File and Directories

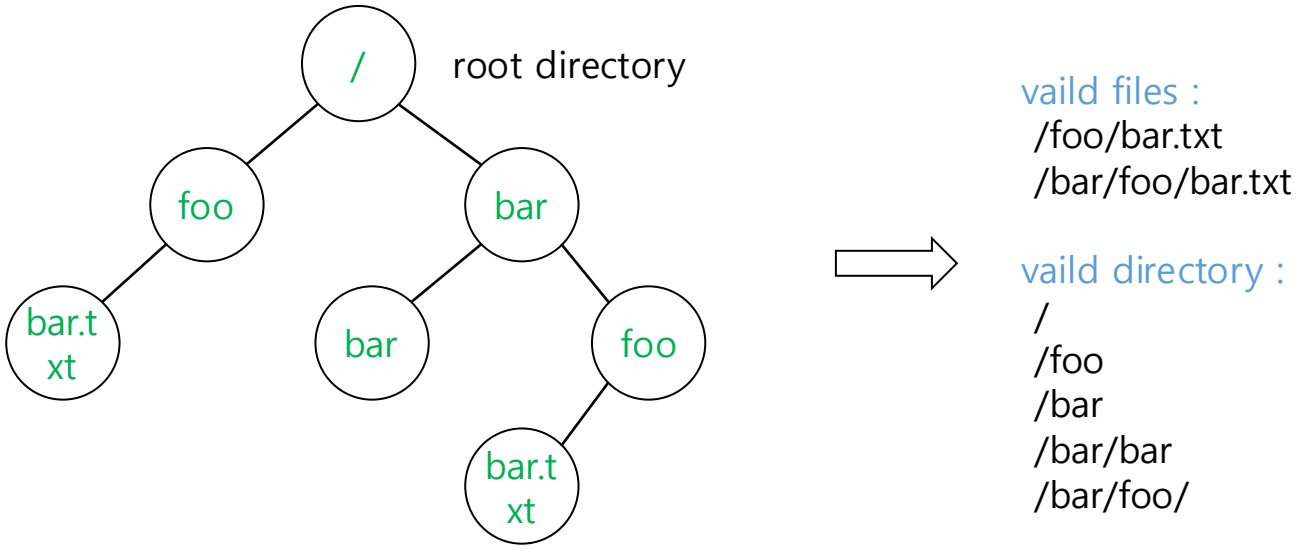
Operating System (OS)

안인규

강의자료는 KAIST 원유집 교수님 OS 강의 자료를 참고했습니다.

Concepts

- File
  - File is simply a linear array of bytes (everything is a stream of bytes)
  - Each file has low-level name as 'inode number' (커널이 실제로 파일을 식별하는 integer이며 unique)
- Directory (in Linux)
  - A file
  - A list of <user-readable filename, low-level filename(=inode)> pairs



An Example Directory Tree

vaild files :
/foo/bar.txt
/bar/foo/bar.txt

vaild directory :
/
/foo
/bar
/bar/bar
/bar/foo/

Interface:

- System calls
  - Creating, accessing, and deleting files
  - And, unlinking!

Interface: Creating a file

- Use `open` system call with `O_CREAT` flag

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

  - `O_CREAT` : create file
  - `O_WRONLY` : only write to that file while opened
  - `O_TRUNC` : make the file size zero (remove any existing content)
- open system call returns file descriptor
  - file descriptor is an integer, is used to access files
  - E.g., `read (file descriptor)`
  - File descriptor table

```
struct proc {
...
 struct file *ofile[NOFILE]; // Open files
...
};
```

Interface: Reading and Writing Files

- An Example of reading and writing 'foo' file.
  - `cat`: concatenate multiple files → outputs contents of all files
    - It is commonly used with output redirection '>'

```
prompt> echo hello > foo  //save the output to the file foo
prompt> cat foo           //dump the contents to the screen
hello
prompt>
```

Interface: Reading and Writing Files

- An Example of reading and writing 'foo' file.

```
prompt> echo hello > foo  //save the output to the file foo
prompt> cat foo           //dump the contents to the screen
hello
prompt>
```

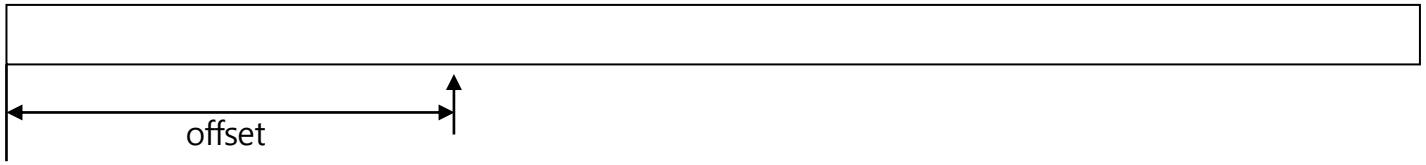
- The result of `strace` to figure out `cat` is doing.

```
prompt> strace cat foo   //strace to figure out what cat is doing
…
open("foo", O_RDONLY|O_LARGEFILE)  = 3
read(3, "hello\n", 4096)           = 6
write(1, "hello\n", 6)             = 6
hello
read(3, "", 4096)                  = 0
close(3)                           = 0
..
prompt>
```

  - `open()`: open file for reading with `O_RDONLY` and `O_LARGEFILE` flags.
    - returns file descriptor 3 (0,1,2, is for standard input/output/error)
  - `read()`: read bytes from the file.
  - `write()`: write buffer to standard output.

Reading and Writing Files (Cont.)

- OFFSET



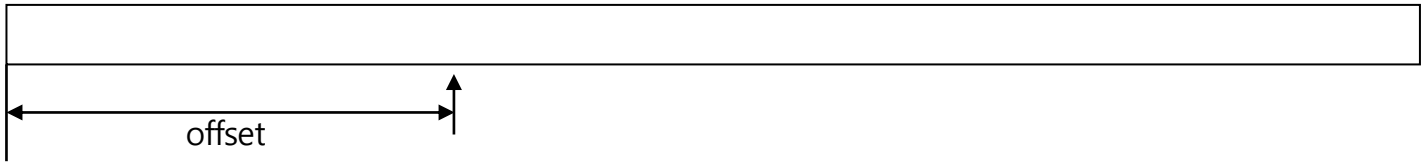
  - When a file is open, "an offset" is allocated
  - The position of the file where we start read and write (read() 및 write() 호출 시, 현재 offset에서부터 읽거나 씀)
  - Updated after read/write (offset은 read() 및 write() 후 자동으로 증가)

```
struct {
        struct spinlock lock;
        struct file file[NFILE];
} ftable;
```

```
struct file {
    int ref;
    char readable;
    char writable;
    struct inode *ip;
    uint off;
};
```

Reading and Writing Files (Cont.)

- OFFSET



  - When a file is open, "an offset" is allocated
  - The position of the file where we start read and write
  - Updated after read/write (offset은 read() 및 write() 후

```
struct {
        struct spinlock lock;
        struct file file[NFILE];
} ftable;
```

```
struct file {
    int ref;
    char readable;
    char writable;
    struct inode *ip;
    uint off;
};
```

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`read(fd, buffer, 100);`	100	100
`read(fd, buffer, 100);`	100	200
`read(fd, buffer, 100);`	100	300
`read(fd, buffer, 100);`	0	300
`close(fd);`	0	–

System Calls	Return Code	OFT[10] Current Offset	OFT[11] Current Offset
`fd1 = open("file", O_RDONLY);`	3	0	–
`fd2 = open("file", O_RDONLY);`	4	0	0
`read(fd1, buffer1, 100);`	100	100	0
`read(fd2, buffer2, 100);`	100	100	100
`close(fd1);`	0	–	100
`close(fd2);`	0	–	–

Reading and Writing Files (Cont.)

- How to read or write to a specific offset within a file ?

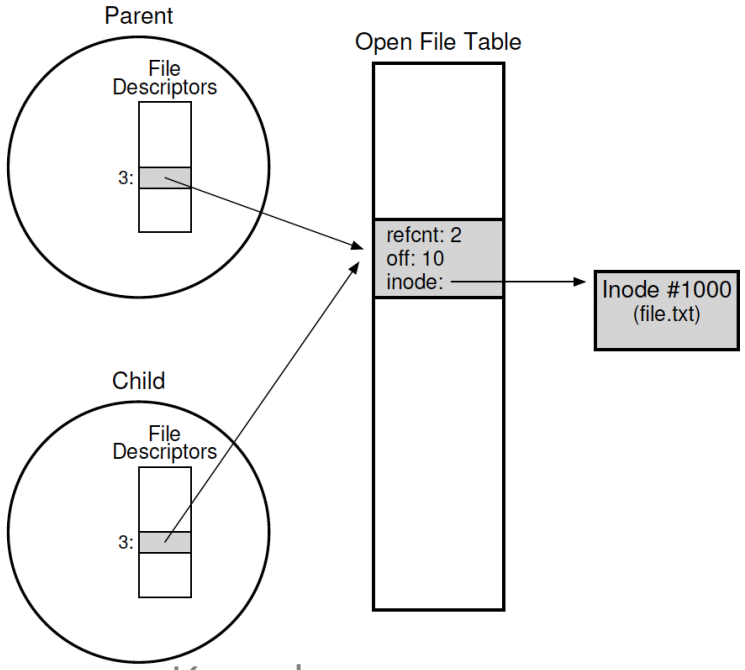
```
off_t lseek(int fd, off_t offset /*location */, int whence);
```

  - Third argument is how the seek is performed.
    - `SEEK_SET` : to offset bytes.
    - `SEEK_CUR`: to its current location plus offset bytes.
    - `SEEK_END`: to the size of the file plus offset bytes.
  - Does it perform Disk I/O? **NO**

System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

fork() and dup()

- Child process inherits the file descriptor table of the parent. (fd table을 복사해 물려받음)




```
struct {
        struct spinlock lock;
        struct file file[NFILE];
} ftable;
```

```
struct file {
    int ref;
    char readable;
    char writable;
    struct inode *ip;
    uint off;
};
```

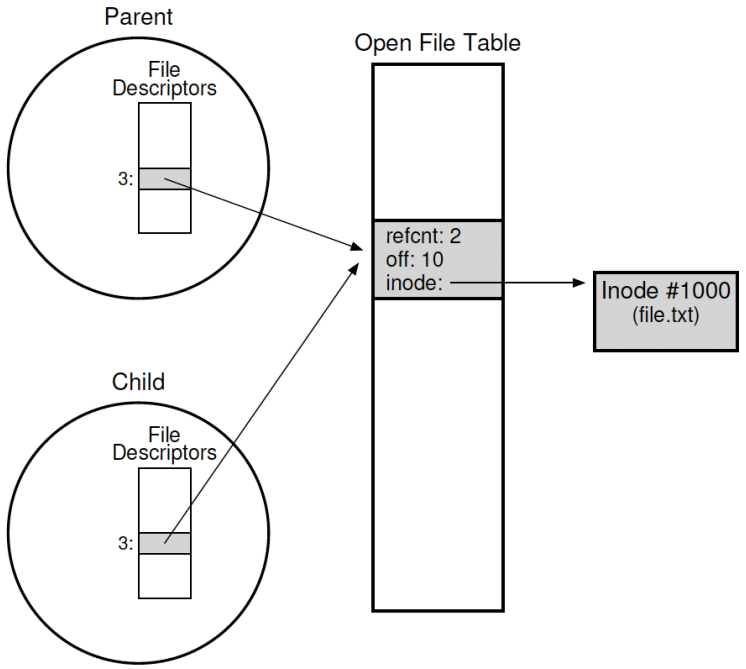
- Duplicating a file descriptor

```
int main(int argc, char *argv[]) {
    int fd = open("README", O_RDONLY);
    assert(fd >= 0);
    int fd2 = dup(fd);
    // now fd and fd2 can be used interchangeably
    return 0;
}
```

fork() and dup()

```
int main(int argc, char *argv[]) {
    int fd = open("file.txt", O_RDONLY);
    assert(fd >= 0);
    int rc = fork();
    if (rc == 0) {
        rc = lseek(fd, 10, SEEK_SET);
        printf("child: offset %d\n", rc);
    } else if (rc > 0) {
        (void) wait(NULL);
        printf("parent: offset %d\n",
               (int) lseek(fd, 0, SEEK_CUR));
    }
    return 0;
}
```

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```



fsync()

- **Persistency**
  - `write()` : write data to the buffer. Later, save it to the storage. (e.g., 백그라운드 flusher thread가 주기적으로 dirty buffer를 디스크에 기록)
  - some applications require more than eventual guarantee.
- `fsync()` : the writes are forced immediately to disk
  - Return 0 on success, -1 on failure

```
int fsync(int fd /*for the file referred to by the specified file*/)
```

- An Example of `fsync()`.

```
1    int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC);
2    int rc = write(fd, buffer, size);
3    rc = fsync(fd);
```

  - If a file is created, it needs to be durably a part of the directory.
    - Above code requires `fsync()` to directory also.

```
1    int dirfd = open(".", O_DIRECTORY | O_RDONLY);
2    rc = fsync(dirfd);
```

Renaming Files

- `rename()` : rename a file to different name.
  - It implemented as an atomic call
    (The file name must be "foo" or "bar" if the system crashes during the renaming)
  - E.g., change from foo to bar

```
promt > mv foo bar
```

- Saving a file in an editor
  (Imagine that your are using a file editor (e.g., vim), and you insert a line into the middle of a file)

```
1   int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC);
2   write(fd, buffer, size); // write out new version of file
3   fsync(fd);
4   close(fd);
5   rename("foo.txt.tmp", "foo.txt");
```

Thanks to the atomic call (rename), the file "foo.txt" is guaranteed to be either the original "foo.txt" or the new "foo.txt" with the inserted line

Getting Information About Files

- `stat()`: Show the File metadata
  - metadata is information about each file, e.g., size, permission, ..
- `stat` structure is below:
  - It is located in the user space (where does it get the file metadata?)

```
1     struct stat {
2         dev_t st_dev; /* ID of device containing file */
3         ino_t st_ino; /* inode number */
4         mode_t st_mode; /* protection */
5         nlink_t st_nlink; /* number of hard links */
6         uid_t st_uid; /* user ID of owner */
7         gid_t st_gid; /* group ID of owner */
8         dev_t st_rdev; /* device ID (if special file) */
9         off_t st_size; /* total size, in bytes */
10        blksize_t st_blksize; /* blocksize for filesystem I/O */
11        blkcnt_t st_blocks; /* number of blocks allocated */
12        time_t st_atime; /* time of last access */
13        time_t st_mtime; /* time of last modification */
14        time_t st_ctime; /* time of last status change */
15  };
```

Getting Information About Files (Cont.)

- An example of `stat()`
  - All information is in **a inode**

```
prompt> echo hello > file
prompt> stat file
```

The inode is on the disk!

```
File: 'file'
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ root) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

Getting Information About Files (Cont.)

- An example of `stat()`
  - All information is in **a inode**
  - **The inode is different from the entry of the file description table**

```
prompt> echo hello > file
prompt> stat file
```

The inode is on the disk!

```
File: 'file'
Size: 6 Blocks: 8 IO Block: 4096 regular file
Device: 811h/2065d Inode: 67158084 Links: 1
Access: (0640/-rw-r-----) Uid: (30686/ root) Gid: (30686/ remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

```
struct {
        struct spinlock lock;
        struct file file[NFILE];
} ftable;
```

```
struct file {
    int ref;
    char readable;
    char writable;
    struct inode *ip;
    uint off;
};
```

Removing Files

- The result of `strace` to figure out what `rm` is doing.
  - `rm` is Linux command to remove a file
  - `rm` calls `unlink()` to remove a file.

```
1   prompt> strace rm foo
2   ...
3   unlink("foo")
4   ...
5   prompt>
```

Making Directories

- `mkdir()`: Make a directory
  - When a directory is created, it is empty
  - Empty directory have two entries: . (itself), ..(parent)

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                          = 0
...
prompt>
```

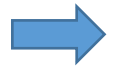
```
1  prompt> ls -al
2  total 8
3  drwxr-x--- 2 root root 6 Apr 30 16:17 ./
4  drwxr-x--- 26 root root 4096 Apr 30 16:17 ../
```

Reading Directories

- `readdir()`
  - It reads each entry (file or folder) in the directory

  - Directory is a file, but with a specific structure.
  - When reading a directory, we use specific system call other than read().
  - A sample code to read directory entries.

```
1   int main(int argc, char *argv[]) {
2    DIR *dp = opendir("."); /* open current directory */
3    assert(dp != NULL);
4    struct dirent *d;
5    while ((d = readdir(dp)) != NULL) { /* read one directory entry */
6     printf("%d %s\n", (int) d->d_ino, d->d_name);
7    }
8    closedir(dp); /*close current directory */
9    return 0;
10  }
```



```
> ls -i

67158084 file1.txt
67158085 foo
67158086 ..
67158087 .
```

It is exactly what the program `ls` does

Reading Directories

- Structure of the directory entry

```
struct dirent {
    char d_name[256]; /* filename */
    ino_t d_ino; /* inode number */
    off_t d_off; /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type; /* type of file */
};
```

Hard Links

- `link()`: Link old file and a new file.
  - Create hard link named file2.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2 /* create a hard link, link file to file2 */
prompt> cat file2
hello
```

- The result of `link()`
  - Two files have same inode number, but two human name(file, file2)
    → Hard link: 하나의 파일에 여러 개의 이름(경로)를 부여

```
prompt> ls -i file file2
67158084 file  /* inode value is 67158084 */
67158084 file2 /* inode value is 67158084 */
prompt>
```

Hard Links (Cont.)

- Hot to create a file?
  - **Step1.** Make an inode, track all information about the file.
  - **Step2.** Link a human-readable name to file.
  - **Step3.** Put link file into a current directory.
- How to create hard link file? → Skip **Step1**

- After creating a hard link to file, old and new files have no difference (동등한 지위)
- Thus, to remove a file, we call `unlink()`

unlink Hard Links

- What `unlink()` is doing ?
  - Check reference count within the inode number.
  - Remove link between human-readable name and inode number.
  - Decrease reference count
    - When only it reaches zero, It delete a file (free the inode and related blocks)

unlink Hard Links (Cont.)

- The result of `unlink()`

```
prompt> echo hello > file            /* create file*/
prompt> stat file
... Inode: 67158084 Links: 1 ...     /* Link count is 1 */
prompt> ln file file2                /* hard link file2 */
prompt> stat file
... Inode: 67158084 Links: 2 ...     /* Link count is 2 */
prompt> stat file2
... Inode: 67158084 Links: 2 ...     /* Link count is 2 */
prompt> ln file2 file3               /* hard link file3 */
prompt> stat file
... Inode: 67158084 Links: 3 ...     /* Link count is 3 */
prompt> rm file                      /* remove file */
prompt> stat file2
... Inode: 67158084 Links: 2 ...     /* Link count is 2 */
prompt> rm file2                     /* remove file2 */
prompt> stat file3
... Inode: 67158084 Links: 1 ...     /* Link count is 1 */
prompt> rm file3
```

Symbolic Links

- Symbolic link
  - Special file that contains path to the target file/directory. (문자열로 저장)
  - Symbolic links have their own inodes (it is also a file)
  - Hard Link cannot create to a directory (Linux does not allow hard links to directories)
  - Hard Link cannot create to a file to other partition (=different file system) → inode is available within the file system
- An example of symbolic link

```
prompt> echo hello > file
prompt> ln -s file file2  /* option -s : create a symbolic link, */
prompt> cat file2
hello
```

Symbolic Links (Cont.)

- Symbolic link is different file type.

```
prompt> ls -al
drwxr-x--- 2 remzi remzi 29 May 3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May 3 15:14 ../       /* directory */
-rw-r----- 1 remzi remzi 6 May 3 19:10 file          /* regular file */
lrwxrwxrwx 1 remzi remzi 4 May 3 19:10 file2 -> file /* symbolic link */
```

- Symbolic link is subject to the dangling reference.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

Summary

- Create file
- read/write/lseek
- mkdir/readdir
- fsync
- hardlink/softlink