



FILE SYSTEMS

Tanzir Ahmed
CSCE 313 Spring 2021

The UNIX File System

- File Systems and Directories
- UNIX inodes
- File protection/access control

Why Study File Systems?

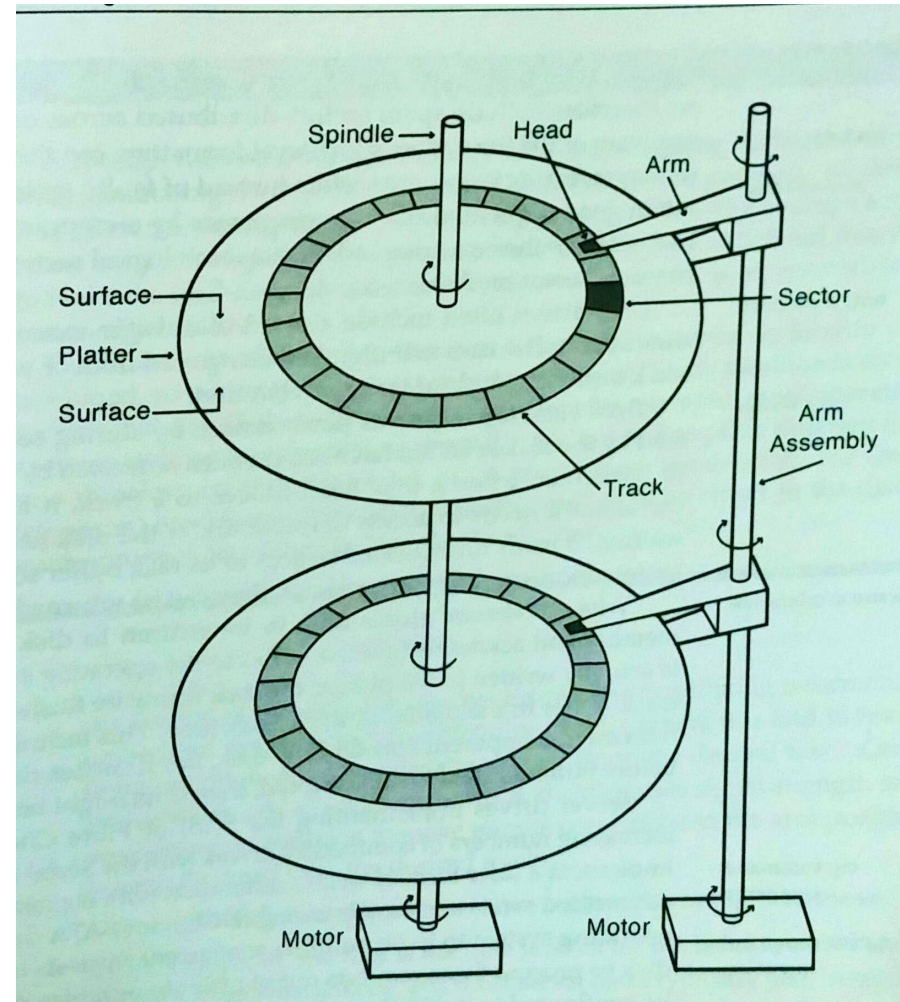
- The main subject is non-volatile storage, e.g.,
 - *Magnetic disk*
 - *Tape drives*
 - *Flash disks*
 - *SSDs*
- These retain data between shutdown and reboots
 - *Important for keeping the OS, programs, and user data*
- But these devices have unique characteristics and limitations not found in main memory
 - First, they *do not allow random accesses* to individual bytes
 - *only allow block level access (512 bytes or multiple)*
 - Second, *accesses are much slower* (10ms compared to 0.1ms of memory)

Why Study File Systems?

- The physical characteristics drive the following keys features of file systems:
 - ***Named data:*** Human-readable names (e.g. file names) given to data
 - ***Performance:*** By grouping, ordering, scheduling disk operations so that high latency is hidden/amortized
 - ***Reliability:*** Unexpected power-cycles do not corrupt the data
 - ***Controlled sharing:*** Determine who can read/write/execute certain files sequentially/simultaneously w/o corrupting

Now, Why Are Disks Slow?

- Consider a typical magnetic disk
 - *Platters are rotating at a constant speed*
 - *Each surface has data in the form of several tracks*
 - *Each track is separated in sectors/blocks*
- To read from disk
 - *Arm moves the disk head to place that on the right track. This is called “seek”*
 - Extremely slow (> 10 ms)
 - *Then, the head waits for the correct sector to come below*
 - Relatively faster because of constant motion (< 10 ms), determined by disk RPM
 - *Read the sector and send to the CPU*
 - Much faster (at SATA rate: $\sim 500\text{MB/s}$)



Disk Access Time

- disk access time = **seek time** + **rotation time** + **transfer time**
- The mechanical movement of the disk arm makes the seek time often the bottleneck
- Sequentially read => less disk seeking
 - *It takes less time to move between adjacent tracks*
- If data is read randomly, the disk arm must seek randomly, leading to very slow operation
 - *Even then, it applies algorithms similar to the ones in elevators to service requests*

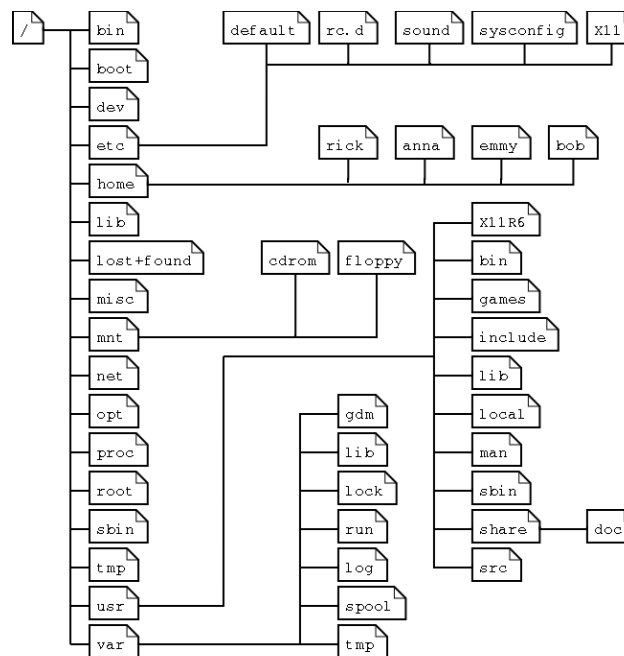
How About SSDs?

- Are they “true random access” – meaning do they take same time as sequential even when accessed randomly?
 - *The answer is NO*
 - *Because accesses are still at sector or page level*
 - *Accessing just 1 byte experiences the full sector latency*
 - *Anticipatory prefetching an adjacent page is a common practice, which hides latency for sequential, but not for random*
 - *Overall, random access is faster than magnetic, but not as fast as sequential*
- From that same logic, even the main memory, which is called Random Access Memory (RAM) not truly random access
 - *You will always have some leverage accessing sequentially*
 - *However, the RAM is much better in random access than any type of disk*

File System Abstraction

- Definition: “File system is an OS abstraction that provides persistent and named data”
- Key components:
 - **Files:** contain *metadata* and actual *data*
 - **Directories:** Special files that contain *names* and *pointers* to actual files

- Because of directories and sub-directories, the File System looks like a **tree**

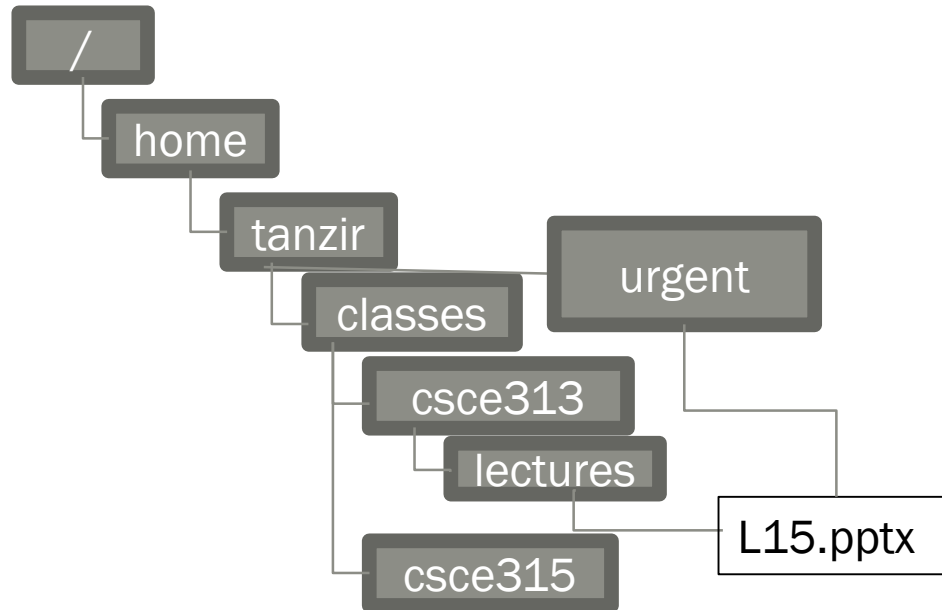


File Systems Terms

- **path:** a string identifying a file (e.g., `/home/tanzir/Work/hw1.cpp`)
- **root directory:** think of the directory as a tree whose root is the *root directory*
 - *Often denoted by the “/”*
- **absolute path:** a path starting with “/”
- **relative path:** a path relative to the **current working directory**. Does not start with a “/”

File System “Tree”

- **hard link:** The mapping between the name and the underlying file
 - *There can be multiple hard links to the same file (e.g., short cuts)*
 - *Means that the directory tree is **not always a tree***



UNIX Directory API: Current Directory

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

```
char * getcwd(char * buf, size_t size);  
/* get current working directory */
```

Example:

```
void main(void) {  
    char mycwd[PATH_MAX];  
  
    if (getcwd(mycwd, PATH_MAX) == NULL) {  
        perror ("Failed to get current working directory");  
        return 1;  
    }  
    printf("Current working directory: %s\n", mycwd);  
    return 0;  
}
```

UNIX Directory API – Open, Read, Close

- Read is **stateful** with a **cursor**
 - *Reading the same directory again gives back the next file in the directory*

```
#include <dirent.h>
int main(int argc, char * argv[]) {
    struct dirent * direntp;
    DIR * dirp = opendir(argv[1]);
    while ((direntp = readdir(dirp)) != NULL)
        printf("%s\n", direntp->d_name);

    closedir(dirp);
    return 0;
}
```

- In the next quiz, you will be asked to implement “ls -l”
 - *That will require many more fields from the `dirent` struct*

UNIX Directory API – Traversal

- Read is stateful with a cursor
 - *Reading the same directory again gives back the next file in the directory*
 - *You can even do “seek” to the beginning using `rewindir()`*

```
#include <dirent.h>

DIR* opendir(const char * dirname);
/* returns pointer to directory object */
struct dirent * readdir(DIR * dirp);
/* read successive entries in directory 'dirp' */
int closedir(DIR *dirp);
/* close directory stream */
void rewinddir(DIR * dirp);
/* reposition pointer to beginning of directory */
```

File System Organization

- First, each sector/block is numbered from 0, 1,
 - *The larger the disk, the more the sectors*
- Then, the file system contains the following components:

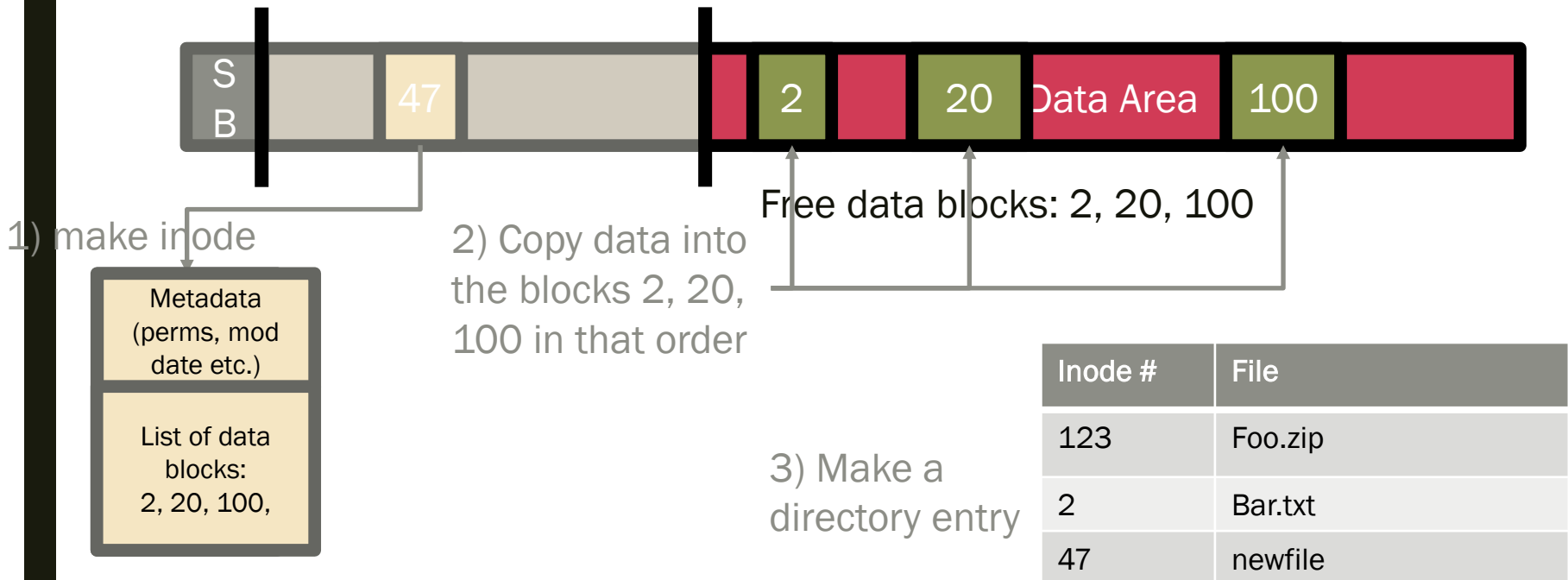
Component	Purpose
Superblock	Contains metadata about the file system. Size is OS dependent
Inode table	An array of inode structs, where each struct contains info of a file (e.g., size, owner id, last modification). Each inode has an identification number, which is also the index into the inode table
Data Area	Contains file content. Each file can be ≥ 1 block



Creating a New File

Let us create a file called “**newfile**” that is 12KB in size, and a disk block is 4KB.

First, we need a free inode to put the file metadata, then find 3 free disk blocks to put the actual data



Steps in Creating a File

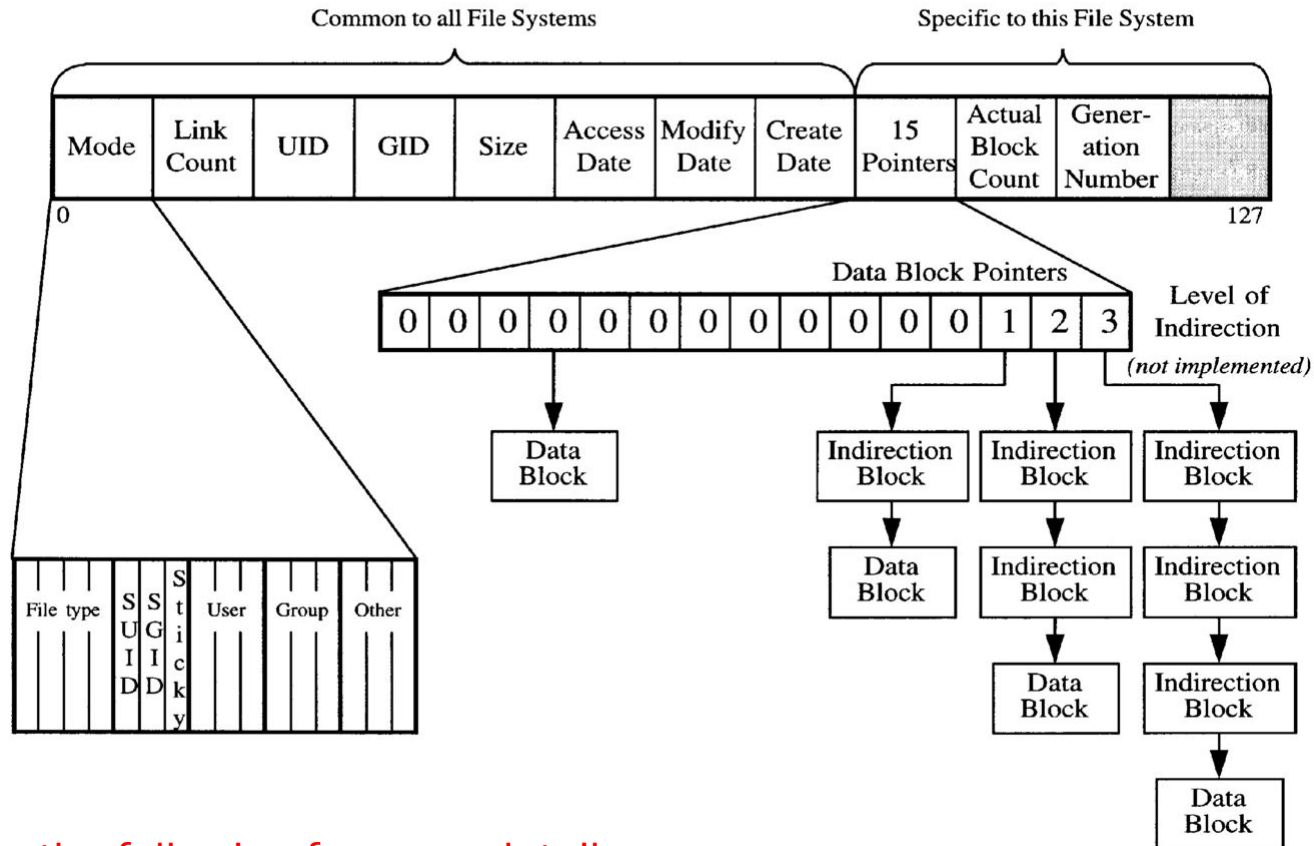
- 1) Store Properties: Kernel looks for a free inode and stores the metadata (e.g., permissions, size, creation date) in that
- 2) Store Data and Record Allocations: Kernel then looks for free disk blocks (and enough of those) and copies content there. Kernel updates the inode with which blocks contain data for that file
- 3) Add file name to directory: Kernel stored the (inode#, filename) pair in the directory entry

Reading a File

- 1) Search the directory for the file name and extract its inode
- 2) Locate and read the inode, find the data block number from there
- 3) Read each data block in sequence and output that
- This is how out `"$cat newfile"` command will work
 - *Output will go to standard output*

What goes inside a inode??

Disk Inode



See the following for more details:

<http://man7.org/linux/man-pages/man7/inode.7.html>

Inode's Features: Protection

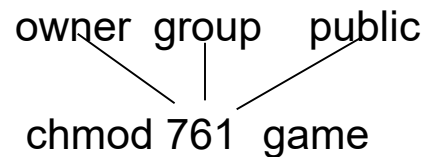
- File owner/creator should be able to control:
 - *what can be done*
 - *by whom*
- Types of access
 - *Read*
 - *Write*
 - *Execute*
 - *Append*
 - *Delete*
 - *List*

Access Lists and Groups

- Mode of access: read, write, execute
- Three classes of users

			RWX
a) owner access	7	⇒	1 1 1
			RWX
b) group access	6	⇒	1 1 0
			RWX
c) public access	1	⇒	0 0 1

- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file: `chgrp G game`

NITIN AGRAWAL
University of Wisconsin, Madison
and

WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH
Microsoft Research

Characteristics of Files

Observations:

- *Most files are small*
- *Most of the space is occupied by the rare big ones*

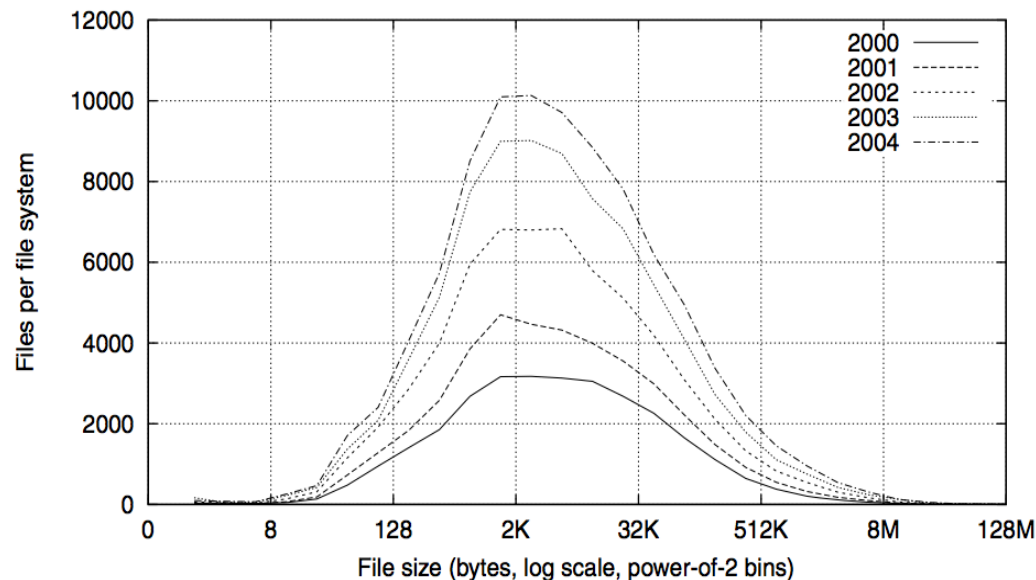
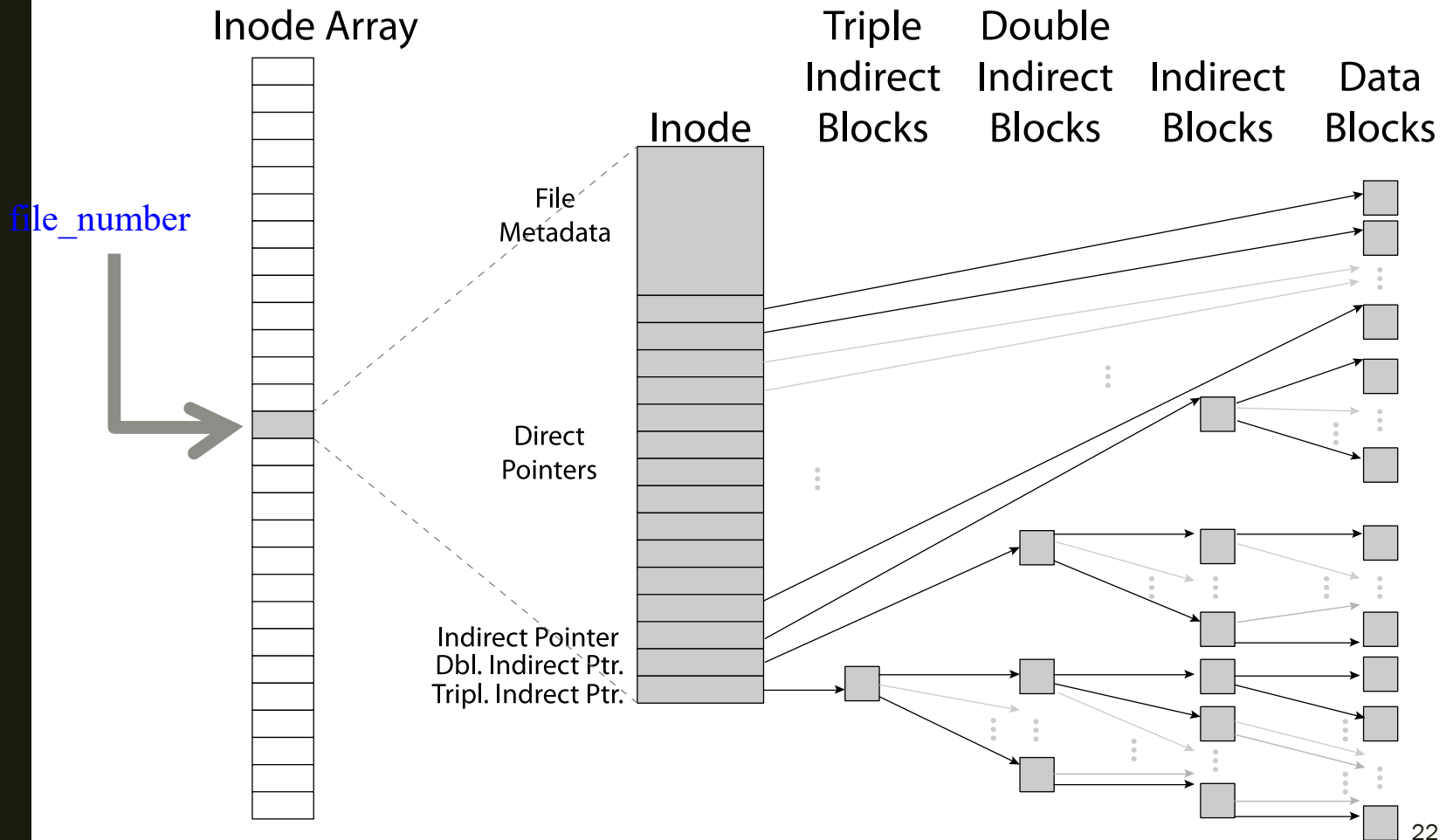


Fig. 2. Histograms of files by size.

Does this help deciding the inode structure?



FFS: Data Storage

Small files: 12 pointers direct to data blocks

Inode Array

Triple Indirect Blocks Double Indirect Blocks Indirect Blocks Data Blocks

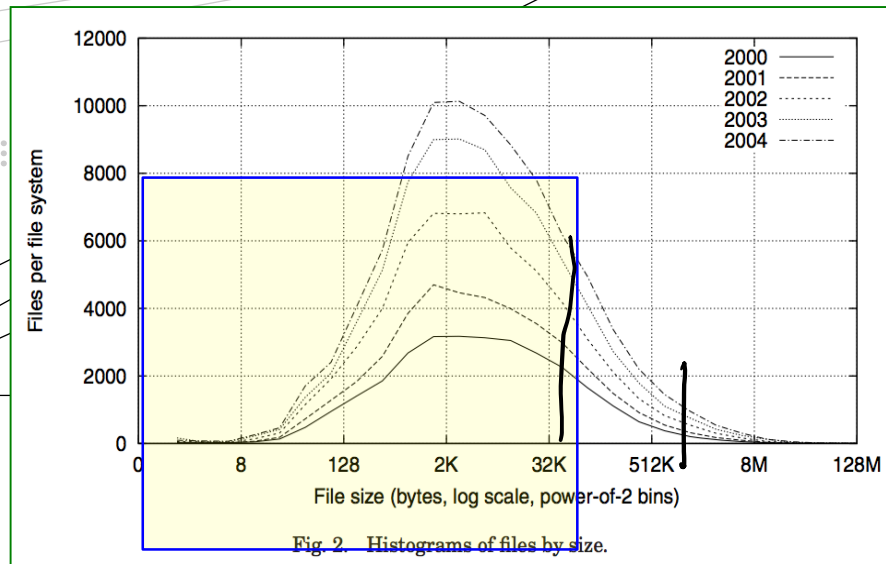
Inode

Direct Pointers

Indirect Pointer
Dbl. Indirect Ptr.
Tripl. Indirect Ptr.

Direct pointers

With 4kB blocks, sufficient
For files up to 48KB



FFS: Data Storage

- Large files: 1,2,3 level indirect pointers

Inode Array

Indirect pointers

- point to a disk block containing only pointers
- eg. 4 kB blocks \Rightarrow 1024 pointers \Rightarrow 4 MB @ level 2
- \Rightarrow 4 GB @ level 3
- \Rightarrow 4 TB @ level 4

A Five-Year Study of File-System Metadata • 9:9

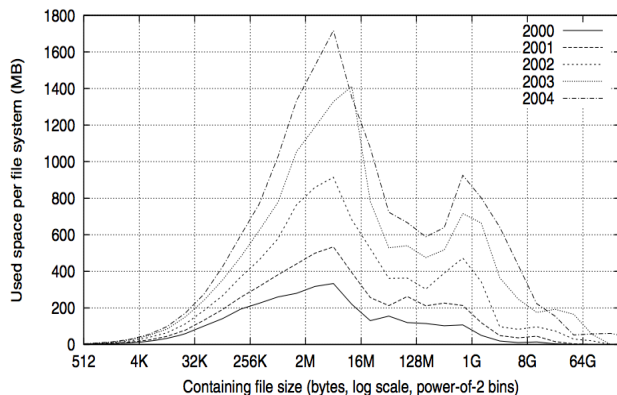
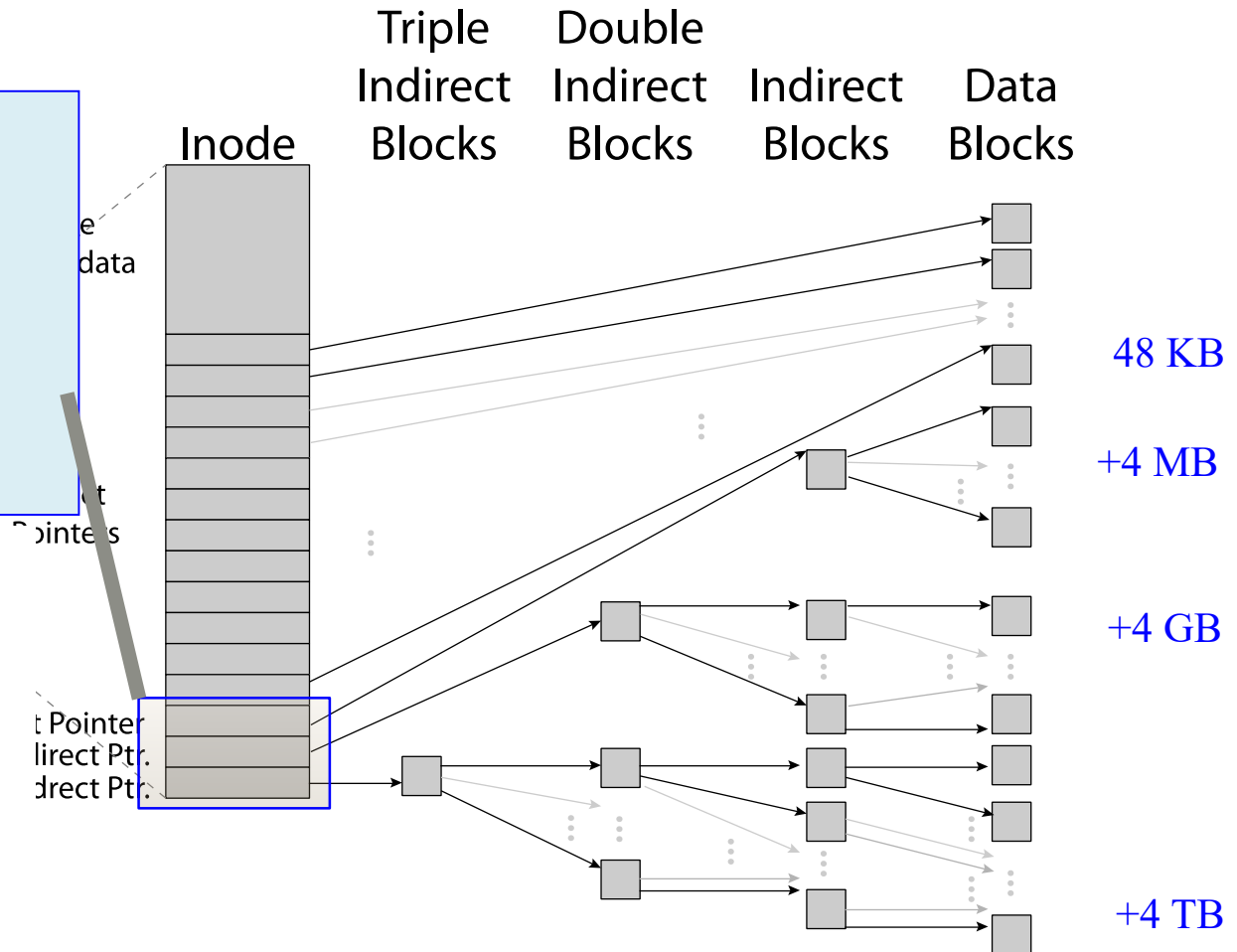
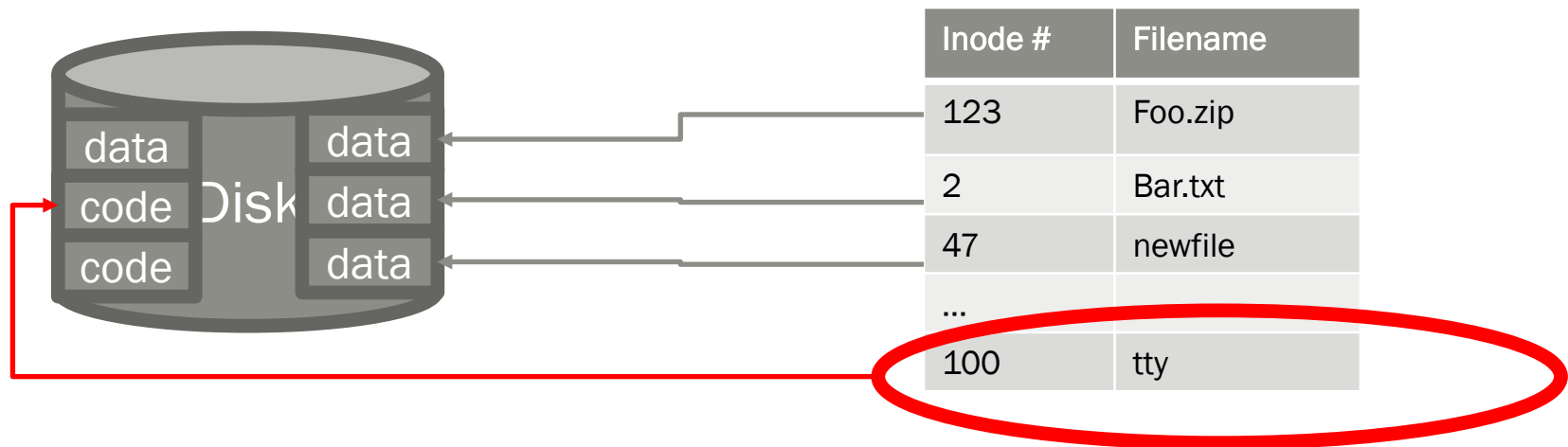


Fig. 4. Histograms of bytes by containing file size.



Device Files and inodes

- How to handle device files (e.g., terminal input, output, printer)?



Inode's for device files and regular files are different:

- They both point to disk blocks
- However, device files' inodes point to **device driver code** instead of regular data blocks

Links

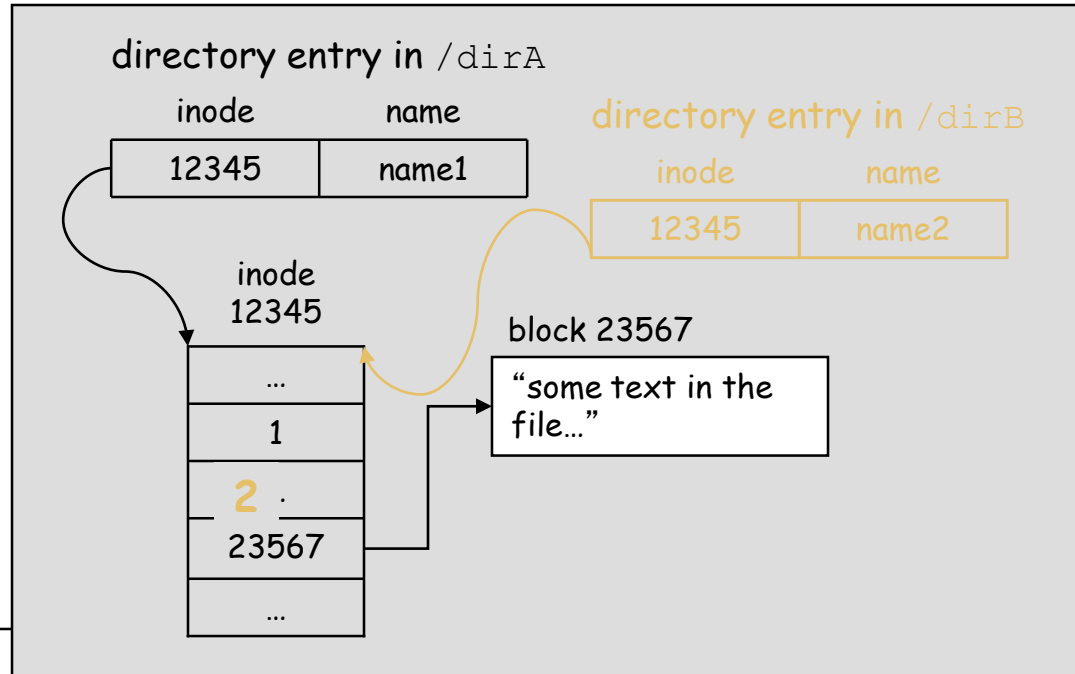
- Hard link

- *Directory entry contains the **inode number***
- *Creates another name (path) for the file*
- *Each is “first class”*

- Soft link or Symbolic Link

- *Directory entry contains the **name of the file***
- *Map one name to another name*

Hard Links



shell command

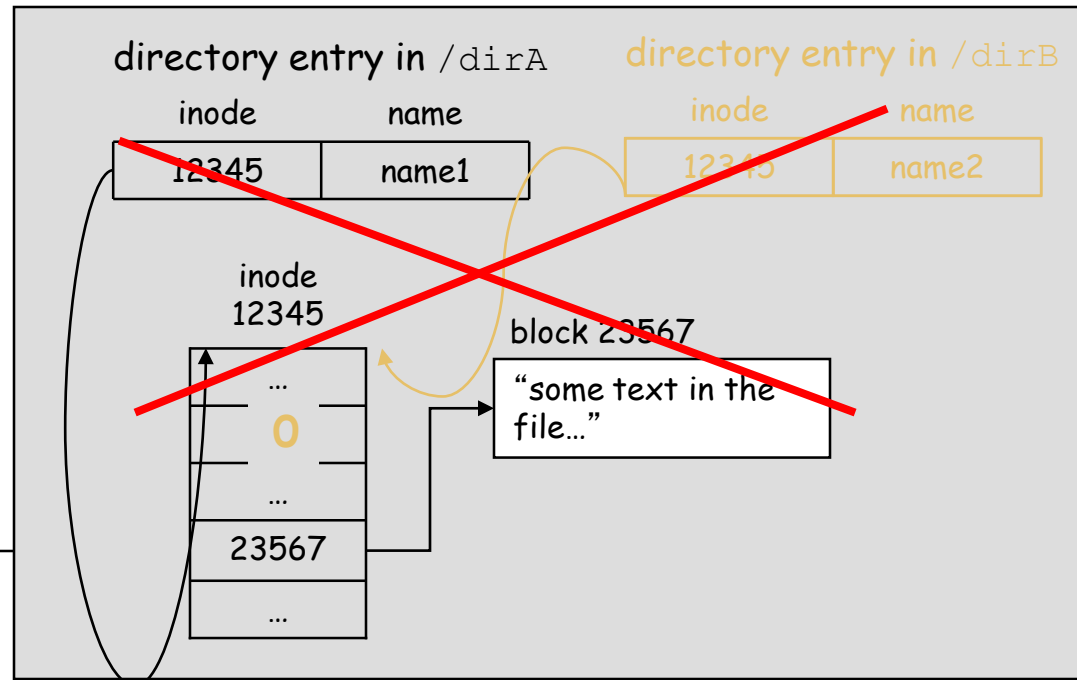
```
ln /dirA/name1 /dirB/name2
```

is typically implemented using the link system call:

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

```
if (link("/dirA/name1", "/dirB/name2") == -1)
    perror("failed to make new link in /dirB");
```

Hard Links: unlink

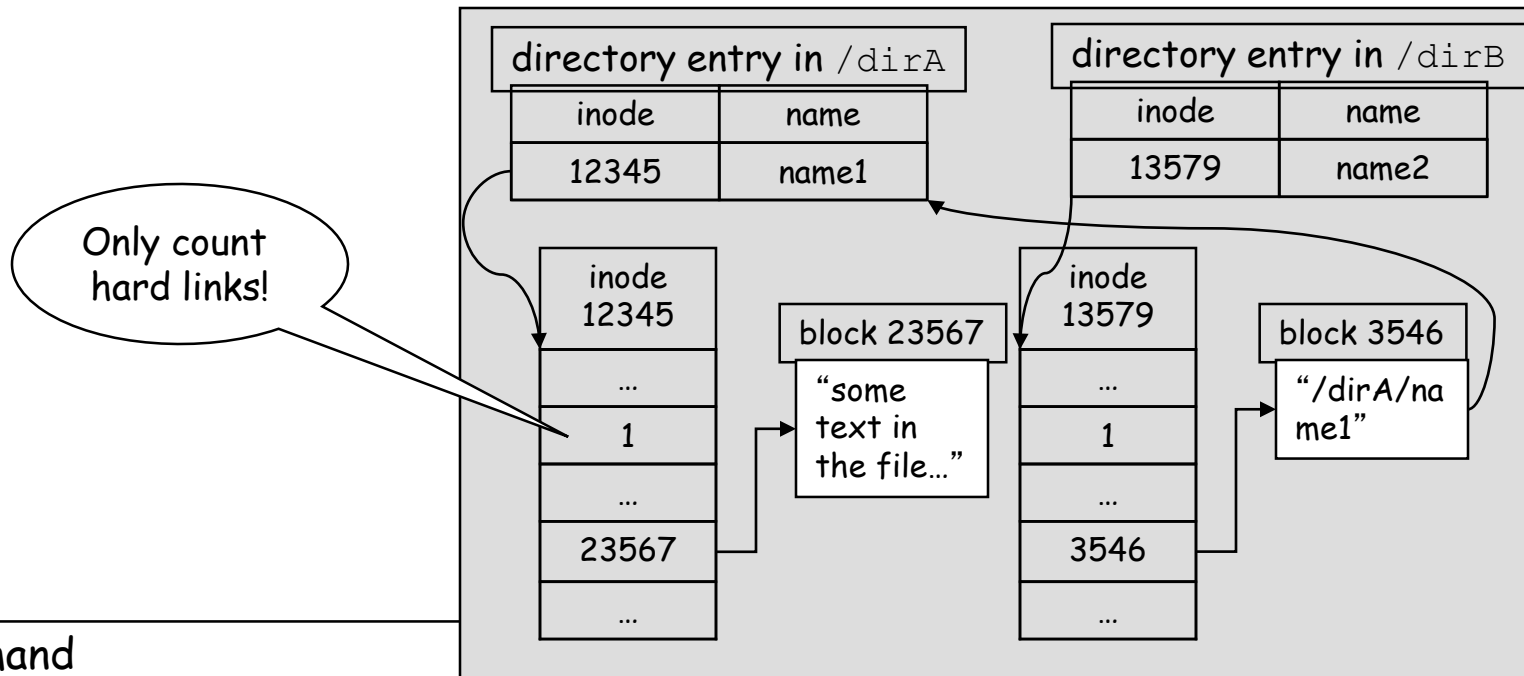


```
#include <stdio.h>
#include <unistd.h>
```

```
if (unlink("/dirA/name1") == -1)
    perror("failed to delete link in /dirA");
```

```
if (unlink("/dirB/name2") == -1)
    perror("failed to delete link in /dirB");
```

Symbolic (Soft) Links



shell command

```
ln -s /dirA/name1 /dirB/name2
```

is typically implemented using the link system call:

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

```
if (symlink("/dirA/name1", "/dirB/name2") == -1)
    perror("failed to create symbolic link in /dirB");
```

Links: Example

```
[tanzir@compute PA7]$ ls -lrt
total 16
-rwxr-xr-x. 1 tanzir CSE_csfac 2251 Apr  7 03:21 semaphore.h
-rwxr-xr-x. 1 tanzir CSE_csfac  520 Apr  7 03:26 BoundedBuffer.h
-rwxr-xr-x. 1 tanzir CSE_csfac  765 Apr  7 03:28 BoundedBuffer.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac  533 Apr 19 20:20 makefile
-rwxr-xr-x. 1 tanzir CSE_csfac 1445 Apr 19 21:02 netreqchannel.h
-rwxr-xr-x. 1 tanzir CSE_csfac 3784 Apr 21 11:31 netreqchannel.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 1012 Apr 21 11:46 dataserver.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 7631 Apr 21 11:57 client.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 1576 Apr 24 17:39 Untitled-1.cpp
[tanzir@compute PA7]$ ln semaphore.h ./sema.h
[tanzir@compute PA7]$ ln -s semaphore.h ./softsema.h
[tanzir@compute PA7]$ ls -lrt
total 18
-rwxr-xr-x. 2 tanzir CSE_csfac 2251 Apr  7 03:21 semaphore.h
-rwxr-xr-x. 2 tanzir CSE_csfac 2251 Apr  7 03:21 sema.h
-rwxr-xr-x. 1 tanzir CSE_csfac  520 Apr  7 03:26 BoundedBuffer.h
-rwxr-xr-x. 1 tanzir CSE_csfac  765 Apr  7 03:28 BoundedBuffer.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac  533 Apr 19 20:20 makefile
-rwxr-xr-x. 1 tanzir CSE_csfac 1445 Apr 19 21:02 netreqchannel.h
-rwxr-xr-x. 1 tanzir CSE_csfac 3784 Apr 21 11:31 netreqchannel.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 1012 Apr 21 11:46 dataserver.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 7631 Apr 21 11:57 client.cpp
-rwxr-xr-x. 1 tanzir CSE_csfac 1576 Apr 24 17:39 Untitled-1.cpp
lrwxrwxrwx. 1 tanzir games 11 Apr 26 19:34 softsema.h -> semaphore.h
```

Files: Big Picture

