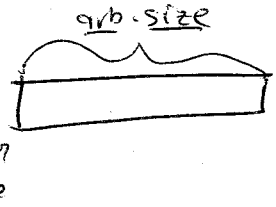
Files

- File / ops
- Meta-Data
- Directories
- Links

Understand disks: SCSI
> remember interface - blocks, read/write
> why not just let users
> use disk (read/write blocks)
> protection, convenience, performance, fair sharing

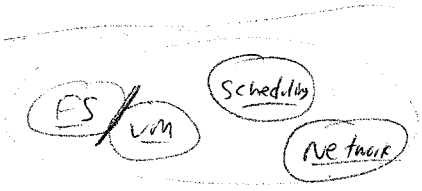
File

User : "named collection of bytes"
=> array of bytes (ordering)
=> randomly accessible
=> permanent (lives across reboots, etc.)
=> convenient



OS : must convert disk/tape blocks => file abstraction
=> must manage disk effectively

Role of FS



Naming
how to select file
find blocks

Protection
only certain users to to certain things

Reliable / Available
don't lose info

Efficient
space
time

File Ops

		unix
Create	find space, allocate name	creat(), open(, O_CREAT)
Access	read, write, seek	open( ) read(), write(), lseek() close( )
Delete	free space	unlink()
Truncate	Set size to 0 (or some lesser size)	open ("file", O_TRUNC), truncate
Rename		rename()
		chmod()

Meta-Data : FS keeps info about each file

- Name of file, type?
- Pointer(s) to data blocks
- Size
- Access / Mod. Time
- Owner, group
- Protection info
- Other: Special file (directory / sym link)

Unix e.g.

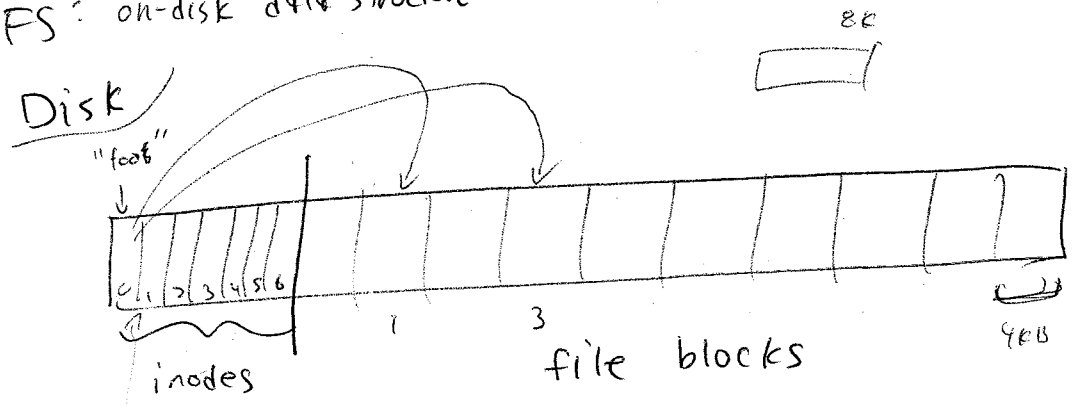
```
ls -lig file
```

<inode #>	<protection bits>	<numLinks>	<username owner>	<uid>	<gid>	<date>	<name>
1000	-rw-rw----	1	remzi	30000	40000	Mar 1,	file

FS must track meta info / file

inode : in Unix

FS: on-disk data structure + access methods



index # (inode number OR i-number)

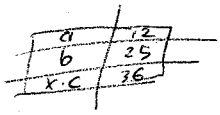
Directories

> Organizing files
Logical grouping

> Single-level directories
Single directory for whole disk
each file unique
Predicate part of disk for directory
(name, index#) pairs

> Two-level
Directory per user VAX/VMS
/remzi/ file1.c
Somewhat more flexible

> Arbitrary Tree e.g. /a/b/c/file.c
> Directory: just a special file
data: <file name, inode #> pairs => name not in inode but in dir => inode
e.g. directory "a" is b/ c/ x.c



> How to distinguish "file" from "dir"?
bit in meta-data
> Special separator used to parse path "/"
> Special directories
Root: top of tree, known index number (2)
.: this dir
..: parent dir
/a/b/./c/../d/foo.c
> E.g. mkdir /a/b/c
root 2: find "a" => 5
5: find "b" => 9
9: ensure "c" doesn't exist
allocate new inode (12), add "c" to "b"'s data

Opening Files

fd = open ("file", O_RDONLY);

read / write (fd, buffer, size); OR

close (fd);

read ("/a/b/c", buffer, size);

expensive

Solution: open file first

search directories, find file

make entry in "open file" table

return index into table (file descriptor / handle)

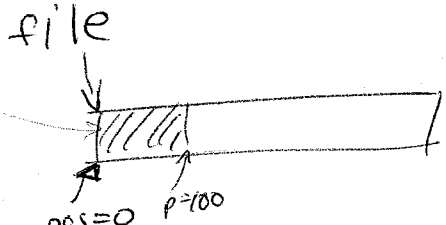
use index to access file

reads/writes => fast

Per-process open file table

current "position" in file

index into system open file table



read (fd, buffer, 100);

lseek (fd, pos, SEEK_SET);

just moves pointer

Global "open file" table

shared

keeps count: how many processes have file open?

Links

> Might want 2/+ names to refer to same file

(could copy ...)
⇒ Links

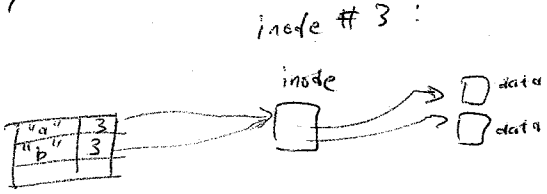
> simple example
> netscape → netscape-4.6.3
→ explorer
> two users sharing same file

Hard link:

("link" system call)

e.g. ln srcFile newLinkFile

⇒ creates another directory entry w/ same index #



remove a file (unlink)
decrement ref count
deallocate when ref == 0

can't refer to directories
(avoid cycles in tree)

Symbolic Link

ln -s srcFile newLinkFile

special new file (bit in meta-data)
contents of file:
name of other file

what happens if original is removed?

Path Names

Absolute Path : start at root /full name

/x/y/z ⇒ cumbersome

Relative : shell

notion : "current" directory / process

look for files in current dir.

cd /x/y/z

open("foo"); ⇒ full path?

open("b/foo.c") ⇒ ?

open("/foo");

Short cuts

~remzi (shell ⇒ /home/remzi)

etc.

Protection

> Types of access

Read, write, execute (run), append, delete, list, ...

> Access Control List (ACL)

List of users w/ rights per file

very flexible (AFS)

cumbersome, hard to manage, does not scale

> Simpler : Bits

owner, group, everybody rwx rwx rwx

easy to implement

not as general