The file

Suntae Hwang
Kookmin University

1 UNIX file access primitives

- A small set of system calls
- UNIX I/O를 위한 기초적 요소를 형성
- 다른 파일 접근방식의 궁극적인 기반

UNIX primitives

이름	의미
open	읽거나 쓰기 위해 파일을 열거나, 또는 빈 파일을 생성한다.
creat	빈 파일을 생성한다.
close	앞서 열려진 파일을 닫는다.
read	파일로부터 정보를 추출한다
write	파일에 정보를 기록한다.
lseek	파일 안의 지정된 바이트로 이동한다.
unlink	파일을 제거한다.
remove	파일을 제거하는 다른 방법
fcntl	한 파일에 연관된 속성을 제어한다.

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

main()
{
int fd;
ssize_t nread;
char buf[1024];

fd = open("data", O_RDONLY);

nread = read(fd, buf, 1024);

close(fd);
}
```

<sys/types.h>에 정의된 type, 일부 기본 type은 <unistd.h>에서도 정의

<fcntl.h>에 정의

File descriptor 값을 return : 시스템에 의해 결정되는 음이 아닌 정수값

문자 또는 바이트의 linear sequence 만을 다룬다.

The open **system call**

- Usage

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

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

The open **system call (cont)**

- Pathname: 파일의 경로이름을 갖고 있는 문자열에 대한 포인터
  - 절대 경로 / 상대 경로
- Flags: 접근 방식 지정
  - `<fcntl.h>`에 정의된 상수
    - O_RDONLY 읽기 전용
    - O_WRONLY 쓰기 전용
    - O_RDWR 읽고 쓰기용
- Mode: security permission과 관계
  - O_CREAT flag와 함께만 사용

```
#include <stdlib.h>                    /* exit 호출을 위한 것임 */
#include <fcntl.h>

char *workfile="junk";                 /* workfile 이름을 정의 */

main()
{
int filedes;

/* <fcntl.h>에 정의된 O_RDWR을 사용하여 open */
/* 파일을 읽기/쓰기로 open한다. */
if((filedes = open(workfile, O_RDWR)) == -1)
{
        printf("Couldn't open %s\n", workfile);
        exit(1);                /* error so exit */
}

/* 이 프로그램의 나머지 부분 */

exit(0);                /* normal exit */
}
```

The open **system call (cont)**

- UNIX convention for ***exit status***
  - 0: successful completion
  - 1(not zero): error occurs
- 프로세스에 동시에 open 될 수 있는 파일의 수에는 제한이 있다
  - POSIX: 최소 20개

Creating a file with open

filedes = open(“tmp/newfile”, O_WRONLY | O_CREAT, 0644);

- /tmp/newfile이 존재하지 않으면 길이가 0인 파일로 새로 생성되어 쓰기 전용으로 open된다.
- Mode는 file access permission을 수록

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

#define PERMS 0644
        /* O_CREAT을 사용하는 open을 위한 permission */

char *filename="newfile";

main()
{
int filedes;

if((filedes = open(filename, O_RDWR|O_CREAT, PERMS)) == -1)
{
        printf("Couldn't create %s\n",filename);
        exit(1);
}

/* 프로그램의 나머지 부분 */
```

Creating a file with open (cont)

- 앞의 예에서 newfile이 이미 존재하면
  - 만일 permission이 허락하면 그 파일은 마치 O_CREAT이 지정되지 않은 것 처럼 open 된다.
    - 예: 쓰기로 open 하려는데 write permission이 있는 경우
  - 이 경우 mode 인수는 아무런 작용을 않는다.

Creating a file with open (cont)

fd=open(“lock”, O_WRONLY | O_CREAT | O_EXCL, 0644);

- O_EXCL (exclusive)
  - 파일 `lock`이 존재하지 않으면, permission 644로 생성하라는 의미
  - 존재하면 실패하고 fd에 –1을 복귀

Creating a file with open **(cont)**

fd=open(“file”, O_WRONLY | O_CREAT | O_TRUNC, 0644);

- O_TRUNC (truncate)
  - O_CREAT과 함께 사용
  - 파일이 존재하고, permission이 허락할 경우 강제로 그 파일을 0 byte로 truncate한다.

The creat **system call**

- Usage

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

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

The creat **system call (cont)**

- open과 대부분 동일
- 그러나 파일이 존재할 경우 file을 항상 truncate한다.

```
filedes = creat("tmp/newfile", 0644)
```

is equivalent to

```
filedes = open("tmp/newfile",
                    O_WRONLY | O_CREAT | O_TRUNC, 0644);
```

The creat **system call (cont)**

- creat은 파일을 항상 write only로 open
  - Creat로 파일을 생성하여 자료를 쓰고 뒤로 돌아와서 읽기를 시도할 수 없다.
  - 이를 위해서는 그 파일을 close하고 다시 open해야 한다.

The close **system call**

- Usage

```
#include <unistd.h>

int close(int filedes);
```

- Close system call
  - 성공하면 0을 반환
  - 오류가 발생하면 -1
- 프로그램 수행이 끝나면 모든 개방된 파일은 자동으로 닫힌다.

The read **system call**

- 파일로부터 임의의 수의 문자나 바이트를 호출 프로그램의 제어하에 있는 버퍼로 복사
- Usage

```
#include <unistd.h>

ssize_t read(int filedes, void *buffer, size_t n);
```

The read system call (cont)

- Read-write pointer (or file pointer)
  - 파일 안에서의 프로세스의 위치를 관리
  - 특정 file descriptor를 통해 읽혀질(쓰여질) 파일의 다음 바이트 위치를 기록
  - 일종의 bookmark
- read로부터의 복귀값이 0인가를 조사하는 것이 프로그램 안에서 파일의 끝을 조사하는 보통의 방법

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

#define BUFSIZE 512

main()
{
        char buffer[BUFSIZE];
        int filedes;
        ssize_t nread;
        long total=0;

        if ((filedes=open("anotherfile",O_RDONLY))==-1)
        {
                printf("error in opening anotherfile\n");
                exit(1);
        }

        while((nread=read(filedes,buffer, BUFSIZE))>0)
                total+=nread;

        printf("total chars in anotherfile: %1d\n",total);
        exit(0);
}
```

The read system call (cont)

- BUFSIZ
  - 진짜 디스크 blocking factor 이용

```
#include <stdio.h>
nread = read(filedes, buffer, BUFSIZ);
```

The write **system call**

- 문자 배열로 선언된 프로그램 버퍼로부터 외부 파일로 자료를 복사
- Usage

```
#include <unistd,.h>
ssize_t write(int filedes, const void *buffer, size_t n);
```

The write **system call (cont)**

- Write 호출이 있을 때 마다 파일의 끝에 자료가 더해지고, read-write pointer는 마지막으로 쓰여진 바이트의 바로 뒤로 전진
- 기존의 파일을 open 하고 write하는 경우 기존의 내용을 replace하고 원래의 파일의 끝에 도달하면 append하기 시작
- 이를 피하려면

```
filedes = open(filename, O_WRONLY | O_APPEND);
```

Example: copyfile

```
/* copyfile – name1을 name2로 복사 */
#include<unistd.h>
#include<fcntl.h>

#define BUFSIZE 512        /* 읽혀질 덩어리 크기 */
#define PERM 0644          /* 새 파일의 file permission */

/* name1을 name2로 복사 */
int copyfile(const char *name1, const char *name2)
{
  int infile, outfile;
  ssize_t nread;
  char buffer[BUFSIZE];

  if((infile=open(name1,O_RDONLY))<0)
            return (-1);
```

Example: copyfile (cont)

```
if((outfile=open(name2,O_WRONLY|O_CREAT|O_TRUNC,PERM))<0)
 {
        close(infile);
        return (-2);
 }

/* name1으로부터 한번에 BUFSIZE 문자를 읽는다 */
 while((nread=read(infile,buffer,BUFSIZE))>0)
 {
        if(write(outfile,buffer,nread )< nread)
        {
                close(infile);
                close(outfile);
                return(-3);          /* 쓰기 오류 */
        }
 }
```

Example: copyfile (cont)

```
  close(infile);
  close(outfile);

  if(nread<0)
          return(-4);          /* 마지막 읽기에서 오류 발생 */
  else
          return(0);
}

main()
{
   return copyfile("test.in","test.out");
/* 0 이 return되면 성공적 수행이고  음수 값이 return 되면
    오류인데 –1 부터 –4 까지 4가지의 오류 상황이 있다. */
}
```

Efficiency: read/write

- 실험 환경
  - large file : 68,307 bytes
  - SVR4 UNIX with disk blocking factor of 512

BUFSIZE	Real time	User time	System time
1	0:24.49	0: 3.13	0:21.16
64	0: 0.46	0: 0.12	0: 0.33
512	0: 0.12	0: 0.02	0: 0.08
4096	0: 0.07	0: 0.00	0: 0.05
8192	0: 0.07	0: 0.01	0: 0.05

- 효율성의 향상은 대부분의 경우 system call의 횟수를 얼마나 줄이느냐에 달려 있다.

lseek **and random access**

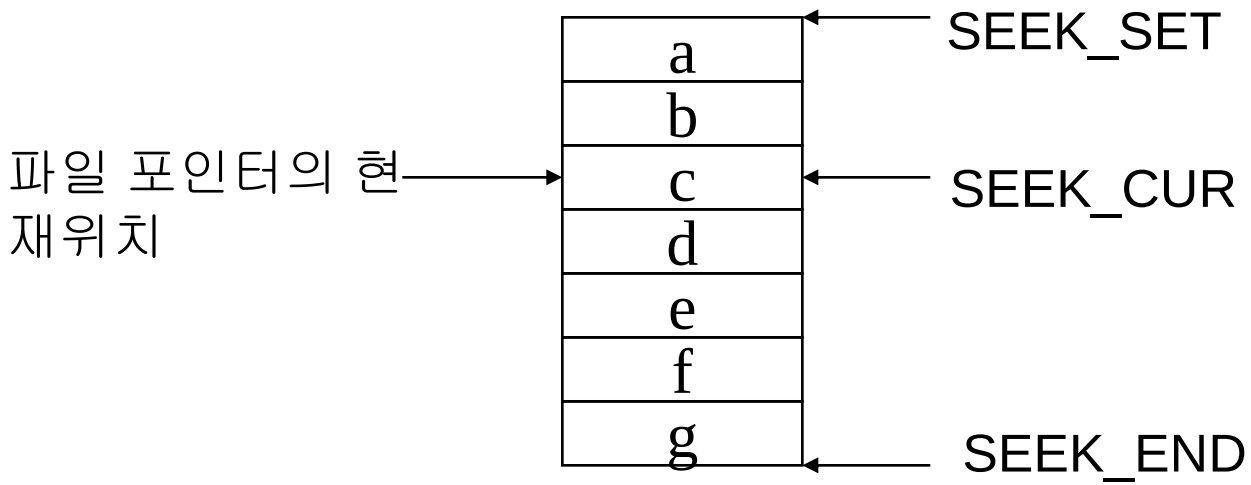
- Read-write pointer의 위치를 변경
  - 즉 다음에 읽거나 쓸 바이트의 위치
  - Random access
- Usage

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

off_t  lseek(int filedes, off_t offset, int start_flag);
```

lseek **and random access (cont)**

- Start flags : `<unistd.h>`에 정의
  - SEEK_SET: offset을 파일의 시작부터 계산
  - SEEK_CUR: offset을 파일 포인터의 현재 위치부터 계산
  - SEEK_END: offset을 파일의 끝부터 계산



lseek **and random access (cont)**

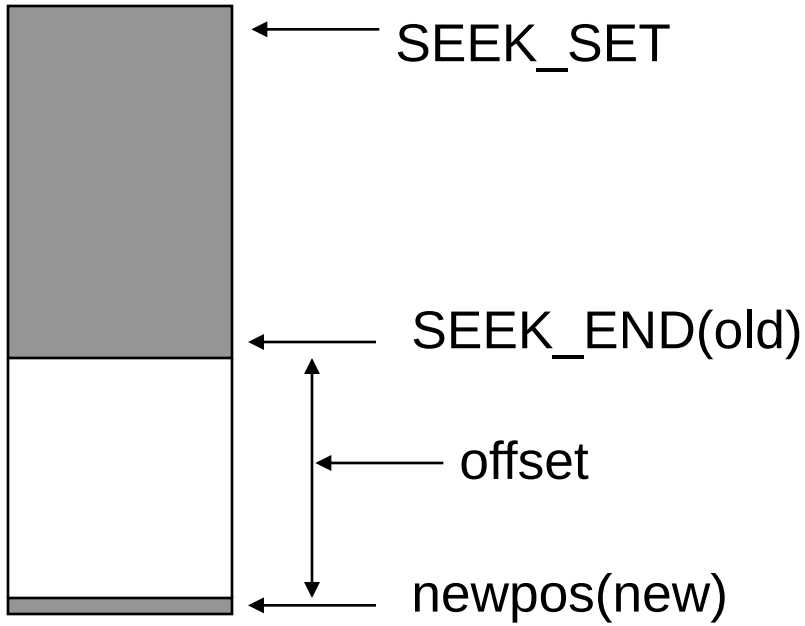
```
off _t newpos;
. . .
newpos = lseek(fd, (off_t)-16, SEEK_END);
```

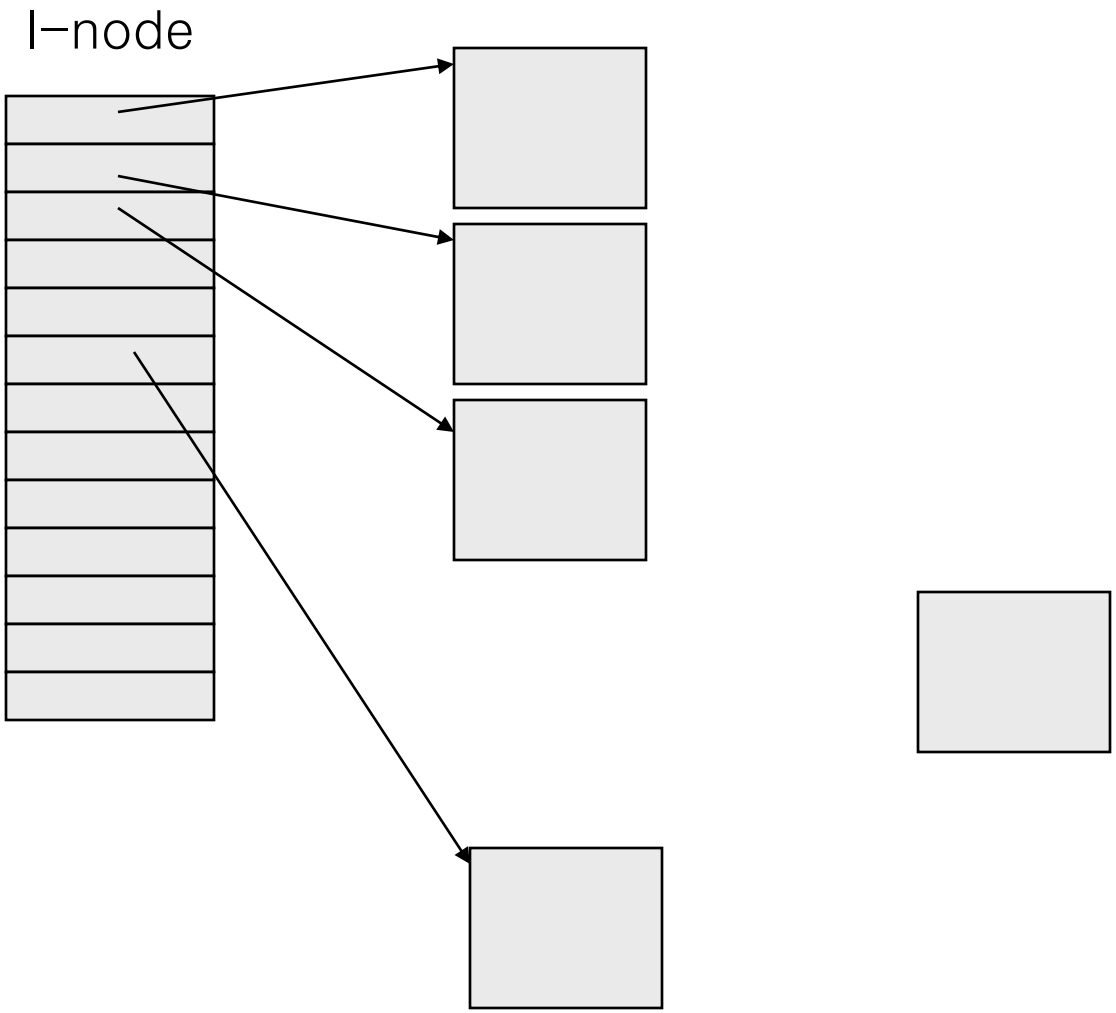
- newpos와 offset 둘다 <sys/types.h>에 정의된 off_t type이며 이는 충분히 큰 type 이다.

- offset은 음수가 될 수 있다.
  - 파일의 시작점 보다 더 앞으로 움직일 때만 오류가 발생

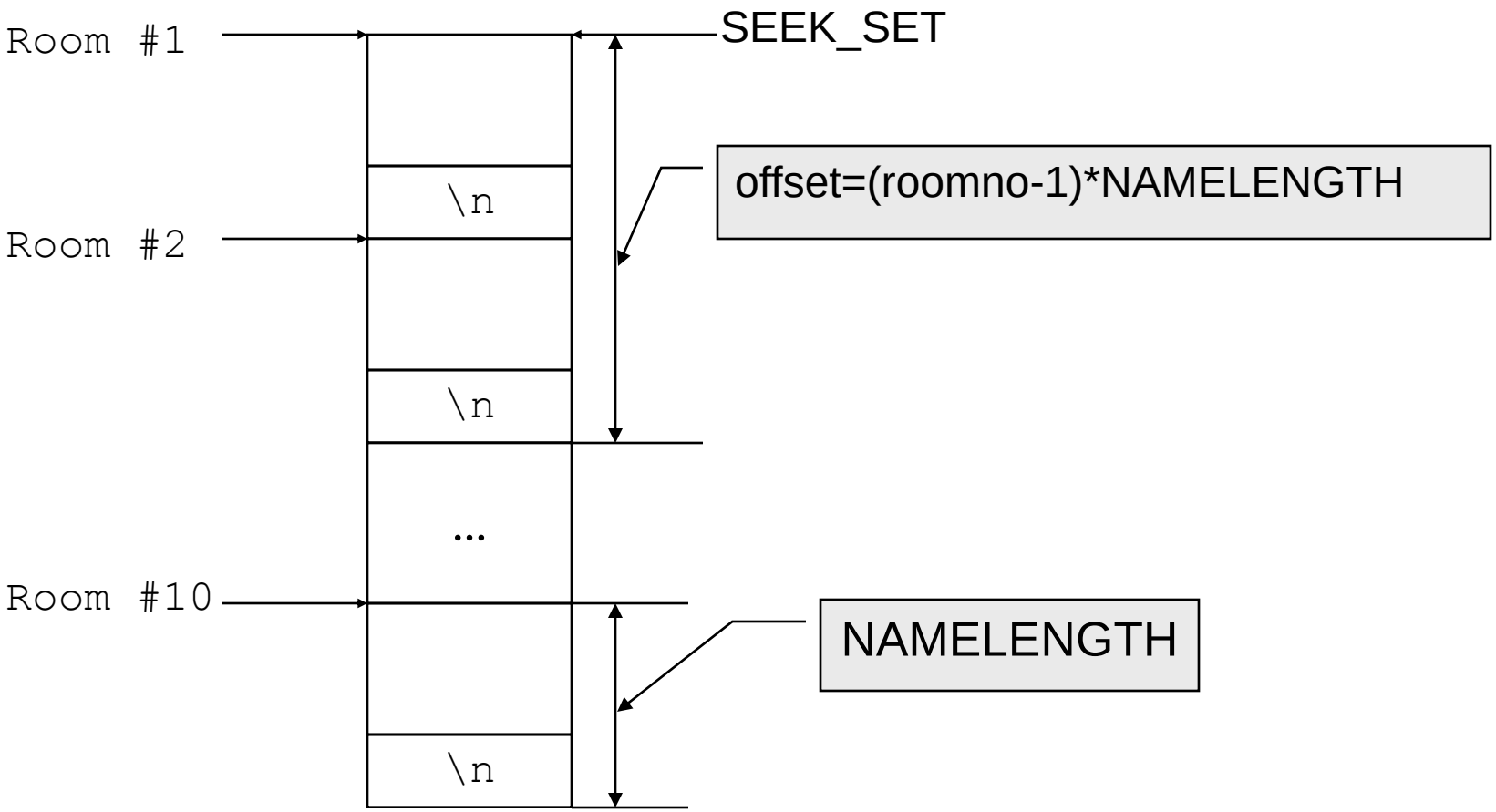
lseek **and random access (cont)**

- 파일의 끝 보다 더 뒤를 지정하는 것이 가능
  - 읽기를 위한 자료는 존재하지 않음
  - 이전의 파일 끝과 새 자료의 시작 위치 사이의 빈 공간은 실제로는 물리적으로 할당되지 않는다.




I–node

Example: Hotel



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

#define NAMELENGTH 41

char namebuf[NAMELENGTH];
int infile = -1;

char *getoccupier(int roomno)
{
  off_t offset;
  ssize_t nread;

  if(infile == -1 &&
       (infile = open("residents", O_RDONLY )) == -1)
      return(NULL);

  offset = (roomno - 1) * NAMELENGTH;

  if(lseek(infile,offset,SEEK_SET) == -1)
      return(NULL);
```

```
    if((nread = read(infile, namebuf, NAMELENGTH)) <= 0)
      return(NULL);

  namebuf[nread - 1] = '\0';
  return(namebuf);
}

#define NROOMS 10

main()
{
  int j;
  char *getoccupier(int), *p;

  for(j=1;j<=NROOMS;j++)
  {
    if(p=getoccupier(j))
      printf("Room %2d, %s\n", j,p);
    else
      printf("Error on room %d\n",j);
  }
}
```

Appending data to a file

- 파일 끝에 자료를 추가할 때

```
lseek(filedes, (off_t)0, SEEK_END);
write(filedes, appbuf, BUFSIZE);
```

- 보다 깔끔한 방법

```
filedes=open(“yetanother”, O_WRONLY | O_APPEND);
. . .
write(filedes, appbuf, BUFSIZE);
```

Deleting a file

- Usage

```
#include <unistd.h>
int unlink(const char *pathname);

#include <stdio.h>
int remove(const char *pathname);
```

  - 예:

```
unlink("/tmp/usedfile");
remove("/tmp/tmpfile");
```

- 빈 디렉토리를 제거하는 것은 remove(path)와 rmdir(path)가 동일하다.
- 디렉토리에 대해서는 unlink 대신 항상 remove를 호출

The fcntl **system call**

- 열려 있는 파일에 대한 제어를 제공
- Usage

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

int fcntl(int filedes, int cmd, …);
```

- cmd는 <fcntl.h>에 정의된 정수로 표현된 command
- …은 cmd를 위한 argument이며 type및 개수는 cmd가 무엇이냐에 달려있다.
- F_GETFL
  - 현재 파일의 status flag를 return
- F_SETFL
  - 파일에 연관된 status flag를 다시 지정
  - 일부만 가능

The fcntl system call (cont)

```
switch (O_ACCMODE & fcntl(filedes, F_GETFL))
  case O_WRONLY:      printf("write-only");
                      break;
  case O_RDWR:        printf("read-write");
                      break;
. . .
 }
```

특정 비트가 설정되어 있는지 테스트

```
if (fcntl(filedes, F_SETFL, O_APPEND) == -1)
   printf("fcntl error\n");
```

F_SETFL을 위한 argument

이후의 write는 파일 끝에 append함

2 Standard input, output and error

- File descriptor: 0, 1, 2
- Standard I/O Library와 혼동하지 말 것
- 키보드 입력, 단말기 출력
- Redirection

```
$ prog_name < infile
$ prog_name > outfile
$ prog_1 | prog_2
$ make > lot.out  2>  log.err
```

The io example

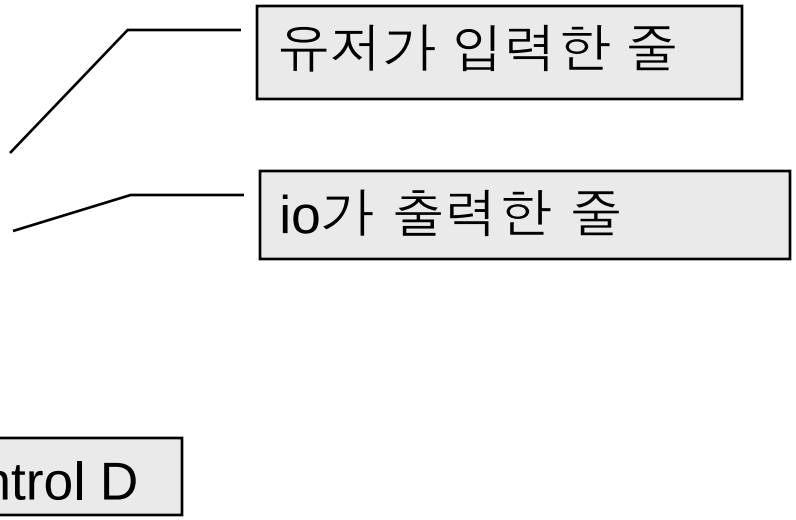
```
#include<stdlib.h>
#include<unistd.h>
#define SIZE 512
main()
{
ssize_t nread;
char buf[SIZE];
while((nread=read(0,buf,SIZE))>0)
        write(1,buf,nread);
exit(0);
}
```

File descriptor 0, 1을 이용
open(), creat()가 없음

The io example(cont)

❑실행 예

```
$ io
This is line 1
This is line 1
This is line 2
This is line 2
^D
$
```



```
$ io < /etc/motd > message

$ io < /etc/motd | wc
```

3 The standard I/O library

- 정수로 된 file descriptor 대신에 `FILE`이라는 structure를 가지고 작업
- fopen, putc, getc, printf, fprintf, …
- Standard I/O 와 System call (open, read, write,…)을 같은 모듈에서 섞어 쓰지 않도록 한다.

The fopen **example**

```
#include<stdlib.h>
#include<stdio.h>
main()
{
FILE *stream;

    if((stream=fopen("junk","r"))==NULL)
    {
       printf("Could not open file junk \n");
       exit(1);
    }
}
```

getc **and** putc

- Usage

```
#include <stdio.h>

int getc(FILE *istream);  /* istream으로부터 한 문자를 읽어라 */

int putc(int c, FILE *ostream);  /* ostream에 한 문자를 넣어라 */
```

filecopy

```
int c;
FILE *istream, *ostream

/* open istream for reading and ostream for writing */
. . .
while ((c=getc(istream))!=EOF)
        putc(c,ostream);
```

In <stdio.h>

```
#define EOF  -1

#define getc(p)     (--(p)->_cnt < 0 ? __filbuf(p) : (int)*(p)->_ptr++)

#define putc(x, p) (--(p)->_cnt < 0 ? __flsbuf((unsigned char) (x), (p)) \
                                    : (int)(*(p)->_ptr++ = (x)))
```

4 The errno variable and system calls

- 파일을 access하는 system call은 어떤 형태로든 실패할 수 있다.
- 이 때 더 많은 정보를 얻을 수 있도록, global variable errno를 제공하여 <errno.h>에 정의된 error code를 수록하도록 한다.

In <error.h>

```
#define EACCESS  13  /* Permission denied */
. . .
#define ENOENT  2  /* No such file or directory */
```

```
#include<stdio.h>
#include<fcntl.h>
#include<errno.h>
main()
{
  int fd;
  if((fd=open("nonesuch",O_RDONLY))==-1) {
     fprintf(stderr,"error %d\n",errno);
     perror(“Error opening nonesuch”);
     }
}
```

```
$ a.out   % if nonesuch doesn’t exist
error 2
Error opening nonesuch: No such file or directory
```

System error message

- perror
  - 자신에게 전달된 문자열 인수, 콜론, 그리고 errno의 현재 값에 연관된 추가 시스템 에러 메시지를 출력한다.

```
perror("error opening nonesuch");
-------------------------------
error opening nonesuch: No such file or directory
```

- strerror
  - System error message string을 얻는다

```
printf("%s\n",strerror(EACCESS));
-------------------------------
Permission denied
```

Writing error messages with fprintf

```
#include<stdio.h>
#include<stdlib.h>
int notfound(const char *progname, const char *filename)
{
  fprintf(stderr, "%s : file %s not found\n", progname,filename);
  exit(1);
}
```