

Ch03 System-Level I/O

I/O 경로 Main memory \leftrightarrow External device (HDD, SSD, Network) \rightarrow Main Memory

File 여러 byte의 sequence \rightarrow B₀, B₁, ..., B_{m-1}
 → 파일 I/O device \rightarrow /dev/sda2 \rightarrow /usr Disk Partition
 → CDRW로도 파일 \rightarrow /dev/hd_Y \rightarrow E10UY
 → kernel도 파일
 → 파일 표현 UNIX I/O (System Call)

`open()`: 파일 열기 `read()`: 파일 읽기
`close()`: 파일 닫기 `write()`: 파일 쓰기.

`lseek()`: "Current File Position"을 찾는다
 : 즉, 입출력 위한 파일 내의 'Offset'를 찾는다.
 : `lseek`을 통해 offset 대신으로 변경 가능

File Types

Regular file: 일반적인 파일, arbitrary data (임의의 정보)

Directory file: 폴더, 관련된 파일의 경로를 위한 인덱스'를 가진 파일

Socket: 네트워크 통신 용도

Regular file

↳ application 파일이 모두 regular file에 해당

→ Binary File → Text File

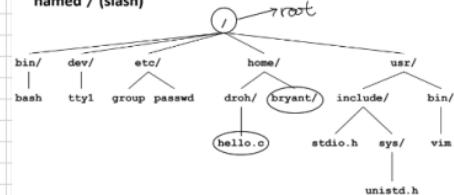
- : 바이너리 파일 형식
- : ASCII, Unicode 등으로 이진화된 파일
- : Sequence of text lines \rightarrow text lines은 whitespace로 끝남
- : EOL (End of Line)
 - \rightarrow Linux & Mac : 0xa = LF
 - \rightarrow Windows : 0xd 0xa (0xd0rn)

Directories

- 파일 이름에 대해 위치를 나타내는 Link의 array
- array of links
- \rightarrow 각각 개별
- \rightarrow 파일 directory

- Absolute Pathname (절대 경로)
 $/home/droh/hello.c$
- Relative Pathname (상대 경로)
 $./droh/hello.c$

■ All files are organized as a hierarchy anchored by root directory named / (slash)



File Open & Close

```
int fd; /* file descriptor */
int fdval; /* return value */

if ((fd = open("/etc/hosts", O_RDONLY)) < 0) {
    perror("open");
    exit(1);
}
```

- 파일을 열수는 것은 kernel에게 "파일 접근을 준비됐어" 알리드 것
- return value 3: File Descriptor 반환
- 1이면 error 발생
- ✓ 3번

0 Standard input (stdin)
 1 Standard output (stdout)
 2 Standard error (stderr)

```
int fd; /* file descriptor */
int retval; /* return value */

if ((retval = close(fd)) < 0) {
    perror("close");
    exit(1);
}
```

- 이와 같은 파일을 또 닫는 것은 최악

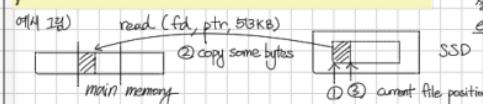
File Read & Write

File Reading

→ current file position (SSD or something) 예상 위치,
 ↳ 파일 내용은 main memory에 \rightarrow current file position을 update
 ↳ 파일 \neq fd update

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */
```

```
/* Open file fd ...
 * Then read up to 512 bytes from file fd */
if ((nbytes = read(fd, buf, sizeof(buf))) < 0) {
    perror("read");
    exit(1);
}
```



→ fd의 current file position #1 sizeof(buf) 만큼
 ↳ buf에 있는 byte가 실제
 → return fd의 size_t인 signed integer
 → nbytes (return 값)이 Biggest error 가능

→ short count
 ↳ 예상보다 더 많은 sizeof(buf)보다
 실제 있는 nbytes (nbytes)가 작은 것은
 error가 아님.

SSD nbytes < sizeof(buf)

File Writing

```
char buf[512];
int fd; /* file descriptor */
int nbytes; /* number of bytes read */
```

```
/* Open the file fd ...
 * Then write up to 512 bytes from buf to file fd */
if ((nbytes = write(fd, buf, sizeof(buf))) < 0) {
    perror("write");
    exit(1);
}
```

Source

→ sizeof(buf) 만큼 buf를 못쓰면서
 ↳ fd의 current file position에서부터 넣고
 한 byte씩 반복

→ nbytes (return 값)이 Biggest error 가능
 → read #1의 short count error가 아님.

Simple Unix I/O example

```
#include <csapp.h>
int main(void)
{
    char c;

    while(Read(STDIN_FILENO, &c, 1) != 0)
        Write(STDOUT_FILENO, &c, 1);
    exit(0);
}
```

lseek()

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence)
```

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

```
int main() {
    int fd = open("example.txt", O_RDWR);
    if (fd == -1) {
        perror("open");
        return 1;
    }
```

// 현재 오프셋 확인

```
off_t pos = lseek(fd, 0, SEEK_CUR);
printf("Current position: %ld\n", pos);
```

// 파일 시작에서 5바이트 앞으로 이동

```
lseek(fd, 5, SEEK_SET);
write(fd, "HELLO", 5); // 파일 처음에서 5번째 바이트에 HELLO
덮어쓰기
```

// 파일 끝에서 0바이트 떨어진 위치로 이동

```
off_t end = lseek(fd, 0, SEEK_END);
printf("File size: %ld bytes\n", end);
```

```
close(fd);
return 0;
```

→ 한글자씩 읽어서 화면에 쓰는 프로그램.
→ 예제의 내용을 참고.
whence → read & write, system call : overhead ↑
↳ 2MMI, read & write 시에는
화면위치로 byte 단위로 읽는 것의 활용.

SEEK_SET 파일 처음
SEEK_CUR 현재 position
SEEK_END 파일 끝

off_t

fd
offset
whence

SEEK_S
SEEK_C
SEEK_E

short count

→ read or write AI 불필요한 수 0.

→ 예제?

- read, 즉 EOD(End of File) 만날 때
- reading text lines from a terminal
- reading & writing network sockets

→ 이런 상황에서는 never 발생

- disk file에서 읽을 때 (EOF 제외)
- disk file에 쓸 때

RIO Package

unbuffered input & output of binary data

rio_read, rio_write

buffered input & output of binary data

rio_readlineb, rio_readnb

↳ thread safe, interleaved arbitrarily on the same descriptor

Unbuffered RIO Input & Output

rio_read → eof 상태로 short count를 return

↳ interleaved arbitrarily on the same descriptor

rio_write → short count return하지 X

```
// 1. 파일디스크립터 식별자, 저장할 메모리 주소, 메모리 크기
ssize_t rio_readn(int fd, void *usrbuf, size_t n)
{
    /**** 1000 ****/
    size_t nleft = n;           // usrbuf의 남은 메모리 공간
    size_t nread;               // (signed) 읽은 Byte 수 저장할 변수
    char *bufp = usrbuf;        // 메모리 주소 (이동)

    /* 더 이상 읽을 데이터 없을 때 까지 */
    while (nleft > 0) {
```

```
    /* fd에서 데이터 읽어 bufp에 저장 (read : 읽은 Byte 수) */
    if ((nread = read(fd, bufp, nleft)) < 0) {
```

```
        /* sys call(read) 중에 시그널 핸들러에 의해 인터럽트된 상황 */
        if (errno == EINTR) // EINTR : 시그널 핸들러에 의해 인터럽트를 나타내는 상수
            nread = 0;
        else
            errno = sys call 이 실패할 때 결정되는 오류 코드 (전역변수)
```

```
        /* sys call 오류발생 */
        else
            return -1;
    }
```

```
    else if (nread == 0) // 더 이상 읽을 데이터가 없음
        break; /* EOF */
```

```
    nleft -= nread;          // 처음 n의 크기에서 읽은 Byte 만큼 빼는 작업 반복
    bufp += nread;           // 읽은 Byte 만큼 메모리 주소 이동
}
```

```
} // 전체 - 남은 공간 = 전체 데이터 크기..? */

return (n - nleft); // Return >= 0 */
/* 전체 - 남은 공간 = 전체 데이터 크기..? */
```

Buffered RIO Input functions

rio_readinitb → 초기화 함수

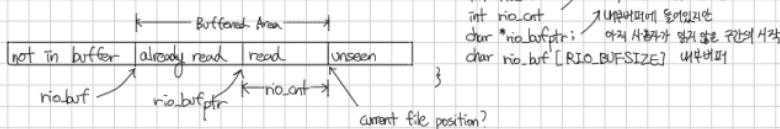
rio_readlineb → 한 줄 단위로, 빠른 반응. file descriptor maxline byte 단위로, variable size
종료조건
↳ maxline 만큼 읽었을 때
↳ EOF 만났을 때
↳ Newline character를 만났을 때 - readnub 처리

rio_readnb → 파일 처리 byte 단위로 일을 했을 때 maxline byte.
종료조건
↳ maxline 만큼 읽었을 때
↳ EOF 만났을 때

→ can be interleaved.
arbitrarily on the same
descriptor (4B 744F 4E5 5)

2개 이상 rio_readnb 를
4번까지 가능?

Buffered I/O: Implementation



```
#include "csapp.h"
```

```

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
        Rio_writen(STDOUT_FILENO, buf, n);
    exit(0);
}
  
```

cpfile.c

■ **Metadata** is data about data, in this case file data

■ **Per-file metadata maintained by kernel**

accessed by users with the **stat** and **fstat** functions

```

/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t      st_dev;      /* Device */
    ino_t      st_ino;      /* inode */
    mode_t     st_mode;     /* Protection and file type */
    nlink_t    st_nlink;    /* Number of hard links */
    uid_t      st_uid;      /* User ID of owner */
    gid_t      st_gid;      /* Group ID of owner */
    dev_t      st_rdev;     /* Device type (if inode device) */
    off_t      st_size;     /* Total size, in bytes */
    unsigned long st_blksize; /* Blocksize for filesystem I/O */
    unsigned long st_blocks; /* Number of blocks allocated */
    time_t     st_atime;    /* Time of last access */
    time_t     st_mtime;    /* Time of last modification */
    time_t     st_ctime;    /* Time of last change */
};
  
```

and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

27/33

Sogang Universi

Example of Accessing File Metadata

```

int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode)) /* Determine file type */
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* Check read access */
        readok = "yes";
    else
        readok = "no";

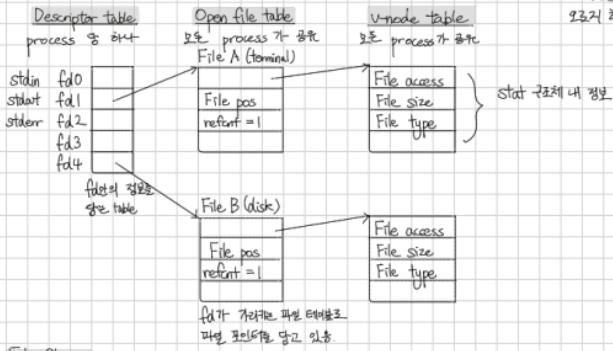
    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
  
```

```

linux> ./statcheck statcheck.c
type: regular, read: yes
linux> chmod 000 statcheck.c
linux> ./statcheck statcheck.c
type: regular, read: no
linux> ./statcheck ..
type: directory, read: yes
  
```

statcheck.c

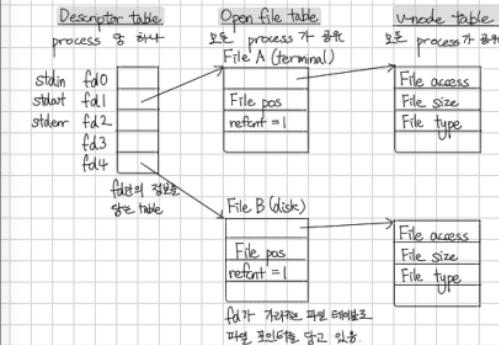
How the Unix Kernel Represents Open Files



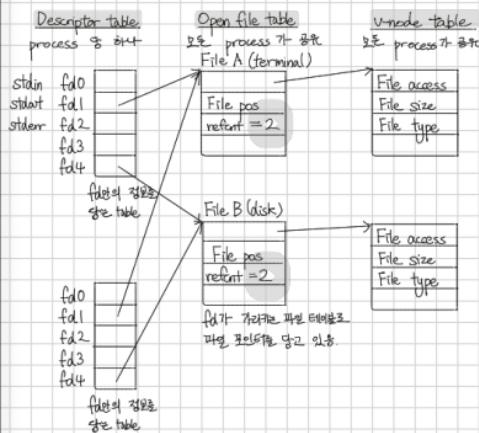
• V-node table은 하나의 파일을 처리
932 2449번 문제

How Process Share Files: fork

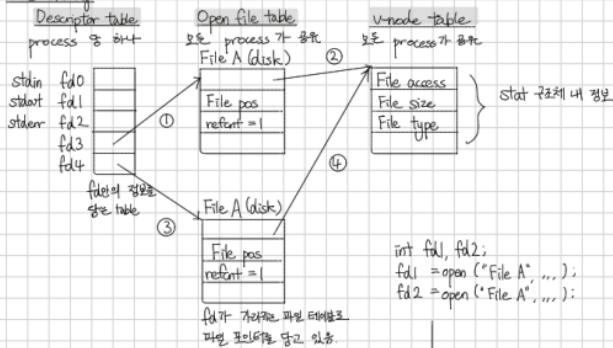
Before fork



After fork



File Sharing



서로 다른 descriptor를 동일한 disk file을 접근

4 되는게 아니라 1이 4 4
8

I/O Redirection

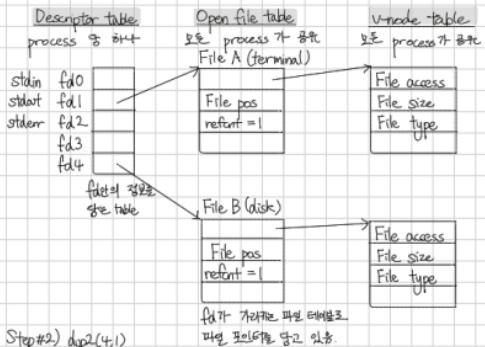
dup2(fd0, newfd) dup2(4,1)

before	
fd0	
fd1	a
fd2	
fd3	
fd4	b

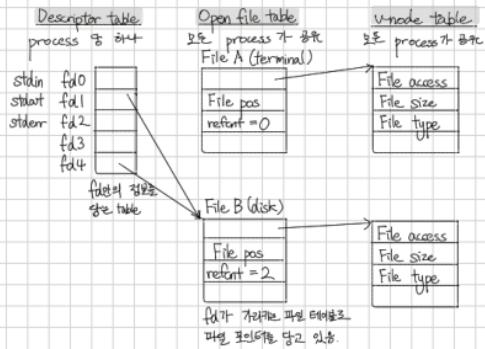
after	
fd0	
fd1	b
fd2	
fd3	
fd4	b

ex)

Step #1)



Step #2) dup2(4,1)



Standard I/O Functions

opening & closing files (fopen & fclose)

reading & writing bytes (fread & fwrite)

reading & writing textlines (fgets & fputs)

formatted reading & writing (scanf & printf)

Standard I/O Streams

Stream API 3 종류: stdin, stdout, stderr

Buffered I/O : Motivation

41 buffered I/O를 선택한 것인가.

1) gets, puts, ungetc / fgets, fputs

↳ 단위로 한문자씩 읽거나 newline마다 읽는다.

2) UNIX I/O는 unbuffered I/O clock cycle↑

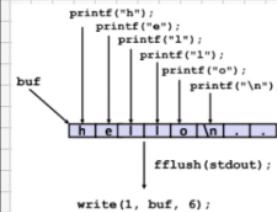
Solution : Buffered read

→ unix I/O의 read는 끝나기 전에 버퍼에 내용을 넣어두고 있다.

→ user-level input function은 이전처럼 1byte씩 가져온다.

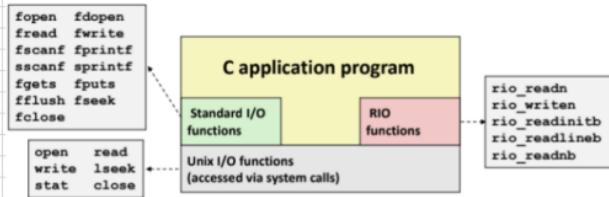
→读后 refill

Buffering in Standard I/O



연속된 printf가 데이터 3개를 동시에 write()로

Unix I/O vs Standard I/O vs RIO



Pros & Cons of Unix I/O

Pros	Cons
→ most general & lowest overhead form of I/O → 대용량 I/O 패키지로 UNIX I/O API 제공	→ short-circuit reading 가능 → 예상치 못한 가능 → unbuffered, 즉 실시간 읽기 가능
→ 파일의 metadata, 경로 정보 포함 (stat, fstat) → 파일	→ metadata를 포함한 경우 → 파일
→ async-signal-safe API, signal handler에서 사용 가능	

Pros & Cons of Standard I/O

Pros	Cons
→ buffered I/O의 efficiency↑, read/write often call → 대용량 읽기 쓰기	→ metadata를 포함한 경우 X → async-signal-safe API X
→ short-circuit 읽기 쓰기 가능	→ network socket API 사용 가능

Caching I/O Functions

general rule : 일반적인 경우 highest-level I/O Function 사용

단기 Standard I/O? : disk + terminal 같은 경우 사용

언제 raw Unix I/O? : signal handler 사용하거나 사용 시
when you need absolute highest performance?

什么时候 RIO ?: reading or writing network sockets
socket API, Standard I/O 대체

Working with Binary Files

- Text oriented I/O ex) fgets, scanf, rio_readlineb
GTL rio_readdn, rio_readdnb
- String functions ex) strton, strcpy, strcat

Fun with File Descriptors (1)

```

#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char c1, c2, c3;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    fd2 = Open(fname, O_RDONLY, 0);
    fd3 = Open(fname, O_RDONLY, 0);
    Dup2(fd2, fd3);
    Read(fd1, &c1, 1); a
    Read(fd2, &c2, 1); a
    Read(fd3, &c3, 1); b
    printf("c1 = %c, c2 = %c, c3 = %c\n", c1, c2, c3);
    return 0;
}
  
```

abcde

ffiles1.c

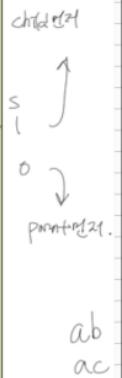
■ What would this program print for file containing "abcde"?

```

#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1;
    int s = getpid() & 0x1;
    char c1, c2;
    char *fname = argv[1];
    fd1 = Open(fname, O_RDONLY, 0);
    Read(fd1, &c1, 1); getpid() % 2 == 1 → 1
    if (fork()) { /* Parent */
        sleep(s);
        Read(fd1, &c2, 1);
        printf("Parent: c1 = %c, c2 = %c\n", c1, c2);
    } else { /* Child */
        sleep(1-s);
        Read(fd1, &c2, 1);
        printf("Child: c1 = %c, c2 = %c\n", c1, c2);
    }
    return 0;
}

```

ffiles2.c



```

#include "csapp.h"
int main(int argc, char *argv[])
{
    int fd1, fd2, fd3;
    char *fname = argv[1];
    fd1 = Open(fname, O_CREAT|O_TRUNC|O_RDWR, S_IRUSR|S_IWUSR);
    Write(fd1, "pqrs", 4);
    fd3 = Open(fname, O_APPEND|O_WRONLY, 0);
    Write(fd3, "jklmn", 5);
    fd2 = dup(fd1); /* Allocates descriptor */
    Write(fd2, "wxyz", 4);
    Write(fd3, "ef", 2);
    return 0;
}

```

ffiles3.c

- What would be the contents of the resulting file?

pqrs jklmn pqrs wxyzn. jklmn. ef

- What would this program print for file containing "abcde"?

Accessing Directories

- Only recommended operation on a directory: read its entries
 - `dirent` structure contains information about a directory entry
 - `DIR` structure contains information about directory while stepping through its entries

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

{
    DIR *directory;
    struct dirent *de;
    ...
    if (!(directory = opendir(dir_name)))
        error("Failed to open directory");
    ...
    while (0 != (de = readdir(directory))) {
        printf("Found file: %s\n", de->d_name);
    }
    ...
    closedir(directory);
}
```