System Programming

3. File IO (2): System Call

Yunseok Rhee

rheeys@hufs.ac.kr

Division of Computer Engineering

Linux File System (1)

- Each file in a file system has its own ***inode***
- An inode is a data structure having all information on a file.
- inodes of all files reside in a ***disk***
- inode contents (C struct)
  - file name
  - file type (regular, directory,...)
  - file owner id
  - access permission
    rwxr-xr-x (for owner, group, others)
  - creation/modified time
  - file size
  - file data block addr. table (see the next page!)
  - ...

Linux File System (2)

- File types
  - regular file
  - directory file
  - FIFO file (pipe)
  - special files (IO devices)
  - symbolic link files

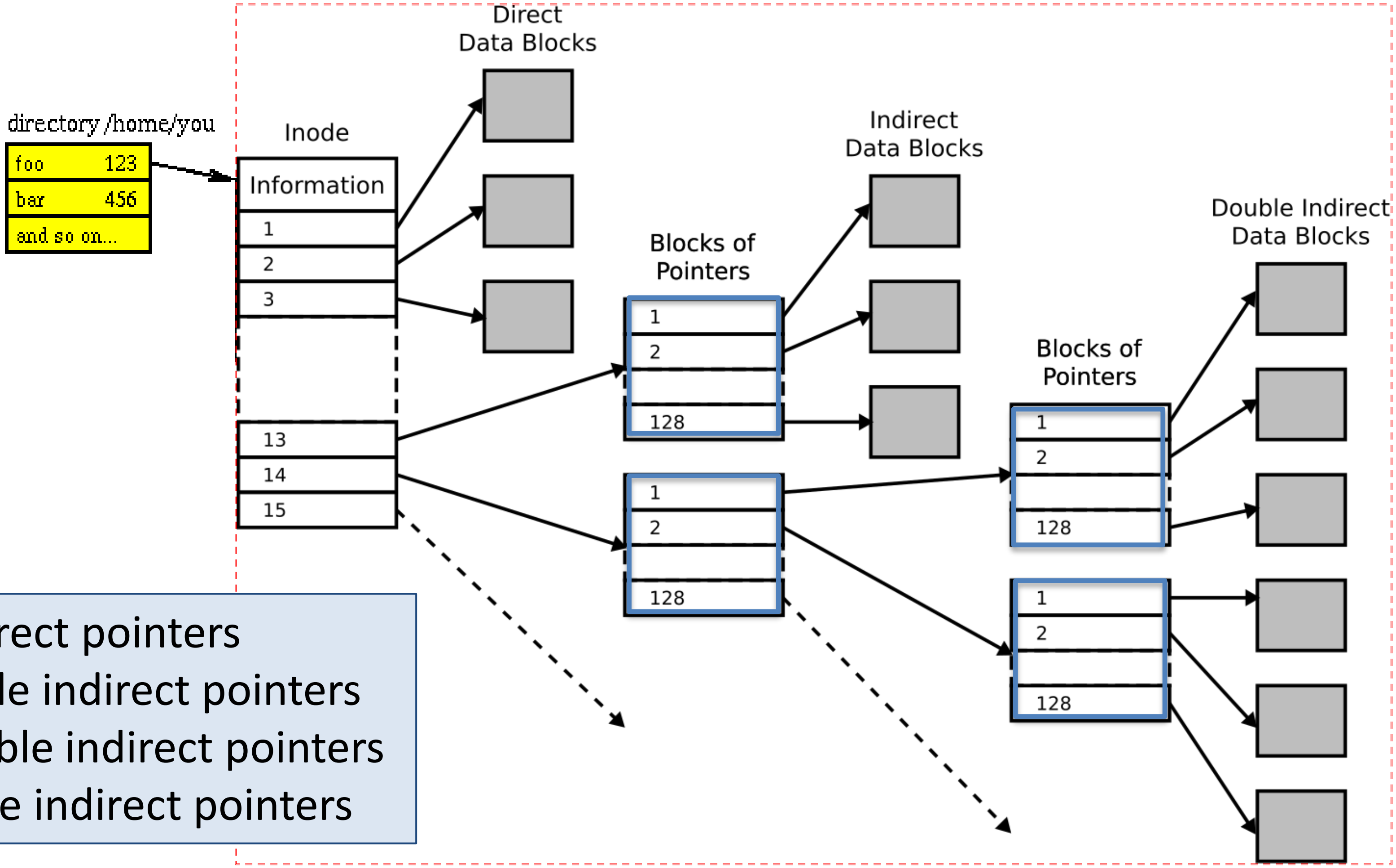
A “john” directory file

i	. (john)
j	.. (parent)
k	File name A
l	File name B
m	File name C

- Directory file
  - A directory is just a file whose content is the list of (inode #, file name) in the same directory.
  - inode is a data structure which contains all the information about the file and file data blocks
  - inode # is a unique file id number in the file system
  - “`ls -al john`” is a shell command that just displays the “john” directory file

Inode structure example

- block size: 512 byte, block pointer: 4 bytes



1-12: direct pointers
13: single indirect pointers
14: double indirect pointers
15: triple indirect pointers

Linux File Types

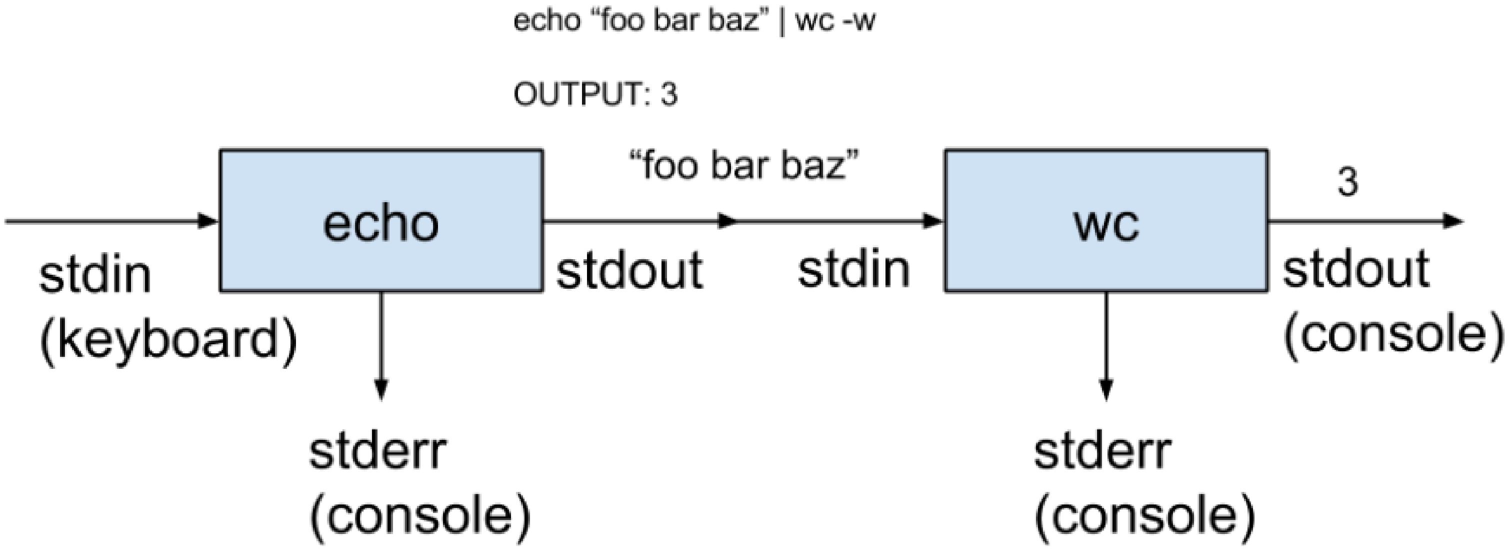
- Ordinary File (Regular File)
  - Text, binary files
- Directory File
  - A file that includes the set of (*file-name, inode #*) of the directory.
- Character Special File
  - Character-oriented device (e.g. Keyboard)
- Block Special File
  - Block-oriented device (e.g. HDD file systems, eth0)
- FIFO file
  - Named *pipe / Unnamed pipe*

  cf. pipe in a process is usually unnamed.
- Symbolic link file
  - a file which points to another file

  cf. *hardlink* is NOT a file.

File Descriptor (1)

- A *file descriptor* (or *file handle*) is a small, non-negative integer which identifies a file to the kernel.
  - Traditionally, `stdin`, `stdout` and `stderr` are 0, 1 and 2 respectively.



```
echo “foo bar baz” | wc -w

OUTPUT: 3
```

  - Relying on “magic numbers” is BAD.
    - Use `STDIN_FILENO`, `STDOUT_FILENO` and `STDERR_FILENO` defined in or stdin, stdout, and stderr defined in .

File Descriptor (2)

- Maximum number of files
  - a process can open 1024 files
  - we can check the system resource configuration

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 194273
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
…….
```

Basic File I/Os

- 5 fundamental Unix/Linux file I/Os
  - `open(2)`
  - `close(2)`
  - `lseek(2)`
  - `read(2)`
  - `write(2)`

File open (1)

```
#include <fcntl.h>

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

- parameters
  - *path*: name of the file to open or create
  - *oflag*: file open options
  - *mode*: access permission (at file creation)
- return
  - *file descriptor* if OK
  - -1 on error

File open (2)

- *oflag* options
  - must be one of these

option1	meaning	<fcntl.h> defined
O_RDONLY	open for reading only	0
O_WRONLY	open for writing only	1
O_RDWR	open for reading & writing	2

  - and can be OR’ed with any of these (by “|”)

option2	meaning
O_CREAT	create a file if the file does not exist.
O_EXCL	used with O_CREAT, return an error if the file already exists.
O_TRUNC	if the file exists, make it empty.
O_APPEND	write from the end of the file.
O_SYNC	do disk synchronization when does file I/O.

File access modes (1)

mode	meaning
S_ISUID	set-user-id at execution
S_ISGID	set-group-id at execution
S_ISVTX	set sticky bit
S_IRWXU	owner RWX
S_IRUSR	owner R
S_IWUSR	owner W
S_IXUSR	owner X
S_IRWXG	group RWX
S_IRGRP	group R
S_IWGRP	group W
S_IXGRP	group X
S_IRWXO	others RWX
S_IROTH	others R
S_IWOTH	others W
S_IXOTH	others X

will be explained later

File close

```
#include <unistd.h>

int close(int fd);
```

- parameters
  - *fd*: file descriptor
- return
  - 0 if OK
  - -1 on error

File open example

open-ex.c

```
#include <fcntl.h>

int main(int argc, char * argv[])
{
        FILE *fpo;    // file pointer
        int fdo;      // file descriptor

        if(argc != 2) {
                perror(argv[0]);
                return 1;
        }
        if((fdo = open(argv[1], O_RDWR | O_CREAT | O_TRUNC,
                                    S_IRUSR | S_IWUSR)) == -1) {
                perror(argv[1]);
                return 1;
        }
        if((fpo = fdopen(fdo, "r+")) == NULL) {
                perror("fdopen");
                return 2;
        }
        fprintf(fpo, "Hello, world! \n");
        fclose(fpo);
}
```

```
$./a.out test.txt
$ cat test.txt
Hello, world!
$
```

File creation

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

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

- parameters
  - *path* : file path name
  - *mode* : access permission
- return
  - *file descriptor* if OK
  - -1 on error

File seeking

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

off_t lseek(int fd, off_t offset, int whence);
```

- parameters
  - *fd* : file descriptor
  - *offset* : offset from the beginning to seek (move)
  - *whence* : SEEK_SET, SEEK_CUR, SEEK_END
- return
  - new offset value if OK
  - -1 on error

File create/lseek example

create-ex.c

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

int main(void)
{         int fd;
          char buf1[] = "Test1 data";
          char buf2[] = "Test2 data";

          if ((fd == creat ("test.txt", S_IRUSR | S_IWUSR | S_IRGRP |
                          S_IROTH)) < 0) {
                    printf("creat error");
                    return 1;
          }
          write(fd, buf1, 10);
          if(lseek(fd, 6L, SEEK_SET) == -1) {
                    printf("lseek error");
                    return 2
          }
          write(fd, buf2, 10);

          return 0;
}
```

```
$ ls
a.out   test.c
$ ./a.out
$ ls
a.out    test.c  test.txt
$ cat  test.txt
Test1 Test2 data
$
```

File reading

```
#include <unistd.h>

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

- parameters
  - *fd* : file descriptor
  - *buf* : buffer address
  - *nbyte* : number of bytes to read
- return
  - number of bytes read successfully if OK
  - -1 on error

File writing

```
#include <unistd.h>

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

- parameters
  - *fd* : file descriptor
  - *buf* : buffer address
  - *nbyte* : number of bytes to write
- return
  - number of bytes written successfully if OK
  - -1 on error

File copy example (1)

fcopy2-ex.c

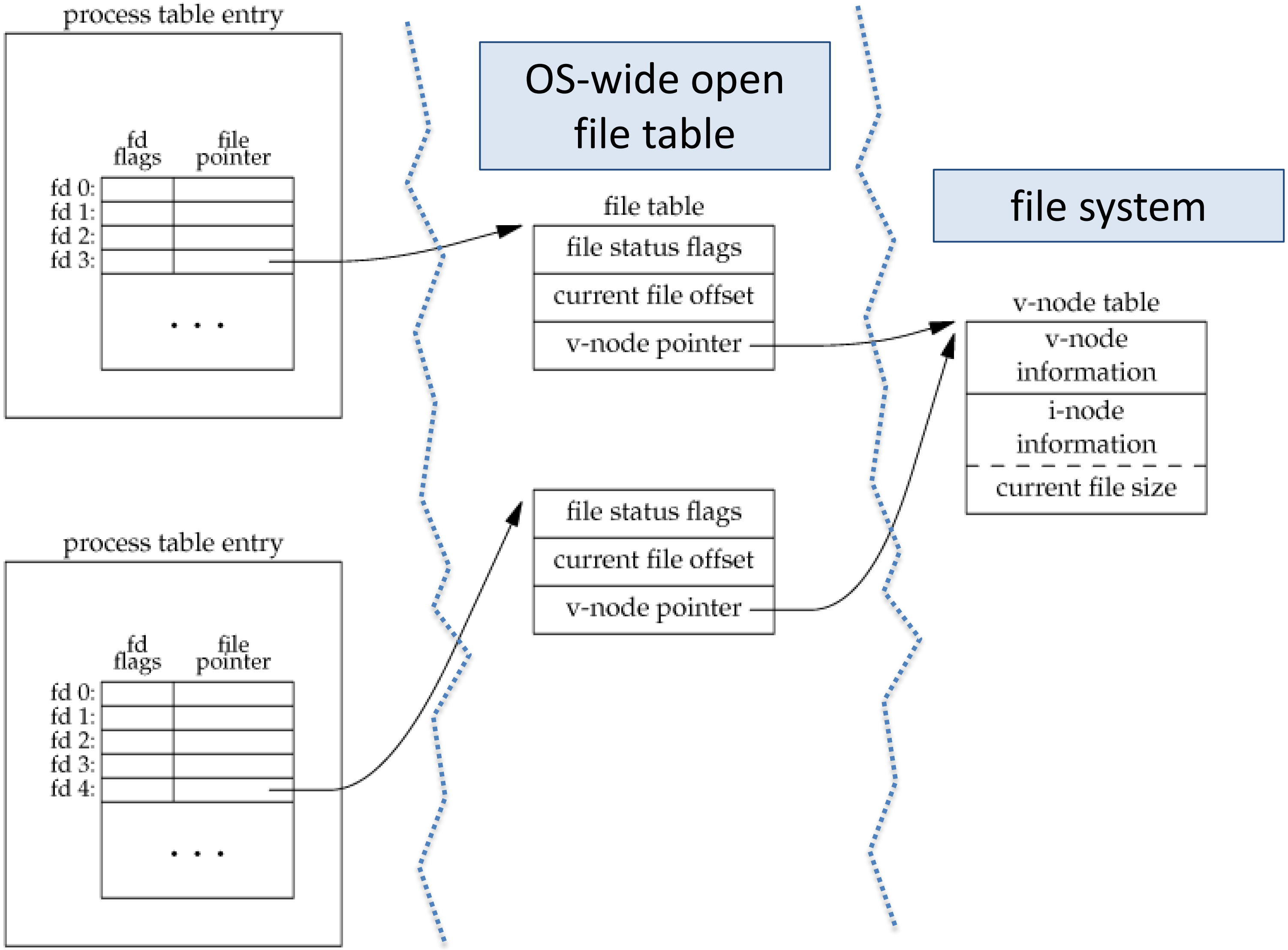
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#inlclude <fcntl.h>

#define BUFFER_SIZE   1024

int main(int argc, char *argv[])
{
        int fdi, fdo;
        char buf[BUFFER_SIZE];
        ssize_t n;

        if(argc != 3) {
                perror(argv[0]);
                return 1;
        }
```

File copy example (2)

fcopy2-ex.c

```
        if((fdi = open(argv[1], O_RDONLY)) == -1) {
                perror(argv[1]);
                return 2;
        }

        if((fdo = open(argv[2], O_WRONLY | O_CREAT | O_EXCL,
                              S_IRUSR | S_IWUSR)) == -1) {
                perror(argv[2]);
                return 3;
        }
        while((n = read(fdi, buf, BUFFER_SIZE)) > 0)
                write(fdo, buf, n);

        close(fdi);
        close(fdo);
        return 0;
}
```

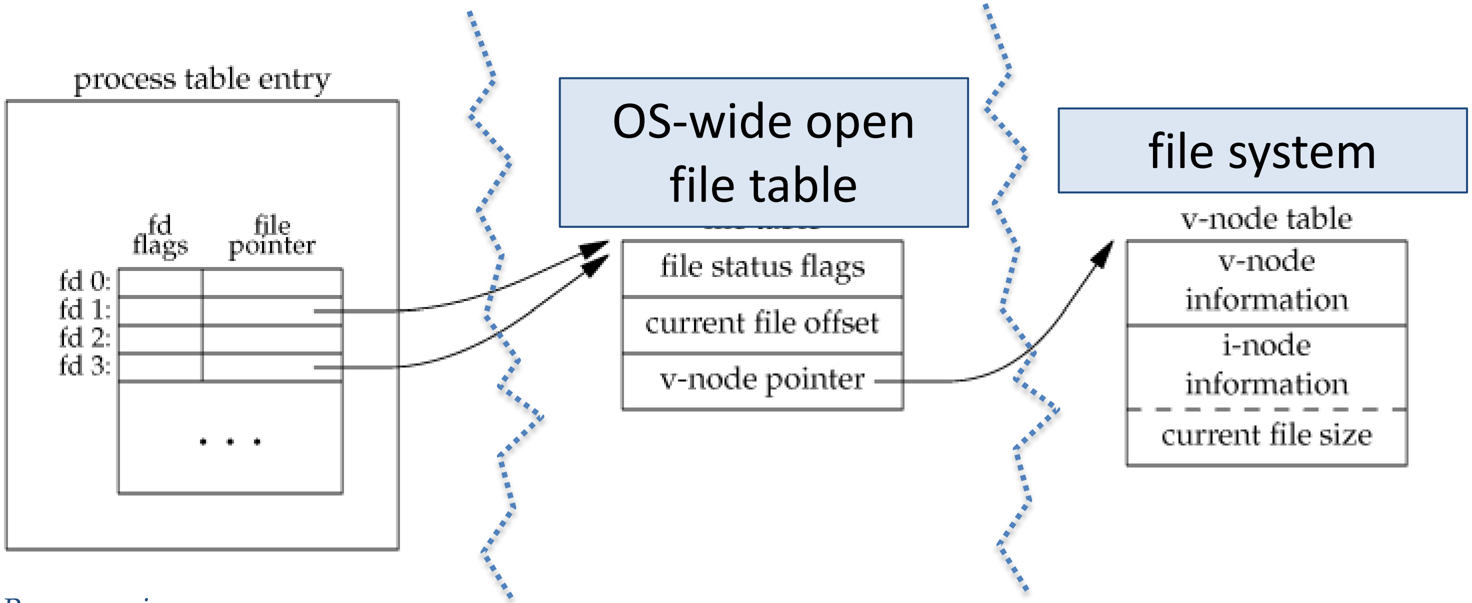
File descriptors & File table



Duplication of file descriptor (1)

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

- parameters
  - *fd* : file descriptor to duplicate
- return
  - newly duplicated file descriptor if OK
  - -1 on error



Duplication of file descriptor (2)

```
#include <unistd.h>
int dup2(int fd1, int fd2);
```

- parameters
  - *fd1* : source file descriptor
  - *fd2* : destination file descriptor
- return
  - copied file descriptor (should be same as fd2) if OK
  - -1 on error
- note
  - functionally same as dup except that dup2 designates destination file descriptor (fd2) the user wants

I/O redirection example (1)

io-redir.c

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

int main()
{
        int backup_des, stdout_des, ofdes;

        stdout_des = fileno(stdout);
        backup_des = dup(stdout_des);
```

I/O redirection example (2)

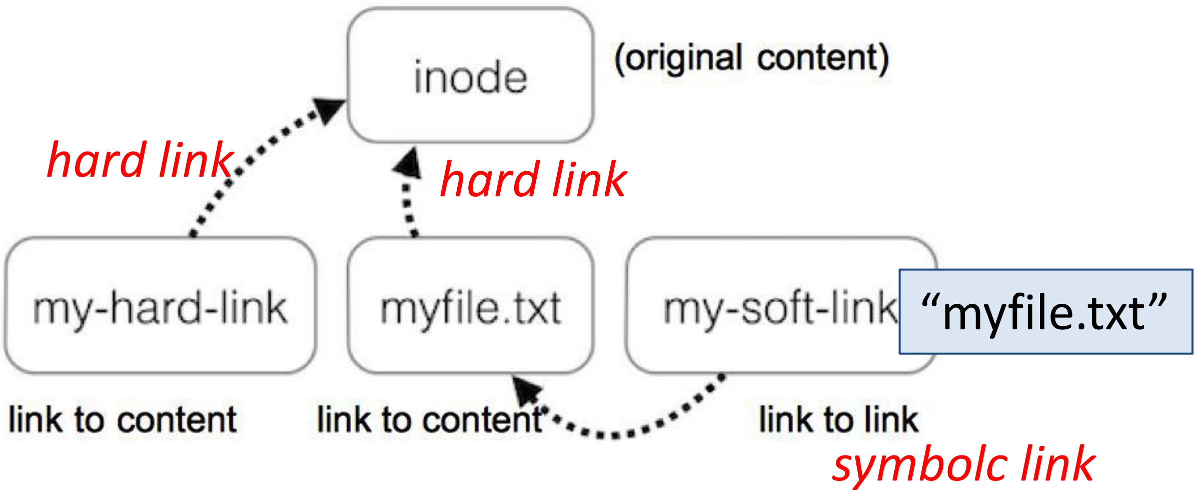
io-redir.c

```
        printf("Hello, world! (1)\n");

        ofdes = open("test.txt", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);

        dup2(ofdes, stdout_des);
        printf("Hello, world! (2)\n");

        dup2(backup_des, stdout_des);
        printf("Hello, world! (3)\n");

        close(ofdes);
}
```

Link (1)

- Symbolic link
  - also called *soft link*
  - a ***file*** which records a path name to a target file
  - if the target file is removed, the link is not valid any longer, but the symbolic link file still exists
- Hard link
  - another link which points to an existing inode which is already used by another file
  - an inode can be shared by two or more filenames (hardlinks)
  - once a file is update, the update can be seen in all the hardlink files
  - though a file is removed, another hardlink can access the file

Link (2)



- Linux command
  - symbolic link:

```
$ ln -s original_file   symbolic_link_name
```

  - hard link:

```
$ ln original_file   hard_link_name
```

Hard link

```
#include <unistd.h>

int link(const char *existing, const char *new_link);
```

- parameters
  - *existing* : original file name
  - *new_link* : new link name which will share the inode of existing file
- return
  - 0 if OK
  - -1 on error

Hard link example (1)

hlink-ex.c

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

int main(int argc, char *argv[])
{      if(argc != 3) {
             perror("argument error");
             return 1;
       }
       if (link(argv[1], argv[2]) < 0) {
             perror("link fail");
             return 2;
       }
}
```

Hard link example (2)

- *Run & Results*

```
$ ls -l my*
drwxr--r-x 3  oskernel  oskernel  512  Jul 9 21:58  .
-rw-r--r–  1  oskernel  oskernel   15  Jul 9 21:58  myfile

$ ./a.out  myfile  myhardlink

$ ls -l my*
drwxr-xr-x 3   oskernel  oskernel  512  Jul 9 22:01   .
-rw-r--r-- 2   oskernel  oskernel   15  Jul 9 22:01   myfile
-rw-r--r-- 2   oskernel  oskernel   15  Jul 9 22:01   myhardlink

$
```

Symbolic link

```
#include <unistd.h>

int symlink(const char *existing, const char *link_name);
```

- parameters
  - *existing* : original file name
  - *link_name* : link name which points the existing file
- return
  - 0 if OK
  - -1 on error

Symbolic link example (1)

symlink-ex.c

```c
#include <unistd.h>

int main(int argc, char *argv[])
{
        if(argc != 3) {
                perror("argument error");
                return 1;
        }
        if (symlink(argv[1], argv[2]) < 0) {
                perror("symlink fail");
                return 2;
        }
}
```

Symbolic link example (2)

- *Run & Results*

```
$ ls -l my*
-rw-r--r-    1  oskernel  oskernel    22    Jul 7 22:21    myfile

$ ./a.out myfile mylink

$ ls -l my*
-rw-r--r--   1  oskernel  oskernel    22    Jul 7 22:21    myfile
lrwxrwxrwx   1  oskernel  oskernel     6    Jul 9 22:18    mylink
```

- *What does the file “mylink” contain?*
  - *“myfile” → thus, the file size is 6 bytes.*

system calls and symbolic links

system calls which does **NOT** follow a symbolic link	system calls which follow a symbolic link
lchown, lstat, remove, readlink, rename, unlink	access, chdir, chmod, chown, creat, exec, link, mkdir, mkfifo, mknod, open, opendir, pathconf, stat, truncate

these system calls handle a symbolic link itself as a FILE!

Following a link (1)

```
#include <unistd.h>

int readlink(const char *path, void *buf, size_t bufsize);
```

- parameters
  - *path* : link name
  - *buf* : buffer address (original file's name)
  - *bufsize* : buffer size
- return
  - number of bytes read if OK
  - -1 on error

Following a link example (1)

readlink-ex.c

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

#define BUFFER_SIZE 100

int main(int argc, char *argv[])
{
          char buf[BUFFER_SIZE];
          int read_size = 0;

          if(argc != 2) {
                    perror("argument error");
                    return 1;
          }
          if ((read_size = readlink(argv[1], buf, BUFFER_SIZE)) < 0) {
                    perror("readlink");
                    return 2;
          }

          buf[read_size] = '\0' ;
          printf("%s\n", buf);
}
```

Following a link example (2)

Run & Results

```
$ ls  -ld  myfile  mylink
-rw-r--r-- 1  oskernel   oskernel   22   Jul 7 22:21   myfile
lrwxrwxrwx 1  oskernel   oskernel    6   Jul 7 22:18   mylink -> myfile

$ ./a.out mylink
myfile

$
```

File information retrieval (1)

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

int stat(const char *path, struct stat *buf);
```

- parameters
  - *path* : file path name
  - *buf* : address of `struct stat` (which contains a file information)
- return
  - 0 if OK
  - -1 on error

File information retrieval (2)

```
int lstat(const char *path, struct stat *buf);
```

- basically, same as stat(), but
  - if the file is a symbolic link, retrieve the information of the link file itself (does not follow the link)
- parameters
  - *path* : file path name
  - *buf* : address of `struct stat` (which contains a file information)
- return
  - 0 if OK
  - -1 on error

File information retrieval (3)

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

- parameters
  - *fd* : file descriptor
  - *buf* : address of `struct stat` (which contains a file information)
- return
  - 0 if OK
  - -1 on error

struct stat *fields*

```
struct stat   {
       dev_t  st_dev;        // device
       ino_t  st_ino;        // i-node #
       mode_t st_mode;       // access mode
       nlink_t st_nlink;     // number of hard links
       uid_t  st_uid;        // owner id
       gid_t  st_gid;        // group owner
       dev_t  st_rdev;       // device type (if inode device)
       off_t  st_size;       // total size of file
       long   st_blksize;    // block size for I/O
       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 (including
                             // ownership change)
};
```

Macros for struct stat

Macro	Functions
S_ISREG(st_mode)	return true if the file is *regular file*
S_ISDIR(st_mode)	return true if the file is *directory*
S_ISCHR(st_mode)	return true if the file is *character device file*
S_ISBLK(st_mode)	return true if the file is *block device file*
S_ISFIFO(st_mode)	return true if the file is *FIFO file*
S_ISLNK(st_mode)	return true if the file is *link file*
S_ISCOCK(st_mode)	return true if the file is *socket file*

File stat example (1)

fstat-ex.c

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

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

        if(argc != 3) {
                perror("argument error");
                return 1;
        }
        if (!strcmp(argv[1], "stat")) {
                if (stat(argv[2], &statbuf) < 0) {
                        perror("stat");
                        return 2;
                }
        }
```

File stat example (2)

fstat-ex.c

```
        else if (!strcmp(argv[1], "fstat")) {
                int filedes = open(argv[2], O_RDWR);
                if (fstat(filedes, &statbuf) < 0) {
                        perror("stat");
                        return 3;
                }
        }
        else if(!strcmp(argv[1], "lstat")) {
                if (lstat(argv[2], &statbuf) < 0) {
                        perror("lstat");
                        return 4;
                }
        }
        if(S_IREG(statbuf.st_mode))
                printf("%s is Regular File\n", argv[2]);
        if(S_ISDIR(statbuf.st_mode))
                printf("%s is Directory\n", argv[2]);
        if(S_ISLNK(statbuf.st_mode))
                printf("%s is Link File\n", argv[2]);
}
```

File stat example (3)

- Run & Results

```
$ ls -ld mydir myfile mylink
drwxr-xr-x 2    root root   512       Jul 9 19 : 59     mydir
-rw-r--r-- 1    root root   26        Jul 2 23 : 41     myfile
lrwxrwxrwx 1    root root   12        Jul 9 19 : 59     mylink ->mydir/myfile

$ ./a.out stat myfile
myfile is Regular File

$ ./a.out stat mydir
mydir is Directory

$ ./a.out stat mylink
mylink is Reqular File

$ ls -l mydir
-rwxrwxrwx 1 peace peace 0 Jul 9 19 : 59 myfile

$ ./a.out lstat mylink
Mylink is Link File

$ ./a.out fstat mylink
mylink is Regular File
```

Process’s Creator (1)

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

uid_t getuid(void)   // process creator’s uid
```

- return
  - user ID of the process if OK
  - -1 on error

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

uid_t getgid(void)   // process creator’s gid
```

- return
  - group ID of the process if OK
  - -1 on error

Process's Creator (2)

```
uid_t geteuid(void)
```

- return
  - effective user ID of the process if OK
  - -1 on error
- note
  - process's effective user id is used as a key for a kernel's protection system, and normally *uid* = *euid*,
  - but sometimes *euid* is a different one from *uid* for the dynamic protection system,

```
uid_t getegid(void)
```

- return
  - effective group ID of the process if OK
  - -1 on error

Process's Creator (3)

- When a process is created, the user(creator)'s IDs are assigned to the process.
- But, in the following cases, a process's effective IDs are set to a **file owner**'s IDs
  - if we run a program file with S_ISUID (or S_ISGID) bit, the process's UID is not my UID, but the file owner's UID (or S_ISGID).
- How to set the bits (example)

```
$ chmod u+s a.out
$ chmod g+s a.out
```

- Example

```
$ ls –al /bin
-rwsr-xr-x  1 root root   26492 Dec 1  2017 mount
```

  - this mount command is executed, the mount process's UID is set to root, not the user. → i.e. with the root's authority, the mount will be run!

Sticky bit: S_ISVTX

- In a directory with sticky bit
  - users can make their own files or subdirectories to the directory
  - but, each file can be only by its owner or supervisor.
- Example

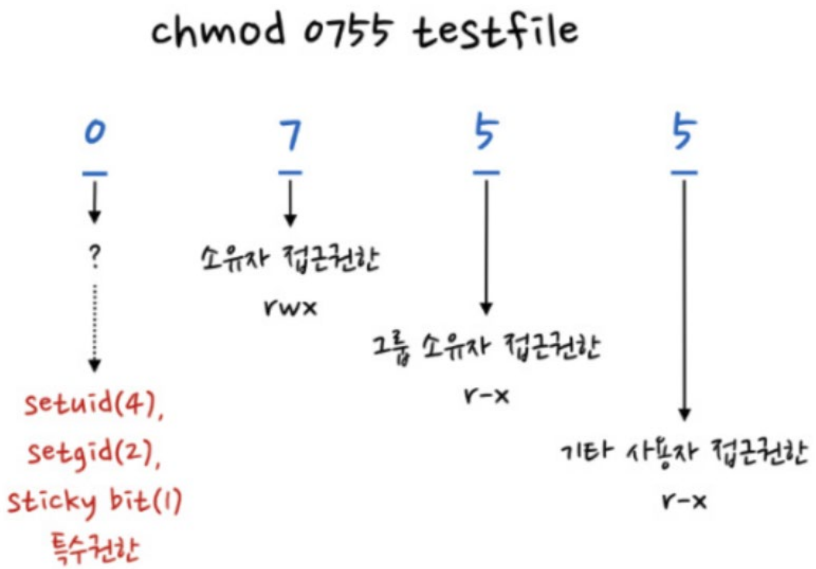
```
$ chmod o+t /test
$ touch /test/file1
$ rm /test/file1 → OK!
$ touch /test/file1
```

```
....
(another user login)
$ touch /test/file2
$ rm /test/file2 → OK!
$ rm /test/myfile → Failed!
```

File permission attributes

파일종류	특수권한			소유자 접근권한			그룹 소유자 접근권한			기타 사용자 접근 권한		
-,d,c,b,s,l,p	4	2	1	4	2	1	4	2	1	4	2	1
	setuid	setgid	sticky bit	r	w	x	r	w	x	r	w	x

d: directory
c: character device file
b: block device file
s: socket
l: symbolic link



File access of a process

- File access (read/write/execute) is allowed in the following cases
  - if the effective UID of the process is 0 (supervisor)
  - if the effective UID of the process is equal to that of file owner, and if the access permission bit of owner is SET
  - if the effective GID of the process is equal to that of file owner, and if the access permission bit of group is SET
  - if other's access permission bit is SET

File ownership & IDs

- File owner ID
  - set with the effective UID of the creating process
- File group ID
  - set with the effective GID of the creating process
  - but, if set-group-ID bit is set in the creating directory, the file’s GID is set to the directory’s GID.

File's access permission

```
#include<unistd.h>

int access(const char *path, int amode);
```

- check if the process can access a file in the path
- parameter
  - *path* : path name
  - *amode* : access mode for the process to check

amode	meaning
R_OK	*READ permission check*
W_OK	*WRITE permission check*
X_OK	*Execute or Exploration permission check*
F_OK	*File existence check*

- return
  - 0 if OK
  - -1 on error

Permission check example

access-ex.c

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

int main(int argc, char *argv[])
{
        if(argc < 2) {
                perror("argument error");
                return 1;
        }
        if(access(argv[1], F_OK) == 0) {
                printf("%s : File Exists\n", argv[1]);
        if(access(argv[1], R_OK) == 0)
                printf("%s : Read\n", argv[1]);
        if(access(argv[1], W_OK) == 0)
                printf("%s : Write\n", argv[1]);
        if(access(argv[1], X_OK) == 0)
                printf("%s : Execute\n", argv[1]);
        }
        else printf("%s : NOT exist\n", argv[1]);
}
```

Default permission change

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

mode_t umask(mode_t cmask);
```

- By default
  - by default, a file’s permission is set to 0666 (rw-rw-rw)
  - by default, a directory’s permission is set to 0777 (rwxrwxrwx)
- umask() changes the default permission
  - set *unmask* which masks off (i.e. not permits)
  - e.g. if umask is 0022, a new file permission is set to 0644

```
$ umask
  0022
$ touch test
$ ls -al test
-rw-r--r--  root root .........
```

- parameter
  - *cmask* : new umask
- return
  - previous umask

umask value

- OR’ed combination of these modes

mode	meaning
S_IRWXU	owner RWX
S_IRUSR	owner R
S_IWUSR	owner W
S_IXUSR	owner X
S_IRWXG	group RWX
S_IRGRP	group R
S_IWGRP	group W
S_IXGRP	group X
S_IRWXO	others RWX
S_IROTH	others R
S_IWOTH	others W
S_IXOTH	others X

File permission change

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

int chmod(const char *path, mode_t mode);
int fchmod(int fd, mode_t mode);
```

- change the permission mode of a file in the path
  - file owner or supervisor(root) can do this
- parameters
  - *path* : path name of a file
  - *mode* : the new access permission to change
    - mode is also OR’ed combination of the access modes (see p.7)
  - *fd* : file descriptor
- return
  - 0 if OK, -1 on error

Permission change example (1)

chmod-ex.c

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

int main(int argc, char *argv[])
{
        struct stat statbuf;
        if(argc != 2) {
                perror("argument error");
                return 1;
        }
        if (lstat(argv[1], &statbuf) < 0) {
                perror("lstat");
                return 2;
        }
        if (S_ISREG(statbuf.st_mode)) {
                if(chmod(argv[1], (statbuf.st_mode & ~S_IXGRP)) < 0) {
                        perror("chmod");
                        return 3;
                }
        }
        else printf("%s is not reqular file\n", argv[1]);
}
```

Permission change example (2)

- Run & Results

```
$ ls -ld myfile mydir
drwx------ 2 root root 512 Jul 15 15 : 13 mydir
-rwx--x--- 1 root root 0   Jul 15 15 : 12 myfile

$ ./a.out myfile
$ ./a.out mydir
mydir is not a regular file

$ ls -ld myfile mydir
drwx------ 2 root root 512 Jul 15 15 : 13 mydir
-rwx------ 1 root root 0   Jul 15 15 : 12 myfile

$
```

Ownership change

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

int chown(const char *path, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
                                 // doesn’t follow links
int fchown(int fd, uid_t owner, gid_t group);
```

- parameters
  - *path* : path name of a file
  - *owner* : owner’s UID
  - *group* : group’s GID
  - *fd* : file descriptor
- return
  - 0 if OK, -1 on error

Ownership change example (1)

chown-ex.c

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

int main(int argc, char *argv[])
{
        int owner_id, group_id, filedes ;

        if(argc != 5) {
                perror("argument error");
                return 1;
        }
        owner_id = atoi(argv[3]);
        group_id = atoi(argv[4]);
        if (strcmp(argv[1], "chown") == 0) {
                if (chown(argv[2], owner_id, group_id)) {
                        perror("chown");
                        return 2;
                }
                printf("chown %s to %s, %s\n", argv[2], argv[3], argv[4]);
        }
```

Ownership change example (2)

chown-ex.c

```
        else if (strcmp(argv[1], "fchown") == 0) {
                filedes = open(argv[2], O_RDWR);
                if (fchown(argv[2], owner_id, group_id)) {
                        perror("chown");
                        return 3;
                }
               printf("fchown %s to %s, %s\n", argv[2], argv[3], argv[4]);
        }
        else if (strcmp(argv[1], "lchown") == 0) {
                if (lchown(argv[2], owner_id, group_id)) {
                        perror("lchown");
                        return 4;
                }
               printf("lchown %s to %s, %s\n", argv[2], argv[3], argv[4]);
        }
}
```

Ownership change example (3)

- *Run & Results*

```
$ ls -l my*
drwxr-xr-x 2      oskernel   oskernel   512  Jul 9 21 : 20   mydir/
-rw-r--r– 1       oskernel   oskernel   0    Jul 7 15 : 37   myfile
lrwxrwxrwx 1      oskernel   oskernel   13   Jul 9 21 : 20   mylink -> ~mydir/myfile1

$ id -u cisc
1703

$id -g cisc
511

$ ./a.out chown myfile 1703 511
chown myfile to 1703, 511

$ ls -l my*
drwxr-xr-x 2   oskernel   oskernel    512    Jul 9 21 : 20   mydir/
-rw-r--r-- 1   cisc          cisc       0    Jul 7 15 : 37   myfile
lrwxrwxrwx 1   oskernel   oskernel     13    Jul 9 21 : 20   mylink -> ~mydir/myfile1
```