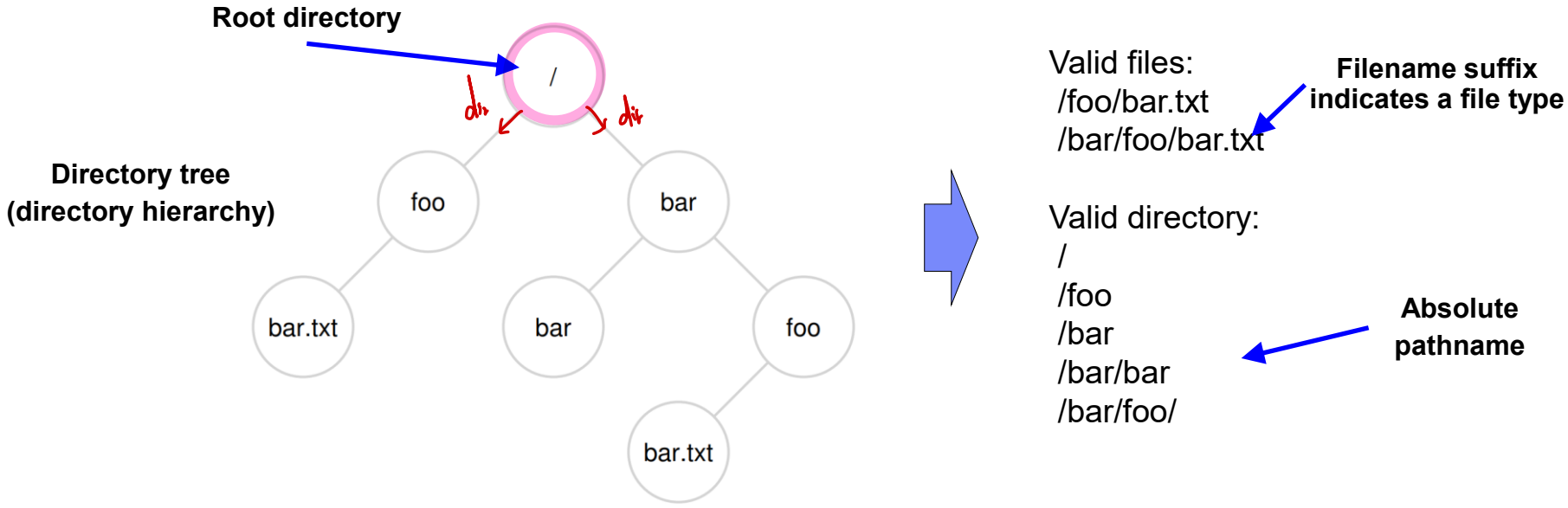
Interlude: Files and Directories

Prof. Yongtae Kim

Computer Science and Engineering
Kyungpook National University

Files and Directories

- **Two key abstractions have developed storage virtualization**
  - The first is the file, simply a linear array of bytes, which can be read and written Each file has some kind of low-level name, usually a number (inode number)
  - OS does not know much about the structure of the file, rather, the file system has a responsibility to store data persistently on disk
  - The second abstraction is that of a directory, which also has low-level name
  - Each directory contains a list of (user-readable name, low-level name) pairs
  - Each entry in a directory refers to either files or other directories



Creating Files

- **`open()` system call with `O_CREATE` flag can create a new file**
  - e.g.) Creating a file "foo"

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

  - The second parameter flags:

Flag	Meaning
O_CREATE	To creates the file if it does not exist
O_WRONLY	To ensures that the file can only be written to
O_TRUNC	To make file size zero (removing any existing contents) if exist

  - The third parameter specifies permissions (making readable/writable by owner)
  - `open()` returns a file descriptor, which is an integer, private per process
  - The file descriptors are used to access files and managed by OS on a per-process basis

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

xv6's proc

Reading and Writing Files

- **Reading and writing example using `echo` and `cat`**
  - We redirect the **`echo`**'s output to the file foo
  - Then, we use **`cat`** to see the contents of the file

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

- **How does the `cat` program access the file foo?**
  - **`strace`** (Linux tool) can trace every system call made by a program while it runs, and dump the trace to the screen; other systems have similar tools (e.g. **`dtruss`** on Mac)

```
prompt> strace cat foo
...
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>
```

1) **`cat`** opens the file for reading (O_RDONLY), 64-bit offset (O_LARGEFILE), returns a file descriptor of 3 (0: stdin, 1:stdout, 2: stderr)
2) **`cat`** uses the **`read()`** system call, which returns the number bytes it read, to repeatedly read some bytes from a file
3) "hello" is written to the screen by **`write()`** system call with the file descriptor 1 (stdout)
4) **`close()`** system call closes the file foo by indicating the descriptor 3

Reading and Writing, But Not Sequentially

- **Files can be accessed at a specific offset within a file**
  - To do so, **`lseek()`** system call is used: `off_t lseek(int fildes, off_t offset, int whence);`
  - **`offset`** indicates the file offset to a particular location within the file
  - **`whence`** determines exactly how the seek is performed

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
  location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
  the file plus offset bytes.
```

  - **`offset`** is kept in that **`struct file`**, as referenced from **`struct proc`**

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

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

  - Examples

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	–

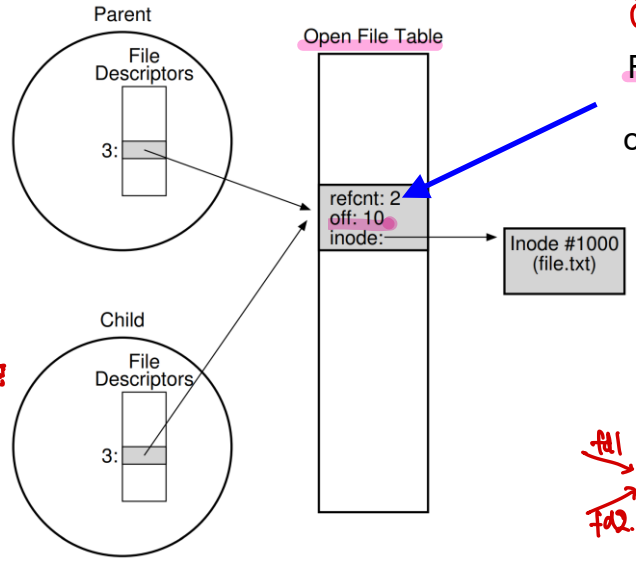
System Calls	Return Code	Current Offset
`fd = open("file", O_RDONLY);`	3	0
`lseek(fd, 200, SEEK_SET);`	200	200
`read(fd, buffer, 50);`	50	250
`close(fd);`	0	–

Shared File Table Entries: `fork()` and `dup()`

- **Open file table keeps all files currently opened in the system**
  - An entry in the open file table can be shared
  - e.g.) **`fork()`** creates a child process and both parent and child call **`lseek()`**

```
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>
```



- **`dup()`** system call allows a process to create a new file descriptor that refers to the same underlying open file

```
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;
}
```

Writing Immediately with `fsync()` & Renaming Files

- **The file system will buffer some writes in memory for some time (e.g. 5 or 30 seconds) for performance reason**
  - The write(s) will actually be issued to the storage device and only in rare cases data can be lost (e.g. machine crashes)
  - However, some applications require more than this eventual guarantee (DBMS)
  - `fsync()` forces all dirty data to disk and returns once all the writes are done

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC,
              S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

- **`rename()` system call changes a file to different name**
  - This call is implemented as an atomic call with respect to system crashes; thus, the file will either be named the old or the new, and no in-between state

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

Getting Information About Files & Removing Files

- **The file system to keep a fair amount of information about each file it is storing: metadata**
  - To see the metadata for a file, `stat()` or `fstat()` system calls can be used
  - These calls take a pathname to a file and fill in a `stat` structure below

```
struct stat {
  dev_t     st_dev;     // ID of device containing file
  ino_t     st_ino;     // inode number
  mode_t    st_mode;    // protection
  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 ID (if special file)
  off_t     st_size;    // total size, in bytes
  blksize_t st_blksize; // blocksize for filesystem I/O
  blkcnt_t  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 status change
};
```

```
prompt> echo hello > file
prompt> stat file
  File: 'file'
  Size: 6   Blocks: 8   IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084    Links: 1
Access: (0640/-rw-r-----) Uid: (30686/remzi)
  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
```

  - Each file system usually keeps this information in a structure called an inode

- **To remove a file, mysteriously-named system call `unlink()` with a file name to be deleted as a parameter can be used**

```
prompt> strace rm foo
...
unlink("foo")
...
```

Making, Reading, and Deleting Directories

- **Beyond files, a set of directory-related system calls enable you to make, read, and delete directories**
  - To create a directory, a single system call, **`mkdir()`**, is available
  - When created, it is empty; an empty directory has two entries: one entry that refers to itself (dot "."), and one entry that refers to its parent (".." dot-dot)

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

```
prompt> ls -a
./  ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi    6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

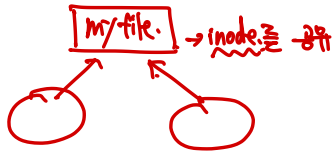
  - To prints the contents of a directory, three calls, **`opendir()`**, **`readdir()`**, and **`closedir()`**, can be used

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino,
               d->d_name);
    }
    closedir(dp);
    return 0;
}
```

```
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
};
```

  - Finally, you can delete a directory with a call to **`rmdir()`** system call, which requires that the directory be empty

Hard Links



- **`link()` system call takes an old pathname and a new one**
  - When you "link" a new file name to an old, you essentially create another way to refer to the same file and these files share the same inode number

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

```
prompt> ls -i file file2
67158084 file
67158084 file2
prompt>
```

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

  - When creating file, two things happen:
    1) making a structure (inode) that tracks virtually all information of file (e.g. size)
    2) linking a human-readable name to the file, and putting the link into a directory
  - When the file system unlinks file, it checks a reference count (called link count) within the inode number and decreases the count
  - Only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly delete the file
  - This tells why the system call to delete a file is **`unlink()`**

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084    Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084    Links: 2 ...
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084    Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084    Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084    Links: 1 ...
prompt> rm file3
```

Symbolic Links

- **There is one other type of useful link: symbolic (soft) link**
  - Hard link is limited; can't link to a directory or to files in other disk partitions
  - Creating a soft link looks the same with hard link but actually quite different

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

```
prompt> stat file
  ... regular file ...
prompt> stat file2
  ... symbolic link ...
```

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

  - Symbolic links are a third type the file system and have own inode numbers
  - They can link between different file systems and link to a directory
  - The size of symbolic linked file is small (e.g. 4 bytes) because a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file
  - Therefore, linking to a longer pathname makes the link file size bigger

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi  6 May  3 19:17 alongerfilename
lrwxrwxrwx 1 remzi remzi 15 May  3 19:17 file3 ->
                                         alongerfilename
```

```
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
```

  - Finally, because of the way symbolic links are created, they leave the possibility for what is known as a dangling reference

Permission Bits and Access Control Lists

- **The files are commonly shared among different users and processes and are not (always) private**
  - Thus, a more comprehensive set of mechanisms for enabling various degrees of sharing are usually present within file systems
  - The first form of such mechanisms is the classic UNIX permission bits

```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

owner group

Permission of owner group other

ability to r: read w: write x: execute -: none

owner, group의 애들 상관없.

  - The owner of the file can readily change the permissions: `prompt> chmod 600 foo.txt`
  - For regular files, the presence of execute bit determines if program can be run

```
prompt> ./hello.csh
hello, from shell world.
```

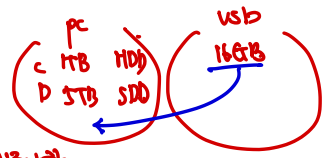
// 0

```
prompt> chmod 600 hello.csh
prompt> ./hello.csh
./hello.csh: Permission denied.
```

owner도 뺄 수 있어.

  - For directories, the execute bit behaves a bit differently
    ➔ it enables a user (or group, or everyone) to do things like change directories into the given directory, and, in combination with the writable bit, create files therein

Making and Mounting a File System



- **How to assemble a full directory tree from many underlying file systems?**
  - This task is accomplished via first making file systems, and then mounting them to make their contents accessible
  - To make a file system, most file systems provide a tool, referred to as **`mkfs`**
  - Once a device, such as a disk partition (e.g. /dev/sda1), and a file system type (e.g. ext3) are given, it writes an empty file system, starting with a root directory
  - Once created, it needs to be made accessible within the uniform file-system tree, which can be achieved by mount program
  - e.g.) unmounted ext3 file system, stored in device partition /dev/sda1 whose root directory has two directories a, b; this is mounted to /home/users

```
prompt> mount -t ext3 /dev/sda1 /home/users
prompt> ls /home/users/
a b
```

We can access:
/home/users/a
/home/users/b

  - When running mount program, you will see like

```
/dev/sda1 on / type ext3 (rw)
proc on /proc type proc (rw)
sysfs on /sys type sysfs (rw)
/dev/sda5 on /tmp type ext3 (rw)
/dev/sda7 on /var/vice/cache type ext3 (rw)
tmpfs on /dev/shm type tmpfs (rw)
AFS on /afs type afs (rw)
```