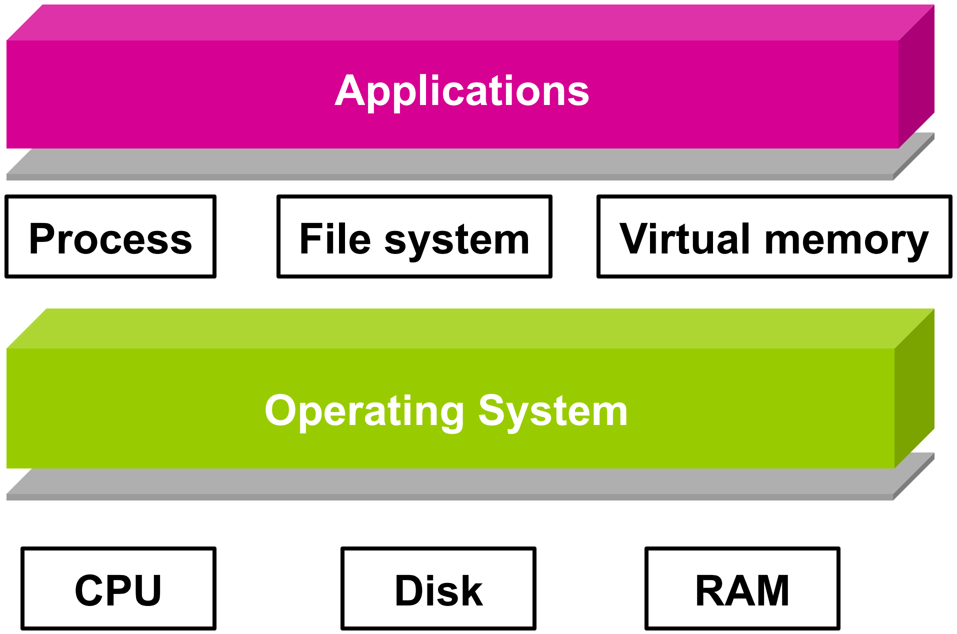
CS 153
Design of Operating Systems

Fall 20

Lecture 24: File Systems

Instructor: Chengyu Song

OS Abstractions



File Systems

- File systems
  - Implement an abstraction (files) for persistent storage
  - Organize files logically (directories)
  - Permit sharing of data between processes, people, and machines
  - Protect data from unwanted access (security)

Files

- A file is a sequence of bytes with some properties
  - Owner, last read/write time, protection, etc.
- A file can also have a type
  - Understood by the file system
    - » Block, character, device, portal, link, etc.
  - Understood by other parts of the OS or applications
    - » Executable, library, source, object, text, etc.

File Types

- A file’s content type can be encoded in its name or contents
  - Encodes type in name
    - » .com, .exe, .bat, .dll, .jpg, etc.
  - Encodes type in contents
    - » Magic numbers, initial characters (e.g., #! for shell scripts)
- Q: are these encoding method reliable?
  - No! Check polyglot files.

Basic File Operations

Unix

- creat(name)
- open(name, how)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)

NT

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

Directories

- Directories serve two purposes
  - For users, they provide a structured way to organize files
  - For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk
- Most file systems support multi-level directories
  - Naming hierarchies (/, /usr, /usr/local/, …)
- Most OS support the notion of a current directory
  - Relative paths specified with respect to current directory
  - Absolute paths start from the root of directory tree

Directory Internals

- A directory is a list of entries
  - <name, location>
  - Name is just the name of the file or directory
  - Location depends upon how file is represented on disk
- List is usually unordered (effectively random)
  - Entries usually sorted by program that reads directory
- Directories typically stored in files
  - Only need to manage one kind of secondary storage unit

Basic Directory Operations

Unix

- Directories implemented in files
  - Use file ops to create dirs
- C runtime library provides a higher-level abstraction for reading directories
  - opendir(name)
  - readdir(DIR)
  - seekdir(DIR)
  - closedir(DIR)

Windows

- Explicit dir operations
  - CreateDirectory(name)
  - RemoveDirectory(name)
- Very different method for reading directory entries
  - FindFirstFile(pattern)
  - FindNextFile()

Path Name Translation

- Let’s say you want to open “/one/two/three”
- What does the file system do?
  - Open directory “/” (well known, can always find)
  - Search for the entry “one”, get location of “one” (in dir entry)
  - Open directory “one”, search for “two”, get location of “two”
  - Open directory “two”, search for “three”, get location of “three”
  - Open file “three”
- Q: what about “..” and “.”?

Path Name Translation

- Systems spend a lot of time walking directory paths
- Q: what can we do to make this faster?
  - Add fd = open(path) and change read/write to use fd
  - OS will cache prefix lookups for performance
    - /a/b, /a/bb, /a/bbb, etc., all share "/a" prefix

File Sharing

- File sharing is important for getting work done
  - Basis for communication between processes and users
- Two key issues when sharing files
  - Semantics of concurrent access
  - Protection

Concurrent Access

- Based on your understanding about race conditions
  - What happens when one process/thread reads while another writes?
  - What happens when two processes/threads open a file for writing?
- POSIX 2.9.7 Thread Interactions with Regular File Operations
  - All of the functions chmod(), close(), fchmod(), fcntl(), fstat(), ftruncate(), lseek(), open(), read(), readlink(), stat(), symlink(), and write() shall be atomic with respect to each other in the effects specified in IEEE Std 1003.1-2001 when they operate on regular files. If two threads each call one of these functions, each call shall either see all of the specified effects of the other call, or none of them.

Protection

- File systems implement some kind of protection system
  - Who can access a file
  - How they can access it
- More generally…
  - Objects are “what”, subjects are “who”, actions are “how”
- A protection system dictates whether a given action performed by a given subject on a given object should be allowed
  - You can read and/or write your files, but others cannot
  - You can read “/etc/motd”, but you cannot write to it

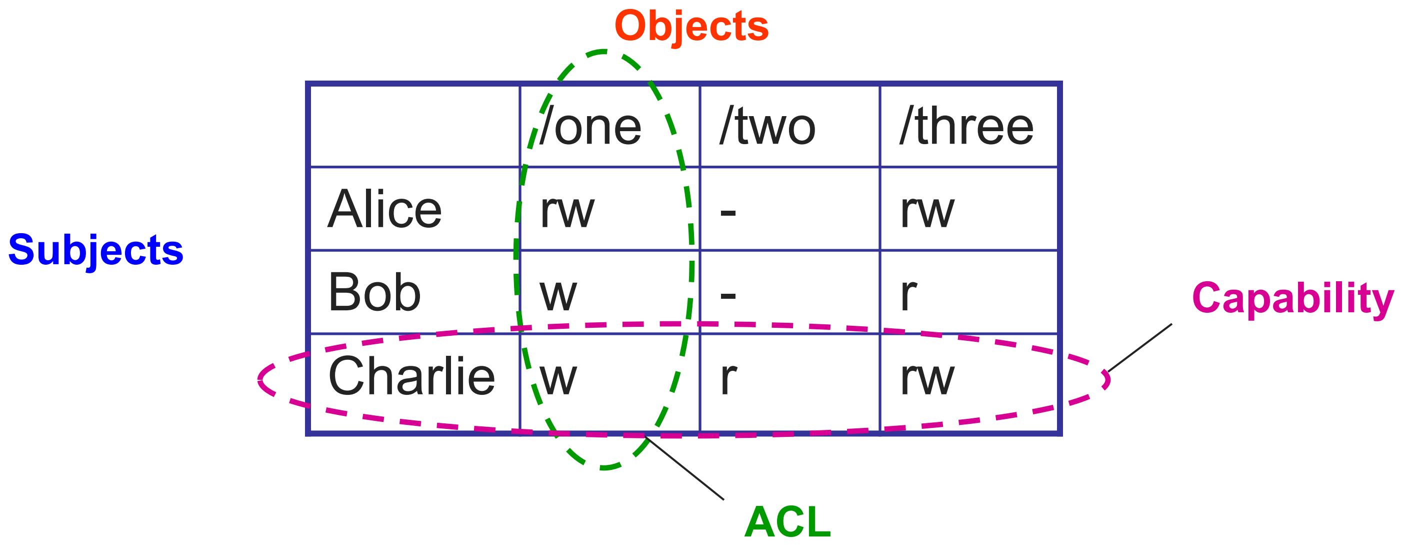
Representing Protection

Access Control Lists (ACL)

- For each object, maintain a list of subjects and their permitted actions

Capabilities

- For each subject, maintain a list of objects and their permitted actions



	/one	/two	/three
Alice	rw	-	rw
Bob	w	-	r
Charlie	w	r	rw

ACLs and Capabilities

- The approaches differ only in how table is represented
  - What approach does Unix use?
  - What approach does Mobile permission system use?
- Capabilities are easier to transfer
  - They are like keys, can handoff, does not depend on subject
- In practice, ACLs are easier to manage
  - Object-centric, easy to grant, revoke
  - To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem
- ACLs have a problem when objects are heavily shared
  - The ACLs become very large

Unix Access Control Model

- Unix uses ACLs
  - Rights are associated with objects
    - » Essentially files, everything is a file in Unix
- Three sets of rights
  - Owner, Group, Others
  - Use groups to limit the length of ACL
    - » Files (objects) can only belong to one group
    - » Process (subjects) can belong to multiple group

```
$ ls -l fs.pdf
-rw-r--r--  1 csong  csprofs  fs.pdf
$ id
uid=123(csong) gid=12(csprofs) groups=12(csprofs),34(csoffice)
```

Unix Access Rights

- Files
  - Literal: read, write, execute (rwx)
- Directories
  - Read: read/list file names, but not metadata
  - Write: create, rename, and delete files
  - Execute: read file metadata and change current directory to it
- Who can set the access rights and group?
  - Owner!

Discretionary Access Control

- Both ACLs and capabilities are DAC
- DAC has a special user: root/Administrator
  - root is not subject to access rights
    - They can read/write/execute any file
  - root can change the owner/group of any file
  - root can change the access rights of any file
  - Why?
- This is why attackers like root

Users and Processes

- Although ACLs use users as subjects, the OS actually uses processes as subjects
  - Processes act on the behalf of users, like a **trusted** proxy
- Then how to decide the identity (user/group) of a process?
  - R1: by default, a newly created process inherits its parent process' identity, unless R2 or R3
  - R2: a process can change its identity through setuid()/setgid() system calls
  - R3: exec a setuid/setgid program will change the identity of the process to the owner/group of the executable file

Boot and login

- Process tree in Unix
    - The first process is init
        - Created with root privilege
        - Spawns other daemons (services), including the login daemon
    - The login daemon
        - Authenticates the user
        - Forks itself, changes its identity to the authenticated user
        - Starts a shell (desktop GUI)

Security Problems of DAC

- Root/Administrator has too much power
  - Solution: mandatory access control (MAC)
- Programs could be malicious
  - Solution: sandbox

File System Layout

How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
    - Disk space is allocated in granularity of blocks
- A “Master Block” determines location of root directory
    - At fixed disk location, sometimes replicated for reliability
- A free map determines which blocks are free, allocated
    - Usually a bitmap, one bit per block on the disk
    - Also stored on disk, cached in memory for performance
- Remaining blocks store files (and dirs), and swap!

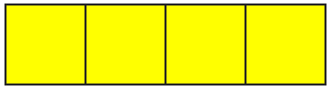
File systems

- File system design: how to allocate and keep track of files and directories
- Does it matter? What is the difference?
  - Performance, reliability, limitations on files, overhead, …
- Many different file systems have been proposed and continue to be proposed
- Let’s talk about some general ideas first

Disk Layout Strategies

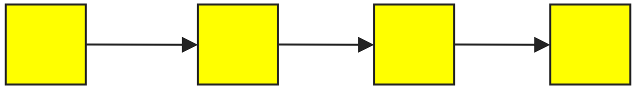
- Files span multiple disk blocks
- How do you find all of the blocks for a file?

1. Contiguous allocation



   » Like memory
   » Fast, simplifies directory access
   » Inflexible, causes fragmentation, needs compaction

2. Linked structure

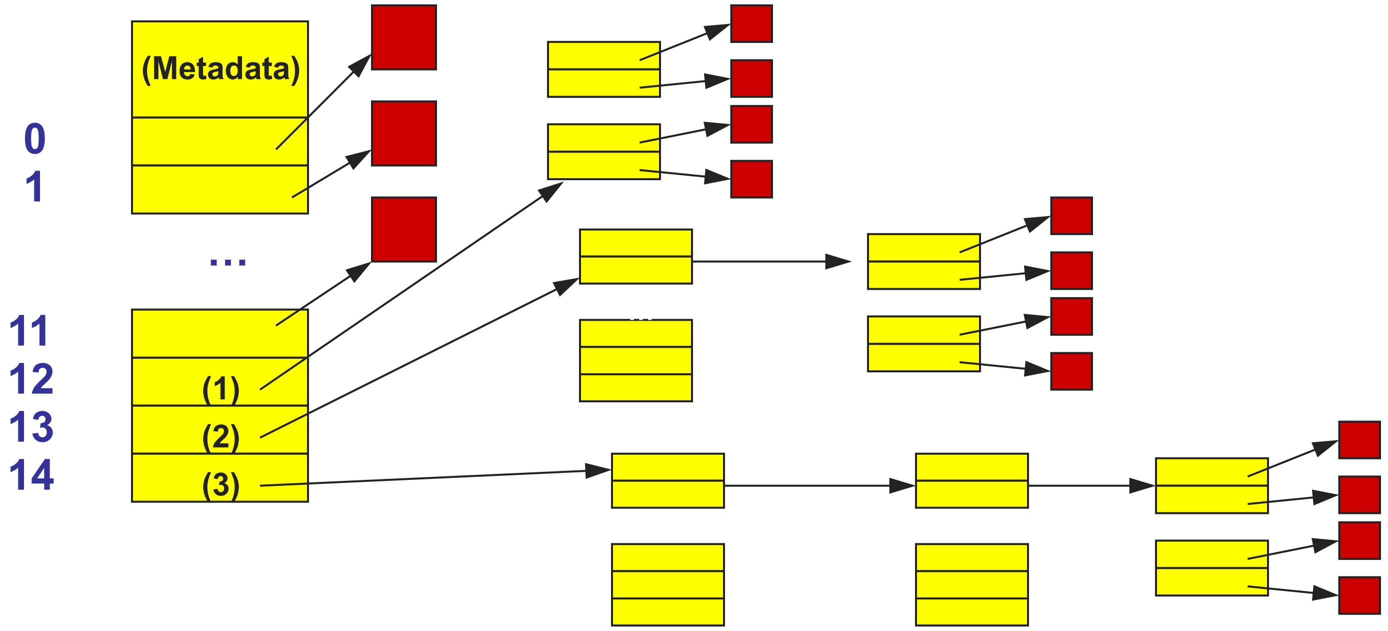


   » Each block points to the next, directory points to the first
   » Bad for random access patterns

3. Indexed structure (indirection, hierarchy)
   » An "index block" contains pointers to many other blocks
   » Handles random better, still good for sequential
   » May need multiple index blocks (linked together)

Unix Inodes

- Unix inodes implement an indexed structure for files
  - Also store metadata info (protection, timestamps, length, ref count…)
- Each inode contains 15 block pointers
  - First 12 are direct blocks (e.g., 4 KB blocks)
  - Then single, double, and triple indirect



Unix Inodes and Path Search

- Unix Inodes are not directories
- Inodes describe where on disk the blocks for a file are placed
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- Directory entries map file names to inodes
  - To open "/one", use Master Block to find inode for "/" on disk
  - Open "/", look for entry for "one"
  - This entry gives the disk block number for the inode for "one"
  - Read the inode for "one" into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file
- This is why we have *open* in addition to *read* and *write*

Symbolic and hard links

A link is a pointer to a file.

- Basically create a file that points at another file
- Two types:
  - Symbolic or soft link (file points to the other file's meta data)
    - This metadata index the file
  - Hard link (file points to the other file's data directly)
    - Repeats the indexing information

Hard Links

- Hard link is a reference to the physical data on a file system
- All named files are hard links
- More than one name can be associated with the same physical data
- Hard links can only refer to data that exists on the ***same*** file system
- You can **not** create hard link to a directory

Hard Links

- Example:
  - Assume you used “vi” to create a new file, you create the first hard link (vi myfile)
  - To Create the 2nd, 3rd and etc. hard links, use the command:

    » `ln myfile `***`link-name`***

Display Hard Links info

- Create a new file called “myfile”
- Run the command “ls -il” to display the ***i-node number*** and ***link counter***

```
38753 -rw-rw-r--  1 uli  uli    29 Oct 29 08:47 myfile
  ^               ^
  |-- inode #     |-- link counter (one link)
```

Display Hard Link Info

- Create a 2nd link to the same data:

  ln myfile mylink

- Run the command “ls -il”:

```
38753 -rw-rw-r-- 2 uli  uli    29 Oct 29 08:47 myfile
38753 -rw-rw-r-- 2 uli  uli    29 Oct 29 08:47 mylink
 ^                ^
 |-- inode #      |--link counter (2 links)
```

Removing a Hard Link

- When a file has more than one link, you can remove any one link and still be able to access the file through the remaining links.
- Hard links are a good way to backup files without having to use the copy command!

Symbolic Links

- Also Known As (a.k.a.): soft links or symlinks
- A symbolic link is an indirect pointer to a file – a pointer ***to*** the hard link ***to*** the file
- You can create a symbolic link to a directory
- A symbolic link can point to a file on a different file system
- A symbolic link can point to a non-existent file (referred to as a “broken link”)

Symbolic Links

- To create a symboic link to the file “myfile”, use

```
ln -s myfile symlink          or

ln --symbolic myfile symlink
```

```
[uli@seneca courses] ls -li myfile
44418 -rw-rw-r-- 1 uli uli   49 Oct 29 14:33 myfile


[uli@seneca courses] ln -s myfile symlink
[uli@seneca courses] ls -li myfile symlink
44418 -rw-rw-r-- 1 uli uli  49 Oct 29 14:33 myfile
44410 lrwxrwxrwx 1 uli uli   6 Oct 29 14:33 symlink -> myfile
```

Different i-node

File type: (symbolic link)

counter

Can we create loops?

- Yes, with symbolic links
  - E.g., /usr/nael/hi/there/link_to_hi@
  - Try it ☺
  - If you do a recursive command it will get stuck…
- Not possible with hard links since we cannot create a hard link to a directory
  - There is no difference between the hard link and the original file
  - Bad idea to allow loops/links to directories

Improving Performance

- Disk reads and writes take order of milliseconds
    - Very slow compared to CPU and memory speeds
- How to speed things up?
    - File buffer cache
    - Cache writes
    - Read ahead

File Buffer Cache

- Applications exhibit significant locality for reading and writing files
- Idea: Cache file blocks in memory to capture locality
  - This is called the file buffer cache
  - Cache is system wide, used and shared by all processes
  - Reading from the cache makes a disk perform like memory
  - Even a 4 MB cache can be very effective
- Issues
  - The file buffer cache competes with VM (tradeoff here)
  - Like VM, it has limited size
  - Need replacement algorithms again (LRU usually used)

Caching Writes

- On a write, some applications assume that data makes it through the buffer cache and onto the disk
  - As a result, writes are often slow even with caching
- Several ways to compensate for this
  - "write-behind"
    - » Maintain a queue of uncommitted blocks
    - » Periodically flush the queue to disk
    - » Unreliable
  - Log-structured file system
    - » Always write next block after last block written
    - » Complicated

Read Ahead

- Many file systems implement "read ahead"
  - FS predicts that the process will request next block
  - FS goes ahead and requests it from the disk
  - This can happen while the process is computing on previous block
    - » Overlap I/O with execution
  - When the process requests block, it will be in cache
  - Compliments the disk cache, which also is doing read ahead
- For sequentially accessed files can be a big win
  - Unless blocks for the file are scattered across the disk
  - File systems try to prevent that, though (during allocation)

FFS, JFS, LFS, RAID

- Now we’re going to look at some example file and storage systems
  - BSD Unix Fast File System (FFS)
  - Journaling File Systems (JFS)
  - Redundant Array of Inexpensive Disks (RAID)

Fast File System

- The original Unix file system had a simple, straightforward implementation
  - Easy to implement and understand
  - But very poor utilization of disk bandwidth (lots of seeking)
- BSD Unix folks did a redesign (mid 80s) that they called the Fast File System (FFS)
  - Improved disk utilization, decreased response time
  - McKusick, Joy, Leffler, and Fabry
- Now the FS to which all other Unix FS's are compared
- Good example of being device-aware for performance

Data and Inode Placement

Original Unix FS had two placement problems:

1. Data blocks allocated randomly in aging file systems
   - Blocks for the same file allocated sequentially when FS is new
   - As FS "ages" and fills, need to allocate into blocks freed up when other files are deleted
   - Problem: deleted files essentially randomly placed
   - So, blocks for new files become scattered across the disk
2. inodes allocated far from blocks
   - All inodes at beginning of disk, far from data
   - Traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks

Both of these problems generate many long seeks

Cylinder Groups

- BSD FFS addressed these problems using the notion of a cylinder group
  - Disk partitioned into groups of cylinders
  - Data blocks in same file allocated in same cylinder
  - Files in same directory allocated in same cylinder
  - inodes for files allocated in same cylinder as file data blocks
- Free space requirement
  - To be able to allocate according to cylinder groups, the disk must have free space scattered across cylinders
  - 10% of the disk is reserved just for this purpose
    - Only used by root – this is why “df” may report >100%

Other Problems

- Small blocks (1K) caused two problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)
- Fix: Use a larger block (4K)
  - Very large files, only need two levels of indirection for 2^32
  - Problem: internal fragmentation
  - Fix: Introduce "fragments" (1K pieces of a block)
- Problem: Media failures
  - Replicate master block (superblock)
- Problem: Device oblivious
  - Parameterize according to device characteristics

The Results

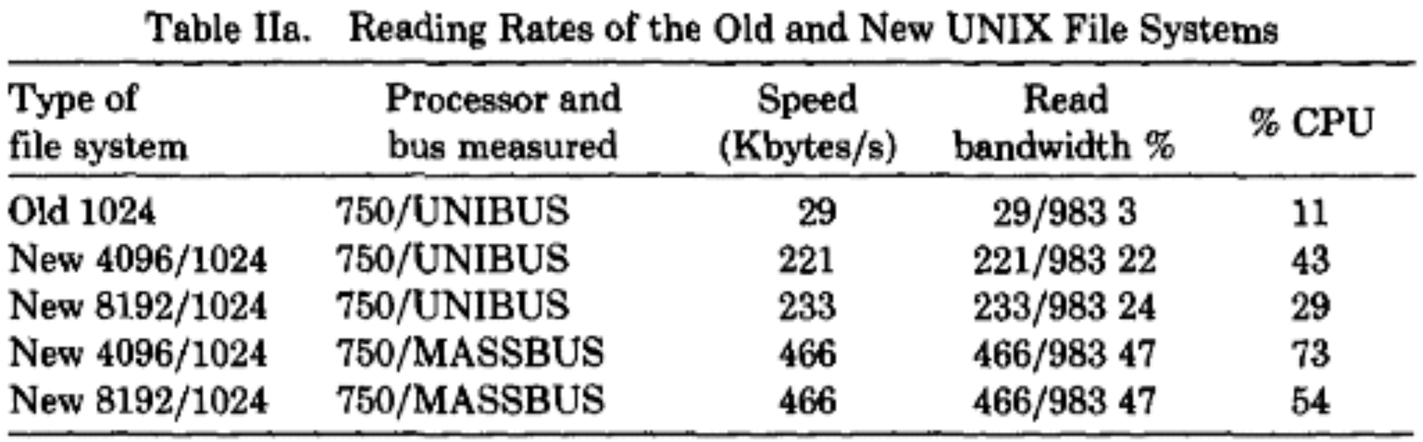


Table IIa. Reading Rates of the Old and New UNIX File Systems

Type of file system	Processor and bus measured	Speed (Kbytes/s)	Read bandwidth %	% CPU
Old 1024	750/UNIBUS	29	29/983 3	11
New 4096/1024	750/UNIBUS	221	221/983 22	43
New 8192/1024	750/UNIBUS	233	233/983 24	29
New 4096/1024	750/MASSBUS	466	466/983 47	73
New 8192/1024	750/MASSBUS	466	466/983 47	54

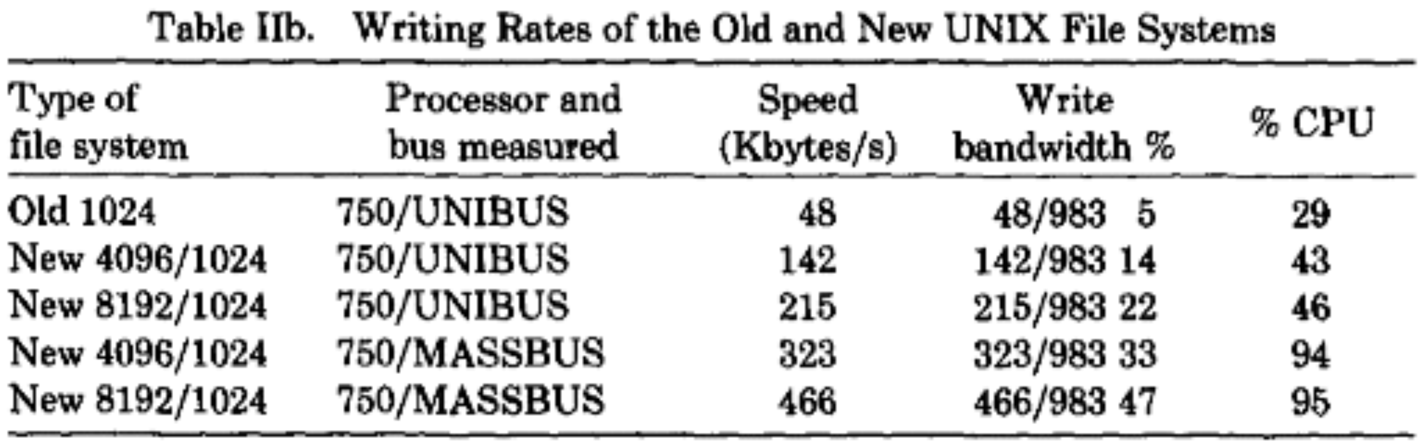


Table IIb. Writing Rates of the Old and New UNIX File Systems

Type of file system	Processor and bus measured	Speed (Kbytes/s)	Write bandwidth %	% CPU
Old 1024	750/UNIBUS	48	48/983 5	29
New 4096/1024	750/UNIBUS	142	142/983 14	43
New 8192/1024	750/UNIBUS	215	215/983 22	46
New 4096/1024	750/MASSBUS	323	323/983 33	94
New 8192/1024	750/MASSBUS	466	466/983 47	95

Problem: Crash Consistency

- Updates to data and meta data are not atomic
- Consider, what happens when you delete a file
    1. Remove directory entry
    2. Remove the inode(s)
    3. Mark the free map (for all the i-node and data blocks you freed)
    4. What happens if you crash somewhere in the middle?

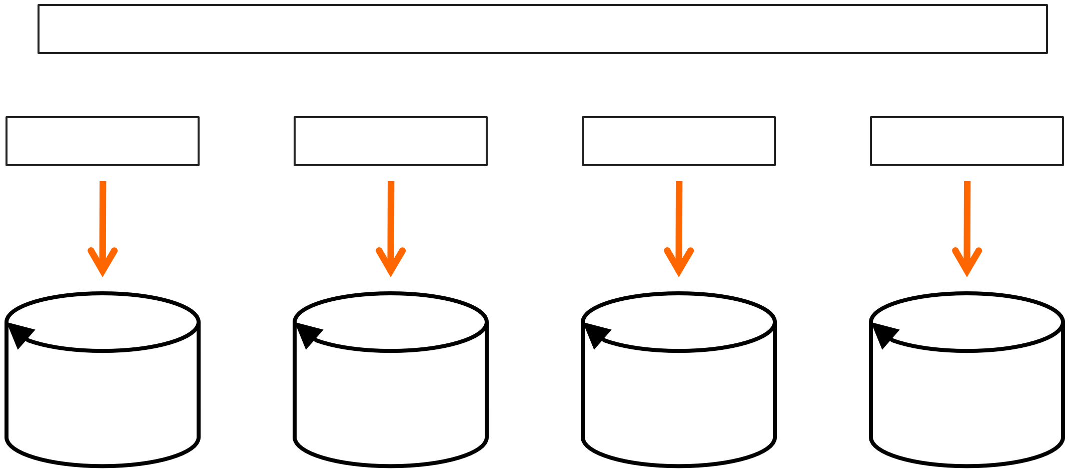
Journaling File Systems

- Journaling File systems make updates to a log
    - Log plans for updates to a journal first
    - When a crash happens you can replay the journal to restore consistency
- What if we crash when writing journal?
    - Problem. Possible solution, bracket the changes
        - Introduce checksum periodically
        - Replay only parts where there is checksum that matches
- Journal choices (regular file? Special partition?)
- Log meta-data and data?

RAID

- Redundant Array of Inexpensive Disks (RAID)
  - A storage system, not a file system
  - Patterson, Katz, and Gibson (Berkeley, 1988)
- Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
  - Files are striped across disks
  - Each stripe portion is read/written in parallel
  - Bandwidth increases with more disks

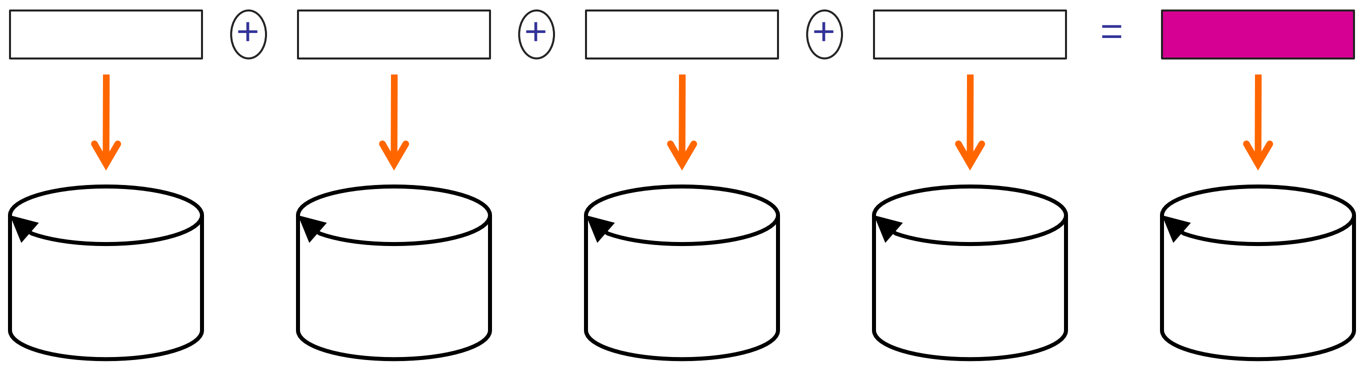
RAID



RAID Challenges

- Small files (small writes less than a full stripe)
  - Need to read entire stripe, update with small write, then write entire stripe out to disks
- Reliability
  - More disks increases the chance of media failure (MTBF)
- Turn reliability problem into a feature
  - Use one disk to store parity data
    - XOR of all data blocks in stripe
  - Can recover any data block from all others + parity block
  - Hence "redundant" in name
  - Introduces overhead, but, hey, disks are "inexpensive"

RAID with parity



RAID Levels

- In marketing literature, you will see RAID systems advertised as supporting different “RAID Levels”
  - RAID 0: Striping
    - Good for random access (no reliability)
  - RAID 1: Mirroring
    - Two disks, write data to both (expensive, 1X storage overhead)
  - RAID 2,3 and 4: bit, byte and block level parity. Rarely used.
  - RAID 5, 6: Floating parity
    - Parity blocks for different stripes written to different disks
    - No single parity disk, hence no bottleneck at that disk
  - RAID “10”: Striping plus mirroring
    - Higher bandwidth, but still have large overhead
    - See this on PC RAID disk cards

Other file system topics

- Network File systems (NFS)
  - Can a file system be shared across the network
  - The file system is on a single server, the clients access it remotely
- Distributed file systems: Can a file system be stored (and possibly replicated) across multiple machines
  - What if they are geographically spread?
  - Hadoop Distributed File System (HDFS), Google File System (GFS)
- File systems is an exciting research area
  - Take cs202 if interested!