 반환 값을 잘 활용하면 파일의 크기를 구하는 프로그램도 작성할 수 있다.

File IO - write( )

- 파일에 데이터를 쓰는 함수는 write 로, fd 식별자의 "읽기/쓰기 포인터" 를 기준으로 buf 를 nbytes 만큼 파일에 쓴다.

write 함수
기능
	파일에 데이터를 쓴다.
기본형
	ssize_t write(int fd, const void *buf, size_t nbytes);
	fd: 쓸 파일의 파일 식별자
	buf: 데이터를 저장하고 있는 변수
	nbytes: 쓸 데이터의 바이트 수
반환값
	성공: 파일에 쓴 데이터의 바이트 수
	실패: -1
헤더파일
	<unistd.h>

File IO - write( )

- 예제

```c
#include <fcntl.h>
#include <unistd.h>

main()
{
    int fd;
    char buf[128] = "Operating system concepts\n";

    if ((fd = open("newfile", O_WRONLY | O_CREAT, 0644)) == -1) {
        perror("open failed");
        exit(1);
    }

    if (write(fd, buf, 128) == -1) {
        perror("write failed");
        exit(1);
    }

    close(fd);
    exit(0);
}
```

- 본 예제는 단순히 버퍼에 저장된 변수 값을 파일에 쓴다.
- O_WRONLY | O_CREAT 조건으로 파일을 쓰기 전용으로 생성한 후 반환되는 파일 식별자로 write 함수를 처리한다.
- 수행하면 "newfile" 이라는 파일이 생성되고, 파일의 내용은 buf의 내용으로 저장된다.
- 여기서, O_CREAT 부분을 O_APPEND 로 바꾸면 이미 존재하는 파일의 뒷 부분에 쓰기가 가능하다.

File IO - creat( )

- open 함수가 아닌 creat 함수에 의해서도 파일을 생성할 수 있다.

creat 함수
기능
　　파일에 생성한다.
기본형
　　int creat(const char *pathname, mode_t mode);
　　pathname: 생성하고자 하는 파일 이름
　　mode: 생성할 파일에 대한 접근권한
반환값
　　성공: 양의 정수 (파일 식별자)
　　실패: -1
헤더파일
　　<sys/types.h>
　　<sys/stat.h>
　　<unistd.h>

File IO - creat( )

- 예제

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define MAX 1024

main(int argc, char *argv[])
{
    int fd1, fd2, count;
    char buf[MAX];

    if (argc != 3) {
        perror("argc is not 3");    exit(1);
    }

    if ((fd1 = open(argv[1], O_RDONLY)) == -1) {
        perror("open failed");
        exit(1);
    }

    if ((fd2 = creat(argv[2], 0644)) == -1) {
        perror("creat failed");
        exit(1);
    }

    while ((count = read(fd1, buf, MAX)) > 0) {
        if (write(fd2, buf, count) != count) {
            perror("write failed");
            exit(1);
        }
    }

    close(fd1);
    close(fd2);
    exit(0);
}
```

- 본 예제는 argv[1] 의 파일 이름을 갖는 파일을 읽은 후, argv[2] 의 파일명으로 새로운 파일을 생성한 후, argv[1] 의 파일 식별자로 파일의 내용을 읽고, argv[2] 에 읽은 내용을 쓰는 복사의 코드이다.

File IO - lseek( )

- 앞서 "읽기/쓰기 포인터"로 파일의 위치를 나타낼 수 있다고 하였다. 여기서, lseek 함수를 이용하면 이러한 "읽기/쓰기 포인터"의 위치를 프로그래머의 임의로 변경할 수가 있다.

- 함수의 내용은 fildes 파일의 읽기/쓰기 포인터를 whence를 기준으로 offset 만큼 이동하라. 라는 뜻이다.

lseek 함수
기능
	읽기/쓰기 포인터의 위치를 임의로 변경한다.
기본형
	off_t lseek(int fildes, off_t offset, int whence);
	fildes: 읽기/쓰기 포인터의 위치를 임의로 변경할 파일의 파일 식별자
	offset: 이동할 바이트 수
	whence: 시작 지점
반환값
	성공: 변경된 읽기/쓰기 포인터
	실패: -1
헤더파일
	<sys/types.h>
	<unistd.h>

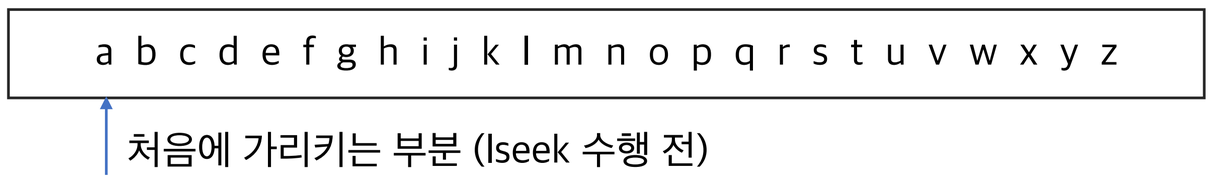
File IO - lseek( )

- offset 매개 변수는 포인터가 이동할 바이트 수로 양수면 뒤로, 음수면 앞으로 이동한다. whence 매개 변수는 기준점을 어디로 둘 지에 관한 것으로 아래와 같은 매크로 값을 갖는다.

매크로	의미
SEEK_SET	파일의 시작
SEEK_CUR	현재 읽기/쓰기 포인터가 가리키는 부분
SEEK_END	파일의 끝

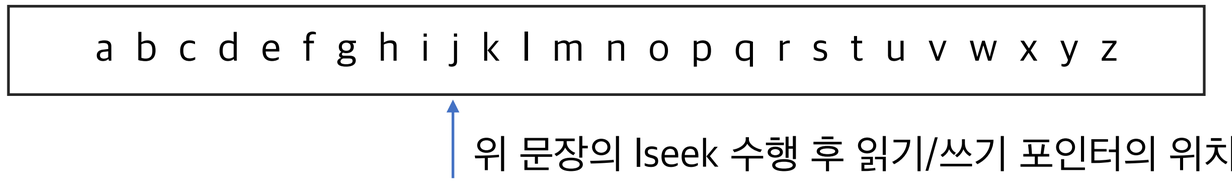
File IO - lseek( )

alphabet 파일



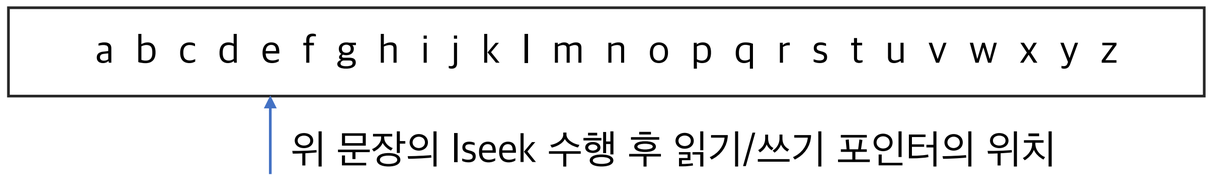
pos = lseek(fd, 9, SEEK_CUR);

alphabet 파일



pos = lseek(fd, -5, SEEK_CUR);

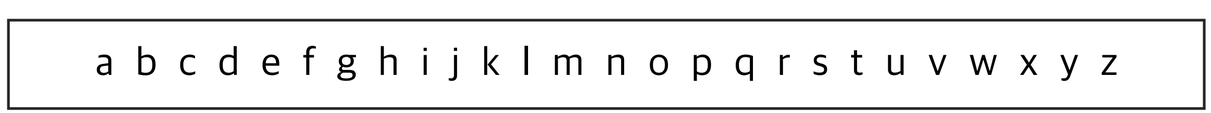
alphabet 파일



File IO - lseek( )

pos = lseek(fd, -5, SEEK_END);

alphabet 파일

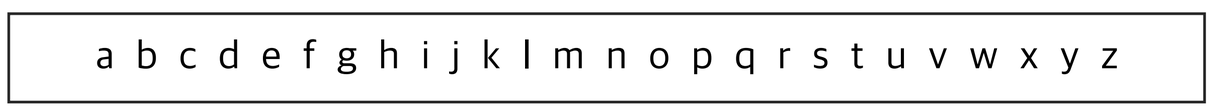


위 문장의 lseek 수행 후 읽기/쓰기 포인터의 위치

여기서, v를 가리키지 않는 이유는 파일의 맨 끝은 항상 널문자로 끝나기 때문이다.

pos = lseek(fd, 2, SEEK_SET);

alphabet 파일



위 문장의 lseek 수행 후 읽기/쓰기 포인터의 위치

File IO - lseek( )

- 예제

```
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

#define MAX 10

main()
{
    int fd, count;
    char buf[MAX];

    if ((fd = open("alphabet", O_RDONLY)) == -1) {
        perror("open failed");
        exit(1);
    }

    lseek(fd, 5, SEEK_SET);
    count = read(fd, buf, MAX);

    write(STDOUT_FILENO, buf, count);

    close(fd);
    exit(0);
}
```

- 본 예제는 "alphabet" 파일을 읽은 후 "읽기/쓰기 포인터"를 5칸 이동시킨 후 STDOUT_FILENO(콘솔출력) 으로 MAX 만큼 출력하는 예제이다.
- 실행 결과로 "abcde" 는 나타나지 않고, 'e' 부터 10개가 출력된다.

Outline

- Substance
  - **File IO**
  - **File Information & Directory Management**

File Information & Directory Control

- 리눅스에서는 파일 단위로 작업을 수행한다.
- 대부분의 작업에서 파일과 디렉토리를 처리하는 ls, rm, mkdir, rmdir, cd, link 등의 명령어가 자주 사용된다.
- 이런 명령어들은 파일과 디렉토리 관련 함수를 기반으로 구현된다.

File Information

- 프로그래머는 파일의 크기가 얼마인지, 파일이 수정된 시각은 언제인지, 파일을 만든 사람은 누구인지 등 파일에 대한 여러가지 정보를 알고싶을 때가 있다.

- 이 때, stat, fstat, lstat 함수를 사용하면 파일의 다양한 정보를 알아낼 수 있다.

stat, fstat, lstat 함수
기능
	파일 정보를 얻는다.
기본형
	int stat(const char *file_name, struct stat *buf);
	int fstat(int filedes, struct stat *buf);
	int lstat(const char *file_name, struct stat *buf);
	file_name: 정보를 얻고자하는 파일의 이름
	filedes: 정보를 얻고자하는 파일의 파일 식별자
	buf: 파일 정보를 가리키는 포인터 변수
반환값
	성공: 0
	실패: -1
헤더파일
	<sys/types.h>
	<sys/stat.h>
	<unistd.h>

File Information

- stat 과 fstat 의 차이점은 stat은 파일 이름을 이용하고 fstat은 파일 식별자를 이용한다. 즉, fstat은 열린 파일에 대해서만 동작이 가능하다.

- stat 과 lstat 은 매개 변수의 형식이 같으나, lstat은 심볼릭 링크된 파일의 링크 정보 자체를 전달한다. stat은 심볼릭 링크가 가리키는 파일에 대한 정보를 전달한다.

File Information

- stat 구조체는 다음과 같이 정의되어 있다. 모두를 이해하기 보다는 변수명에 주목한다.

```
struct stat
  {
    __dev_t st_dev;             /* Device.  */
#ifndef __x86_64__
    unsigned short int __pad1;
#endif
#if defined __x86_64__ || !defined __USE_FILE_OFFSET64
    __ino_t st_ino;             /* File serial number.  */
#else
    __ino_t __st_ino;                   /* 32bit file serial number.    */
#endif
#ifndef __x86_64__
    __mode_t st_mode;                   /* File mode.  */
    __nlink_t st_nlink;                 /* Link count.  */
#else
    __nlink_t st_nlink;         /* Link count.  */
    __mode_t st_mode;           /* File mode.  */
#endif
    __uid_t st_uid;             /* User ID of the file's owner. */
    __gid_t st_gid;             /* Group ID of the file's group.*/
#ifdef __x86_64__
    int __pad0;
#endif
    __dev_t st_rdev;            /* Device number, if device.  */
#ifndef __x86_64__
    unsigned short int __pad2;
#endif
#if defined __x86_64__ || !defined __USE_FILE_OFFSET64
    __off_t st_size;                    /* Size of file, in bytes.  */
#else
    __off64_t st_size;                  /* Size of file, in bytes.  */
#endif
    __blksize_t st_blksize;     /* Optimal block size for I/O.  */
#if defined __x86_64__  || !defined __USE_FILE_OFFSET64
    __blkcnt_t st_blocks;               /* Number 512-byte blocks allocated. */
#else
    __blkcnt64_t st_blocks;             /* Number 512-byte blocks allocated. */
#endif
#ifdef __USE_XOPEN2K8
    /* Nanosecond resolution timestamps are stored in a format
       equivalent to 'struct timespec'.  This is the type used
       whenever possible but the Unix namespace rules do not allow the
       identifier 'timespec' to appear in the <sys/stat.h> header.
       Therefore we have to handle the use of this header in strictly
       standard-compliant sources special.  */
    struct timespec st_atim;            /* Time of last access.  */
    struct timespec st_mtim;            /* Time of last modification.  */
    struct timespec st_ctim;            /* Time of last status change.  */
# define st_atime st_atim.tv_sec        /* Backward compatibility.  */
# define st_mtime st_mtim.tv_sec
# define st_ctime st_ctim.tv_sec
#else
    __time_t st_atime;                  /* Time of last access.  */
    __syscall_ulong_t st_atimensec;     /* Nscecs of last access.  */
    __time_t st_mtime;                  /* Time of last modification.  */
    __syscall_ulong_t st_mtimensec;     /* Nsecs of last modification.  */
    __time_t st_ctime;                  /* Time of last status change.  */
    __syscall_ulong_t st_ctimensec;     /* Nsecs of last status change.  */
#endif
#ifdef __x86_64__
    __syscall_slong_t __glibc_reserved[3];
#else
# ifndef __USE_FILE_OFFSET64
    unsigned long int __glibc_reserved4;
    unsigned long int __glibc_reserved5;
# else
    __ino64_t st_ino;                   /* File serial number.  */
# endif
#endif
  };
```

File Information

- 예제

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>

main(int argc, char *argv[])
{
    struct stat st;

    if (stat("alphabet", &st) == -1) {
        perror("stat failed");
        exit(1);
    }

    printf("%ld byte, user-id %d, group-id %d, modify time %s",
        st.st_size, st.st_uid, st.st_gid, ctime(&st.st_mtime));
    exit(0);
}
```

- stat 함수와 구조체는 복잡한 선언에 비해 사용은 간단한 편이다.

- 앞서 만든 "alphabet" 파일을 stat 함수를 통해 파일의 정보를 가져온 후, 저장되는 구조체 변수 st 의 멤버를 참조만 하고있다.

- 이를 실행하면, alphabet 파일에 대한 정보가 출력된다.

File Information

- 특별히, stat 구조체의 st_mode 변수는 파일의 형식과 접근 권한 등의 정보를 나타낸다.

매크로	16진수	의미
S_IFMT	0xF000	파일 유형
S_IFIFO	0x1000	FIFO 파일
S_IFCHR	0x2000	문자 특수 파일
S_IFDIR	0x4000	디렉토리
S_IFBLK	0x6000	블록 특수 파일
S_IFREG	0x8000	정규 파일
S_IFLNK	0xA000	심볼릭 링크
S_IFSOCK	0xC000	소켓

File Information

- 예제

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

main(int argc, char *argv[])
{
    int i;
    struct stat st;

    if (stat("alphabet", &st) == -1) {
        perror("stat failed");
        exit(1);
    }
    printf("%s's mode: %x\n", "alphabet", st.st_mode);

    exit(0);
}
```

```
s_guest@A1409-OSLAB-01:~/lab6$ ./a.out
alphabet's mode: 81b4
```

- 본 예제에서 alphabet 파일의 st_mode 값은 81b4 (16진수) 를 나타내고 있다.

- 앞선 표에서 S_IFMT 는 파일유형의 의미로 0xF000을 나타낸다고 하였다. 파일의 종류를 알기 위에서 출력 결과인 81b4 와 S_IFMT 변수를 AND(&) 연산을 하면 어떤 파일인지 알 수 있다.

	st_mode	1000 0001 1011 0100
&	S_IFMT	1111 0000 0000 0000
		1000 0000 0000 0000

- 위와 같이 & 연산을 하면 1000 0000 0000 0000 (0x8000) 이 나온다. 이 값은 표에서 S_IFREG 와 같으므로 정규 파일을 의미한다.

Directory Management

- 파일을 효율적이고 구조적으로 관리하기 위해서는 디렉토리를 이용하는 것이 바람직하다.
- 디렉토리를 생성하는 mkdir 함수의 선언은 다음과 같다.

mkdir 함수
기능
	디렉토리를 만든다.
기본형
	int mkdir(const char *pathname, mode_t mode);
	pathname: 만들고자하는 디렉토리 이름
	mode: 만들고자하는 디렉토리의 접근권한
반환값
	성공: 0
	실패: -1
헤더파일
	<sys/stat.h>
	<sys/types.h>
	<fcntl.h>
	<unistd.h>

Directory Management

- 디렉토리를 삭제하는 rmdir 함수의 선언은 다음과 같다.

```
rmdir 함수
기능
        디렉토리를 삭제한다.
기본형
        int mkdir(const char *pathname);
        pathname: 만들고자하는 디렉토리 이름
반환값
        성공: 0
        실패: -1
헤더파일
        <unistd.h>
```

Directory Management

- 예제

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

main()
{
    if (mkdir("my_directory", 0755) == -1) {
        perror("mkdir failed");
        exit(1);
    }

    if (rmdir("my_directory") == -1) {
        perror("rmdir failed");
        exit(2);
    }

    exit(0);
}
```

- 본 예제는 "my_directory" 라는 이름의 디렉토리를 0755 의 권한을 부여하여 생성했다가 바로 삭제한다.
- 따라서, 실행결과는 아무것도 나오지 않는다.
- mkdir, rmdir의 사용법을 숙지한다.

Directory Management

- 예제

```c
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

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

    if (mkdir("english", 0755) == -1) {
        perror("mkdir failed");
        exit(1);
    }

    if (chdir("english") == -1) {
        perror("chdir failed");
        exit(1);
    }

    if ((fd = open("alphabet", O_WRONLY | O_CREAT, 0644)) == -1) {
        perror("open failed");
        exit(1);
    }

    write(fd, "abcd", strlen("abcd"));

    close(fd);
    exit(0);
}
```

- 본 예제는 "english" 라는 디렉토리를 0755 의 권한을 부여하여 생성한 후, english 디렉토리로 이동하여 alphabet 이라는 파일을 만들고 abcd 를 파일에 쓴 후 종료한다.

- 따라서, 실행한 후 english 라는 디렉토리가 생기고 그 안에 "abcd" 내용을 갖는 alphabet 이라는 파일이 생성된다.

Directory Management

- 현재 작업 중인 디렉토리를 알고자 할 때는 getcwd 함수를 사용한다. 함수 호출이 성공하면 buf 에 현재 작업 중인 디렉토리 이름이 저장된다.

getcwd 함수
기능
	현재 작업 디렉토리 이름을 얻어온다.
기본형
	char *getcwd(char *buf, size_t size);
	buf: 현재 작업 디렉토리 이름을 저장하는 공간
	size: 디렉토리 이름의 예상되는 길이
반환값
	성공: 현재 작업 디렉토리 이름
	실패: NULL
헤더파일
	<unistd.h>

Directory Management

- 예제

```c
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

#define MAX 100

main(int argc, char *argv[])
{
    char name[MAX];

    getcwd(name, MAX);
    printf("directory name: %s\n", name);

    mkdir("new", 0755);
    chdir("new");

    getcwd(name, MAX);
    printf("directory name: %s\n", name);

    chdir("..");
    getcwd(name, MAX);
    printf("directory name: %s\n", name);

    rmdir("new");

    exit(0);
}
```

Directory Management

- 파일을 여는 함수에는 open 함수가 있다. 디렉토리 역시 opendir 이라는 함수로 디렉토리를 열어줄 수 있다.

```
opendir 함수
기능
        디렉토리를 연다.
기본형
        DIR *opendir(const char *name);
        name: 열고자하는 디렉토리 이름
반환값
        성공: DIR 구조체 포인터
        실패: NULL
헤더파일
        <sys/types.h>
        <dirent.h>
```

Directory Management

- 파일을 닫는 함수에는 close 함수가 있다. 디렉토리 역시 closedir 이라는 함수로 디렉토리를 닫을 수 있다.

```
closedir 함수
기능
        디렉토리를 닫는다.
기본형
        int closedir(DIR *dir);
        dir: 닫고자하는 디렉토리 정보에 대한 포인터
반환값
        성공: 0
        실패: -1
헤더파일
        <sys/types.h>
        <dirent.h>
```

Directory Management

- opendir, closedir 함수에서 사용하는 DIR 구조체는 다음과 같은 구조를 갖고있다.

```
typedef struct {
    int d_fd;       /* file descriptor */
    int d_loc;      /* offset in block */
    int d_size;     /* size of data */
    char *d_buf;    /* directory block */
} DIR;
```

- DIR 구조체 역시 각 변수를 참조를 하면된다.

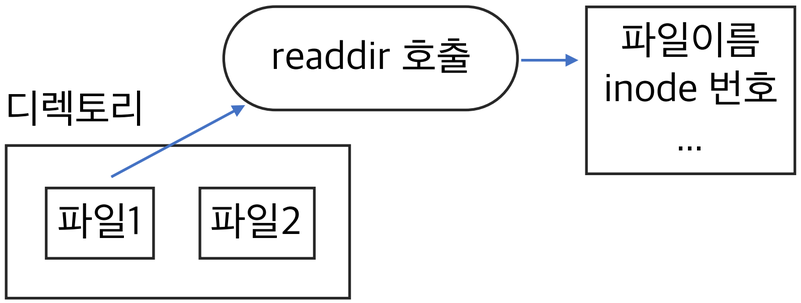
Directory Management

- 디렉토리에 존재하는 항목(파일) 각각에 대한 정보를 읽어오기 위해서는 readdir 함수를 이용한다.
- dir 은 읽고자하는 디렉토리 정보를 가리키는 포인터로 opendir 호출에 의해 반환된 값이다.

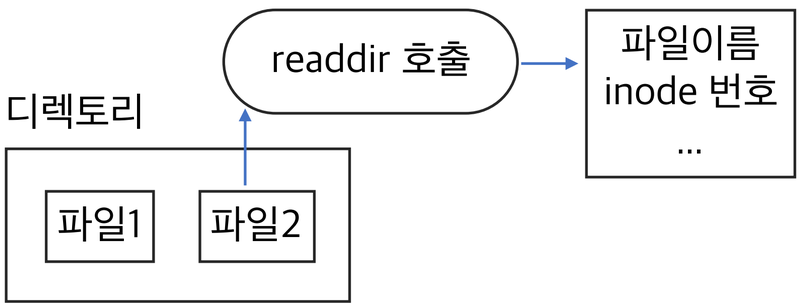
readdir 함수
기능
	디렉토리를 읽는다.
기본형
	struct dirent *readdir(DIR *dir);
	dir: 읽고자하는 디렉토리
반환값
	성공: struct dirent의 포인터
	실패: NULL
헤더파일
	<sys/types.h>
	<dirent.h>

Directory Management

- readdir 을 처음 호출하면 첫 번째 항목의 정보에 대한 포인터가 반환된다.



- 한 번 더 호출하면 다음 항목의 정보에 대한 포인터가 반환된다.



Directory Management

- 예제

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

main()
{
    DIR *dp;
    struct dirent *dirp;

    if ((dp = opendir(".")) == NULL) {
        perror("opendir failed");
        exit(1);
    }

    while (dirp = readdir(dp)) {
        printf("%s ", dirp->d_name);
    }
    printf("\n");

    closedir(dp);
    exit(0);
}
```

- DIR 구조체는 디렉토리의 정보를, dirent 구조체는 한 개의 항목을 나타내는 구조체이다.
- 본 예제는 현재 디렉토리인 "." 에 대한 파일 목록을 보여준다.
- 디렉토리도 파일과 마찬가지로 열었으면 반드시 닫아주어야 한다.

Next Week

- Mid term exam… :)