chapter04. UNIX I/O

objectives

1. Learn the basics of device-independent I/O
2. Experiment with read and write
3. Explore ways to monitor multiple descriptors
4. Use correct error handling
5. Understand inheritance of file descriptors

Device Terminology 1

Peripheral device

- 컴퓨터의 I/O device → piece of H/W accessed by a computer system.

  eg. disks, tapes, CD_ROMS, screens, keyboards, printers, mouse devices, network interfaces

- Processer와 communicate

Device driver

- S/W module로 H/W 제어하는 device (OS가 관리 → rapping 해서 H/W 숨겨 놓음)
- user program perform control and I/O to these devices through system calls to OS modules.
- hides the details of device operation + protects the device from unauthorized use.
- interface만 제공

Device Terminology 2

Unix

- 굉장히 간단한 programmer device interface를 가지고 있음.
  - open, close, read, write, and ~~ioctl~~ → system call func.

    → 이 다섯개의 함수를 통해 대부분의 device에 접근 가능(I/O device는 전부!)

- **Unix files** : is a sequence of m bytes
  - unix 자체에서는 file 구분을 x
  - user가 자체적으로 file 구분 짓고:사용함(ex. 확장자 이용)
  - **I/O devices**
    - hard disk partition : /dev/sda2
    - terminal : /dev/tty2
  - **kernel**
    - kernel memory image : /dev/kmem
    - kernel data structures : /proc

Special files : 주변 장치를 나타냄

- all devices are represented by files, called special files
  - that are located in the /dev directory ☆

ex) disk files and other devices : 이러한 방식으로 접근, named

UNIX I / O

: the elegant mapping of files to devices allows a simple, low-level interface

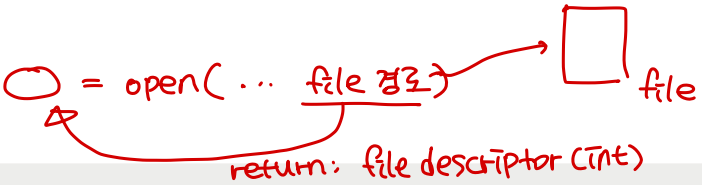
- key idea : all input and output is handled in a consistent and uniform way. ☆

- **Basic Unix I/O operations (system calls)**
  1. Opening and closing files
     - open() : an application announces its intention to access an I/O device
     - close() : an application has finished accessing the file(open 한 file은 close 할 것)
  2. Changing the current file position
     - seek → lseek(in Linux)
       - kernel : maintains a file position k, initially 0 for each open file.
         - the file position : a byte offset from the begining of a file
       - an application can set the current file position k explicitly by calling seek func.
  3. Reading and writing files
     - read() : read n bytes from a file to memory
     - write() : write n bytes from memory to a file

Reading

Unix provides sequential access to files and other devices through the read and write func.

Read func




```
#include <unistd.h>


//locaion of current file offset에서 memory area로 specifed length만큼 읽기
ssize_t read(int fildes, void *buf, size_t nbyte);

//ssize_t : a signed integer data type used for the number of bytes read
//size_t : a unsigned integer data type used for the number of bytes read

//fildes : open()의 return 값, 즉 file descripter
//buf : file로부터 읽어들인 data 저장할 pointer(user의 buffer), buf size >= nbyte
//      초기화되지 않은 pointer 값을 제공하면 x
//nbyte : 읽어들일 양(>0) => buf의 크기만큼 nbyte를 지정해야 함
```

data type

parameter

error 발생시 -1 return

```
//error 발생 시 -1 return
```

- return value
  - the number of bytes actually read, if successful ⇒ 1 byte라도 읽으면 성공
  - -1 and sets errno if unsuccessful
- read operation
  - for a regular file : 요청된 것보다 더 적게 return 될 수도 있음
    - 보통 한 줄씩 모든 파일 내용을 읽을 때 잘 발생
    - request를 완전히 만족하기 전에 file의 끝에 도달했을 때도 가능
  - return 0 for a regular file : end-of-file을 알림
    - 보통 read return 값이 0이 될 때가 읽기 종료 조건이 됨

file descriptor

represents a file or device that is open

- shell로부터 program을 실행시킬 때 program은 3개의 open stream으로 시작 가능
  - I/O에 대해서 자동으로 입출력 가능 → 따로 open 호출하지 않더라도 바로 read 실행 가능
  - parameter에서 file descripter 위치에 넣으면 됨
  - 각각의 stream은 숫자가 define

◦ Open Stream

1. STDIN_FILENO
   - standard input
   - corresponding to keyboard input
   - in legacy code it is represented by 0
2. STDOUT_FILENO (→ write 함수 실행 시에도 가능)
   - standard output
   - corresponding to screen output

- in legacy code it is represented by 1

3. STDERR_FILENO
   - standard error device
   - programs should never close it (→ file 우선순위가 높을 경우 사용할 것)
   - in legacy code it is represented by 2

Example

```
//#1

char buf[100];
ssize_t bytesread;

bytesread = read(STDIN_FILENO, buf, 100);

//reads at most 100 bytes into buf from standard input


//#1 - warning ver.

char *buf; => 선언X → warning!
ssize_t bytesread;

bytesread = read(STDIN_FILENO, buf, 100);

//buf가 가리키는 메모리가 x -> arr 선언 or malloc과 같이 동적 할당이 필요함
```

⇒ result가 예측불가능 해짐
(아마 대부분이 memory access violation을 생성)

```
//#2 - Readline.c

#include <errno.h>
#include <unistd.h>

//read와 유사하게 작성 -> line 단위로 읽어 옴
int readline(int fd, char *buf, int nbytes){
  //file descripter, buf(data save), nbytes(한 줄 읽을 때의 upper bound)
  int numread = 0; //지금까지 읽어들인 byte 수 = 기존 read return 값
  int returnval; //찐 read 함수의 return 값

  while(numread < nbutes -1){
    //while 종료 조건 : numread가 nbytes-1과 같아지면 => 한 줄이 너무 길어지면
    //                  작을 때 까지만 입력 받도록!! => Null이 들어갈 한 자리는 남아야 하기 때문

    returnval = read(fd, buf + numread, 1); //numread = idx를 나타냄
    if((returnval == -1) && (errno == EINTR)) //interrupt 처리
```

3 nbutes -1
1) read
2) err check
EINTR → interrupt

```
    continue;

    //error handling
  if((returnval == 0) && (numread == 0))
    return 0;
  if(returnval == 0) //end of file
    break;
  if(returnval == -1) //input file error(즉, new line char가 없음)
    return -1;

  numread++; //읽은만큼 byte++

  if(buf[numread-1] == '\n'){ //line의 끝에 도달했는지 체크(즉, new line char check)
    buf[numread] = '\0'; // "\" + "NULL" 자리가 필요함 -> string으로 구성하여 return
    return numread;
   }
  }
  errno = EINVAL;
  return -1;
}

//#3 - readline 호출 example

int bytesread;
char mybuf[100];

bytesread = readline(STDIN_FILENO, mybuf, sizeof(mybuf));

//mybuf : %s로 point -> NULL character가 들어갔기에 string으로 취급 가능
//array 크기 : 100 -> 99byte보다 작을 동안 입력 받아 mybuf에 작성
```

- error handling
  1. an error occured on read() ⇒ returnval == -1 조건
  2. at least one byte was read and an end-of-file occurred before a newline was read

     ⇒ returnval ==0 조건
  3. nbytes-1 bytes were read and no newline was found ⇒ while문 종료 조건

Writing

write function

attempts to ouput nbytes from the user buffer buf to the file represented by file descriptor fildes.

⇒ user buffer에서 filedes가 나타내는 file로 nbytes ouput을 할당

- 0보다 크고 nbyte보다 작은 value를 return → error가 아님

0 < value < nbyte

```
#include <unistd.h>

//Memory area에서 current file offset으로 specified length만큼 쓰기

ssize_t write(int fildes, const void *buf, size_t nbyte);

//fildes : file 위치(file descriptor)
//buf : arr 주소 pointer 값
//nbyte : 불러오려는 byte 수

//ssize_t : the number of bytes actually written
```

parameter

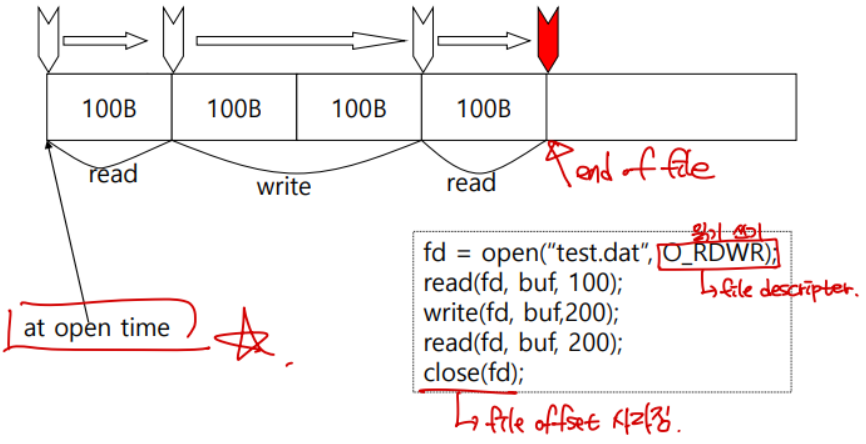
data type →

file offset

: current I/O position → open된 file의 current location (like. location of VCR Header)

↳ 읽고자 하는 지점

- open 되면 초기화 됨
- 처음 open 하면 0 ★
- read, write와 같은 I/O 작업 후에 자동으로 이동 → 다음 I/O operation start location



Example 1 - warning

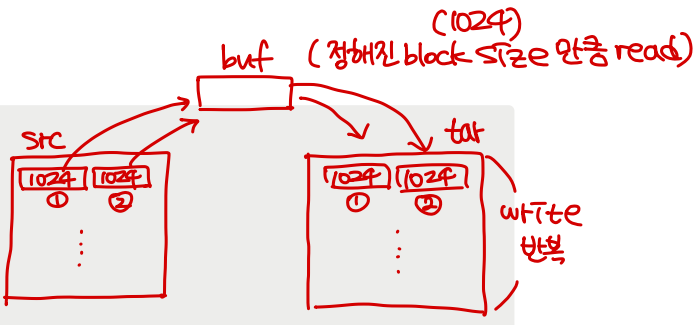
: read 실패 혹은 write의 nbytes보다 적게 읽었을 때의 handling이 없음

```
#define BLKSIZE 1024
//최대 크기 define

char buf[BLKSIZE];

read(STDIN_FILENO, buf, BLKSIZE); //standard I/O
write(STDOUT_FILENO, buf, BLKSIZE); //standard I/O

//write -> read가 BLKSIZE bytes 만큼 buf를 채울 것이라고 가정한 뒤 시행됨
//but, read가 실패할 수도 있고 BLKSIZE만큼 읽지 않을지도 모름
=>write ouput-> garbage
```

Example2 - warning

```
#define BLKSIZE 1024
char buf[BLKSIZE];
ssize_t bytesread;

bytesread = read(STDIN_FILENO, buf, BLKSIZE);
if(bytesread > 0)
  write(STDOUT_FILENO, buf, bytesread);

//write -> BLKSIZE가 아닌 실제 read한 byte 수만큼 요청
//        but, write가 실제로 요청된 모든 byte를 출력한다는 보장 x
//read나 write가 signal에 의해 interrupted 될 수 도 있고
//이 경우에 interrupted call이 -1을(EINTR이면 read를 다시 호출해서 제대로 실행) return
```

Example3(read and write)




```
#include <errno.h>
#include <unistd.h>
#define BLKSIZE 1024

int copyfile(int fromfd, int tofd){
  //fromfd(src) : read, tofd(target) : write
  //src -> 정해진 block size만큼 buf에 복사 -> target에 write => 다 복사될 때까지 반복
  char *bp;
  char buf[BLKSIZE];
  int bytesread; //실제로 읽은 byte 수
  int byteswritten = 0; //target에 쓴 byte 수
  int totalbytes = 0; //복사된 총 byte 수
```

```
  for( ; ; ){
    //source 끝나면 빠져나감

    while(((bytesread = read(fromfd, buf, BLKSIZE)) == -1) && (errno == EINTR));
    //fromfd로부터 buf에 BLKSIZE만큼 read 후 bytesread에 읽은 byte 수 할당 && interrupt handling
    //(src fd)
    if(bytesread <= 0) //real error or end-of-file on fromfd
      break;

    bp = buf;   //char pointer : buffer 내부값, 처음에는 시작 지점 가리킴

    while(bytesread > 0){ //read한 bytesread가 0이 될 때까지(target에 써야 되는 잔여 byte)
      while(((byteswritten = write(tofe, bp, bytesread)) == -1) && (errno == EINTR));
      //tofd로부터 bp부터 bytesread만큼 쓴 후 byteswritten에 쓴 byte 수 할당 && interrupt handling
      //(target fd)

      if(byteswritten <= 0) //real error on tofd
        break;

      totalbytes += bytewritten; //쓴 만큼 update -> src byte와 동일하면 끝
      bytesread -= bytewritten; //읽은 만큼 update -> 0이 되면 끝
      bp += byteswritten; //bp : pointer가 이동하여 다음에 써야 할 data 시작 위치
    }
    if(byteswritten == -1) //real error on tofd
      break;
  }
  return totalbytes;
}
```

- copies bytes from the file, fromfd to the file, tofd.
- the func restarts read and write if either is interrupted by a signal.
- write statement : specifies the buf by a pointer, bp. rather than by buf

restart read/write after a signal

- r_read(), r_write : greatly simplify programs that need to read and write while handling signals.
  - problem of r_read()
    - only restarts if interrupted by a signal and often reads fewer bytes than requested.

  ⇒ readblock() : 요청한 byte 다 읽을 때까지 작동함 → I/O utility 함수

- readblock()
  - a version of read that continues reading until the requested number of bytes is read or an error occurs.
  - return values
    - 0 → if an end-of-file occurs before any bytes are read
    - requested number of bytes → successful
    - -1 → if error

- readwrite() → using r_read(), r_write() ⇒ 읽고 쓰는 과정 더 간단하게!
  - reads bytes from one file descriptor and writes all of the bytes read to another one.
  - PIPE_BUF : useful for writing to pipes since a write to a pipe of PIPE_BUF bytes or less is atomic. → a buffer size of readwrite()

- copyfile() : a version that uses readwrite() func.

Opening

Open

reading, writing을 위해서는 file open이 먼저 필요

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

int open(const char *path, int flag);
int open(const char *path, int flag, mode_t mode);

//kernel return -> a small non-negative integer called a file descriptor
//return value = -1 -> error

//path : point to the pathname of the file or device.
//flag : specifies status flags and access modes for the opened file
```

return

parameter

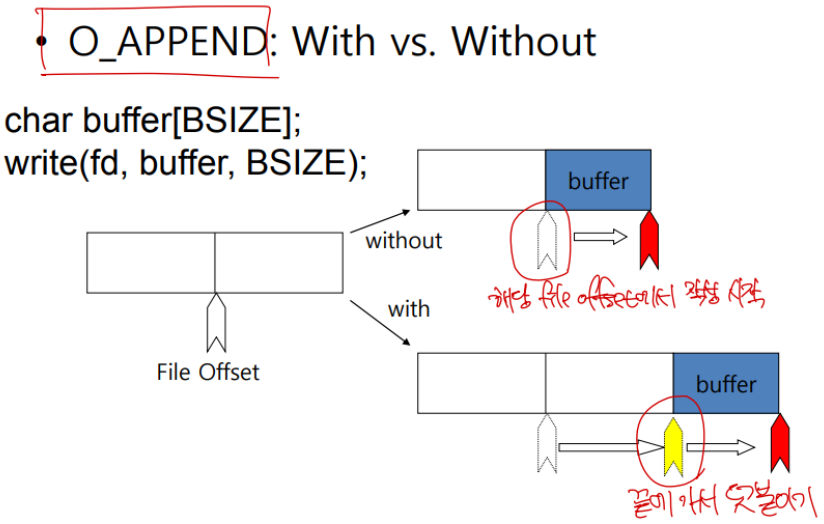
```
//mode : access permission -> mode_t data type
//만약 file을 creating -> must include a third parameter to spcify access permission
```

Flag

construct the flag argument by taking the bitwise OR(|) of the desired combination of the access mode and the additional flags.

ex) O_WRONLY | O_APPEND

- access mode
  - must specity exactly one of access mode.
    - O_RDONLY : read only
    - O_WRONLY : write only
    - O_RDWR : read, write
  - additional flags
    - write mode
      - O_APPEND : the file offset is moved to the end of the file before a write
        - file offset을 file의 끝으로 옮긴 후 작성을 시작



      - O_TRUNC : truncates the length of a regular file opened for writing to 0

    - file의 크기를 0 → file의 기존 data를 삭제
- O_CREAT : create the file if it does not exist. you must use a third argument to designate the permissions.
    - file이 존재하지 않는다면 file 생성. 3번째 parameter를 사용하여 permission 디자인
- O_EXCL : used with O_CREAT, return an error if file exists.
    - O_CREAT를 사용하되 파일이 존재하면 에러를 반환 → 기존 file 사용 x

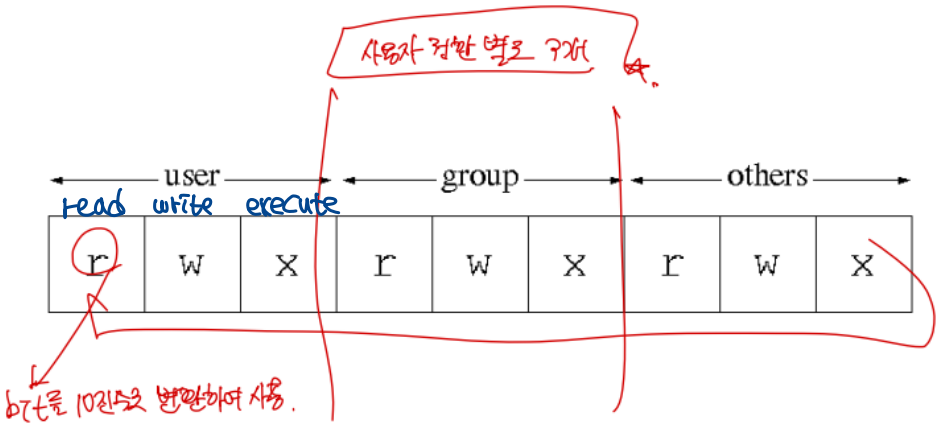
- example : 덮어쓰는 문제를 피하고 싶다면 → O_CREAT | O_EXCL 사용
    - 만약 file이 이미 존재한다면 에러를 반환

Permission mask

- 각각의 file은 3개의 클래스가 연관되어 있음 → a user(or owner), a group, everybody else
- 가능한 permissions or privlieges : read(r), write(w), execute(x)

```
zziglet@DESKTOP-LQTP9AI:~/usp_all/chapter04$ ls -l
total 688
-rw-r--r-- 1 zziglet zziglet   870 Jul 12  2018 README
-rw------- 1 zziglet zziglet   870 Oct 10 01:57 README2
-rw-r--r-- 1 zziglet zziglet  4971 Jul 12  2018 atomic_logger.c
-rw-r--r-- 1 zziglet zziglet   213 Jul 12  2018 atomic_logger.h
-rwxr-xr-x 1 zziglet zziglet 16200 Oct  3 18:14 bufferinout
```

- 가능한 permission들은 각각의 클래스에게 각각 분리되어 있음
- file을 O_CREAT flag로 open → 반드시 세번째 argument(mode_t) 로 permission을 지정해야 함.



- 만약 user → r, group → rw, others → x 라면 100 / 110 / 001 이기에 461 로 변환하여 사용

Historical Numeric representation

Octal digit	Text equivalent	Binary value	Meaning
0	---	000	All types of access are denied
1	--x	001	Execute access is allowed only
2	-w-	010	Write access is allowed only
3	-wx	011	Write and execute access are allowed
4	r--	100	Read access is allowed only
5	r-x	101	Read and execute access are allowed
6	rw-	110	Read and write access are allowed
7	rwx	111	Everything is allowed

Example

644	owner: read and write permissions, group: only read permissions, others: only read permissions.
755	owner: read, write and execute permissions, group: read and execute permissions, others: read and execute permissions.

- POSIX symbolic names
  - POSIX : permission bit마다 상응하는 mask들의 이름을 정의함.
  - 이 이름들은 sys/stat.h에 define

S_IRUSR	read permission bit for owner
S_IWUSR	write permission bit for owner
S_IXUSR	execute permission bit for owner
S_IRWXU	read, write, execute for owner
S_IRGRP	read permission bit for group
S_IWGRP	write permission bit for group
S_IXGRP	execute permission bit for group
S_IRWXG	read, write, execute for group
S_IROTH	read permission bit for others
S_IWOTH	write permission bit for others
S_IXOTH	execute permission bit for others
S_IRWXO	read, write, execute for others
S_ISUID	set user ID on execution
S_ISGID	set group ID on execution

Example

```
int fd;
mode_t fdmode = (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);

if((fd = open("info.dat", O_RDWR | O_CREAT, fdmode)) == -1)
①//"info.dat" 를 현재 directory에 생성
②//이미 존재하면 그 위에 overwritten -> 새로 만들어야 하면 fdmode로 permission
//새로운 파일이면 user에겐 읽기,쓰기 권한 / everybody else에겐 읽기만 허용

  perror("failed to open info.dat");
```

Example - copyfilemain.c

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

#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_EXCL)
//write only, 덮어쓰기 방지
#define WRITE_PERMS (S_IRUSR | S_IWUSR)
```

```
//user -> read, write => 600

int main(int argc, char *argv[]){
  //$ copyfilemain (src file) (tar file) -> 즉 입력 받은 argument가 3개여야 함!
  int bytes;
  int fromfd, tofd;

  if(argc != 3){ //argument가 3개인지 확인
    fprintf(stderr, "Usage: %s from_file_to_file\n", argv[0]);
    return 1;
  }

  if((fromfd = open(argv[1], READ_FLAGS)) == -1){ //argv[1] -> src
    perror("Failed to open input file");
    return 1;
  }

  if((tofd = open(argv[2], WRITE_FLAGS, WRITE_PERMS)) == -1){ //argv[2] -> tar
    perror("Failed to create output file");
    return 1;
  }

  bytes = copyfile(fromfd, tofd); //from -> to로 copy
  printf("%d bytes copied from %s to %s\n", bytes, argv[1], argv[2]);
  return 0; //return이 자동으로 process close
  //만약 close하지 않고 다른 process가 진행되면 메모리 낭비 발생
}
```

Closing

open()을 한 이후에는 main의 return 값에서 open file들과 관련된 모든 자원을 지우는 clean up이 필수적으로 시행되어야 함.

close

```
#include <unistd.h>

int close(int filedes);
//OS kernel에게 open된 file 정보를 다시 system에게 반환해달라고 요청
//kernel은 file의 더 이상 사용하지 않는 file resource들을 지울 수 있음.
```

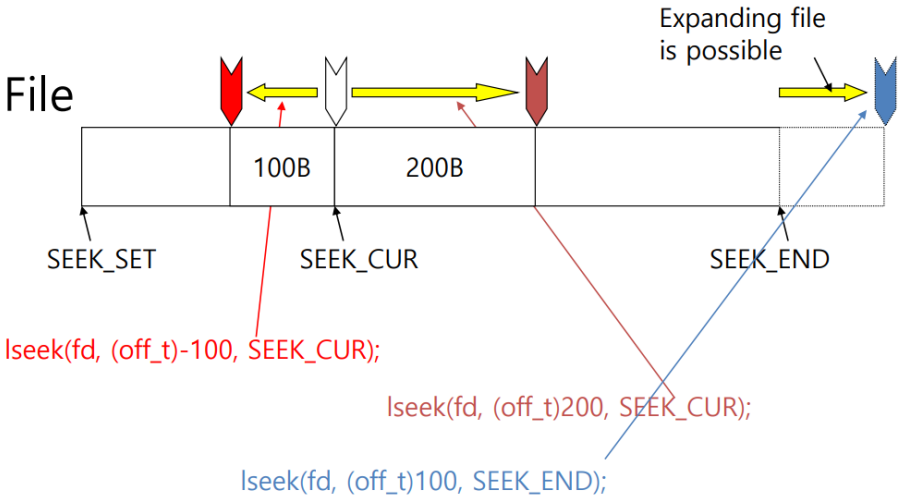
r_close : interrupt 처리

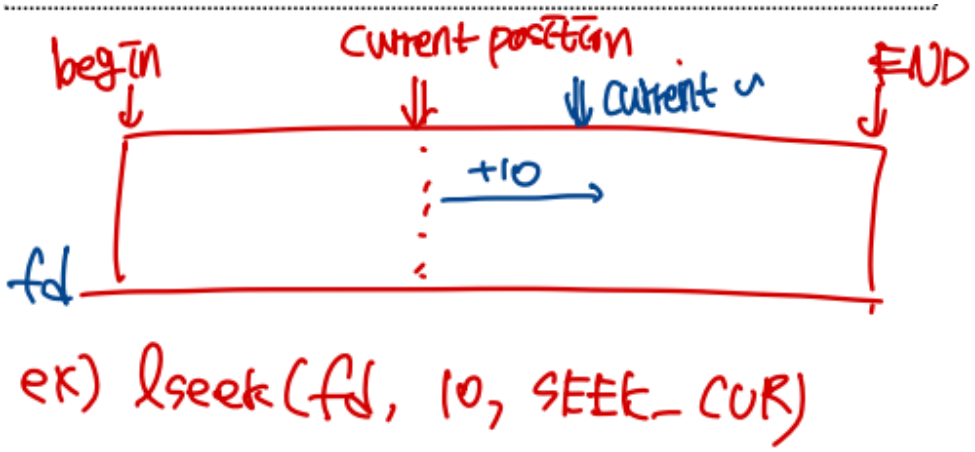
```
#include <errno.h>
#include <unistd.h>
```

```
int r_close(int fd){
  int retval;

  while (retval = close(fd), retval == -1 && errno == EINTR);
  return retval;
}
```

lseek

file offset(읽고자 하는 file 위치)을 임의의 위치로 고정

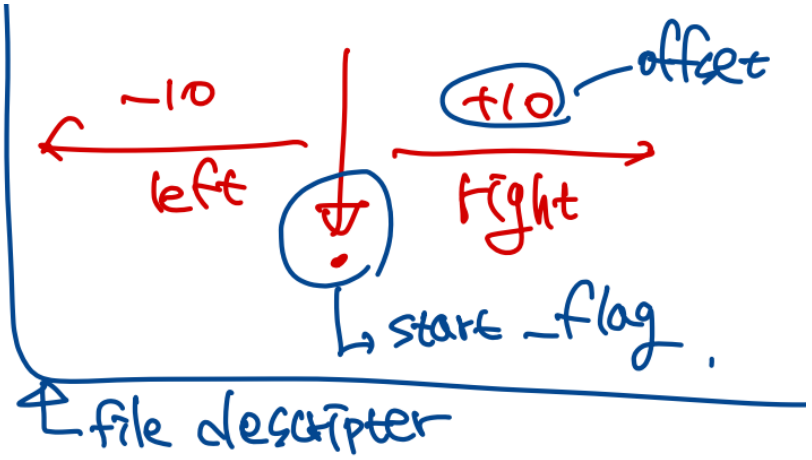





```
#include <sys/types.h>
#include <unistd.h>

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

//filedes : file descripter -> open 될 file 가리킴
```

Parameter

```
//off_t : otffset type(integer number)
//start_flag : 시작지점 (기준지점)
//      - SEEK_SET : file beginning
//      - SEEK_CUR : File offset
//      - SEEK_END : file end
```



- distance
  - negative : left
  - postive : right

File representation

여러 process가 같은 file을 열어서 I/O를 수행하는 경우 → table 동작에 대한 이해도가 높아야 함.

File descriptors vs File pointers

- Files
  - designated within C either by file pointers or by file descriptors
- **File pointers**
  - used by the standard I/O library functions for ISO C

    ex) fopen, fscanf, fprintf, fread, fwrite, fclose(C library) → file pointer return

    / stdin, stdout, stderr (defined in stdio.h)

    → C library ⇒ file pointer return (FILE *fp)
- **File descriptors**

- used by the UNIX I/O functions

  ex) open, read, write, close, ioctl → file descriptors return

  / STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO(defined in unistd.h)

  → 표준 I/O constant 값으로 정의

File descriptors

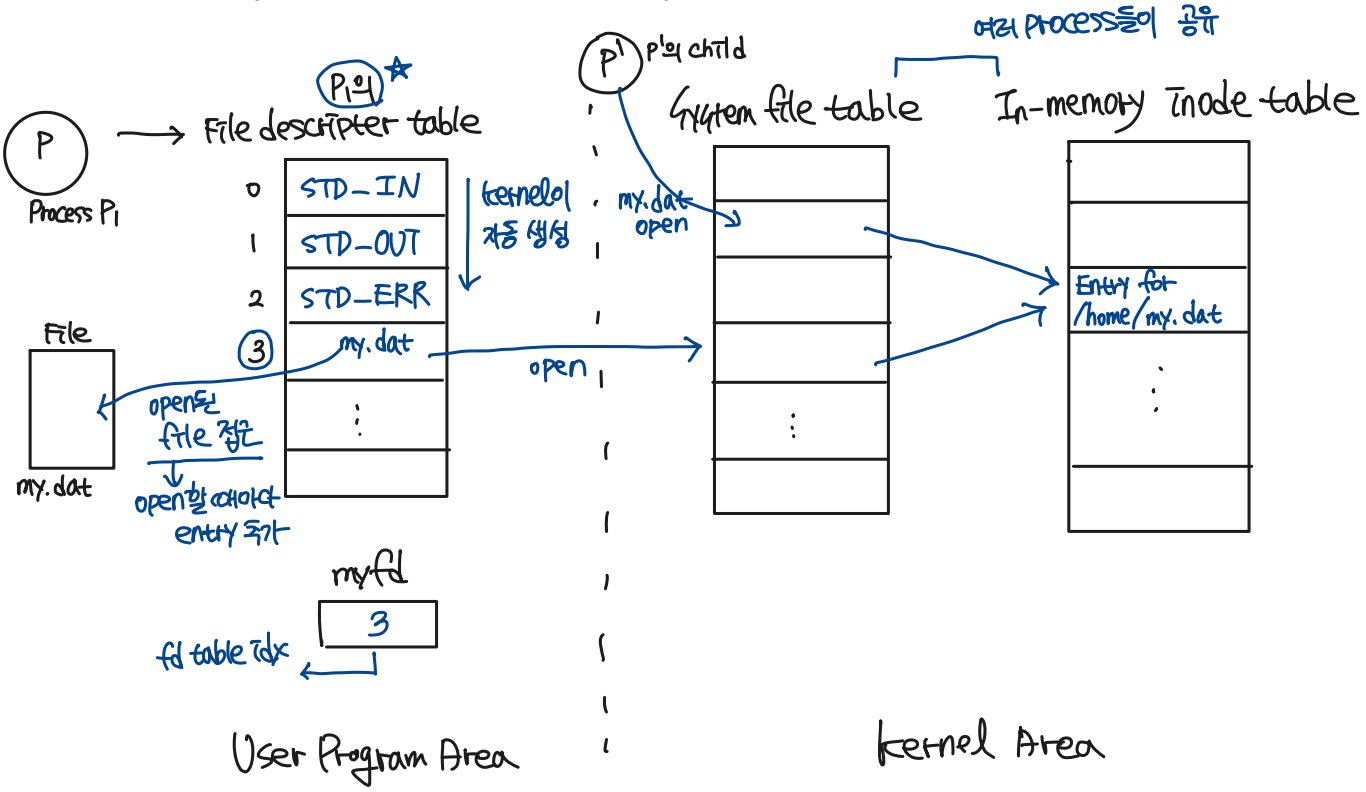
OS kernel이 open된 file을 어떻게 관리하는지? → 특히 severl process

- file descriptors value : 이 process의 file descriptor table의 idx

```
myfd = open("/home/ann/my.dat", O_RDONLY);


//1. myfd에 해당 file contents write를 원함
//2. file descripter : myfd <- file descriptor table index
//3. system file table entry 연결
//4. innode file table entry 연결
//5. innode의 contents of block에 작성
//6. 작성한 만큼 file offset 변경
```

- 아래 3개의 table로 file이 open 되었을 때 open된 file의 정보를 관리하기 위해서 사용

  → file이 open하지 않으면 빈 상태 → entry가 비어 있음.



User

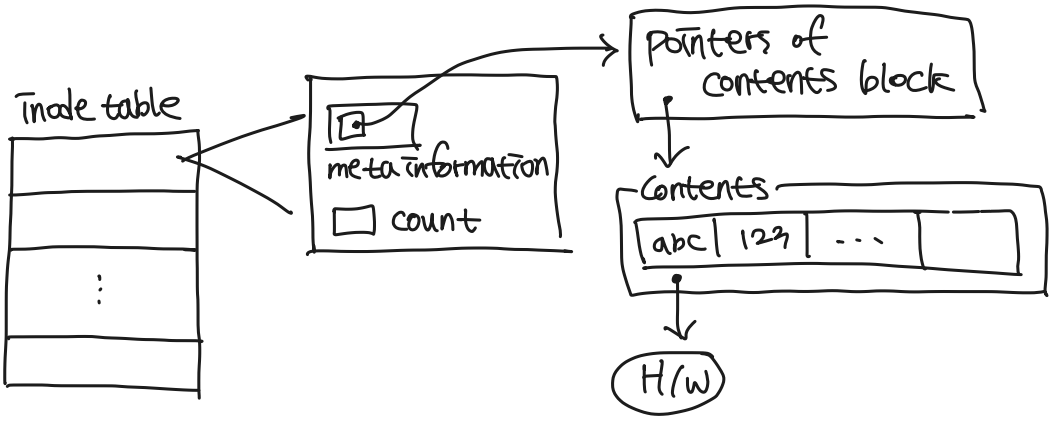
- **file descriptor table** → process마다 가지고 있음.
  - process 생성되면 kernel이 차례대로 알아서 entry 생성(idx 0~2는 standard)
  - idx pointer : 해당 file의 system file table의 entry를 가리키는 pointer

- **myfd** = 3 → open() return value = file descriptor idx = 3

kernel (OS)

- **system file table**
  - open될 때마다 process가 entry 한 개씩 추가
    - open된 file 정보 ≠ 원래 file 정보
  - 같은 entry를 가리키는 경우
    - 다른 process가 각각 open하면 불가능
    - parent가 fork()하여 child를 생성했을 때 → 같은 system file entry
      (fork() : descriptor table이 그대로 복제됨)
  - 저장하는 정보
    1. open된 file의 file offset 저장
    2. access mode 저장
    3. 여러 개의 file descriptor table entry 가리킬 경우 몇 개인지 count도 저장
    4. innode mapping value

- **In-memory innode table**
  - 저장하는 정보
    1. 원래 file에 대한 정보(type, permission, 마지막 접속 시간 등 meta information)
       - pointers of contents block의 pointer들을 가지고 있음



2. 여러 개의 system file table entry가 가리킬 경우 몇 개인지 count도 저장

- open된 file 정보 만들어놓고 contents를 이용해서 I/O 작업 → inode를 직접 이용 x

- **example**

1. what happens when the process executes the close(myfd) function in the previous example? process가 close 호출하면?
   - OS
     1. file descriptor table : deletes the fourth entry
     2. system file table : deletes the corresponding entry or count 1 감소
     3. inode file table : deletes the corresponding entry or count 1 감소 →0되면 사라짐

     ** 무조건 close하였다고 해서 system, innode entry 값이 사라지는 것은 x

2. what happens if two processes open the same file for write?

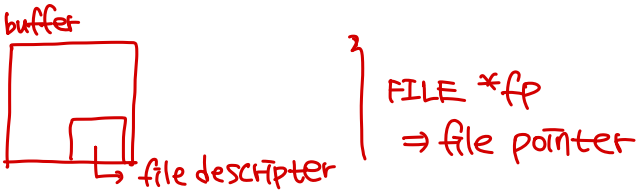
   → 두 개의 process가 같은 file을 open하여 다른 내용을 write
   - 각각의 user가 각 process의 file offset들을 분리하기 때문에 다른 user가 쓴 것 위에 덮어 씀.
   - entry(system file table)가 다르기 때문에 file offset 값이 각자 변함(초기값 : **0**)

3. what would happen if the file offset were stored in the innode table?

   → file offset이 innode table에 저장된다면?

- 두 개의 process가 같은 inode table의 entry 공유
- update된 offset 위치에서 작성
- overwritten이 되지 x

File Pointers



points to a data structure call a FILE structure in the user area of the process

- FILE structure

  contains a buffer and a file descriptor value

  → 무언가 작성하고자 할 때 buf 작성 후 임시로 저장해 놓았다가 추후에 file에 작성

```
FILE *myfp; //file pointer

if((myfp = fopen("/home/ann/my.dat", "w")) == NULL)
  perror("failed to open /home/ann/my.dat); //pointer type == NULL이면 error
else
  fprintf(myfp, "This is a test");
//      file pointer  ,  화면 출력할 printf 함수 parameter


//process 종료 시
//1. buffer 안에 있는 내용 삭제
//2. file에 작성됨
```

- **fprintf()**

  file에 printf 하겠다는 뜻 -> file 종류에 따라 다름

  1. Disk files

     : usually fully buffered instead of actually writing to a disk.

     - When the buffer fills, the I/O subsystem calls write with the file dexcriptor.
     - to avoid the buffering delay, an fflush → 강제로 buf 내용 비우고 실제 file에 작성

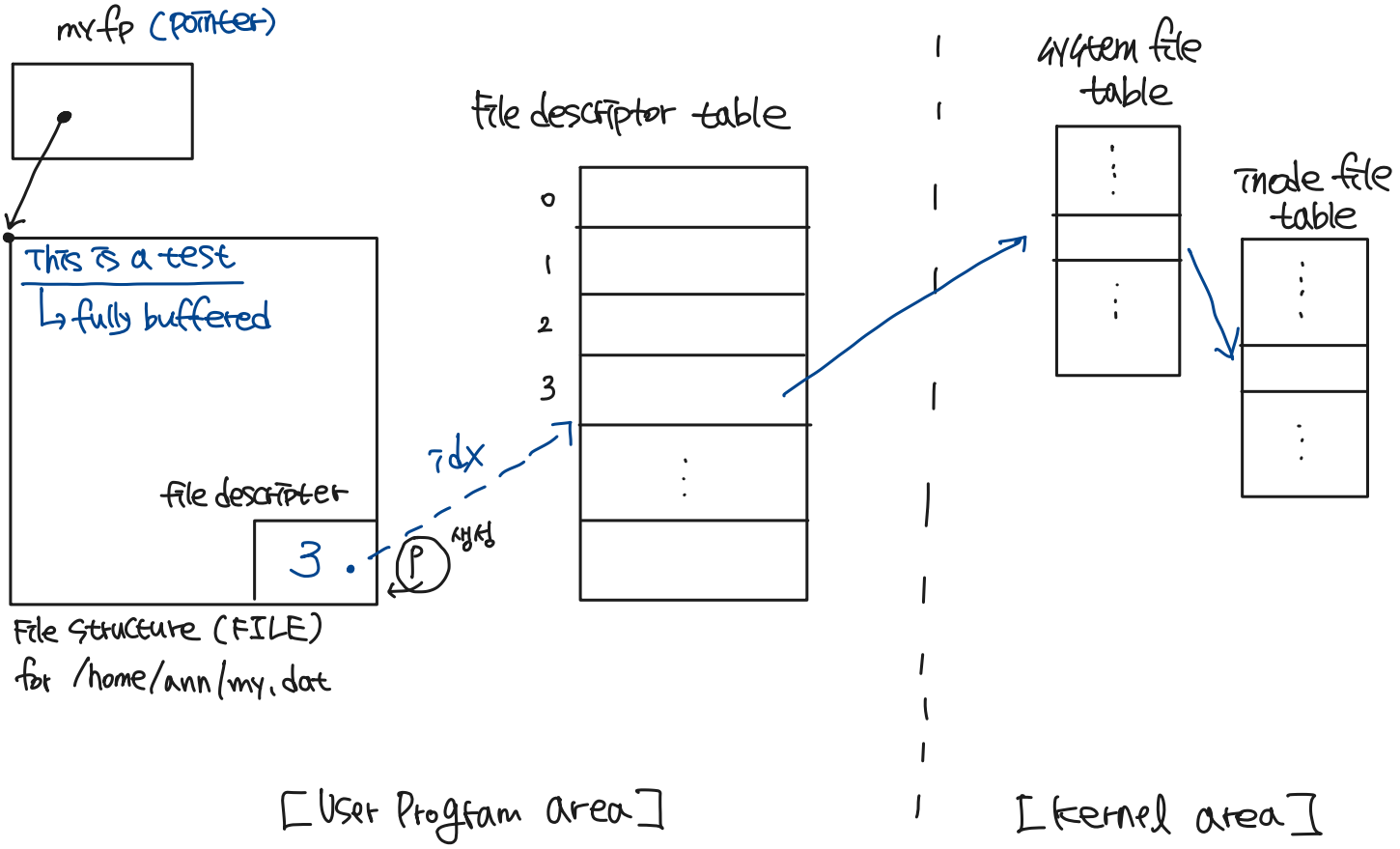
  2. Terminal files (fp parameter가 stdout인 file)

: line buffered (buffered 되지 않은 standard error를 제외)

- stadard error : 보통의 출력 data보다 우선순위가 늦으면 출력됨

  → but. terminal files는 바로 출력되어야 함!

- new line char가 인식되면 print 됨 file로!



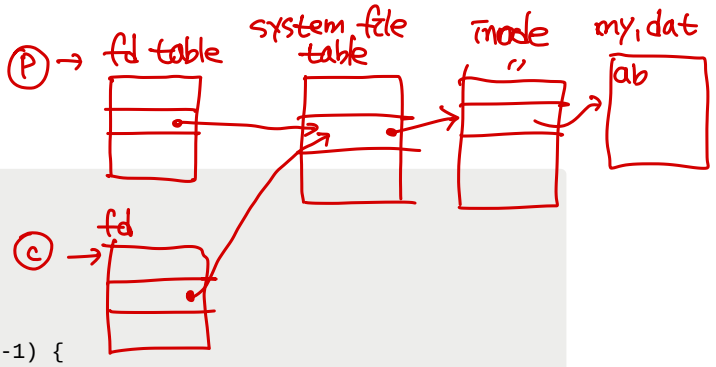
fopen 성공적으로 완료

→ 각 table의 entry 증가 (inode는 존재한다면 생성)

Inheritance of file descriptors

1. open → fork : open 같이

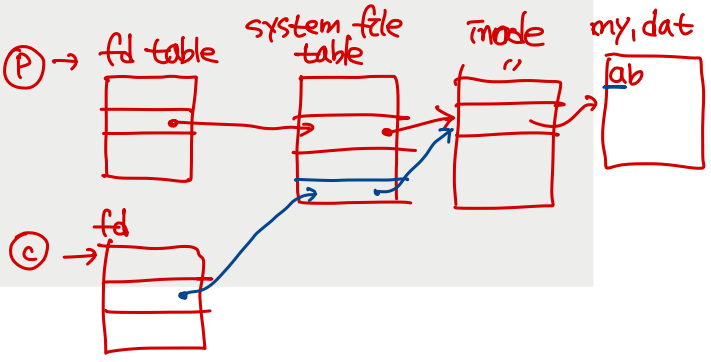
   - two processes share the file offset




```
int main(void) {
  char c = '!';
  int myfd;

  if ((myfd = open("my.dat", O_RDONLY)) == -1) {
    perror("Failed to open file");
    return 1;
  }

  if (fork() == -1) {
    perror("Failed to fork");
    return 1;
  }

  read(myfd, &c, 1);
  printf("Process %ld got %c\n", (long)getpid(), c);
  return 0;
}

[output]
process nnn got a
process mmm got b
```

2. fork → open

   - two processes use different the file offset




```
int main(void) {
  char c = '!';
  int myfd;

  if (fork() == -1) {
    perror("Failed to fork");
    return 1;
```

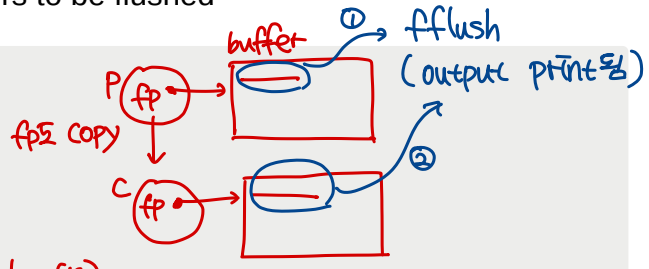
```
  }

  if ((myfd = open("my.dat", O_RDONLY)) == -1) {
    perror("Failed to open file");
    return 1;
  }

  read(myfd, &c, 1);
  printf("Process %ld got %c\n", (long)getpid(), c);
  return 0;
}

[output]
process nnn got a
process mmm got a
```

* 별도의 system file table entry
→ file offset이 0으로 초기화
But. inode는 같은 entry
(file이 같기 때문)
⇒ 같은 값을 read하고 print

Line buffering example

1. regular → fully buffered : buffering ⟶ fork( )
   - Buffering before fork
   - return from main causes the buffers to be flushed

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

int main(void){
  printf("This is my output");
  fork();
  return 0;
}

[output]
This is my output
This is my output
```

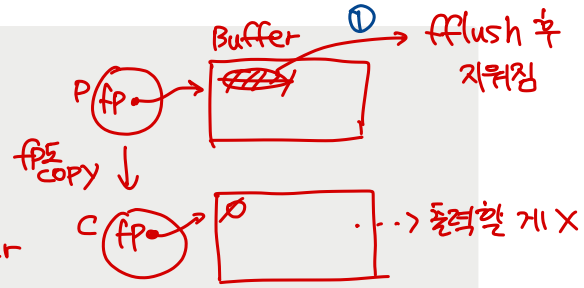


↳ new line char(x)
⇒ regular file : fully buffered

2. terminal → line 단위 : buffering ⟶ fflush ⟶ fork( )
   - Buffering is flushed before fork

     → fork 되기 전에 buffer가 flush 되어 비워진 채로 fork

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

int main(void){
  printf("This is my output\n");
  fork();
  return 0;
}

[output]
This is my output
```



Filters and redirection

Filters

read from standard input, perfom a transformation, and ouput the result to standard output

ex) head : file의 내용을 읽어서 출력 (default : 10) 10줄만 읽음.

tail : head와 동일하지만 끝에서부터 출력

more : 한 화면에서 볼 수 있는 만큼만 출력 후 I/O 작업으로 next page 이동

sort : 정렬 명령어(transformation)

grep : 특정 keyword를 가지고 있는 것만 filtering

…etc.

- all of parameters are communicated as command-line arguments.
- “cat” without input file behaves like a filter
  - take its input from standard input
  - writes its results to standard output

↔ “0” transfromatin, 하나도 fileter하지 않는 filtering

Redirection

방향을 바꾸는 명령어

a program modifies the file descriptor table entry

so that it points to a different entry in the system file table.

> : standard output

- 출력 방향이 바뀜 → 왼쪽 명령어의 결과가 출력 되는 방향이 변경됨

ex) ls > temp.txt : 출력하지 않고 temp.txt의 내용이 ls의 결과로 변경됨.

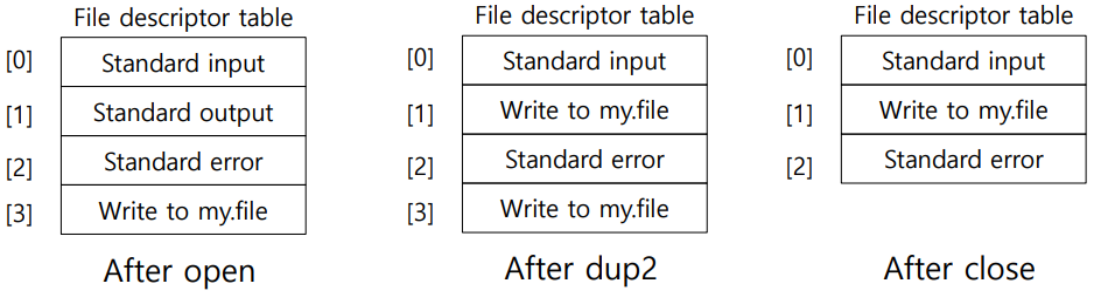
< : standard input

- file에 있는 내용 sorting할 때 사용

EX P → ls > temp.txt
① 기존 ls는 fd table의 STD_OUT으로 출력
② redirection(>) → temp.txt로 변경
③ fd table STD_OUT entry가 temp txt로 변경

- redirection in C ⇒ system call 사용해서 구현 가능 → open된 file에 OS 이용해서 조작

```
#include <unistd.h>

int dup2(int fildes, int fildes2);
//fildes : src file 의 file descriptor
//fildes2 : target file의 file descriptor => filedes 복사하여 entry 내용도 변경

//dup, dup2, dup3 존재 (parameter의 개수에 따라 다름)
//dup( ) : fdt에서 가장 낮은 idx로 변경
//dup( , , flags)
```



```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include "restart.h"
#define CREATE_FLAGS (O_WRONLY | O_CREAT | O_APPEND)
#define CREATE_MODE (S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)

int main(void) {
    int fd;

    fd = open("my.file", CREATE_FLAGS, CREATE_MODE);
    if (fd == -1) {
        perror("Failed to open my.file");
        return 1;
    }
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror("Failed to redirect standard output");
        return 1;
    }
    if (r_close(fd) == -1) {
        perror("Failed to close the file");
        return 1;
    }
    if (write(STDOUT_FILENO, "OK", 2) == -1) {
        perror("Failed in writing to file");
        return 1;
    }
    return 0;
}
```