Engineering & Computer Science
Syracuse University

CIS 657 – Principles of Operating Systems

Topic: Persistence – File and Directories

Endadul Hoque

Acknowledgement

- Youjip Won (Hanyang University)
- OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin)

Persistent Storage

- Keep a data **intact** even if there is a power loss.
  - Hard disk drive
  - Solid-state storage device

- Two key abstractions in the **virtualization of storage**
  - File
  - Directory

File

- A linear array of bytes

- Each file has low-level name as **inode number**
  - A user is not typically aware of this name.

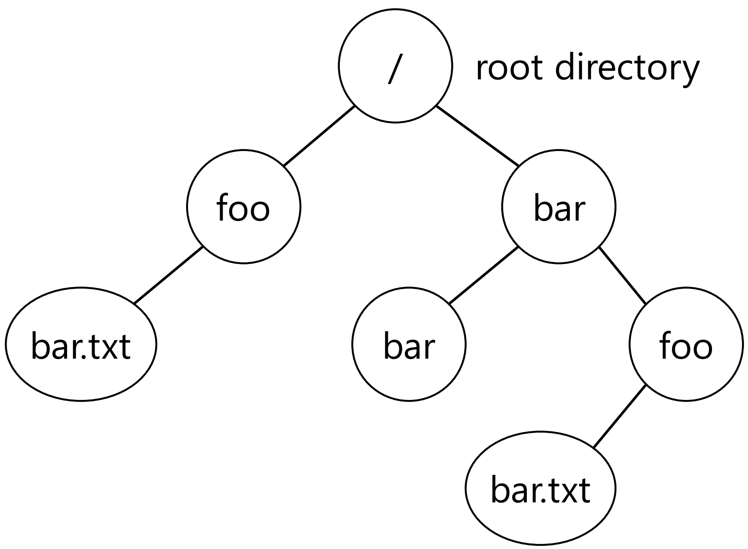
- Filesystem has a responsibility to **store** data **persistently** on disk.

Directory

- Directory is <u>**like a file**</u>, also has a low-level name.
  - It contains a list of (user-readable name, low-level name) **pairs**.
  - Each entry in a directory refers to either *files* or other *directories*.

- Example
  - A directory has an entry ("foo", "10")
    - A file "foo" with the low-level name "10"

Directory Tree (Directory Hierarchy)



An Example Directory Tree

Valid files (absolute pathname) :

/foo/bar.txt
/bar/foo/bar.txt

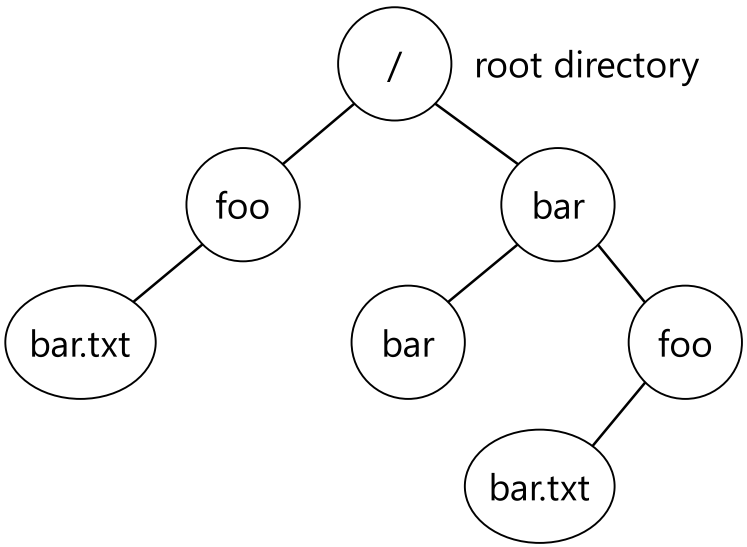
Valid directory :

/
/foo
/bar
/bar/bar
/bar/foo/

Sub-directories

Directories and files can have the same name
as long as they are in different locations in the file-system tree

Directory Tree (Directory Hierarchy)



An Example Directory Tree

Valid files (absolute pathname) :

/foo/bar.txt
/bar/foo/bar.txt

Valid directory :

/
/foo
/bar
/bar/bar
/bar/foo/

Sub-directories

bar.txt

Arbitrary name

Extension (related to the type): more of a convention

Creating Files

- Use `open()` system call with `O_CREAT` flag.

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

  - `O_CREAT` : create file.
  - `O_WRONLY` : only write to that file while opened.
  - `O_TRUNC` : make the file size zero (remove any existing content), if the file already exists.

- `open()` system call returns **file descriptor**.

Creating Files

- Use `open()` system call with `O_CREAT` flag.

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

- `open()` system call returns **file descriptor**.
  - *File descriptor* is an integer, and is used to access files.
  - Private per process
  - Think of it as an "opaque handle" or "a pointer" to manipulate files (e.g., read/write)

Creating Files

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

- *File descriptors* are managed by OS on a per-process basis
- Stored in **`proc`** structure (or PCB)
  - **`ofile`** array keeps track of which files are opened by this process
  - **`ofile[i]`** is a pointer to a **`struct file`** for **`fd i`** (more on this later)
    - **`ofile[0]`** is for **stdin**, **`ofile[1]`** is for **stdout**, **`ofile[2]`** is for **stderr**

Reading and Writing Files

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

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

  - `echo`: redirect the output of echo to the file foo
  - `cat` : dump the contents of a file to the screen

How does the **`cat`** program access the file **`foo`** ?

We can use **`strace`** to trace the **system calls** made by a program.

Reading and Writing Files

```
prompt> strace cat foo
…
open("foo", O_RDONLY|O_LARGEFILE)      = 3
read(3, "hello\n", 4096)               = 6
write(1, "hello\n", 6)                 = 6 // file descriptor 1: standard out
hello
read(3, "", 4096)                      = 0 // 0: no bytes left in the file
close(3)                               = 0
…
prompt>
```

- `open`(file name, flags)
  - Return file descriptor (**3** in example)
  - File descriptor 0, 1, 2, is for standard input/ output/ error.
- `read`(file descriptor, buffer pointer, the size of the buffer)
  - Return the number of bytes it read
- `write`(file descriptor, buffer pointer, the size of the buffer)
  - Return the number of bytes it write

Reading and Writing Files

```
prompt> strace cat foo
…
open("foo", O_RDONLY|O_LARGEFILE)     = 3
read(3, "hello\n", 4096)              = 6
write(1, "hello\n", 6)                = 6 // file descriptor 1: standard out
hello
read(3, "", 4096)                     = 0 // 0: no bytes left in the file
close(3)                              = 0
…
prompt>
```

- `strace` prints about system calls, not library calls
  - Program may not always **directly invoke** these system calls
  - E.g., **`write`**() can be invoked by **`printf`**()

Reading and Writing Files

- Writing a file (A similar set of read steps)
  - A file is opened for writing (`open()`).
  - The `write()` system call is called.
    - Repeatedly called for larger files
  - `close()`

Reading And Writing, But Not Sequentially

- An open file has a **current offset**.
  - Determine **where** the <u>next read</u> or <u>write</u> will begin reading from or writing to within the file.

- Update the current offset
  - **Implicitly**: After a read or write of `N` bytes takes place, `N` is added to the current offset.
  - **Explicitly**: `lseek()`

Reading And Writing, But Not Sequentially

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

- `fildes` : File descriptor
- `offset` : Position the file offset to a particular location within the file
- `whence` : Determine how the seek is performed

From the man page:

```
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.
```

Reading And Writing, But Not Sequentially

- Need to keep track of some essential information about the file (say, **current offset**)

```
struct file {
  int ref;                          // reference counter
  char readable;
  char writable;
  struct inode *ip;                 // file inode
  uint off;                         // current offset
};
```

- All these file structures are stored in a system-wide **Open File Table** (OFT)

```
struct {
  ...
  struct file file[NFILE];
} oftable;
```

Reading And Writing, But Not Sequentially

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	–

- Each `read()` **increments** the offset
  - Making it easy of a process just keep reading the next chunk of the file
- At the end, `read()` **returns 0** indicating that it has reached the end of file

Reading And Writing, But Not Sequentially

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	–	–

- **Two file descriptors** are allocated (3 and 4) even if it is the **<u>same file</u>**
  - Each refers to a **<u>different</u>** entry in the **open file table** (say, 10 and 11)
- Each current offset is updated ***<u>independently</u>***

Reading And Writing, But Not Sequentially

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	–

- **`lseek`**() sets the current offset to 200
- **`read`**() reads the next 50 bytes, and updates the current offset accordingly.

Sharing File Table Entries: fork()

- The mapping of file descriptor to an entry in the open file table is a **one-to-one mapping**.
- Even if some other process reads the same file at the time, each will have its **own entry** in the OFT
- Each logical reading or writing of a file is **independent**
  - Each has its own **current offset** while it accesses the given file.
- There are a few interesting cases where an OFT entry is ***shared***.
  - E.g., in case of `fork()`

Sharing File Table Entries: fork()

```
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;
}
```

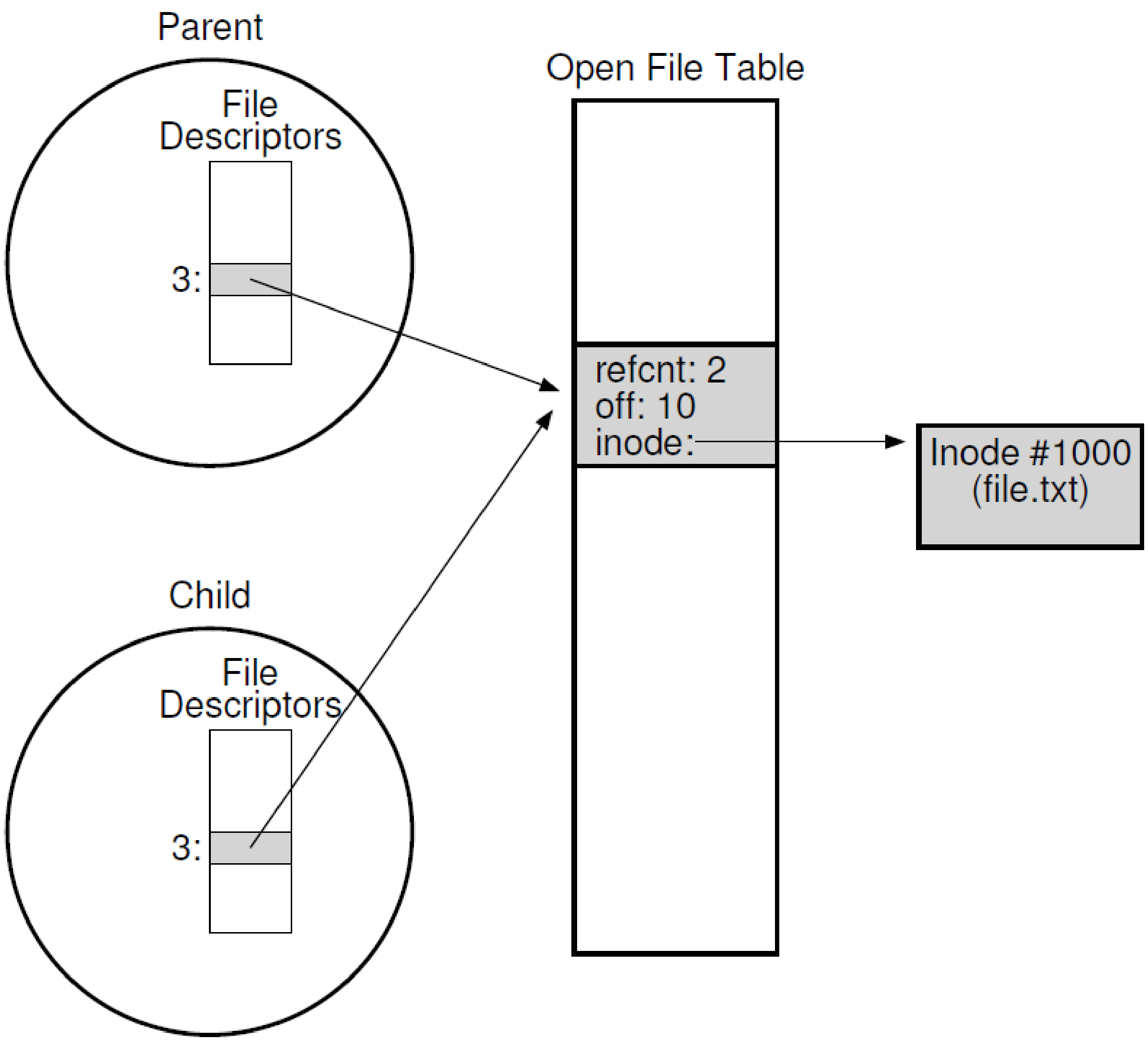
fork-seek.c: The child adjusts the current offset via a call to **lseek**() and then exits. Finally the parent, after waiting for the child, <u>checks the current offset</u> and <u>prints</u>

Sharing File Table Entries: fork()

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

- why is this output?

Sharing File Table Entries: fork()



- The child inherits **ofile** array from the parent
- **ofile[3]** of both processes points to the same OFT entry
- Both update the **same offset**
- When a OFT entry is shared, the **reference count** is incremented
- Only when both close the file (or exit), the OFT entry will be removed

Writing Immediately with `fsync()`

- When a program calls **write**(), it is just telling the file system:
  - please write this data to persistent storage, <u>at some point in the future</u>
- The file system will **buffer** writes in memory for some time.
  - Ex) 5 seconds, or 30
  - Performance reasons
- At that later point in time, the write(s) will **actually be issued** to the storage device.
  - Writes seem to <u>complete quickly</u>.
  - However, in some rare cases data can be **<u>lost</u>** (e.g., the machine crashes).

Writing Immediately with `fsync()`

- However, some applications require more than eventual guarantee.
  - Ex) DBMS requires force writes to disk from time to time.

- `off_t fsync(int fd)`
  - Filesystem forces all dirty (i.e., not yet written) data to disk for the file referred to by the file description.
  - `fsync()` returns **once all** of theses writes are **complete**.

Writing Immediately with `fsync()`

- An Example of `fsync()`.

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

  - In some cases, this code needs to `fsync()` the directory that contains the file `foo`.

Renaming Files

- `rename(char* old, char *new)`
  - Rename a file to different name.
  - It is implemented as an **atomic call**.
    - Ex) Change from `foo` to `bar`:

```
prompt> mv foo bar        // mv uses the system call rename()
```

    - Ex) How to update a file atomically (scenario: text editor):

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

Getting Information About Files

- `stat()`, `fstat()`: Show the file metadata
  - **Metadata** is information about each file.
  - Ex) Size, Low-level name, Permission, …
  - `stat` structure is 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 */
};
```

Getting Information About Files

- To see **stat** information, you can use the command line tool `stat`.

```
prompt> echo hello > ftemp
prompt> stat ftemp
File: ftemp
Size: 6          Blocks: 8        IO Block: 4096   regular file
Device: 10305h/66309d    Inode: 22151497    Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/ endadul)   Gid: ( 1000/ endadul)
Access: 2020-03-16 19:54:57.132234052 -0400
Modify: 2020-03-16 19:54:57.132234052 -0400
Change: 2020-03-16 19:54:57.132234052 -0400
```

  – File system keeps this type of information in a `inode` structure (as a persistent data structure)

Removing Files

- `rm` is Linux command to remove a file
  - `rm` calls `unlink()` to remove a file.

```
prompt> strace rm foo
…
unlink("foo")                  = 0      // return 0 upon success
…
prompt>
```

Why does it call `unlink()`? not "`remove` or `delete`"
We can get the answer later.

System Calls for Directories

- There are system calls to make, read, and delete directories
- Note you **can never write** to a directory <u>**directly**</u>.
- Because the format of the directory is considered <u>file system metadata</u>, the file system considers itself responsible for the **integrity of directory data**;
- You can **only update** a directory <u>**indirectly**</u>
  - by, for example, creating files, directories, or other object types within it.
- In this way, the file system **makes sure** that directory contents are as expected.

Making Directories

- `mkdir()`: Make a directory

```
prompt> strace mkdir foo
…
mkdir(“foo”, 0777)                 = 0
prompt>
```

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

```
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 ../
```

Reading Directories

- A sample code to read directory entries (like `ls`).

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");                  // open current directory
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL)// read one directory entry
    {
        // print out the name and inode number of each file
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);                        // close current directory
    return 0;
}
```

  - The information available within `struct dirent`

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

Deleting Directories

- `rmdir()`: Delete a directory.
  - Require that the directory be **empty**.
    - I.e., Only has "." and ".." entries.
  - If you call `rmdir()` to a **non-empty** directory, it will **fail**.

Hard Links

- `link(old pathname, new one)`
  - **Link** a new file name to an old one
  - Create <u>another way</u> to refer to ***the same file***
  - The command-line link program : `ln`

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

Hard Links

- The way `link` works:
  - **Create** another name in the directory.
  - **Refer** it to the same inode number of the original file.
    - The ***file is not copied*** in any way.
  - Then, we now just have two human names (`file` and `file2`) that both refer to the same file.

Hard Links

- The result of `link()`

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

  - Two files have **same inode** number, but two human name (file, file2).
  - There is **no difference** between `file` and `file2`.
    - Both just link to the underlying metadata about the file.

Hard Links

- Thus, to remove a file, we call `unlink()`.

```
prompt> rm file          # removed 'file'
prompt> cat file2        # Still access the file
hello
```

  - ***link count*** (like **`ref count`** in `struct file`)
    - Track how many different file names have been **linked** to <u>this inode</u>.
    - When `unlink()` is called, the link count decrements.
    - If the link count reaches zero, the filesystem frees the `inode` and related data blocks, thus **truly deletes** the file

Why does `rm` call `unlink()`?

- When we create a file, OS technically does two things
  - First, make a **inode** structure (holds all metadata say, size, disk blocks, and so on)
  - Second, **link** a human-readable name to that file (aka inode) and put that link into a directory

Hard Links

- The result of `unlink()`
  - `stat()` shows the reference count of a file.

```
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 (soft link)

- Symbolic link is more **useful** than Hard link.
  - Hard Link cannot create to a directory.
  - Hard Link cannot create to a file to other partition.
    - Because inode numbers are only **unique** within a file system.

- Create a symbolic link: `ln -s`

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

Symbolic Links

- What is different between *Symbolic link* and *Hard Link*?
  - Symbolic links are **a third type** the file system knows about.

```
prompt> stat file
 ... regular file ...
prompt> stat file2
 ... symbolic link ...      # Actually a file it self of a different type
```

  - The size of symbolic link (`file2`) is **4 bytes**.

```
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
```

  - A symbolic link holds the **pathname** of the linked-to file as the data of the link file.

Symbolic Links

- If we link to a longer pathname, our link file would be 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
```

Symbolic Links

- **Dangling reference**
  - When remove a original file, symbolic link points noting.

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

Making and Mounting a File System

- `mkfs` tool : Make a file system
  - Write an <u>empty file system</u>, starting with ***a root directory***, on to a disk partition.
  - Input:
    - A device (such as a disk partition, e.g., `/dev/sda2`)
    - A file system type (e.g., `ext3`)

Making and Mounting a File System

- `mount()`
  - Take an existing directory as a target **mount point**.
  - Make a new file system available in the directory tree at that point.

  - **Example)**

```
prompt> mount –t ext3 /dev/sda2 /home/users
prompt> ls /home/users
a b
```

    - The pathname `/home/users/` now refers to the root of the newly-mounted directory.

Making and Mounting a File System

- `mount` program: show **what is mounted** on a system.

```
/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)
```

  - `ext3`: A standard disk-based file system
  - `proc`: A file system for accessing information about current processes
  - `tmpfs`: A file system just for temporary files
  - `AFS`: A distributed file system

Permission bits and Access Control

- Will be covered in the next lecture

Reading Material

- **Chapter 39** of OSTEP book – by Remzi and Andrea Arpaci-Dusseau (University of Wisconsin) http://pages.cs.wisc.edu/~remzi/OSTEP/file-intro.pdf

Questions?