LINUX PROGRAMMING

FILE I/O
(2 week)

FILE I/O

FILE DESCRIPTORS

- To the kernel, all open files are referred to by **descriptor**.

- A file descriptor is a **non-negative integer**.

- When we open an existing file or create a new file, the kernel returns a file descriptor to the process.

- When we want to read or write a file, we identify the file with the file descriptor that was returned by **open()** or **creat()**

DESCRIPTORS

- By convention, UNIX System shells associate file descriptor 0 with the **standard input** of a process.
- File descriptor 1 with the **standard output.**
- File descriptor 2 with the **standard error.**

OPEN

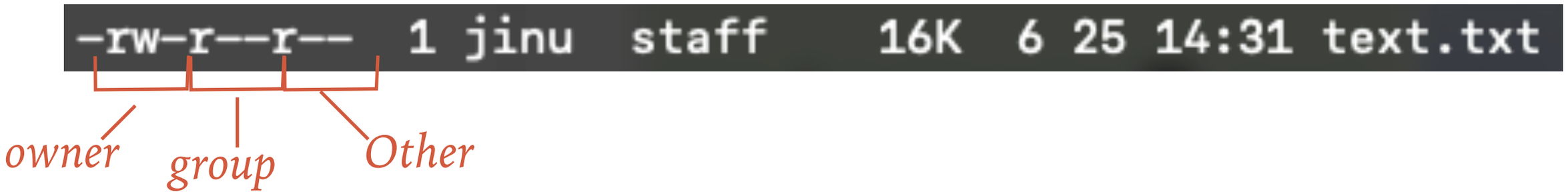
```
#include <fcntl.h>

int open(const char *path, int oflag, ... /* mode_t mode */ );

int openat(int fd, const char *path, int oflag, ... /* mode_t mode */ );
```

Both return: file descriptor if OK, –1 on error

Oflag

- O_RDONLY : Open for reading only.
- O_WRONLY : Open for writing only.
- O_RDWR : Open for reading and writing.
- O_EXEC : Open for execute only.
- O_APPEND : Append to the end of file on each write.
- O_CREAT : Create the file if it doesn’t exist.
- O_TRUNC : If the file exists and if it successfully opened for either write-only or read-write, truncate its length to 0.

ACCESS PERMISSION



- R : Read permission
- W : Write permission
- X : Write permission
- - : not permission

CREAT

```
#include <fcntl.h>

int creat(const char *path, mode_t mode);

                    Returns: file descriptor opened for write-only if OK, –1 on error
```

- A new file can also be created by calling the creat function
- Note that this function is equivalent to

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

CLOSE

```
#include <unistd.h>
int close(int fd);
```

Returns: 0 if OK, –1 on error

- When a process terminates, all of its open files are closed automatically by the kernel.
- Many programs take advantage of this fact and don’t explicitly close open files.

LSEEK

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
                                    Returns: new file offset if OK, –1 on error
```

- If whence is **SEEK_SET**, the file's offset is set to offset bytes from the beginning of the file.
- If whence is **SEEK_CUR**, the file's offset is set to its current value plus the offset. The offset can be positive or negative.
- If whence is **SEEK_END**, the file's offset is set to the size of the file plus the offset. The offset can be positive or negative.

LSEEK

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

void err_sys(const char* x)
{
    perror(x);
    exit(1);
}
char buf1[] = "abcdefghij";
char buf2[] = "ABCDEFGHIJ";

int main(void)
{
    int fd;

    if((fd = creat("text.txt",644))<0)
        err_sys("creat error");

    if(write(fd,buf1,10) != 10)
        err_sys("buf1 write error");
    /* offset now = 10 */

    if(lseek(fd, 16384, SEEK_SET) == -1)
        err_sys("lseek error");

    if(write(fd, buf2, 10) != 10)
        err_sys("buf2 write error");
    /* offset now = 16394 */

    return 0;
}
```

```
jinu@jeongjin-uui-MacBook-Pro [15:24:00] ~/Desktop/git/linuxprogramming/pa2
$ ls -l text.txt
-rw-r--r--  1 jinu  staff    16K  6 25 14:31 text.txt

jinu@jeongjin-uui-MacBook-Pro [15:24:06] ~/Desktop/git/linuxprogramming/pa2
$ od -c text.txt
0000000    a   b   c   d   e   f   g   h   i   j  \0  \0  \0  \0  \0  \0
0000020   \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*
0040000    A   B   C   D   E   F   G   H   I   J
0040012
```

- The **od** utility is a filter which displays the specified files, or standard input if no files are specified, in a user specified format.

READ

```
#include <unistd.h>

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

                                Returns: number of bytes read, 0 if end of file, –1 on error
```

- Data is read from an open file with the **read** function

```
#include <unistd.h>

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

                                Returns: number of bytes written if OK, –1 on error
```

- Data is written to an open file with the **write** function

FILE SHARING

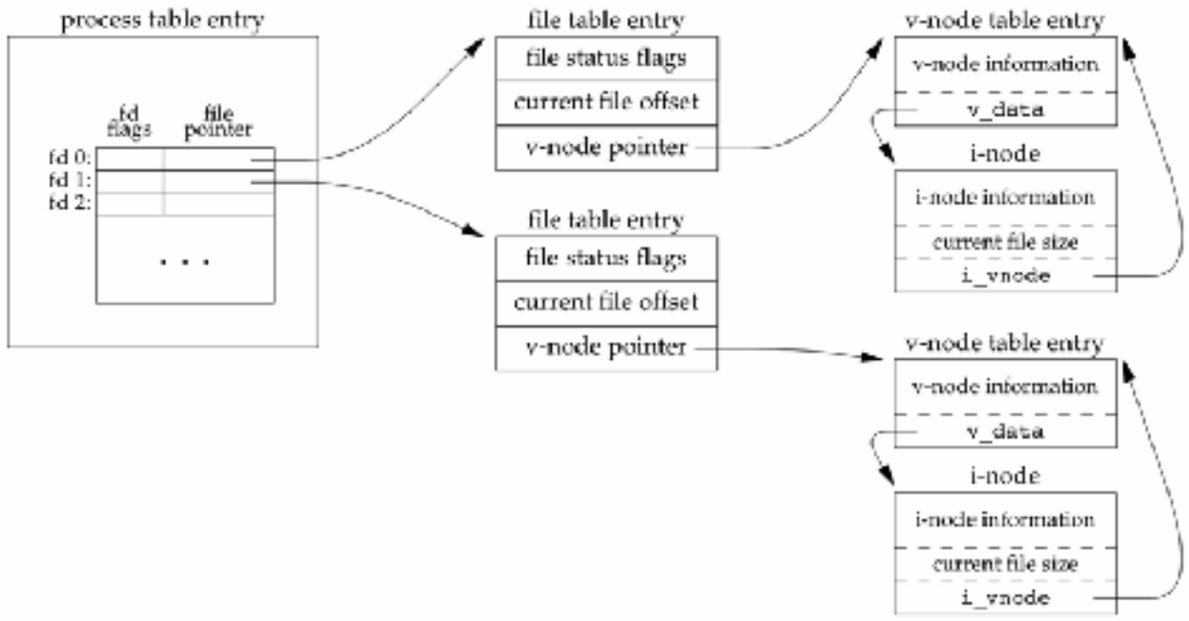


Figure 3.7 Kernel data structures for open files

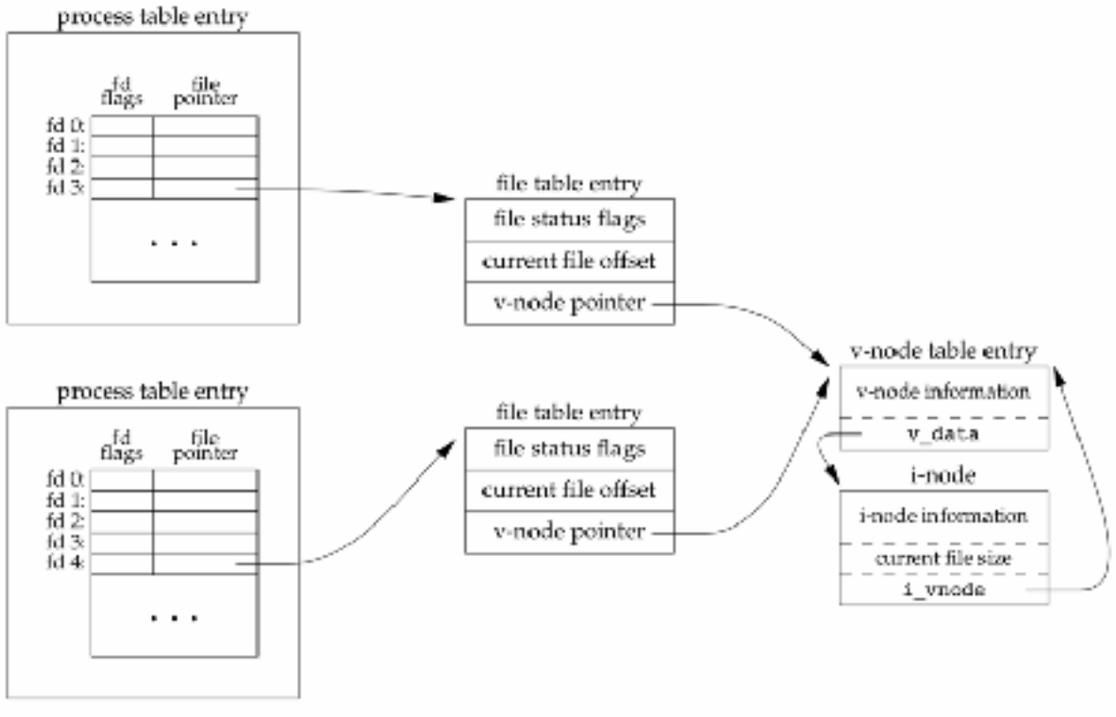


Figure 3.8 Two independent processes with the same file open

The UNIX system supports the sharing of open files among different processes.

Mechanism

- Every process has an entry in the process table.
- The kernel maintains a file table for all open files.
- Each open file has a v-node structure that contains information about the type of file and pointers to function that operate on the file.

DUP & DUP2

```
#include <unistd.h>

int dup(int fd);

int dup2(int fd, int fd2);
```

Both return: new file descriptor if OK, –1 on error

- The new file descriptor returned by dup is guaranteed to be lowest-numbered available file descriptor.
- With dup2, we specify the value of the new descriptor with the fd2 argument. If fd2 is already open, it is first closed.

DUP & DUP2

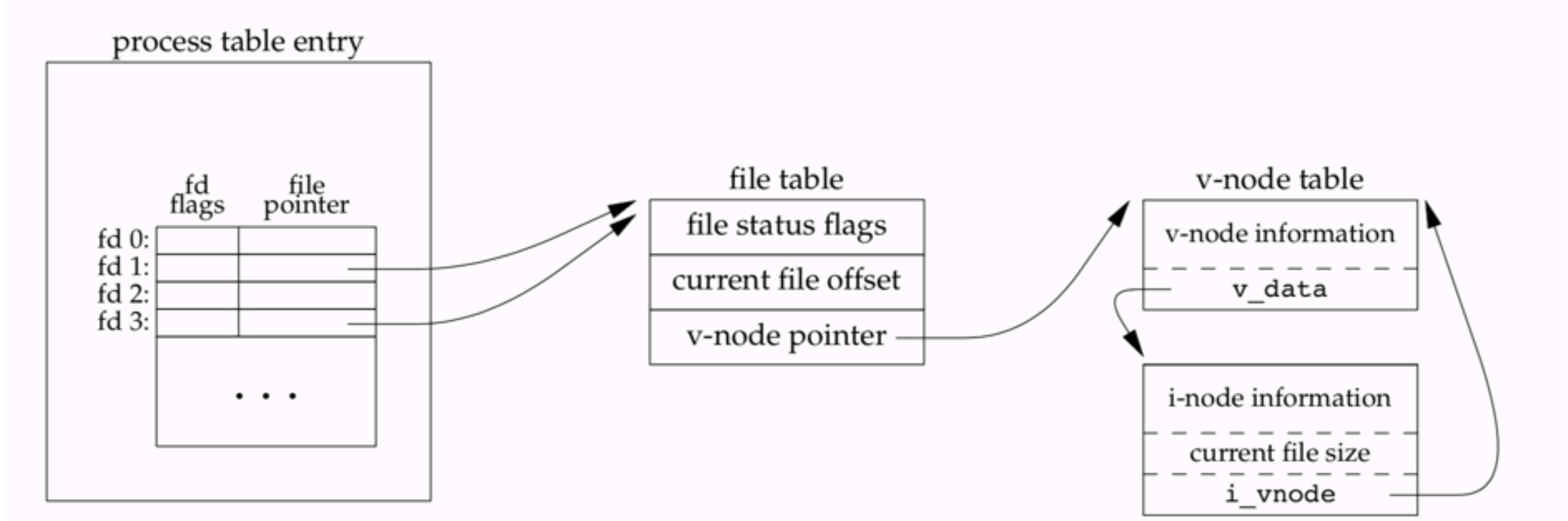


Figure 3.9 Kernel data structures after `dup(1)`

- What if we specify the value of the STDIN or STDOUT with dup2?

DUP2

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

int main(){
    int fd = open("text.txt",O_RDWR|O_TRUNC);

    char buff[] = "hello";

    dup2(fd,1);
    printf("%s",buff);

    dup2(fd,0);
    scanf("%s",buff);

    write(fd,buff,sizeof(buff));

}
```

```
jinu@jeongjin-uui-MacBook-Pro [16:11:12] ~/Desktop/git/linuxprogramming/pa2
$ gcc -o dup dup.c

jinu@jeongjin-uui-MacBook-Pro [16:12:05] ~/Desktop/git/linuxprogramming/pa2
$ ./dup

jinu@jeongjin-uui-MacBook-Pro [16:12:08] ~/Desktop/git/linuxprogramming/pa2
$ cat text.txt
hellohello
```

FILES &DIRECTORY

STAT

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

int stat(const char *restrict pathname, struct stat *restrict buf);

int fstat(int fd, struct stat *buf);

int lstat(const char *restrict pathname, struct stat *restrict buf);

int fstatat(int fd, const char *restrict pathname,
            struct stat *restrict buf, int flag);
```

All four return: 0 if OK, –1 on error

- Given a pathname, the **stat** function returns a structure of information about the named file.
- The **fstat** function obtains information about the file that is already open on the descriptor fd.
- The **lstat** function returns information about the symbolic link, not the file referenced by the symbolic link.

STAT STRUCTURE

```
struct stat {
  mode_t         st_mode;    /* file type & mode (permissions) */
  ino_t          st_ino;     /* i-node number (serial number) */
  dev_t          st_dev;     /* device number (file system) */
  dev_t          st_rdev;    /* device number for special files */
  nlink_t        st_nlink;   /* number of links */
  uid_t          st_uid;     /* user ID of owner */
  gid_t          st_gid;     /* group ID of owner */
  off_t          st_size;    /* size in bytes, for regular files */
  struct timespec st_atim;    /* time of last access */
  struct timespec st_mtim;    /* time of last modification */
  struct timespec st_ctim;    /* time of last file status change */
  blksize_t      st_blksize; /* best I/O block size */
  blkcnt_t       st_blocks;  /* number of disk blocks allocated */
};
```

ACCESS

```
#include <unistd.h>
int access(const char *pathname, int mode);
int faccessat(int fd, const char *pathname, int mode, int flag);
                                        Both return: 0 if OK, –1 on error
```

The access() system call checks the accessibility of the file named by the path argument for the access permissions indicated by the mode argument.

Mode

- R_OK : test for read permission
- W_OK : test for write permission
- X_OK : test for execute permission
- F_OK : the existence test.

ACCESS

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

void err_sys(char *x){
    perror(x);
    exit(1);

}

int main(){

    if(access("text.txt",R_OK)==-1)
        err_sys("READ permission deny");
    else printf("READ OK\n");

    if(access("text.txt",W_OK)==-1)
        err_sys("WRITE permission deny");
    else printf("WRITE OK\n");

    if(access("text.txt",X_OK)==-1)
        err_sys("EXECUTE permission deny");
    else printf("EXECUTE OK\n");

    return 0;
}
```

```
jinu@jeongjin-uui-MacBook-Pro [16:43:05] ~/Desktop/git/linuxprogramming/pa2
$ gcc -o access access.c

jinu@jeongjin-uui-MacBook-Pro [16:43:13] ~/Desktop/git/linuxprogramming/pa2
$ ./access
READ OK
WRITE OK
EXECUTE permission deny: Permission denied

jinu@jeongjin-uui-MacBook-Pro [16:43:17] ~/Desktop/git/linuxprogramming/pa2
$ ls -l
total 88
-rwxr-xr-x  1 jinu  staff   8.4K  6 25 16:42 a.out*
-rwxr-xr-x  1 jinu  staff   8.4K  6 25 16:43 access*
-rw-r--r--  1 jinu  staff   429B  6 25 16:43 access.c
-rw-r--r--  1 jinu  staff   165B  6 25 16:17 dup.c
-rw-r--r--  1 jinu  staff   528B  6 25 14:31 lseek.c
-rw-r--r--  1 jinu  staff   370B  6 25 15:55 read.c
-rw-r--r--  1 jinu  staff    11B  6 25 16:13 text.txt
```

CHMOD

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
int fchmodat(int fd, const char *pathname, mode_t mode, int flag);
                                    All three return: 0 if OK, –1 on error
```

- The **chmod**, **fchmod**, and **fchmodat** functions allow us to change the file access permission for an existing file.

CHOWN

```
#include <unistd.h>

int chown(const char *pathname, uid_t owner, gid_t group);

int fchown(int fd, uid_t owner, gid_t group);

int fchownat(int fd, const char *pathname, uid_t owner, gid_t group,
             int flag);

int lchown(const char *pathname, uid_t owner, gid_t group);
```

All four return: 0 if OK, –1 on error

- The **chown** function allow us the change a file's user ID and group ID, but if either of the arguments owner or group is -1, the corresponding ID is left unchanged.

REMOVE & RENAME & MKDIR & RMDIR

```
#include <stdio.h>

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

Returns: 0 if OK, –1 on error

```
#include <stdio.h>

int rename(const char *oldname, const char *newname);

int renameat(int oldfd, const char *oldname, int newfd,
             const char *newname);
```

Both return: 0 if OK, –1 on error

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

int mkdir(const char *pathname, mode_t mode);

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

Both return: 0 if OK, –1 on error

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

Returns: 0 if OK, –1 on error

CHANGE DIRECTORY

```
#include <unistd.h>

int chdir(const char *pathname);

int fchdir(int fd);
```

Both return: 0 if OK, –1 on error

- Every process has a current working directory. This directory is where the search for all relative pathnames starts
- We can change the current working directory of the calling process by calling the **chdir** of **fchdir** function
- Get current working directory by **getcwd** function.

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

Returns: *buf* if OK, NULL on error

CHANGE DIRECTORY

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

void err_sys(char *x){
    perror(x);
    exit(1);
}
int main(){
    char buff[256];

    if(chdir("..")==-1){
        err_sys("chdir error");
    }

    getcwd(buff,sizeof(buff));

    puts(buff);

    return 0;
}
```

```
jinu@jeongjin-uui-MacBook-Pro [17:14:07] ~/Desktop/git/linuxprogramming/pa2
$ gcc -o chdir chdir.c

jinu@jeongjin-uui-MacBook-Pro [17:14:17] ~/Desktop/git/linuxprogramming/pa2
$ ./chdir
/Users/jinu/Desktop/git/linuxprogramming
```

STANDARD I/O LIBRARY

INTRODUCTION

- In this chapter, we describe the standard I/O library.
- This library is specified by the ISO C standard because it has been implemented on many operating systems other than the UNIX system.
- Additional interfaces are defined as extensions to the ISO C standard by the Single UNIX Specification.
- The standard I/O library handles such details as **buffer allocation** and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size.

STREAMS

- All the I/O routines centered on file descriptors.
- When a file is opened, a file descriptor is returned, and that descriptor is then used for all subsequent I/O operation.
- With the standard I/O library, the discussion centers on **streams**.
- When we open or create a file with the standard I/O library, we say that we have associated a stream with the file.

STREAMS

Three streams are predefined and automatically available to a process :

- Standard Input : **stdin**
- Standard Output : **stdout**
- Standard Error : **stderr**

BUFFERING

- The goal of the buffering provided by the standard I/O library is to use the minimum number of **read** and **write** calls.

Three types of buffering are provided:

- **Fully buffered : with file**

Actual I/O takes place when the standard I/O buffer is filled

- **Line buffered : with stdin, stdout**

The standard I/O library performs I/O when a newline character is encountered on input or output.

- **Unbuffered : with stderr**

The standard I/O library does not buffer the characters

BUFFERING

```c
#include<stdio.h>

int main(){
    int i;
    for(i=0;i<5;i++){
        printf("%d",i);
        sleep(1);
    }

    return 0;
}
```

➤ How does it work?

```c
#include<stdio.h>

int main(){
    int i;
    for(i=0;i<5;i++){
        printf("%d",i);
        fflush(stdout);
        sleep(1);
    }

    return 0;
}
```

➤ We can call the function **flush** to flush a stream.

OPENING A STREAM

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict type);

FILE *freopen(const char *restrict pathname, const char *restrict type,
              FILE *restrict fp);

FILE *fdopen(int fd, const char *type);
```

All three return: file pointer if OK, NULL on error

```
#include <stdio.h>
int fclose(FILE *fp);
```

Returns: 0 if OK, EOF on error

Any buffered output data is flushed before the file is closed. Any input data that may be buffered is discarded.

LINE-AT-A-TIME I/O

```
#include <stdio.h>

char *fgets(char *restrict buf, int n, FILE *restrict fp);

char *gets(char *buf);
```

Both return: `buf` if OK, `NULL` on end of file or error

```
#include <stdio.h>

int fputs(const char *restrict str, FILE *restrict fp);

int puts(const char *str);
```

Both return: non-negative value if OK, `EOF` on error

BINARY I/O

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nobj,
             FILE *restrict fp);

size_t fwrite(const void *restrict ptr, size_t size, size_t nobj,
              FILE *restrict fp);
```

Both return: number of objects read or written

- If we're doing binary I/O, we often would like to read or write an entire structure at a time.
- To do this using **getc** or **puts**, we have to loop through the entire structure, one byte at time, reading or writing each byte.

BINARY I/O

1. Read or write a binary array, For example, to write elements 2 through 5 of a floating-point array, we could write

```
float   data[10];

if (fwrite(&data[2], sizeof(float), 4, fp) != 4)
    err_sys("fwrite error");
```

2. Read or write a structure. For example, we could write

```
struct {
  short  count;
  long   total;
  char   name[NAMESIZE];
} item;

if (fwrite(&item, sizeof(item), 1, fp) != 1)
    err_sys("fwrite error");
```

BINARY I/O

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>

struct User{
    char name[10];
    short age;
};

int main(){
    FILE *fp;

    fp = fopen("userinfo","rw");

    struct User user = {"jinwoo",10},other;

    fwrite(&user,sizeof(user),1,fp);

    fread(&other,sizeof(user),1,fp);

    printf("%d %s\n",other.age,other.name);
    return 0;
}
```

```
jinu@jeongjin-uui-MacBook-Pro [21:02:11] ~/Desktop/git/linuxprogramming/pa2
$ gcc -o fwrite fwrite.c

jinu@jeongjin-uui-MacBook-Pro [21:02:38] ~/Desktop/git/linuxprogramming/pa2
$ ./fwrite
10 jinwoo
```

FORMATTED I/O

```
#include <stdio.h>

int printf(const char *restrict format, ...);

int fprintf(FILE *restrict fp, const char *restrict format, ...);

int dprintf(int fd, const char *restrict format, ...);
```

All three return: number of characters output if OK, negative value if output error

```
int sprintf(char *restrict buf, const char *restrict format, ...);
```

Returns: number of characters stored in array if OK, negative value if encoding error

```
int snprintf(char *restrict buf, size_t n,
             const char *restrict format, ...);
```

Returns: number of characters that would have been stored in array if buffer was large enough, negative value if encoding error

```
#include <stdio.h>

int scanf(const char *restrict format, ...);

int fscanf(FILE *restrict fp, const char *restrict format, ...);

int sscanf(const char *restrict buf, const char *restrict format, ...);
```

All three return: number of input items assigned, EOF if input error or end of file before any conversion

FILENO

```
#include <stdio.h>

int fileno(FILE *fp);
```

Returns: the file descriptor associated with the stream

- Each standard I/O stream has an associated file descriptor, and we can obtain the descriptor for a stream by calling **fileno**.

PRACTICE

- “wiki” 라는 파일의 내용 중 첫 번째 단어, 중간 단어, 마지막 단어를 출력해라.
- 중간 단어는 파일 사이즈가 2000 byte 라고 하면 1000 byte 이후의 단어를 출력하면된다.
- 만약 1000 byte 가 after 라는 단어의 t를 가르키고 있다면, 998byte 부터 읽어서 after을 출력한다.
- Hint, fstat() ㅎㅎ