Recap

HDD

- Interface: an address space divided into sectors

- Main factors impacting performance: mechanical movement
  - Seek time
  - Rotational latency

Examples

	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

Scheduling

Multiple processes making concurrent requests can result in frequent seeking

Order of requests by an application can also increase seek time

Solution is to use a scheduling policy

- Maintain an ordered queue of I/O requests by track
- **Shortest Seek Time First (SSTF)** – pick requests on the nearest track to complete first

Requests (track #s)	1, 20, 5	6	25	4	10	3	2	40		
Requests serviced	1	5	6	4	10	3	2	20	25	40

Scheduling

Solution is to use a scheduling policy (cont')

- Maintain an ordered queue of I/O requests by track
- **SCAN (Elevator)** – the head moves across the tracks in order (sweep 0 -> n, sweep n -> 0, sweep 0 ->n, …) serving requests along the way

Requests (track #s)	1, 20, 5	6	25	4	10	3	2	40		
Requests serviced	1	5	6	20	25	10	4	3	2	40

- **F-SCAN** – a variant of SCAN, which freezes the queue to be serviced when it is doing a sweep (i.e., moving from track 0->n or from n->0); more fair

Requests (track #s)	1, 20, 5	6	25	4	10	2	3	40		
Requests serviced	1	5	20	25	6	4	10	3	2	40

- **C-SCAN (Circular SCAN)** - only sweeps from outer-to-inner and then resets at the outer track to begin again; more fair

Requests (track #s)	1, 20, 5	6	25	4	10	3	2	40		
Requests serviced	1	5	6	20	25	3	4	10	40	2

Files and Directories

(based on Ch. 39)

Slides revised from
Matthew Tancreti
Iowa State University

Files and Directories

File systems provide **persistent storage**

Users have expectation that persistent data is kept intact despite potential system crashes or power outages

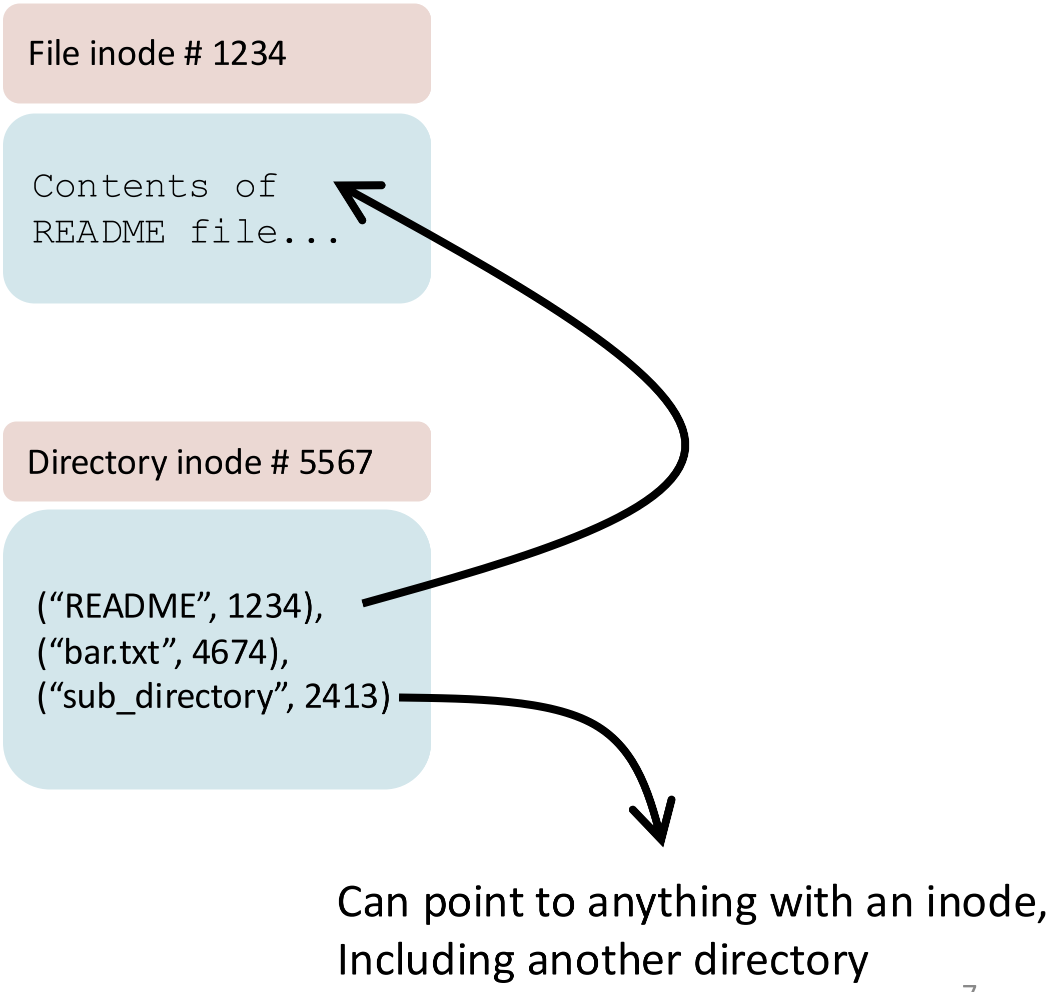
Users also want to be able to organize, search and secure data

What is the file system API?

Files and Directories

File – an array of bytes given an identifier (**inode number** in Unix/Linux)

Directory – a list of (user-readable name, inode number) pairs that is also given an inode



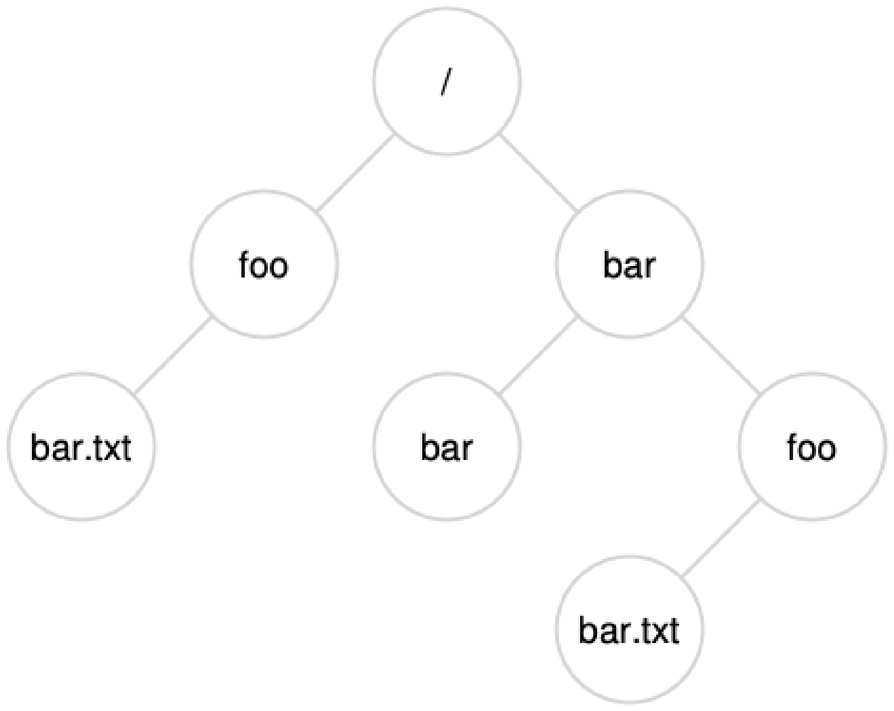
Directory Tree

Directory points to files or other directories resulting in a **directory tree**

The head of the tree is the **root directory**, often referred to as **/**

Same / separator is used to name subsequent directories

We can name a file by its **absolute path** starting at root, for example: **/bar/foo/bar.txt**



POSIX File System API

Creating File

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

Open system call takes a file name and options. The file will be opened in the **current working directory**.

Options:

- O_CREAT – create the file if it does not exist
- O_RDWR – allow read/write from/to the file
- O_TRUNC – zero out the contents of the file
- S_IRUSR – user (caller) has read permission
- S_IWUSR – user (caller) has write permission

Returns a **file descriptor** – an identifier the process uses to reference a resource when making system calls (read(), write(), etc.). Think of file descriptor as pointer to an object that stores information about an open file.

Open File Descriptors

Information about opened files are stored in the Process Control Block (PCB)

In xv6, the file descriptor is used as an index to an array of open files, the array is stored in the process structure (i.e., PCB)

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

File Read and Write

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

Attemps to read up to count bytes from file descriptor fd into the buffer starting at buf.

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

Writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd. The write is promised to happen but may not right now.

int fsync(int fd);

Transfers (flushes) all modified data of the file referred to by the file descriptor fd to the disk device so that all changed information can be retrieved even if the system crashes or is rebooted. Zero is returned on success.

Close File Descriptor

int close(int fd);

Closes a file descriptor, so that it no longer refers to any file and may be reused.

Tracing System Calls of Example Program

```
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>
```

Sequential vs Random Read/Writing

By default, the first read/write begins at byte 0 of the file. Every read/write after that starts at the next byte following the previous read/write.

Change the location of the next read/write with lseek.

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

Whence describes how to apply the offset

- 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.

Support for Radom Access

Process Control Block (PCB)

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

Open file

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

offset for next read/write

File Sequential Access

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	--

File Sequential Access with 2 File Descriptors

System Calls	Return Code	OFT[10].CO	OFT[11].CO
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	--	--

File Radom Access

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	--

Sync/Flush Writes

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

Support for Concurrent Access

What if multiple processes access file at same time? Race condition?

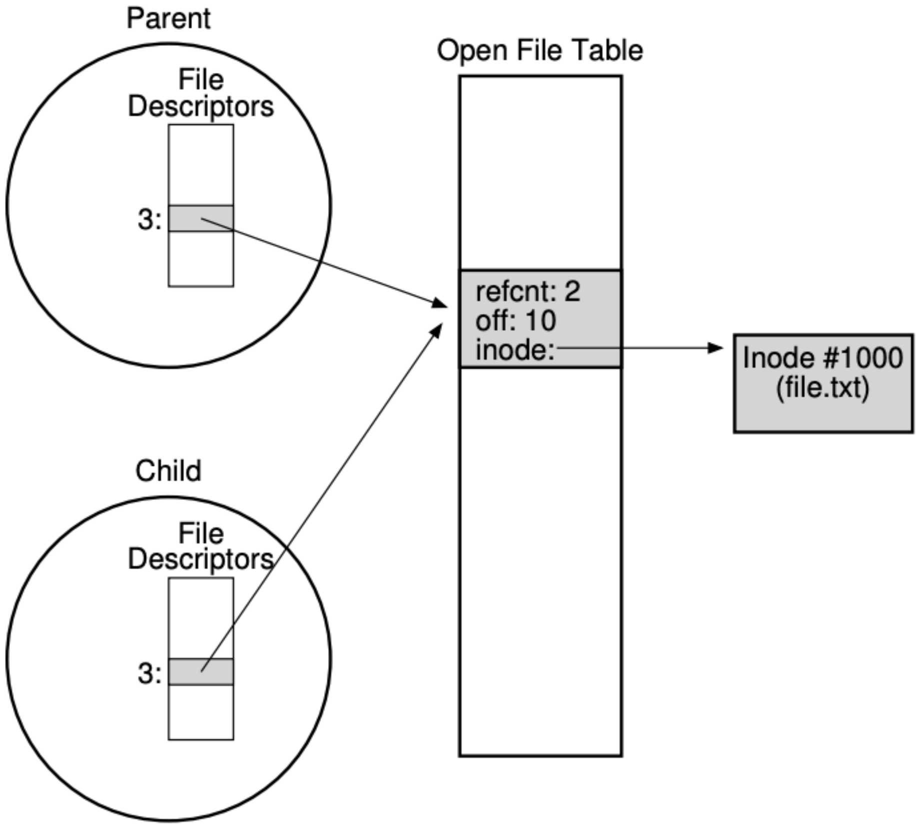
- Each process has a list of pointers to open files in its PCB
- The kernel also keeps a list of all files open by processes

xv6 uses this list of open files to place a lock such that only one process can be accessing a file at one time

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

Shared File Table Entries

A child inherits its parents open file descriptors from the fork



Permission Bits and Access Control

Files are owned by a user and a group

Permissions for read, write and execute are specified for the user, group and all other users

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

permissions user group file size date modified

Command Line

Useful Linux commands for files and directories

mv – rename a file

rm – remove a file

mkdir – make a directory

ls – list the contents of a directory

rmdir – remove a directory

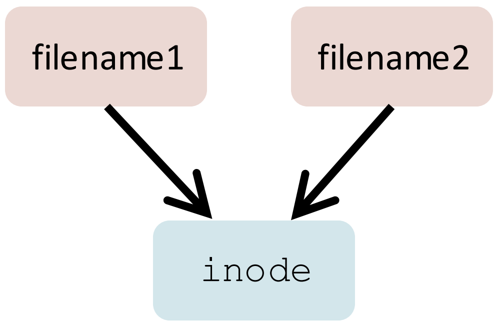
stat – display file or file system status

Hard Links vs Soft Links

Hard link connects a filename to an inode

`ln filename1 filename2`

make filename2 point to the same inode as filename1



Soft (symbolic) link gives an alias to a name

`ln -s filename1 filename2`

Make filename2 another name for filename1