CS162
Operating Systems and Systems Programming
Lecture 19

File Systems (Con't), MMAP, Transactions, COW

April 8th, 2015

Prof. John Kubiatowicz

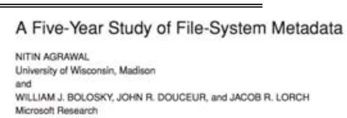
http://cs162.eecs.Berkeley.edu

Recall: Building a File System

- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.
- File System Components
  - Disk Management: collecting disk blocks into files
  - Naming: Interface to find files by name, not by blocks
  - Protection: Layers to keep data secure
  - Reliability/Durability: Keeping of files durable despite crashes, media failures, attacks, etc
- User vs. System View of a File
  - User's view:
    - » Durable Data Structures
  - System's view (system call interface):
    - » Collection of Bytes (UNIX)
    - » Doesn't matter to system what kind of data structures you want to store on disk!
  - System's view (inside OS):
    - » Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)
    - » Block size ≥ sector size; in UNIX, block size is 4KB

Recall: Characteristics of Files

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


A Five-Year Study of File-System Metadata

NITIN AGRAWAL
University of Wisconsin, Madison
and
WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH
Microsoft Research

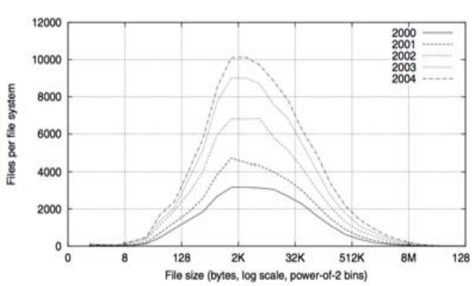


Fig. 2. Histograms of files by size.

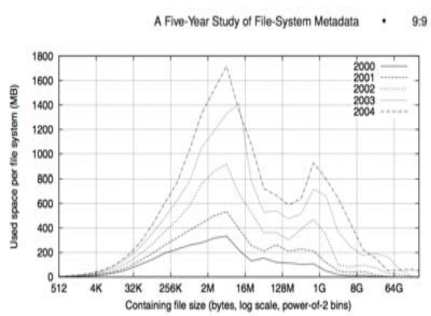
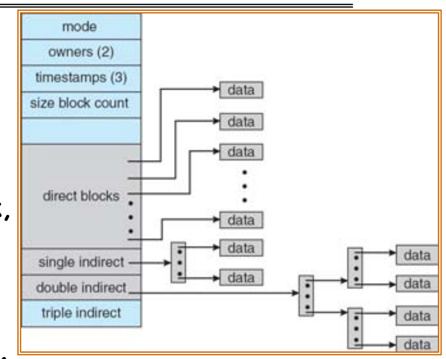


Fig. 4. Histograms of bytes by containing file size.

Recall: Multilevel Indexed Files (Original 4.1 BSD)



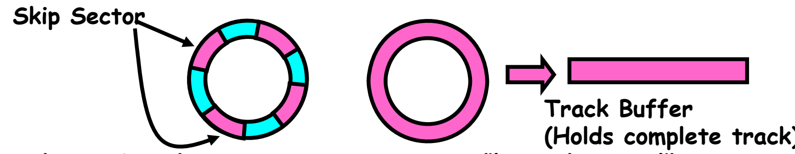
- Sample file in multilevel indexed format:
  - 10 direct ptrs, 1K blocks
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data
- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect block (four I/Os per block!)

UNIX BSD 4.2

- Same as BSD 4.1 (same file header and triply indirect blocks), except incorporated ideas from Cray DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned next slide)
- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  - How much contiguous space do you allocate for a file?
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: store files from same directory near each other
- Fast File System (FFS)
  - Allocation and placement policies for BSD 4.2

Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



  - Solution1: Skip sector positioning ("interleaving")
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution2: Read ahead: read next block right after first, even if application hasn't asked for it yet.
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
- Important Aside: Modern disks+controllers do many complex things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

Where are inodes stored?

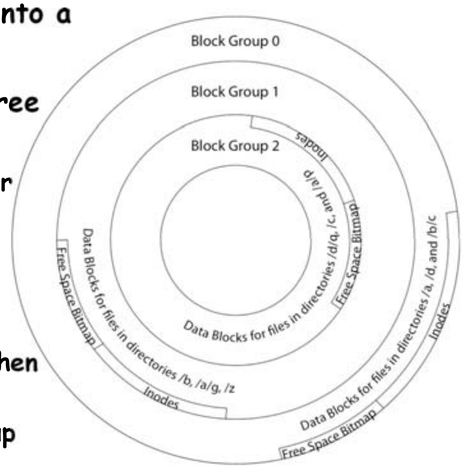
- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored anywhere near the data blocks. To read a small file, seek to get header, seek back to data.
  - Fixed size, set when disk is formatted. At formatting time, a fixed number of inodes were created (They were each given a unique number, called an "inumber")

Where are inodes stored?

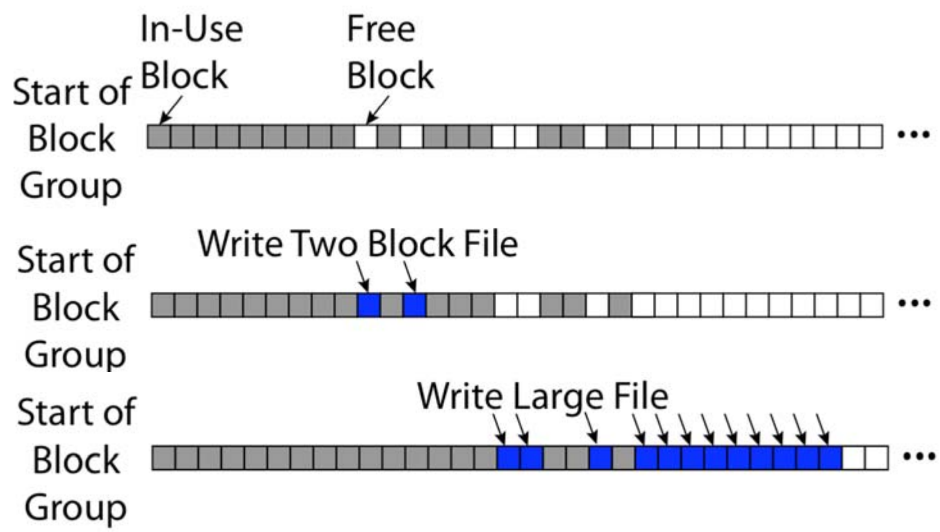
- Later versions of UNIX moved the header information to be closer to the data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the file (makes an ls of that directory run fast).
  - Pros:
    - » UNIX BSD 4.2 puts a portion of the file header array on each of many cylinders. For small directories, can fit all data, file headers, etc. in same cylinder $\Rightarrow$ no seeks!
    - » File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
    - » Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)
  - Part of the Fast File System (FFS)
    - » General optimization to avoid seeks

4.2 BSD Locality: Block Groups

- File system volume is divided into a set of block groups
  - Close set of tracks
- Data blocks, metadata, and free space interleaved within block group
  - Avoid huge seeks between user data and system structure
- Put directory and its files in common block group
- First-Free allocation of new file blocks
  - To expand file, first try successive blocks in bitmap, then choose new range of blocks
  - Few little holes at start, big sequential runs at end of group
  - Avoids fragmentation
  - Sequential layout for big files
- **Important: keep 10% or more free!**
  - **Reserve space in the BG**



FFS First Fit Block Allocation

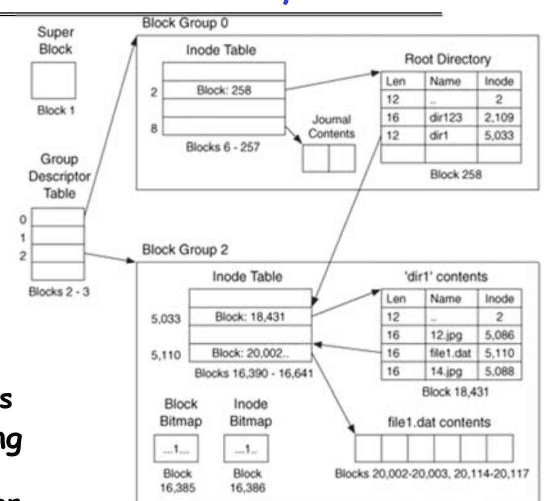


FFS

- Pros
  - Efficient storage for both small and large files
  - Locality for both small and large files
  - Locality for metadata and data
- Cons
  - Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
  - Inefficient encoding when file is mostly contiguous on disk (no equivalent to superpages)
  - Need to reserve 10-20% of free space to prevent fragmentation

Linux Example: Ext2/3 Disk Layout

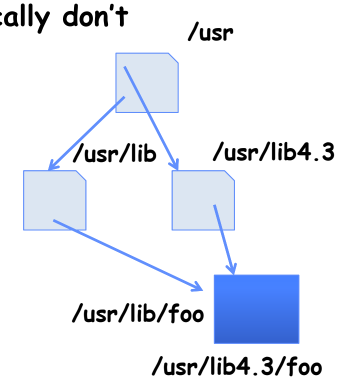
- Disk divided into block groups
  - Provides locality
  - Each group has two block-sized bitmaps (free blocks/inodes)
  - Block sizes settable at format time: 1K, 2K, 4K, 8K…
- Actual Inode structure similar to 4.2BSD
  - with 12 direct pointers
- Ext3: Ext2 w/Journaling
  - Several degrees of protection with more or less cost



- Example: create a *file1.dat* under */dir1/* in Ext3

A bit more on directories

- Stored in files, can be read, but typically don't
  - System calls to access directories
  - Open / Creat traverse the structure
  - mkdir /rmdir add/remove entries
  - Link / Unlink
    - » Link existing file to a directory
      - Not in FAT !
    - » Forms a DAG

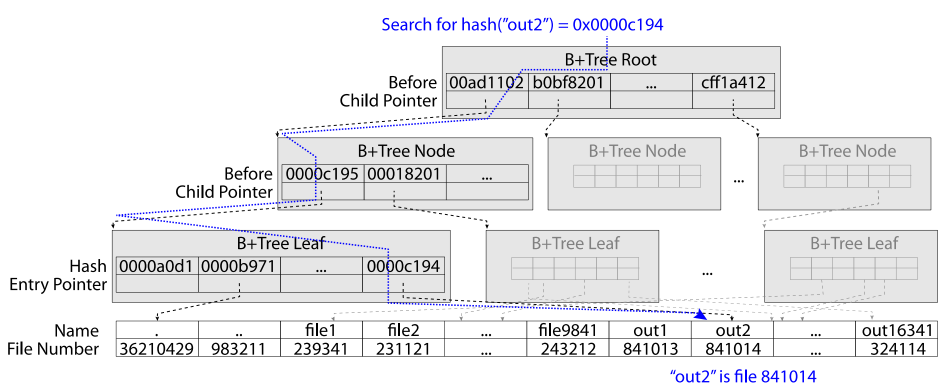


- When can file be deleted?
  - Maintain ref-count of links to the file
  - Delete after the last reference is gone.
- libc support
  - `DIR * opendir (const char *dirname)`
  - `struct dirent * readdir (DIR *dirstream)`
  - `int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)`

Links

- Hard link
  - Sets another directory entry to contain the file number for the file
  - 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

Large Directories: B-Trees (dirhash)



Administrivia

- Midterm II
  - Wednesday, 4/22
  - Topics up until Monday class (4/20)
  - 1 page of hand-written notes, both sides
- HW 4 handed out next Monday

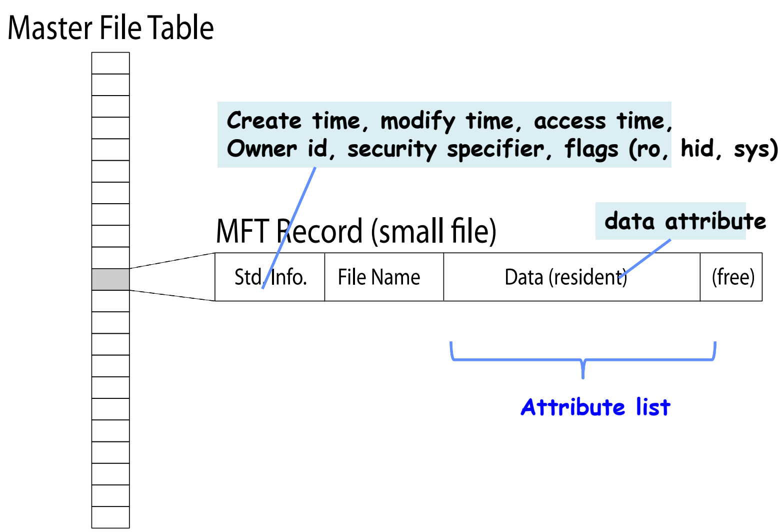
NTFS

- New Technology File System (NTFS)
  - Common on Microsoft Windows systems
- Variable length extents
  - Rather than fixed blocks
- Everything (almost) is a sequence of <attribute:value> pairs
  - Meta-data and data
- Mix direct and indirect freely
- Directories organized in B-tree structure by default

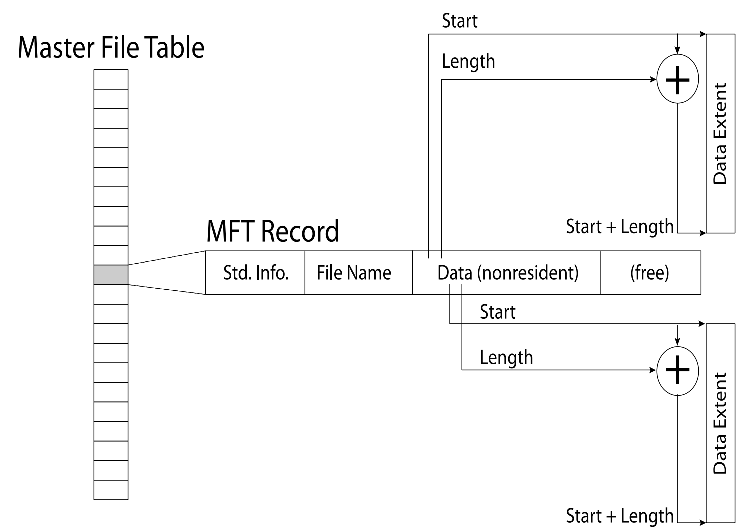
NTFS

- Master File Table
  - DataBase with Flexible 1KB entries for metadata/data
  - Variable-sized attribute records (data or metadata)
  - Extend with variable depth tree (non-resident)
- Extents – variable length contiguous regions
  - Block pointers cover runs of blocks
  - Similar approach in Linux (ext4)
  - File create can provide hint as to size of file
- Journalling for reliability
  - Discussed later

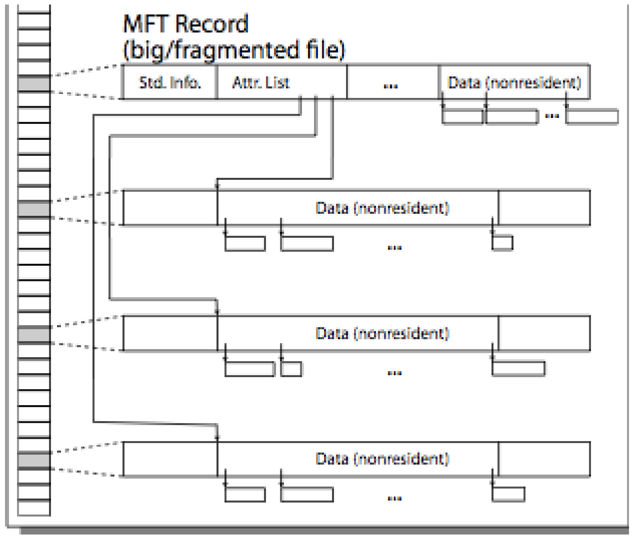
NTFS Small File

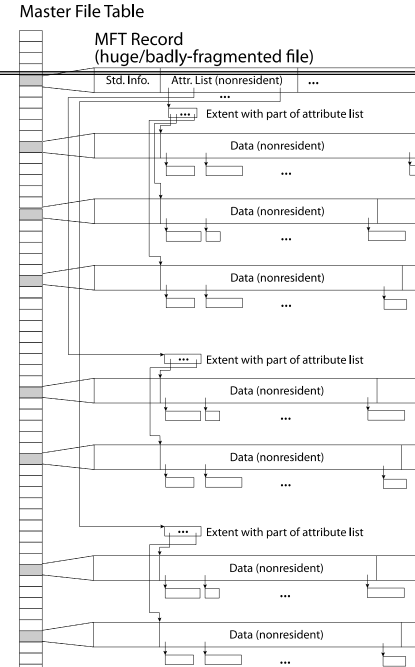


NTFS Medium File

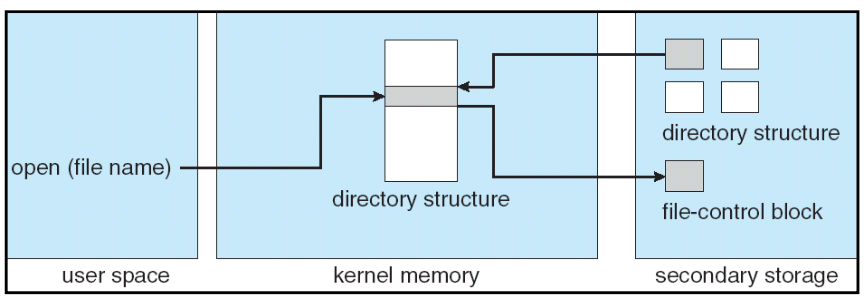


NTFS Multiple Indirect Blocks



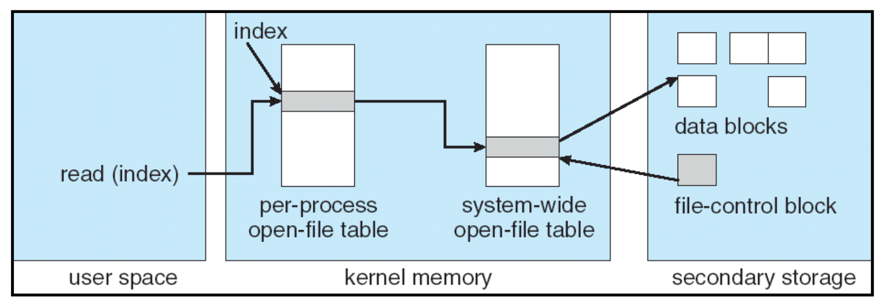


In-Memory File System Structures



- **Open system call:**
  - **Resolves file name, finds file control block (inode)**
  - **Makes entries in per-process and system-wide tables**
  - **Returns index (called "file handle") in open-file table**

In-Memory File System Structures



- **Read/write system calls:**
  - **Use file handle to locate inode**
  - **Perform appropriate reads or writes**

Authorization: Who Can Do What?

- How do we decide who is authorized to do actions in the system?
- **Access Control Matrix:** contains all permissions in the system
  - Resources across top
    - » Files, Devices, etc…
  - Domains in columns
    - » A domain might be a user or a group of users
    - » E.g. above: User D3 can read F2 or execute F3
  - In practice, table would be huge and sparse!

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Authorization: Two Implementation Choices

- **Access Control Lists:** store permissions with object
  - Still might be lots of users!
  - UNIX limits each file to: r,w,x for owner, group, world
  - More recent systems allow definition of groups of users and permissions for each group
  - ACLs allow easy changing of an object's permissions
    - » Example: add Users C, D, and F with rw permissions
- **Capability List:** each process tracks which objects has permission to touch
  - Popular in the past, idea out of favor today
  - Consider page table: Each process has list of pages it has access to, not each page has list of processes …
  - Capability lists allow easy changing of a domain's permissions
    - » Example: you are promoted to system administrator and should be given access to all system files

Authorization: Combination Approach

- Users have capabilities, called "groups" or "roles"
  - Everyone with particular group access is "equivalent" when accessing group resource
  - Like passport (which gives access to country of origin)
- Objects have ACLs
  - ACLs can refer to users or groups
  - Change object permissions object by modifying ACL
  - Change broad user permissions via changes in group membership
  - Possessors of proper credentials get access

Authorization: How to Revoke?

- How does one revoke someone's access rights to a particular object?
  - Easy with ACLs: just remove entry from the list
  - Takes effect immediately since the ACL is checked on each object access
- Harder to do with capabilities since they aren't stored with the object being controlled:
  - Not so bad in a single machine: could keep all capability lists in a well-known place (e.g., the OS capability table).
  - Very hard in distributed system, where remote hosts may have crashed or may not cooperate (more in a future lecture)

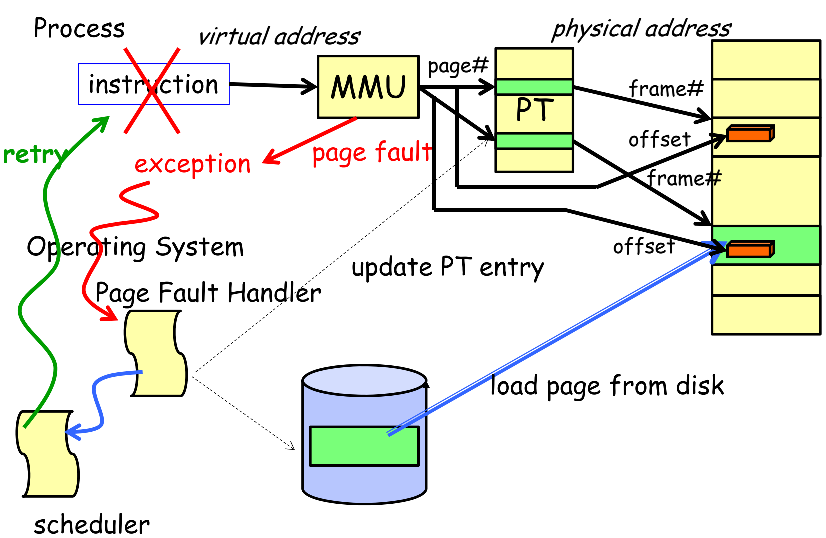
Revoking Capabilities

- Various approaches to revoking capabilities:
  - Put expiration dates on capabilities and force reacquisition
  - Put epoch numbers on capabilities and revoke all capabilities by bumping the epoch number (which gets checked on each access attempt)
  - Maintain back pointers to all capabilities that have been handed out (Tough if capabilities can be copied)
  - Maintain a revocation list that gets checked on every access attempt

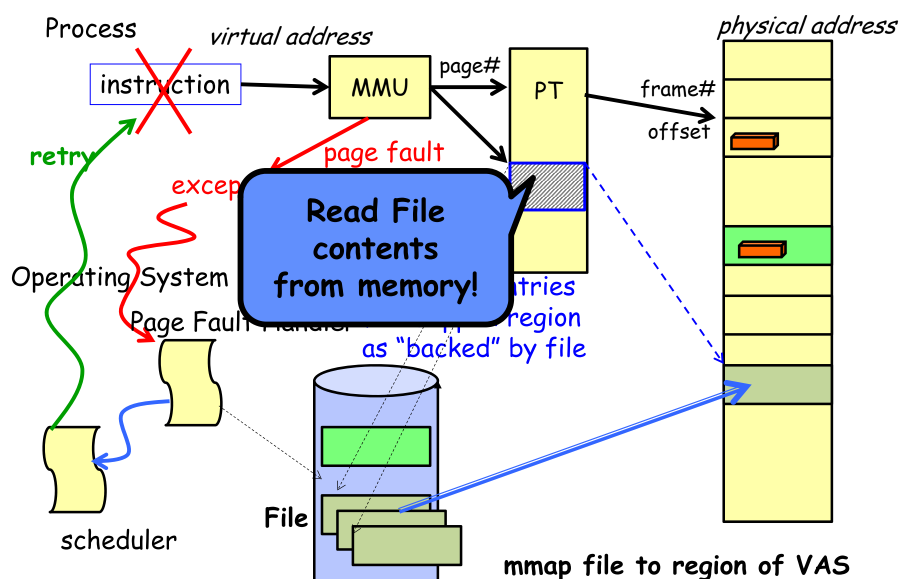
Memory Mapped Files

- Traditional I/O involves explicit transfers between buffers in process address space to regions of a file
  - This involves multiple copies into caches in memory, plus system calls
- What if we could "map" the file directly into an empty region of our address space
  - Implicitly "page it in" when we read it
  - Write it and "eventually" page it out
- Executable file is treated this way when we exec the process !!

Recall: Who does what, when?



Using Paging to mmap files



mmap system call

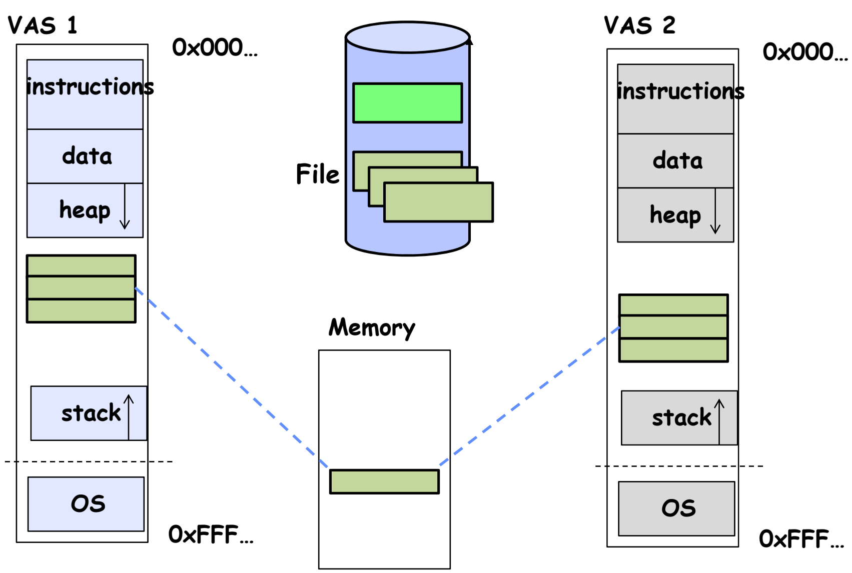
```
MMAP(2)                    BSD System Calls Manual                    MMAP(2)

NAME
     mmap -- allocate memory, or map files or devices into memory

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <sys/mman.h>

     void *
     mmap(void *addr, size_t len, int prot, int flags, int fd,
         off_t offset);

DESCRIPTION
     The mmap() system call causes the pages starting at addr and continuing
     for at most len bytes to be mapped from the object described by fd,
     starting at byte offset offset.  If offset or len is not a multiple of
```

- May map a specific region or let the system find one for you
  - Tricky to know where the holes are
- Used both for manipulating files and for sharing between processes

An example

```
#include <sys/mman.h>

int something = 162;

int main (int argc, char *argv[]) {
  int myfd;
  char *mfile;

  printf("Data  at: %16lx\n", (long unsigned int) &something);
  printf("Heap at : %16lx\n", (long unsigned int) malloc(1));
  printf("Stack at: %16lx\n", (long unsigned int) &mfile);

  /* Open the file */
  myfd = open(argv[1], O_RDWR | O_CREATE);
  if (myfd < 0) { perror((“open failed!”);exit(1); }

  /* map the file */
  mfile = mmap(0, 10000, PROT_READ|PROT_WRITE, MAP_FILE|MAP_SHARED, myfd, 0);
  if (mfile == MAP_FAILED) {perror("mmap failed"); exit(1);}

  printf("mmap at : %16lx\n", (long unsigned int) mfile);

  puts(mfile);
  strcpy(mfile+20,"Let's write over it");
  close(myfd);
  return 0;
}
```


Sharing through Mapped Files



File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- Buffer Cache: Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain “dirty” blocks (blocks yet on disk)
- Replacement policy? LRU
  - Can afford overhead of timestamps for each disk block
  - Advantages:
    - » Works very well for name translation
    - » Works well in general as long as memory is big enough to accommodate a host’s working set of files.
  - Disadvantages:
    - » Fails when some application scans through file system, thereby flushing the cache with data used only once
    - » Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, ‘Use Once’:
    - » File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- Read Ahead Prefetching: fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - » Too many imposes delays on requests by other applications
    - » Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con't)

- Delayed Writes: Writes to files not immediately sent out to disk
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - » Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - » If some other application tries to read data before written to disk, file system will read from cache
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

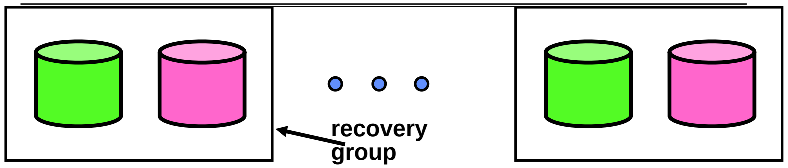
Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

How to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache.
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…
- RAID: Redundant Arrays of Inexpensive Disks
  - Data stored on multiple disks (redundancy)
  - Either in software or hardware
    - » In hardware case, done by disk controller; file system may not even know that there is more than one disk in use

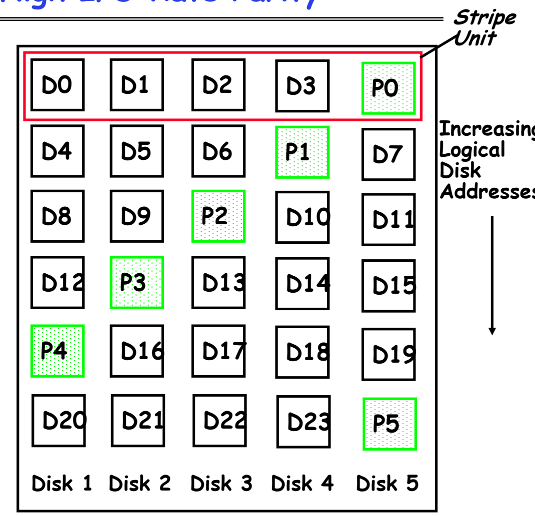
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation fully synchronized (hard to do exactly)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk already attached to system to be used for immediate replacement

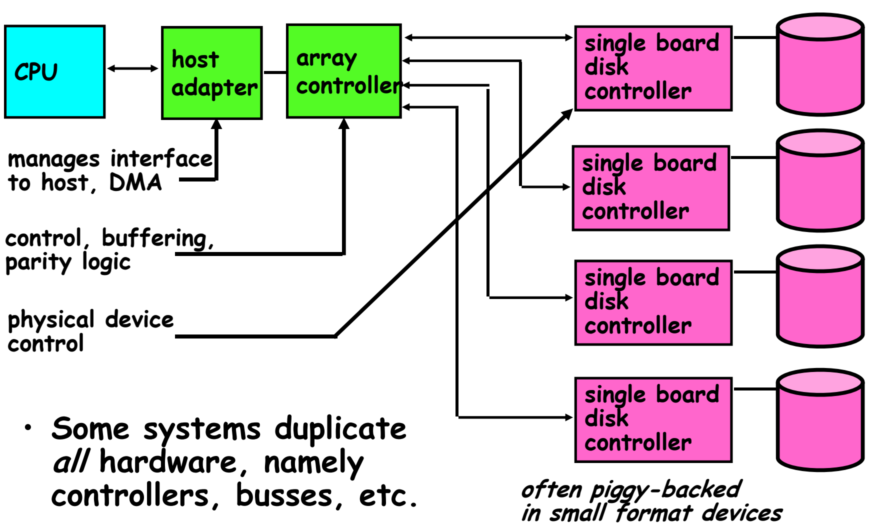
RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - $P0=D0\oplus D1\oplus D2\oplus D3$
  - Can destroy any one disk and still reconstruct data
  - Suppose D3 fails, then can reconstruct: $D3=D0\oplus D1\oplus D2\oplus P0$



- Later in term: talk about spreading information widely across internet for durability.

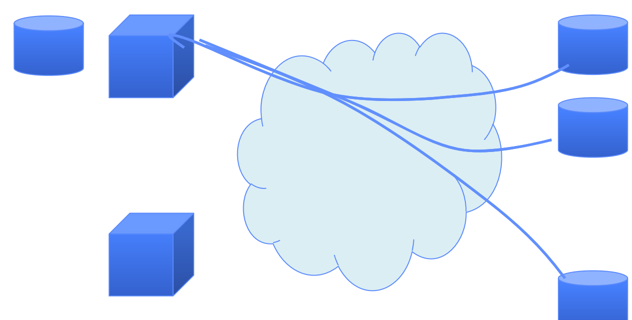
Hardware RAID: Subsystem Organization



- Some systems duplicate *all* hardware, namely controllers, busses, etc.

Higher Durability/Reliability through Geographic Replication

- Highly durable – hard to destroy bits
- Highly available for reads
- Low availability for writes
  - Can't write if any one is not up
  - Or – need relaxed consistency model
- Reliability?



File System Reliability

- What can happen if disk loses power or machine software crashes?
  - Some operations in progress may complete
  - Some operations in progress may be lost
  - Overwrite of a block may only partially complete
- Having RAID doesn't necessarily protect against all such failures
  - Bit-for-bit protection of bad state?
  - What if one disk of RAID group not written?
- File system wants durability (as a minimum!)
  - Data previously stored can be retrieved (maybe after some recovery step), regardless of failure

Achieving File System Reliability

- Problem posed by machine/disk failures
- Transaction concept
- Approaches to reliability
  - Careful sequencing of file system operations
  - Copy-on-write (WAFL, ZFS)
  - Journalling (NTFS, linux ext4)
  - Log structure (flash storage)
- Approaches to availability
  - RAID

Storage Reliability Problem

- Single logical file operation can involve updates to multiple physical disk blocks
  - inode, indirect block, data block, bitmap, …
  - With remapping, single update to physical disk block can require multiple (even lower level) updates
- At a physical level, operations complete one at a time
  - Want concurrent operations for performance
- How do we guarantee consistency regardless of when crash occurs?

Threats to Reliability

- Interrupted Operation
  - Crash or power failure in the middle of a series of related updates may leave stored data in an *inconsistent state*.
  - e.g.: transfer funds from BofA to Schwab. What if transfer is interrupted after withdrawal and before deposit
- Loss of stored data
  - Failure of non-volatile storage media may cause previously stored data to disappear or be corrupted

Log Structured and Journaled File Systems

- Better reliability through use of log
  - All changes are treated as *transactions*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery
- For Journaled system:
  - Log used to asynchronously update filesystem
    - » Log entries removed after used
  - After crash:
    - » Remaining transactions in the log performed ("Redo")
    - » Modifications done in way that can survive crashes
- Examples of Journaled File Systems:
  - Ext3 (Linux), XFS (Unix), etc.

More General Solutions

- Transactions for Atomic Updates
  - Ensure that multiple related updates are performed atomically
  - i.e., if a crash occurs in the middle, the state of the systems reflects either *all or none* of the updates
  - Most modern file systems use transactions internally to update the many pieces
  - Many applications implement their own transactions
- Redundancy for media failures
  - Redundant representation (error correcting codes)
  - Replication
  - E.g., RAID disks

Transactions

- Closely related to critical sections in manipulating shared data structures
- Extend concept of atomic update from memory to stable storage
  - Atomically update multiple persistent data structures
- Like flags for threads, many ad hoc approaches
  - FFS carefully ordered the sequence of updates so that if a crash occurred while manipulating directory or inodes the disk scan on reboot would detect and recover the error, -- fsck
  - Applications use temporary files and rename

Key concept: Transaction

- An atomic sequence of actions (reads/writes) on a storage system (or database)
- That takes it from one consistent state to another

Typical Structure

- **Begin** a transaction – get transaction id
- Do a bunch of updates
  - If any fail along the way, **roll-back**
  - Or, if any conflicts with other transactions, **roll-back**
- **Commit** the transaction

"Classic" Example: Transaction

```
BEGIN;    --BEGIN TRANSACTION
 UPDATE accounts SET balance = balance - 100.00
   WHERE name = 'Alice';

 UPDATE branches SET balance = balance - 100.00
   WHERE name = (SELECT branch_name FROM accounts
   WHERE name = 'Alice');

 UPDATE accounts SET balance = balance + 100.00
   WHERE name = 'Bob';

 UPDATE branches SET balance = balance + 100.00
   WHERE name = (SELECT branch_name FROM accounts
   WHERE name = 'Bob');


COMMIT;    --COMMIT WORK
```

Transfer $100 from Alice's account to Bob's account

The ACID properties of Transactions

- **Atomicity:** all actions in the transaction happen, or none happen
- **Consistency:** transactions maintain data integrity, e.g.,
  - Balance cannot be negative
  - Cannot reschedule meeting on February 30
- **Isolation:** execution of one transaction is isolated from that of all others; no problems from concurrency
- **Durability:** if a transaction commits, its effects persist despite crashes

Poor-man's transactions: Toward Copy-on-Write

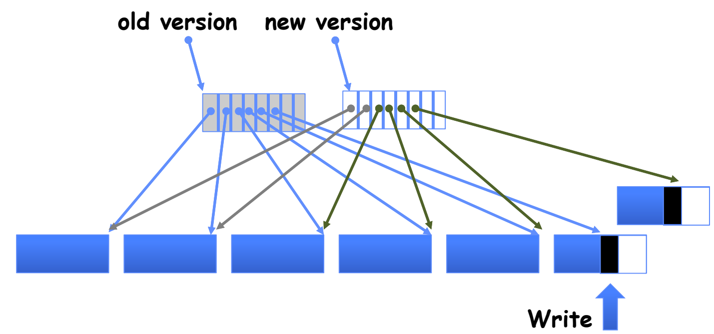
- *Files* are for durable storage **AND** flexible process-independent, protected namespace
- Files grow incrementally as written
  - Update-in-place file systems start with a basic chunk and append (possibly larger) chunks as file grows
  - Transition from random access to large sequential
- *Disks trends:* huge and cheap, high startup
- *Design / Memory trends:* cache everything
  - Reads satisfied from cache, buffer multiple writes and do them all together
- Application trends: make multiple related changes to a file and commit **all or nothing**
  - **What if want to be able to undo changes later?**

Emulating COW @ user level

- Transform file foo to a new version
- Open/Create a new file foo.v
  - where v is the version #
- Do all the updates based on the old foo
  - Reading from foo and writing to foo.v
  - Including copying over any unchanged parts
- Update the link
  - ln –f foo foo.v

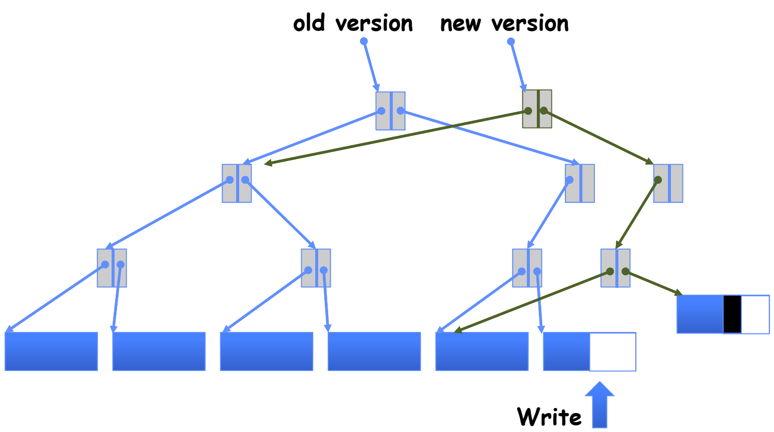
- Does it work?
- What if multiple updaters at same time?
- How to keep track of every version of file?
  - Would we want to do that?

Creating a New Version



- If file represented as a tree of blocks, just need to update the leading fringe

Creating a New Version



- If file represented as a tree of blocks, just need to update the leading fringe

ZFS

- Variable sized blocks: 512 B – 128 KB
- Symmetric tree
  - Know if it is large or small when we make the copy
- Store version number with pointers
  - Can create new version by adding blocks and new pointers
- Buffers a collection of writes before creating a new version with them
- Free space represented as tree of extents in each block group
  - Delay updates to freespace (in log) and do them all when block group is activated

File System Summary (1/2)

- File System:
  - Transforms blocks into Files and Directories
  - Optimize for size, access and usage patterns
  - Maximize sequential access, allow efficient random access
  - Projects the OS protection and security regime (UGO vs ACL)
- File defined by header, called "inode"
- Naming: act of translating from user-visible names to actual system resources
  - Directories used for naming for local file systems
  - Linked or tree structure stored in files
- Multilevel Indexed Scheme
  - inode contains file info, direct pointers to blocks, indirect blocks, doubly indirect, etc..
  - NTFS uses variable extents, rather than fixed blocks, and tiny files data is in the header
- 4.2 BSD Multilevel index files
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc.
  - Optimizations for sequential access: start new files in open ranges of free blocks, rotational Optimization

File System Summary (2/2)

- File layout driven by freespace management
  - Integrate freespace, inode table, file blocks and directories into block group
- Deep interactions between memory management, file system, and sharing
- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?
- RAID: Redundant Arrays of Inexpensive Disks
  - RAID1: mirroring, RAID5: Parity block
- Use of Log to improve Reliability
  - Journaled file systems such as ext3
- Copy-on-write creates new (better positioned) version of file upon burst of writes