File Systems

Fear not Smithers! I'll move heaven and earth to save you! It's still easier then teaching a new assistant my filing system.

— C. Montgomery Burns, *The Simpsons*

File systems

- Files
- Directories & naming
- File system implementation
- Example file systems

Baskin Engineering
UC SANTA CRUZ

Why are file systems important?

Long-term information storage

- Must store large amounts of data
  - Gigabytes → terabytes → petabytes
- Stored information must survive the termination of the process using it
  - Lifetime can be seconds to years
  - Must have some way of finding it!
- Multiple processes must be able to access the information concurrently

Baskin Engineering UC SANTA CRUZ

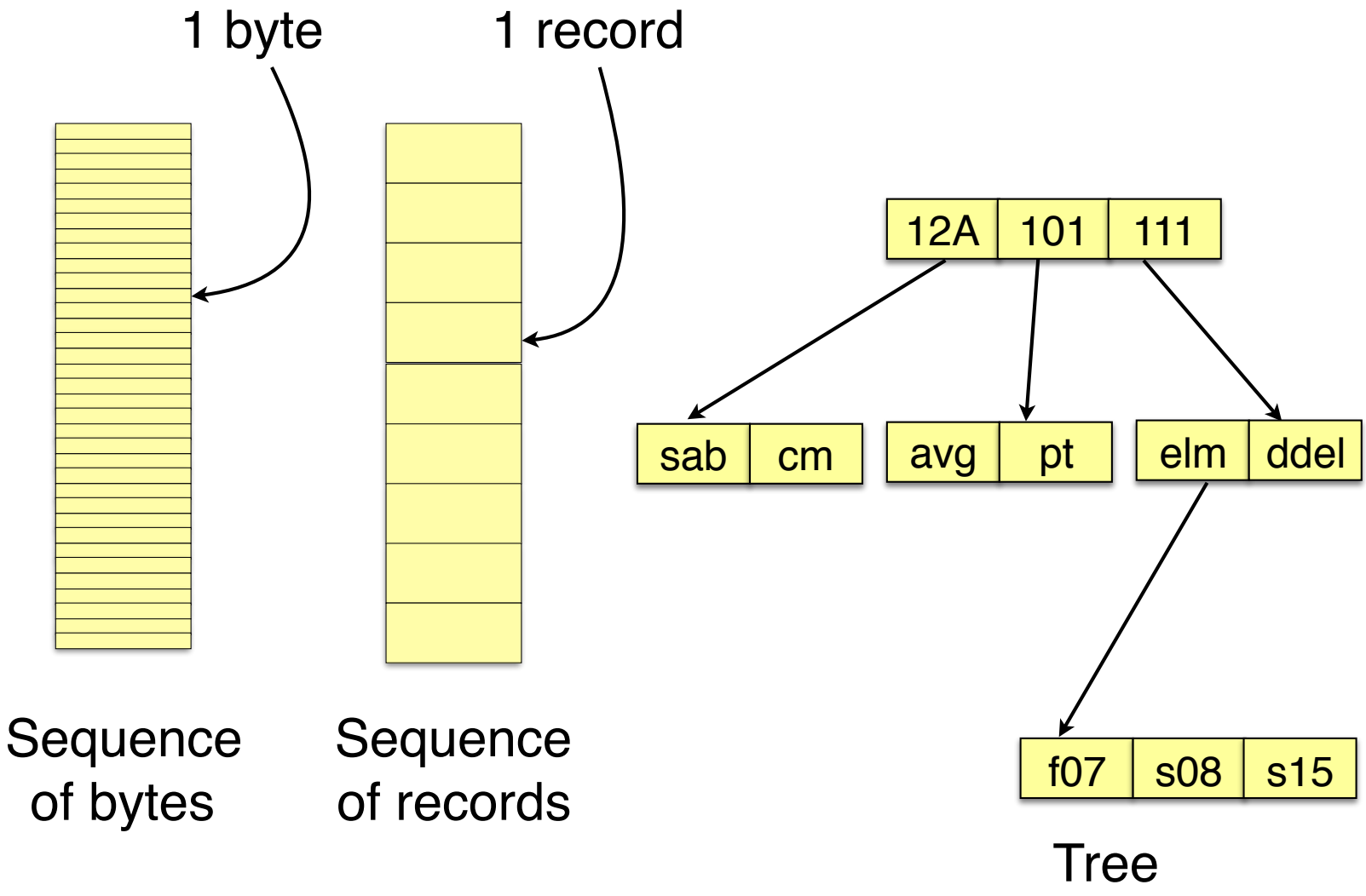
Naming files

- Important to be able to find files after they're created
- Every file has at least one name
- Name can be
  - Human-accessible: "foo.c", "my photo", "Go Slugs!"
  - Machine-usable: 1138, 8675309
- Case may or may not matter
  - Depends on the file system
- Name may include information about the file's contents
  - Certainly does for the user (the name should make it easy to figure out what's in it!)
  - Computer may use part of the name to determine the file type

Typical file extensions

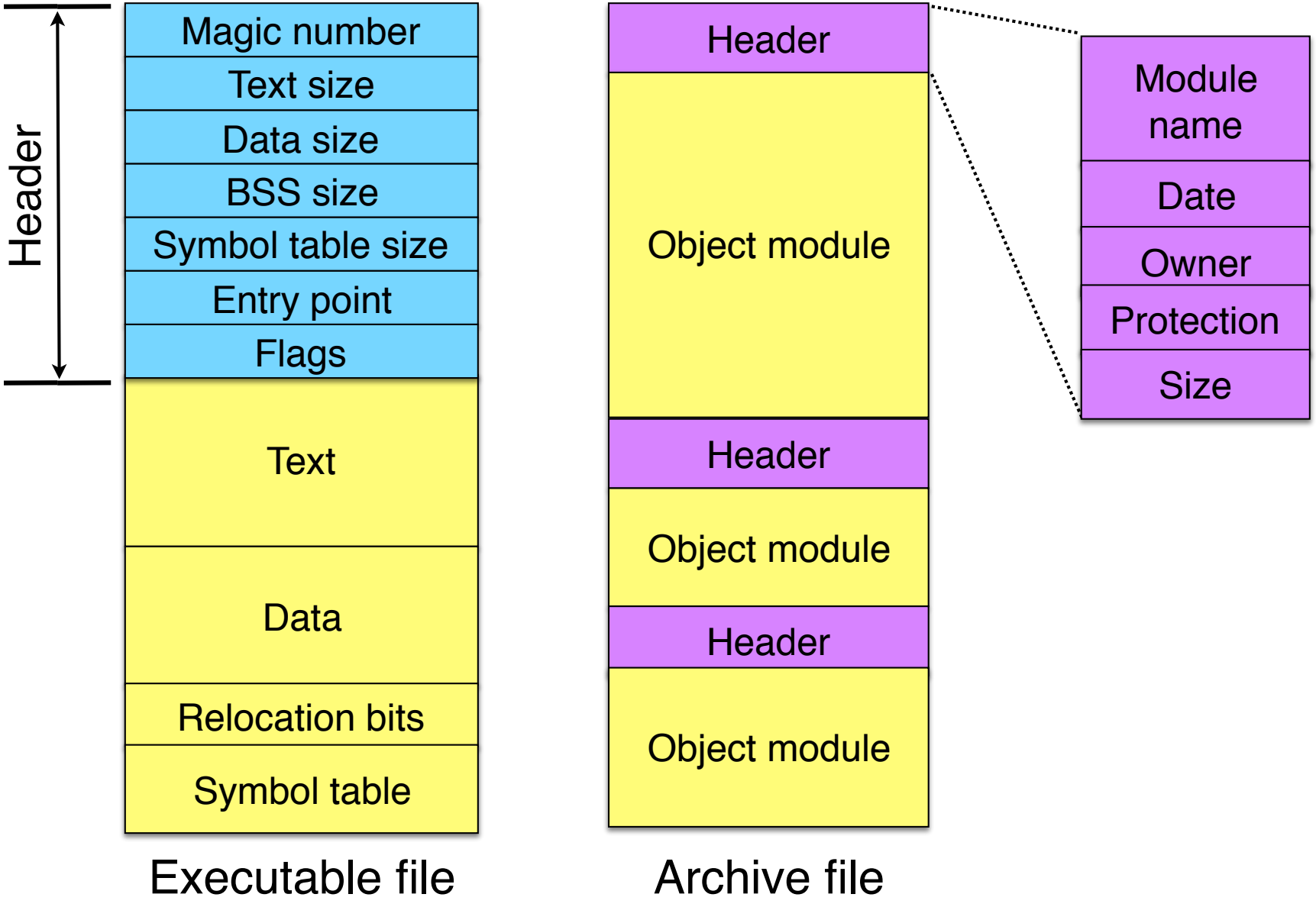
Extension	Meaning
File.txt	Text file
File.c	C source program
File.jpg	JPEG-encoded picture
File.tgz	Compressed tar file
File.html	HTML file
File~	Emacs-created backup
File.tex	Input for TeX
File.mp3	Audio encoded in MPEG layer 3
File.pdf	Adobe Portable Document Format
File.ps	Postscript file

File structures



Baskin Engineering
UC SANTA CRUZ

Internal file structure



Baskin Engineering UC SANTA CRUZ

Accessing a file

- Sequential access
  - Read all bytes/records from the beginning
  - Cannot jump around
    - May rewind or back up, however
  - Convenient when medium was magnetic tape
  - Often useful when whole file is needed
- Random access
  - Bytes (or records) read in any order
  - Essential for database systems
  - Read can be …
    - Move file marker (seek), then read or …
    - Read and then move file marker

File attributes (possibilities)

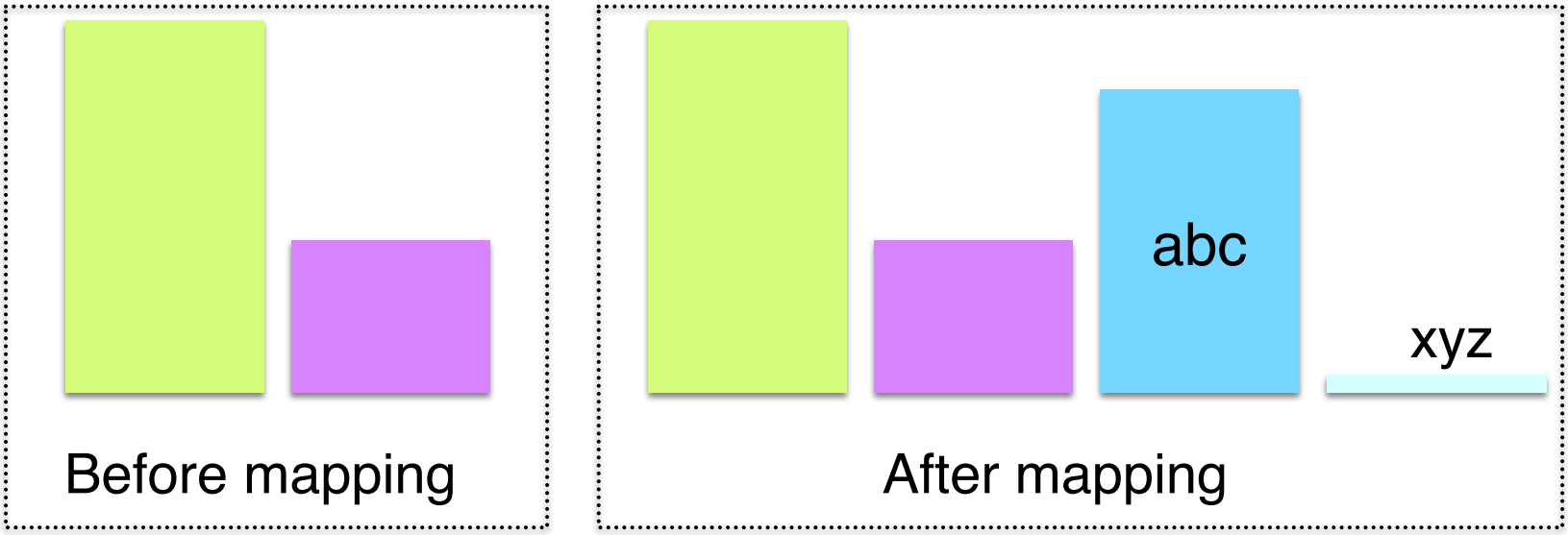
Attribute	Meaning
Protection	Who can access the file and how
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner of the file
Read-only flag	Indicates whether file may be modified
Hidden flag	Indicates whether file should be displayed in listings
System flag	Is this an OS file?
Archived flag	Has this file been backed up?
ASCII/binary flag	Is this file ASCII or binary?
Random access flag	Is random access allowed on this file?
Temporary flag	Is this a temporary file?
Lock flags	Used for locking a file to get exclusive access
Record length	Number of bytes in a record
Key position / length	Location & length of the key
Creation time	When was the file created?
Last access time	When was the file last accessed?
Last modified time	When was the file last modified?
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Baskin Engineering UC SANTA CRUZ

File operations

- Create: make a new file
- Delete: remove an existing file
- Open: prepare a file to be accessed
- Close: indicate that a file is no longer being accessed
- Read: get data from a file
- Write: put data to a file
- Append: like write, but only at the end of the file
- Seek: move the “current” pointer elsewhere in the file
- Get attributes: retrieve attribute information
- Set attributes: modify attribute information
- Rename: change a file’s name

Memory-mapped files



- Segmented process before mapping files into its address space
- Process after mapping
  - Existing file abc into one segment
  - Creating new segment for xyz

Baskin Engineering
UC SANTA CRUZ

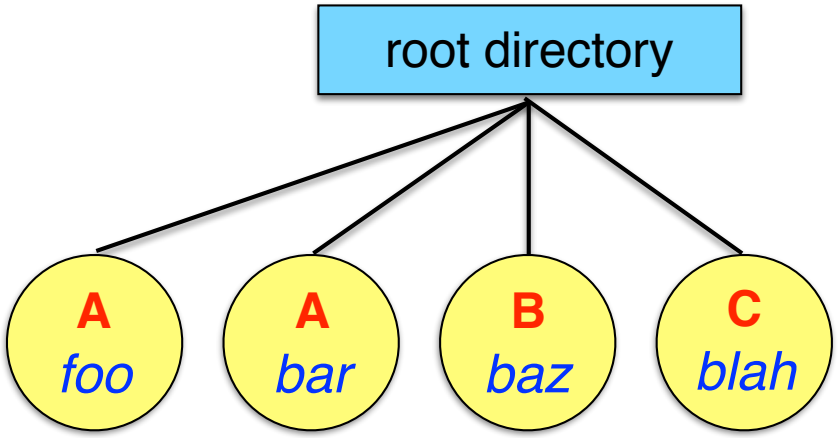
More on memory-mapped files

- Memory-mapped files are a convenient abstraction
  - Example: string search in a large file can be done just as with memory!
  - Let the OS do the buffering (reads & writes) in the virtual memory system
- Some issues come up…
  - How long is the file?
    - Easy if read-only
    - Difficult if writes allowed: what if a write is past the end of file?
  - What happens if the file is shared: when do changes appear to other processes?
  - When are writes flushed out to disk?
- Clearly, easier to memory map read-only files…

Directories

- Naming is nice, but limited
- Humans like to group things together for convenience
- File systems allow this to be done with directories (sometimes called folders)
- Grouping makes it easier to
  - Find files in the first place: remember the enclosing directories for the file
  - Locate related files (or just determine which files are related)

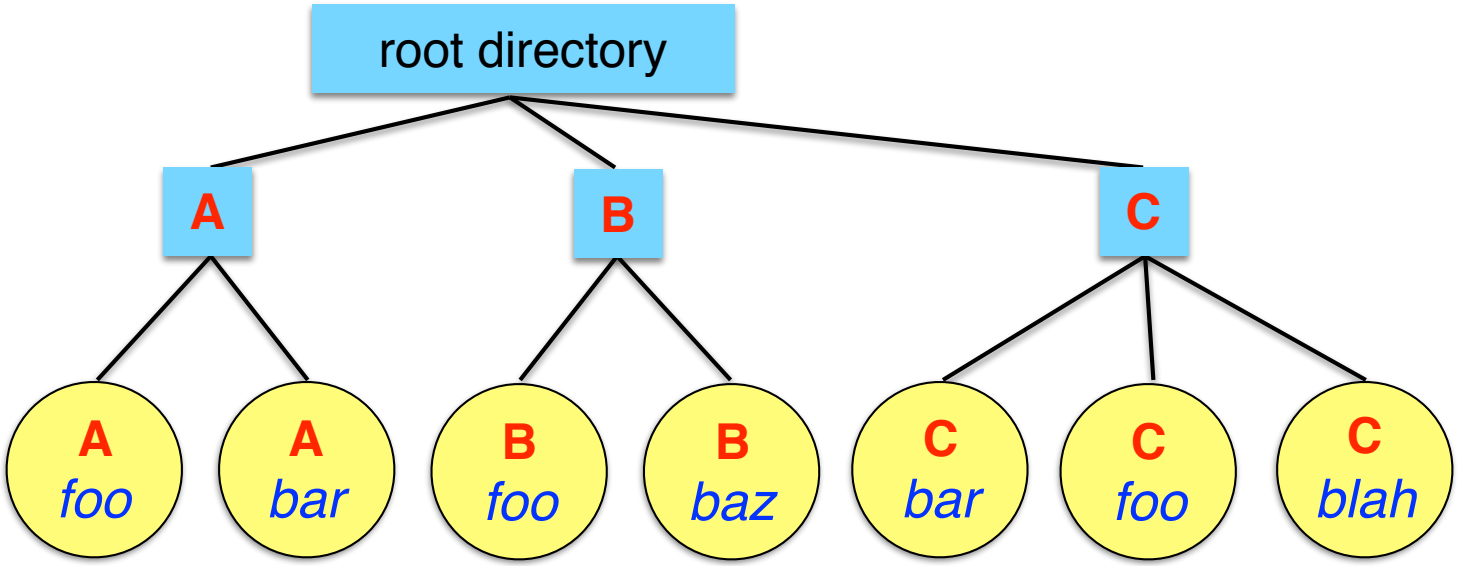
Single-level directory systems



- One directory in the file system
- Example directory
  - Contains 4 files (foo, bar, baz, blah)
  - owned by 3 different people: A, B, and C (owners shown in red)
- Problem: what if user B wants to create a file called foo?

Baskin Engineering UC SANTA CRUZ

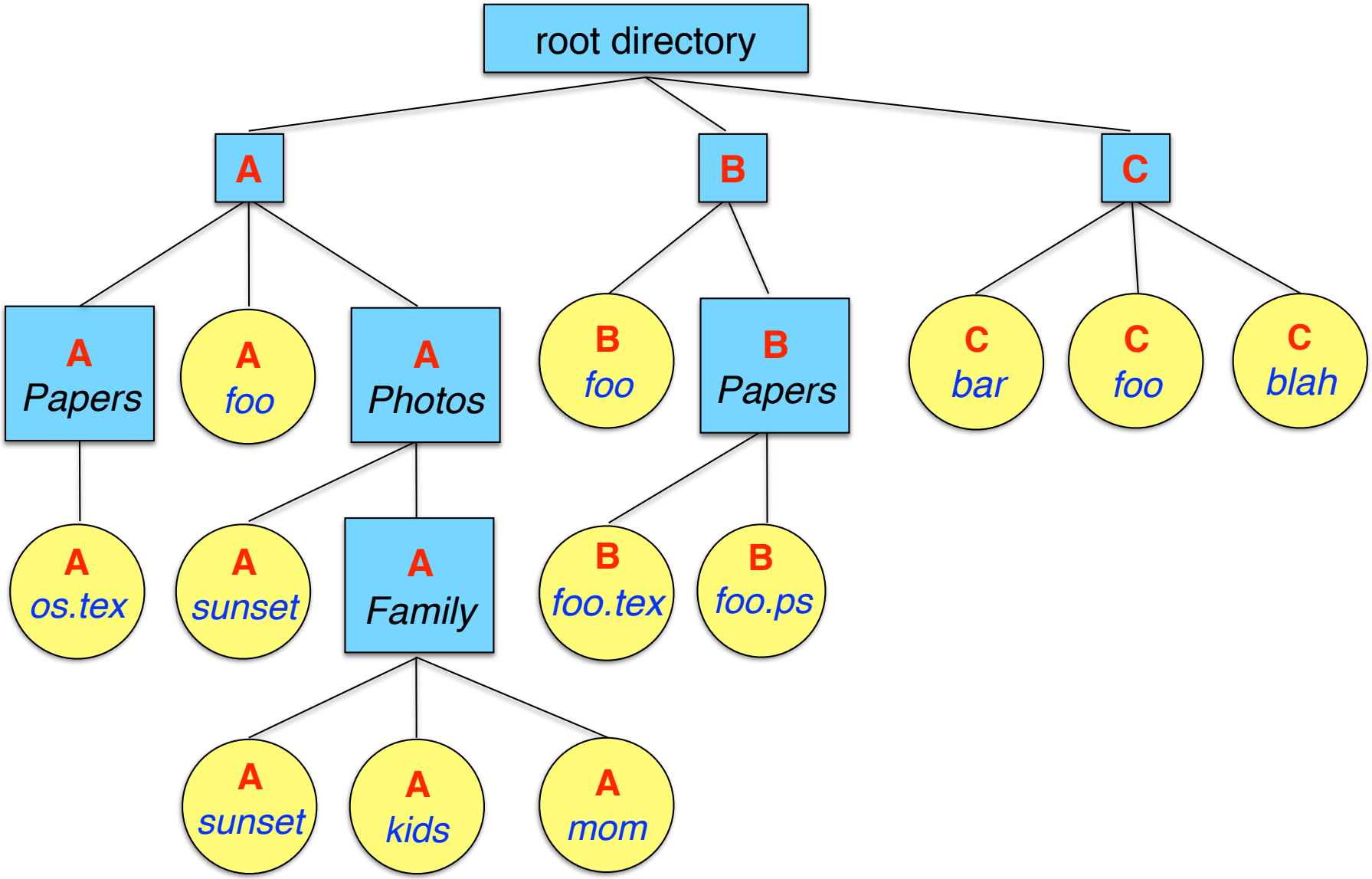
Two-level directory system



- Solves naming problem: each user has her own directory
- Multiple users can use the same file name
- By default, users access files in their own directories
- Extension: allow users to access files in others' directories

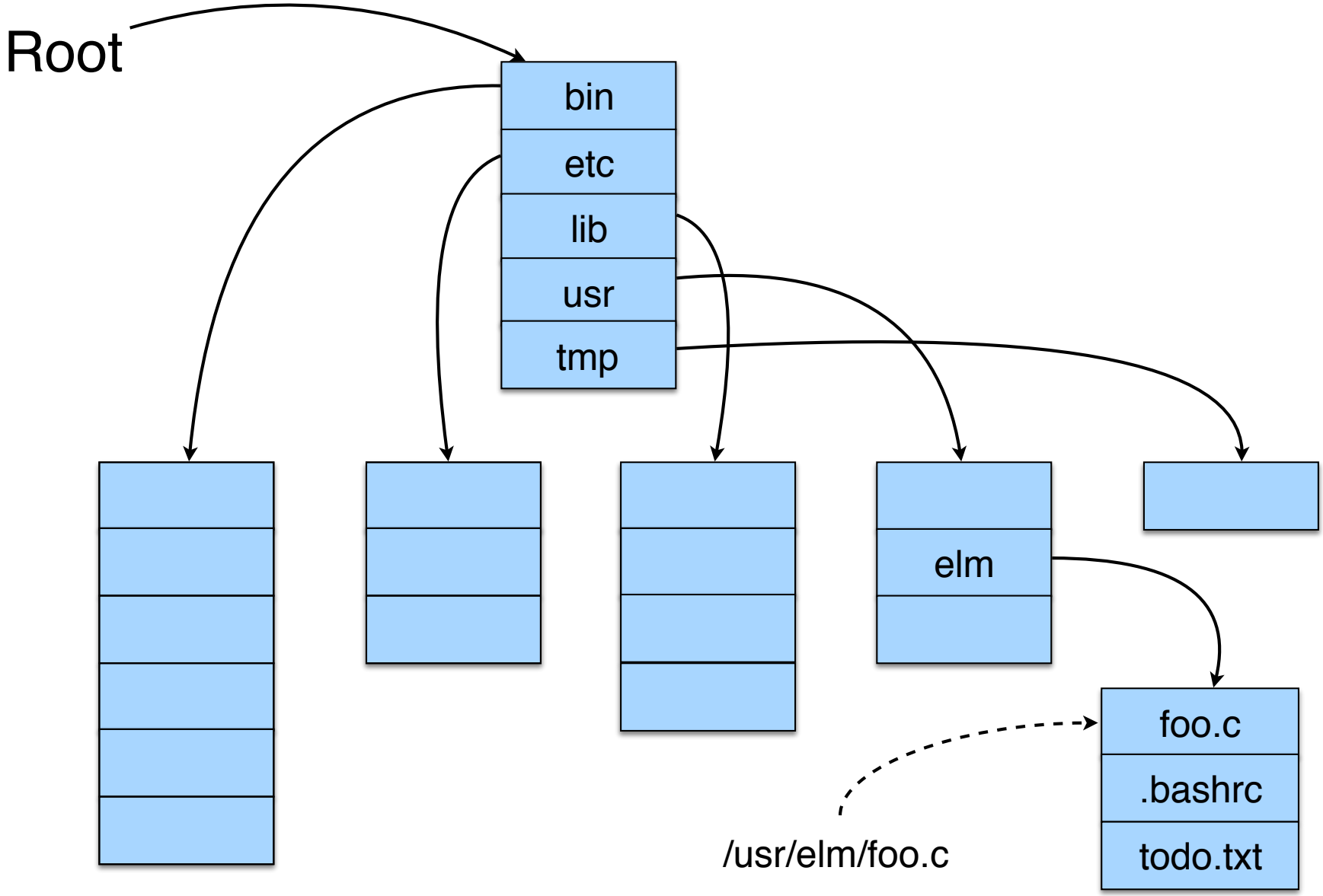
Baskin Engineering UC SANTA CRUZ

Hierarchical directory system



Baskin Engineering
UC SANTA CRUZ

Unix directory tree



Baskin Engineering
UC SANTA CRUZ

Operations on directories

- Create: make a new directory
- Delete: remove a directory (usually must be empty)
- Opendir: open a directory to allow searching it
- Closedir: close a directory (done searching)
- Readdir: read a directory entry
- Rename: change the name of a directory
  - Similar to renaming a file
- Link: create a new entry in a directory to link to an existing file
- Unlink: remove an entry in a directory
  - Remove the file if this is the last link to this file

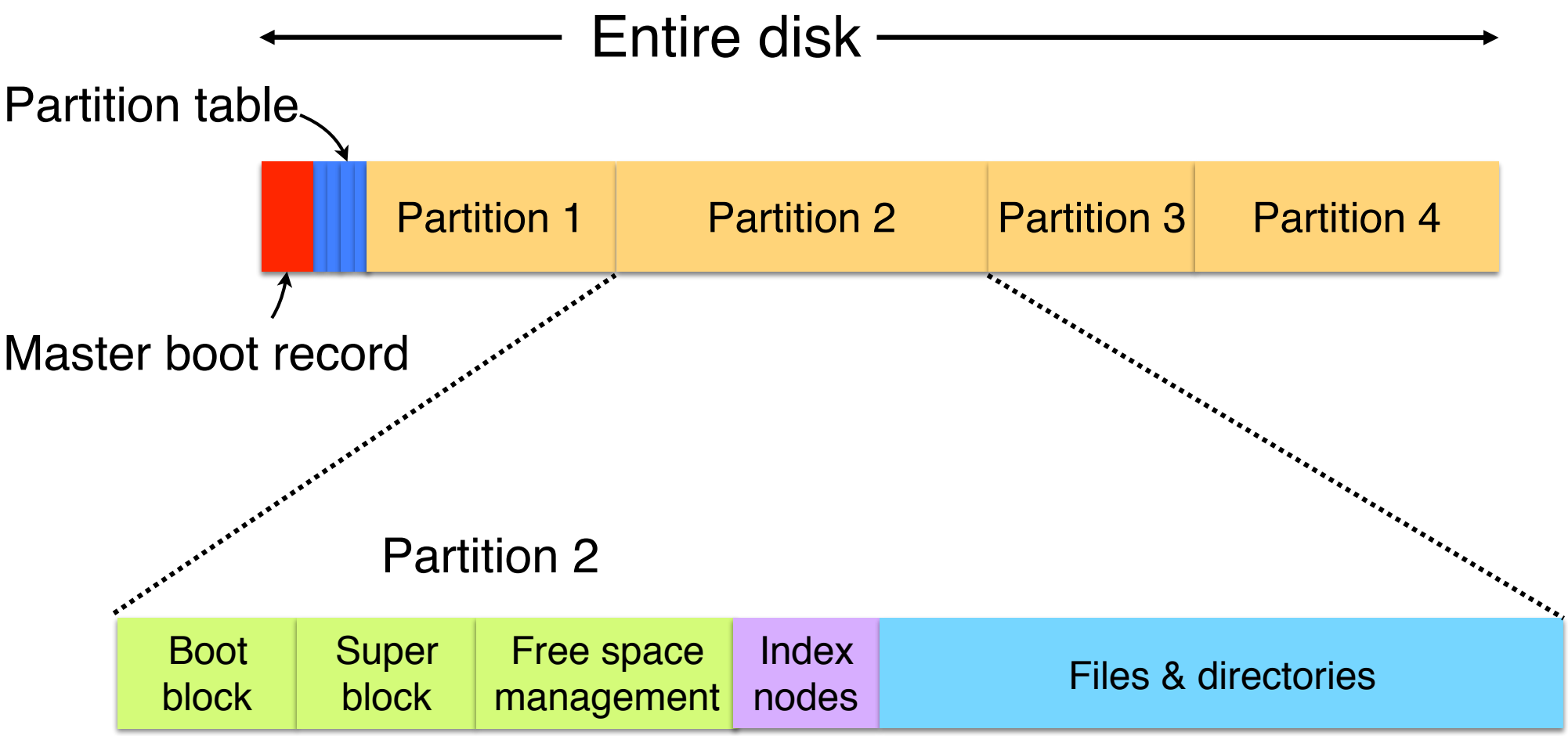
Baskin Engineering UC SANTA CRUZ

File system implementation issues

- How are disks divided up into file systems?
- How does the file system allocate blocks to files?
- How does the file system manage free space?
- How are directories handled?
- How can the file system improve…
  - Performance?
  - Reliability?

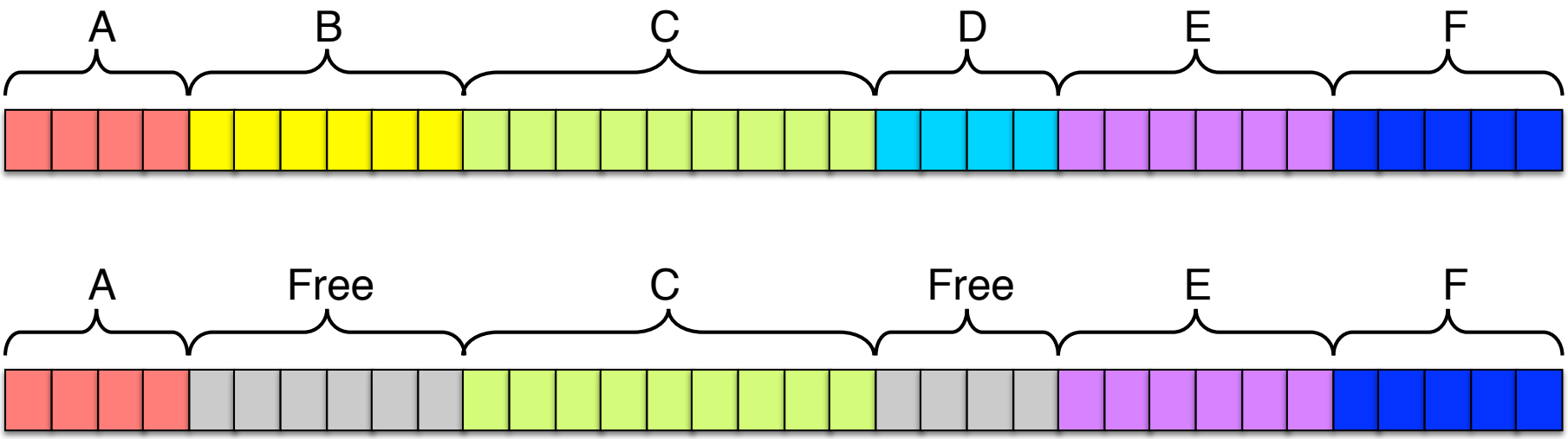
Baskin Engineering UC SANTA CRUZ

Carving up the disk



Baskin Engineering
UC SANTA CRUZ

Contiguous allocation for file blocks



- Contiguous allocation requires all blocks of a file to be consecutive on disk
- Problem: deleting files leaves “holes”
  - Similar to memory allocation issues
  - Compacting the disk can be a very slow procedure…

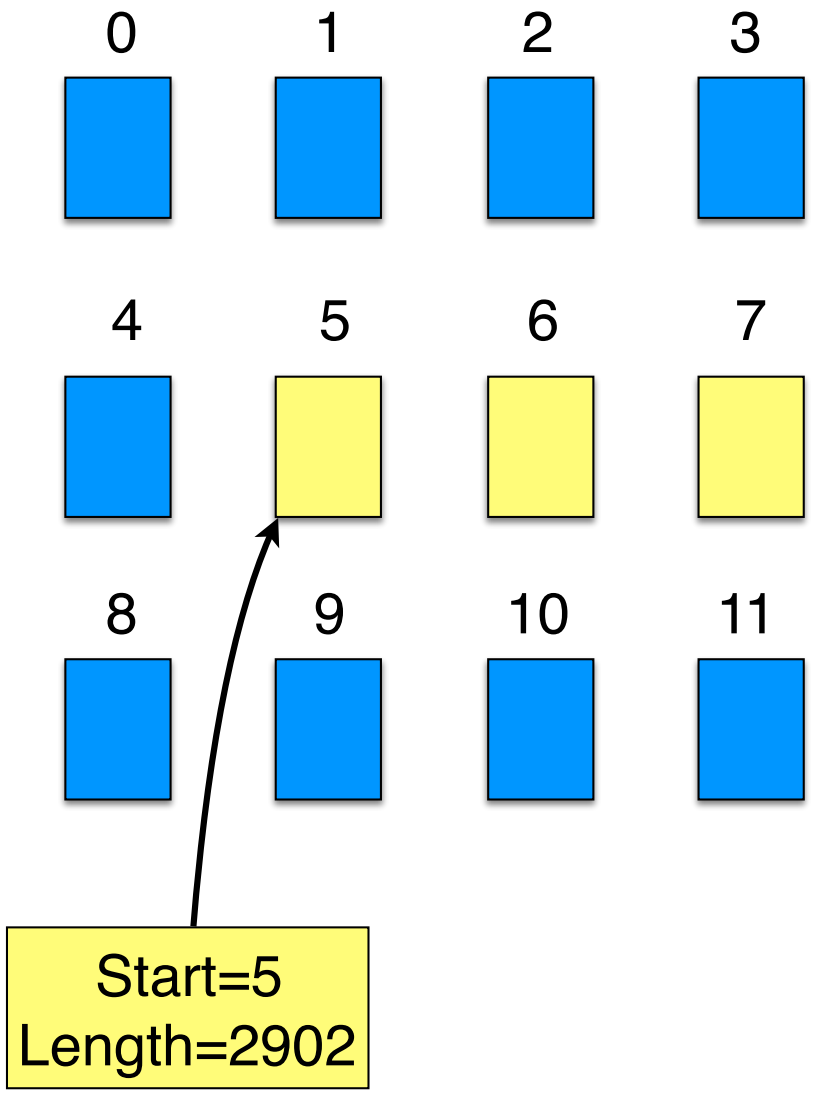
Baskin Engineering UC SANTA CRUZ

Contiguous allocation

- Data in each file is stored in consecutive blocks on disk
- Simple & efficient indexing
  - Starting location (block #) on disk (start)
  - Length of the file in blocks (length)
- Random access well-supported
- Difficult to grow files
  - Must pre-allocate all needed space
  - Wasteful of storage if file isn't using all of the space
- Logical to physical mapping is easy

  blocknum = (pos / 1024)+start;

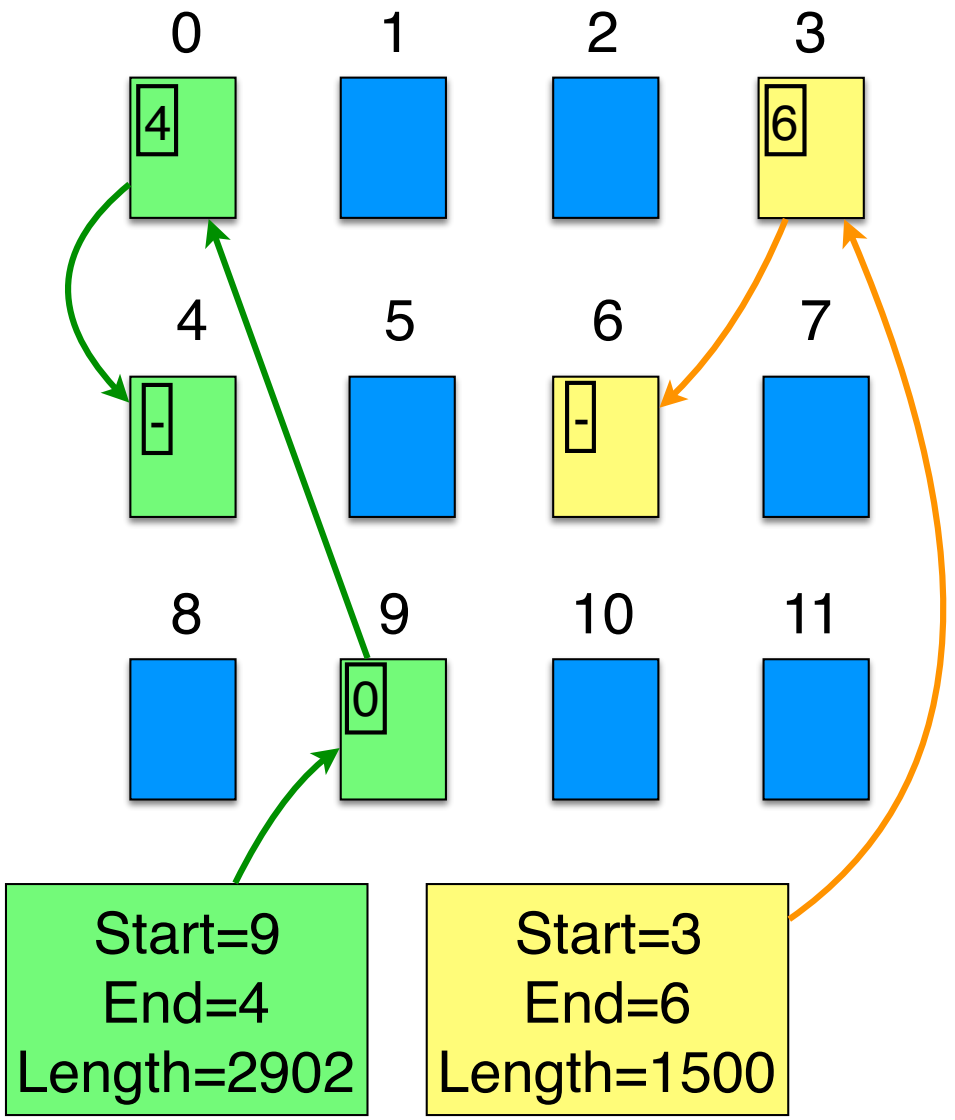
  offset_in_block = pos % 1024;



Baskin Engineering UC SANTA CRUZ

Linked allocation

- File is a linked list of disk blocks
  - Blocks may be scattered around the disk drive
  - Block contains both pointer to next block and data
  - Files may be as long as needed
- New blocks are allocated as needed
  - Linked into list of blocks in file
  - Removed from list (bitmap) of free blocks



Baskin Engineering
UC SANTA CRUZ

Finding blocks with linked allocation

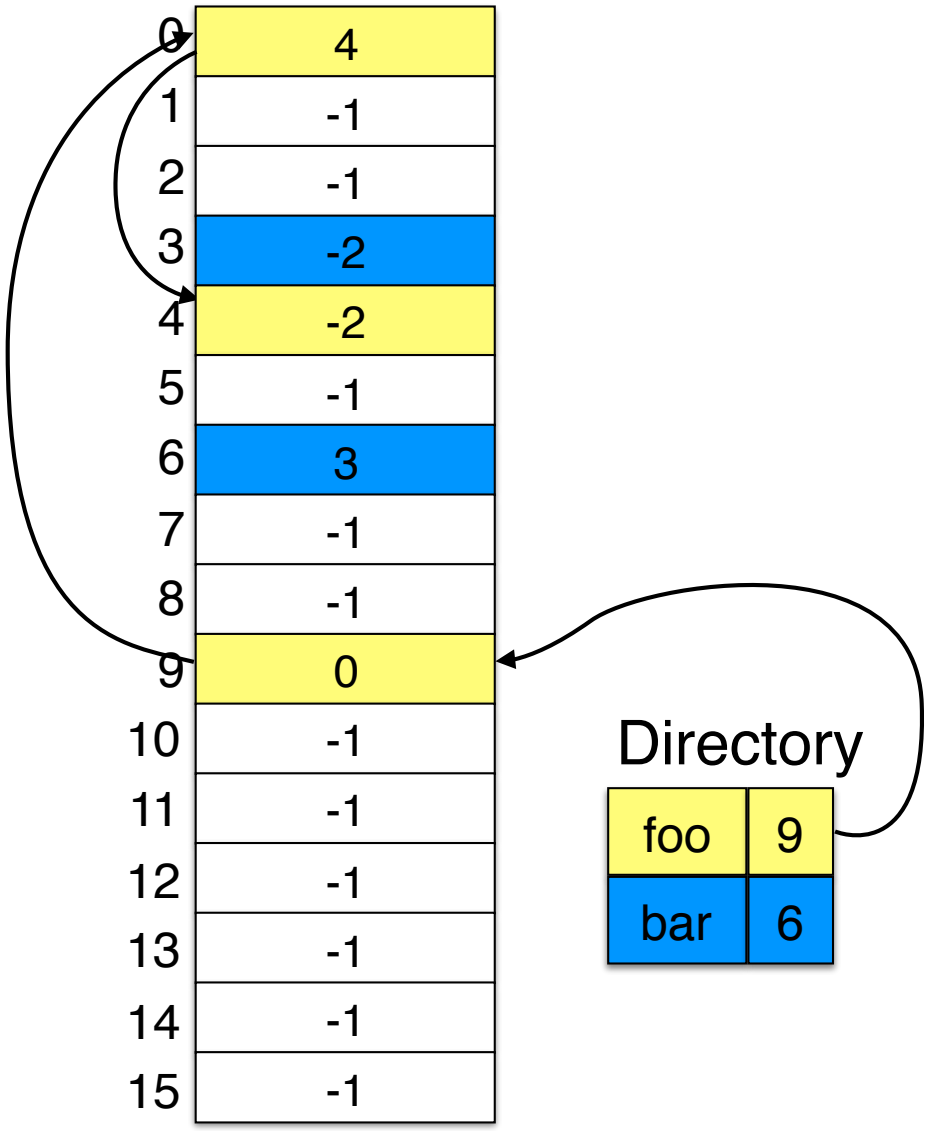
- Directory structure is simple
  - Starting address looked up from directory
  - Directory only keeps track of first block (not others)
- No wasted space—all blocks can be used
- Random access is difficult: must always start at first block!
- Logical to physical mapping is done by

```
block = start;
offset_in_block = pos % 1020;
for (j = 0; j < pos / 1020; j++) {
  block = block->next;
}
```

  - Assumes that next pointer is stored at end of block
  - May require a long time for seek to random location in file

Baskin Engineering UC SANTA CRUZ

Linked allocation using a table in RAM

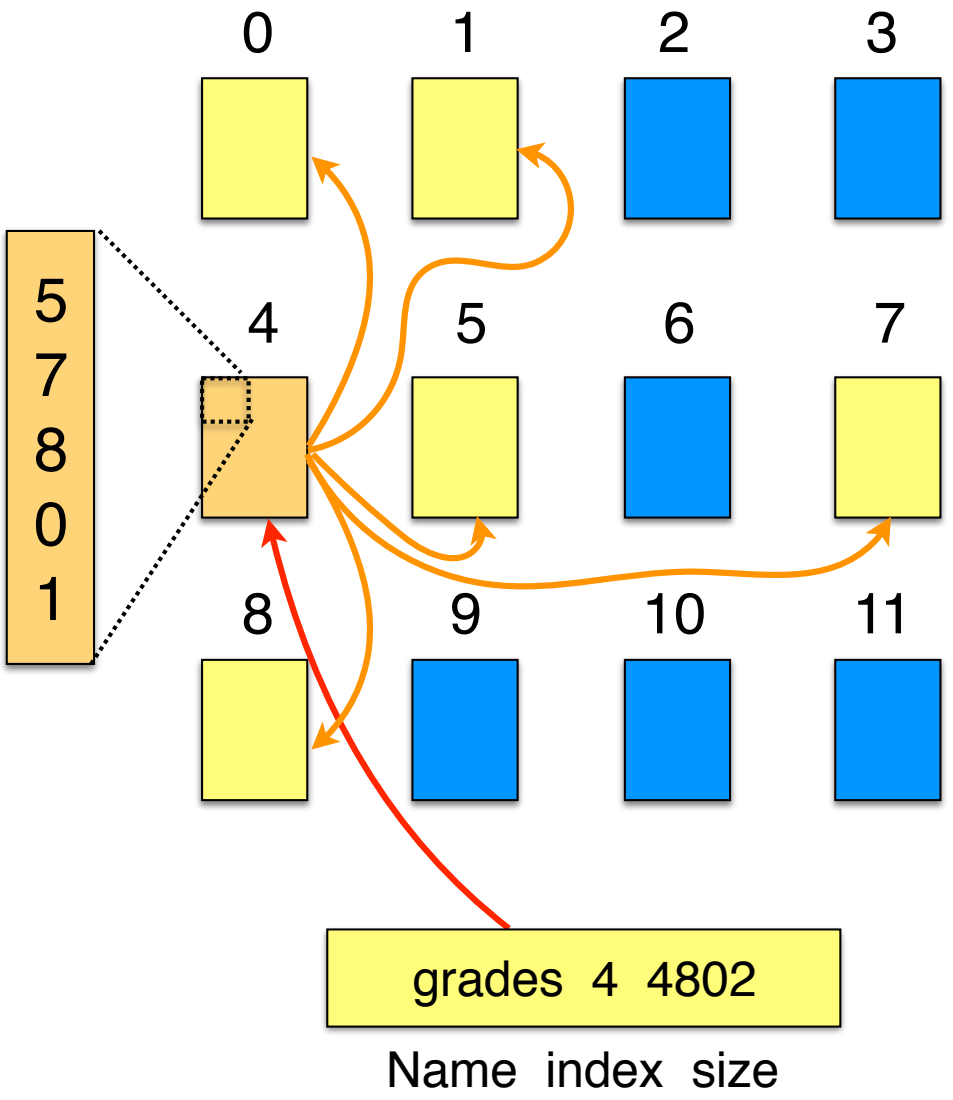


- Links on disk are slow
- Keep linked list in memory
- Advantages
  - Faster!
  - Disk blocks aren't an odd size
- Disadvantages
  - Have to read it from disk at some point (startup?)
  - **Have to keep in-memory and on-disk copy consistent**

Baskin Engineering UC SANTA CRUZ

Using a block index for allocation

- Store file block addresses in an array
  - Array itself is stored in a disk block
  - Directory has a pointer to this disk block
  - Non-existent blocks indicated by -1
- Random access easy
- Limit on file size?



Baskin Engineering UC SANTA CRUZ

Finding blocks with indexed allocation

- Need location of index table: look up in directory
- Random & sequential access both well-supported: look up block number in index table
- Space utilization is good
  - No wasted disk blocks (allocate individually)
  - Files can grow and shrink easily
  - Overhead of a single disk block per file
- Logical to physical mapping is done by
  block = index[pos / 1024];
  offset_in_block = pos % 1024;
- Limited file size: 256 pointers per index block, 1 KB per file block
  ⇒ 256 KB per file limit

Larger files with indexed allocation

- How can indexed allocation allow files larger than a single index block?
- Linked index blocks: similar to linked file blocks, but using index blocks instead
- Logical to physical mapping is done by

```
index = start;
blocknum = pos / 1024;
for (j = 0; j < blocknum /255); j++) {
  index = index->next;
}
block = index[blocknum % 255];
offset_in_block = pos % 1024;
```

- File size is now unlimited
- Random access slow, but only for very large files

Two-level indexed allocation

- Allow larger files by creating an index of index blocks
  - File size still limited, but much larger
  - Limit for 1 KB blocks = 1 KB × 256 × 256 = 2^{26} bytes = 64 MB
- Logical to physical mapping is done by
  blocknum = pos / 1024;
  index = start[blocknum / 256)];
  block = index[blocknum % 256]
  offset_in_block = pos % 1024;
  - Start is the only pointer kept in the directory
  - Overhead is now at least two blocks per file
- This can be extended to more than two levels if larger files are needed...

Baskin Engineering UC SANTA CRUZ

Block allocation with extents

- Reduce space consumed by index pointers
  - Often, consecutive blocks in file are sequential on disk
  - Store ⟨*block*,*count*⟩ instead of just ⟨*block*⟩ in index
  - At each level, keep total count for the index for efficiency
- Lookup procedure is:
  - Find correct index block by checking the starting file offset for each index block
  - Find correct ⟨*block*,*count*⟩ entry by running through index block, keeping track of how far into file the entry is
  - Find correct block in ⟨*block*,*count*⟩ pair
- More efficient if file blocks tend to be consecutive on disk
  - Allocating blocks like this allows faster reads & writes
  - Lookup is somewhat more complex

Baskin Engineering UC SANTA CRUZ

Managing free space: bit vector

- Keep a bit vector, with one entry per file block
  - Number bits from 0 through n-1, where n is the number of blocks available for files on the disk
  - If bit[j] == 0, block j is free
  - If bit[j] == 1, block j is in use by a file (for data or index)
- If words are 32 bits long, calculate appropriate bit by:
  wordnum = block / 32;
  bitnum = block % 32;
- Search for free blocks by looking for words with bits unset:
  words != 0xffffffff
- Easy to find consecutive blocks for a single file
- Bit map must be stored on disk, and consumes space
  - Assume 4 KB blocks, 256 GB disk ➡ 64M blocks
  - 64M bits = 2^{26} bits = 2^{23} bytes = 8 MB overhead

Managing free space: linked list

- Use a linked list to manage free blocks
  - Similar to linked list for file allocation
  - No wasted space for bitmap
  - No need for random access unless we want to find consecutive blocks for a single file
- Difficult to know how many blocks are free unless it's tracked elsewhere in the file system
- Difficult to group nearby blocks together if they're freed at different times
  - Less efficient allocation of blocks to files
  - Files read & written more because consecutive blocks not nearby

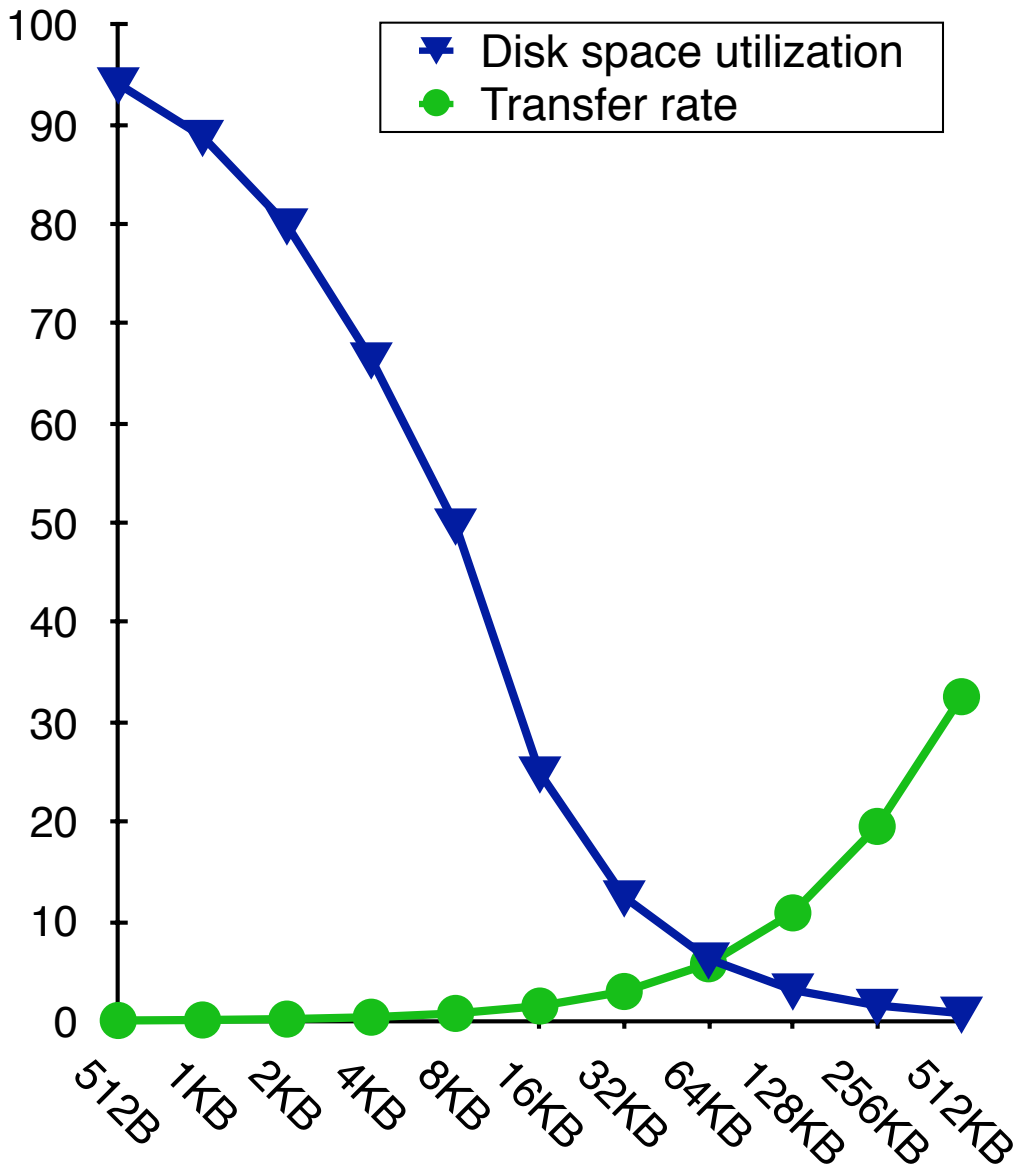
Baskin Engineering UC SANTA CRUZ

Issues with free space management

- OS must protect data structures used for free space management
- OS must keep in-memory and on-disk structures consistent
  - Update free list when block is removed: change a pointer in the previous block in the free list
  - Update bit map when block is allocated
    - Caution: on-disk map must never indicate that a block is free when it's part of a file
    - Solution: set bit[j] in free map to 1 on disk before using block[j] in a file and setting bit[j] to 1 in memory
    - New problem: OS crash may leave bit[j] == 1 when block isn't actually used in a file
    - New solution: OS checks the file system when it boots up…
- Managing free space is a big source of slowdown in file systems

Baskin Engineering UC SANTA CRUZ

Big or small file blocks?

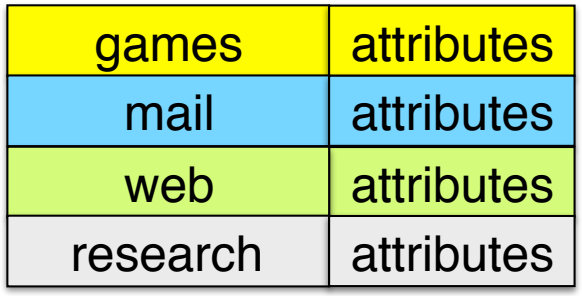


- Green line: transfer efficiency
  - Percent of maximum bandwidth
- Blue line: disk space efficiency
- Files random size 0–8KB
- Larger blocks are
  - Faster: transfer more data per seek
  - Less efficient: waste space

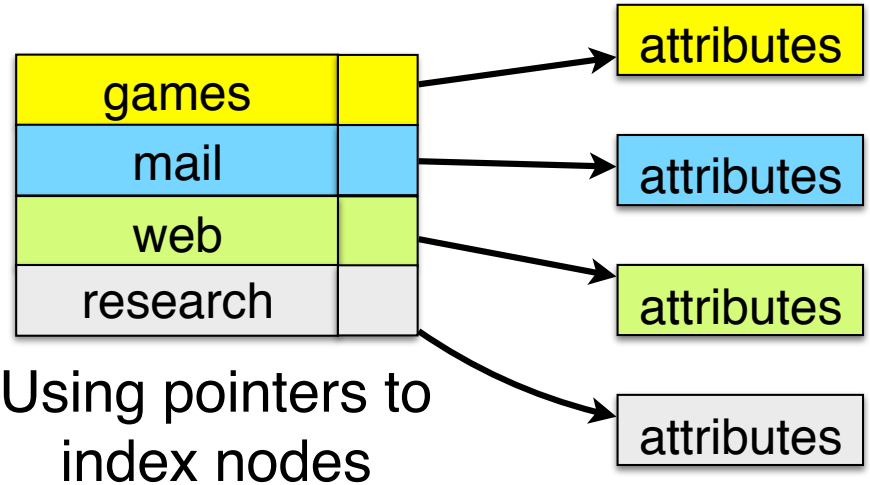
Baskin Engineering
UC SANTA CRUZ

What's in a directory?

- Two types of information
  - File names
  - File metadata (size, timestamps, etc.)
- Basic choices for directory information
  - Store all information in directory
    - Fixed size entries
    - Disk addresses and attributes in directory entry
  - Store names & pointers to index nodes (i-nodes)



Storing all information in the directory



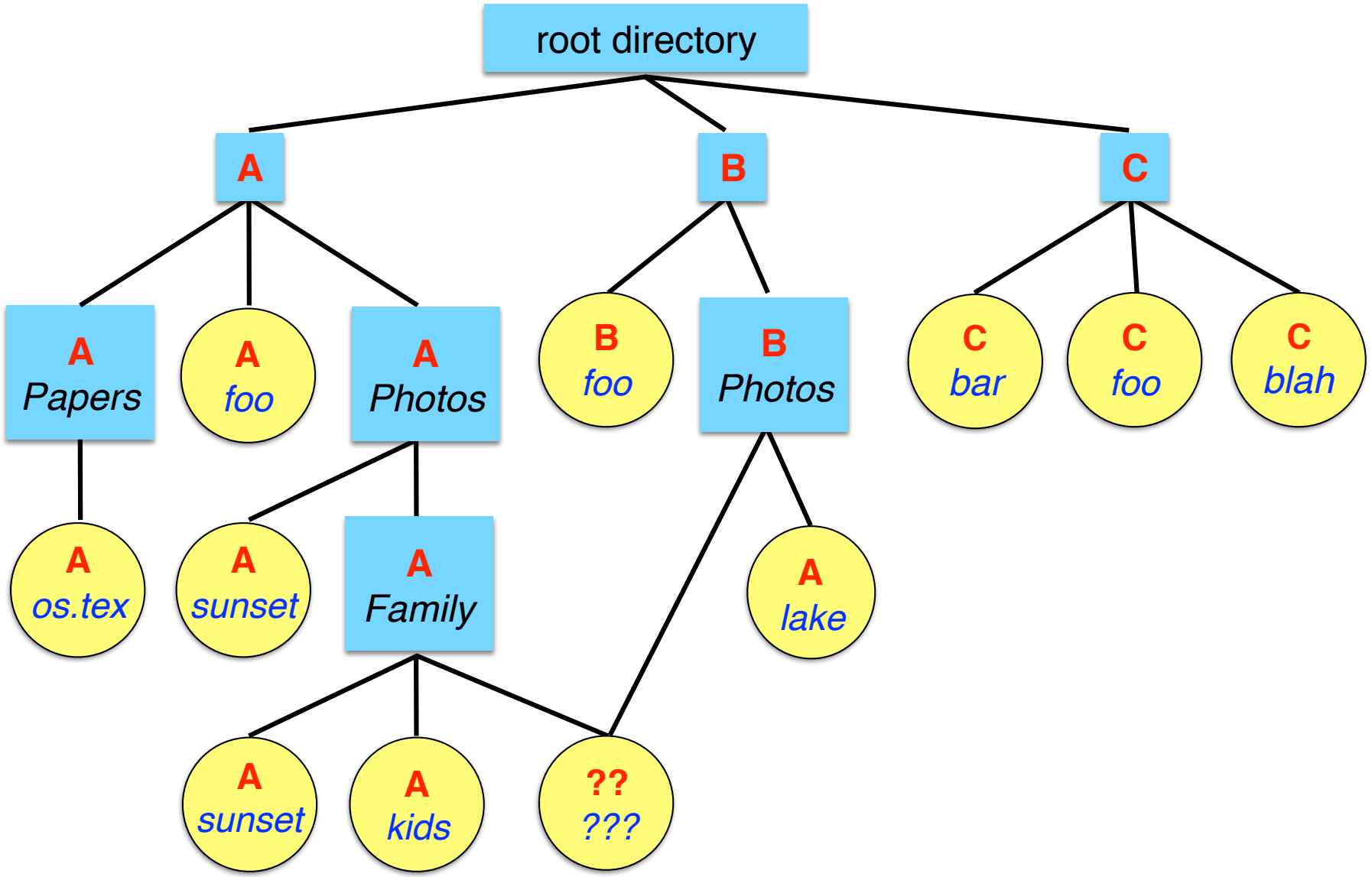
Using pointers to index nodes

Baskin Engineering UC SANTA CRUZ

Directory structure

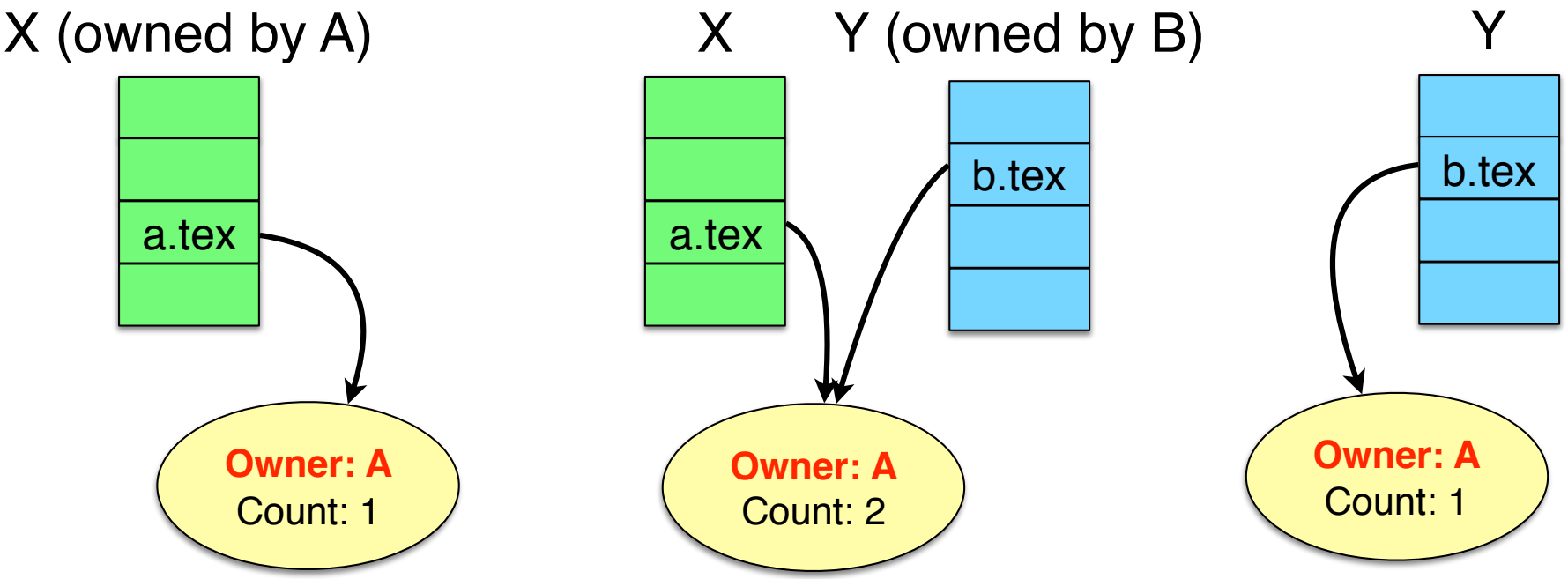
- Structure
  - Linear list of files (often itself stored in a file)
    - Simple to program
    - Slow to run
    - Increase speed by keeping it sorted (insertions are slower!)
  - Hash table: name hashed and looked up in file
    - Decreases search time: no linear searches!
    - May be difficult to expand
    - Can result in collisions (two files hash to same location)
  - Tree
    - Fast for searching
    - Easy to expand
    - Difficult to do in on-disk directory
- Name length
  - Fixed: easy to program
  - Variable: more flexible, better for users

Sharing files



Solution: use links

- A creates a file, and inserts into her directory
- B shares the file by creating a link to it
- A unlinks the file
  - B still links to the file
  - Owner is still A (unless B explicitly changes it)



Baskin Engineering UC SANTA CRUZ

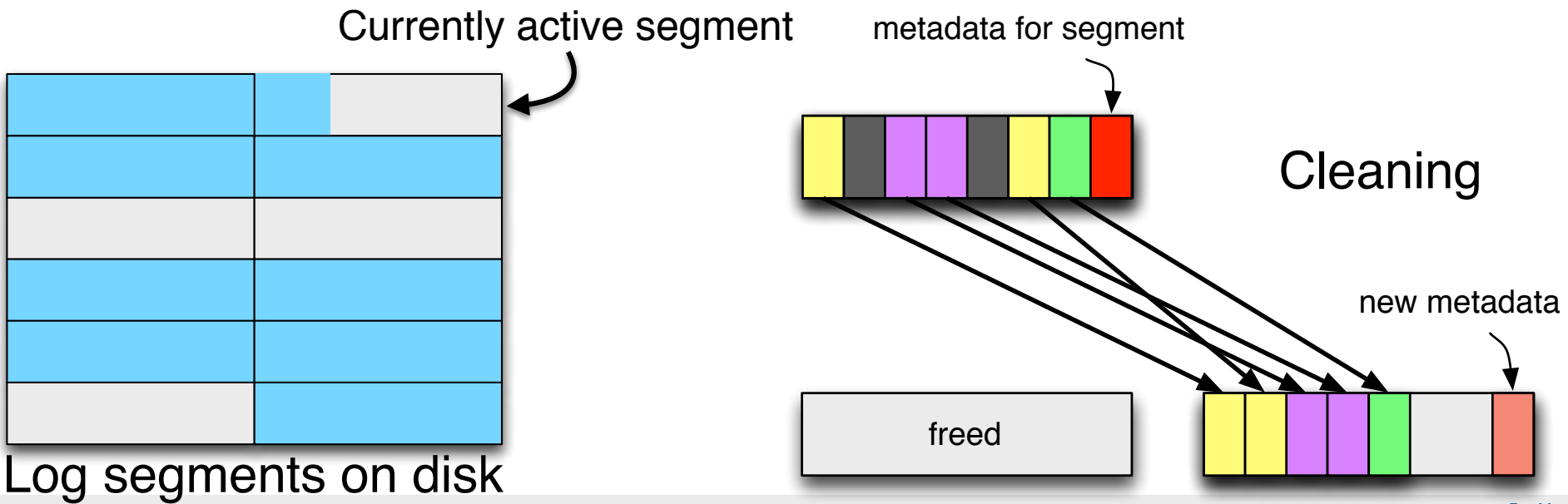
Log-structured file systems

- Trends in disk & memory
  - Faster CPUs
  - Larger memories
- Result
  - More memory ➔ disk caches can also be larger
  - Increasing number of read requests can come from cache
  - Thus, most disk accesses will be writes
- LFS structures entire disk as a log
  - All writes initially buffered in memory
  - Periodically write these to the end of the disk log
  - When file opened, locate i-node, then find blocks
- Issue: what happens when blocks are deleted?

Baskin Engineering UC SANTA CRUZ

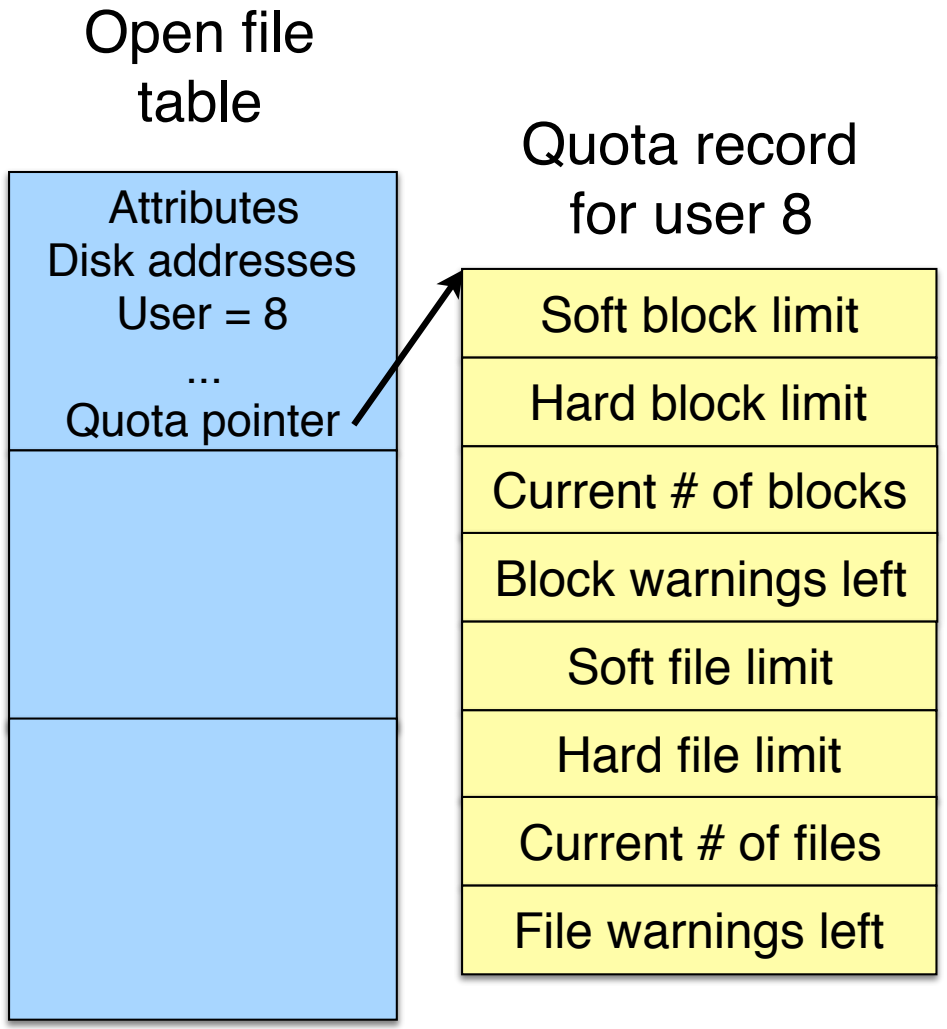
Log-structured file systems

- Divide disk into segments
  - Write data sequentially to segments
  - Read data from wherever it's stored
- Periodically, "clean" segments
  - Go through a segment and copy live data to a new location
  - Mark the segment as free



Disk quotas

- Quotas limit users' allocation
  - Space might be rented
  - Protect the system from rogue programs allocating space
- Hard limits may not be exceeded
- Soft limits generate warnings
  - Number of warnings may be limited
  - Generate too many warnings → treat as hard limit

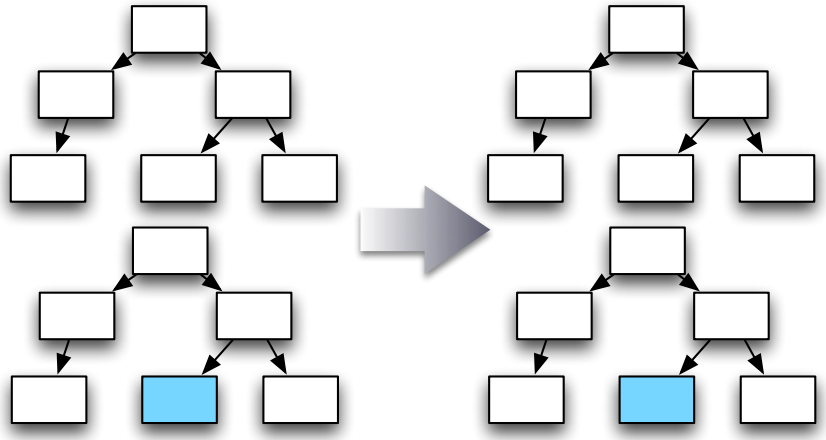


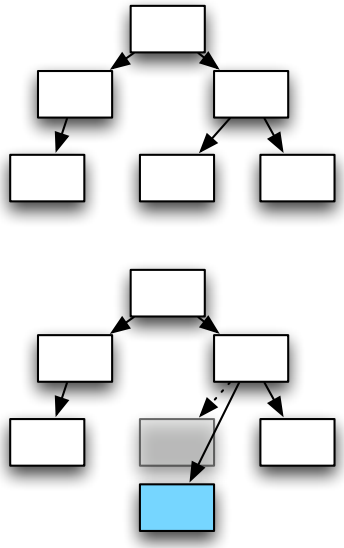
Backing up a file system

- Goal: create an extra copy of each file in the file system
  - Protect against disk failure
  - Protect against human error (rm * .o)
  - Allow the system to track changes over time
- Two basic types of backups
  - Full backup: make a copy of every file in the system
  - Incremental backup: only make a copy of files that have changed since the last backup
    - Faster: fewer files to copy
    - Smaller
- Incremental backups typically require a full backup as a “base”

Backup mechanics

- Actually copy blocks from one file system to another
  - Safe if original FS fails
  - Safe if original FS is corrupted
  - May be difficult to find modified files
  - Somewhat slow
- Snapshot
  - New data doesn't overwrite old data
  - Easy to recover "deleted" files
  - Fast
  - Not as helpful for failed devices or corrupted FS
  - Snapshots can be done with hard links....

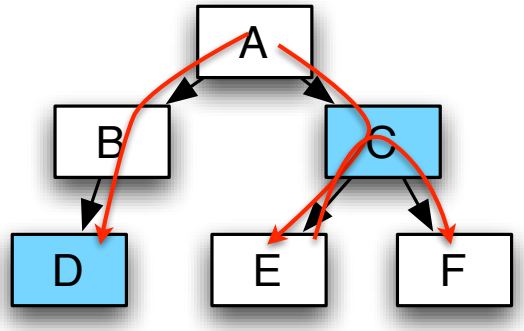


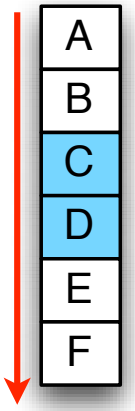


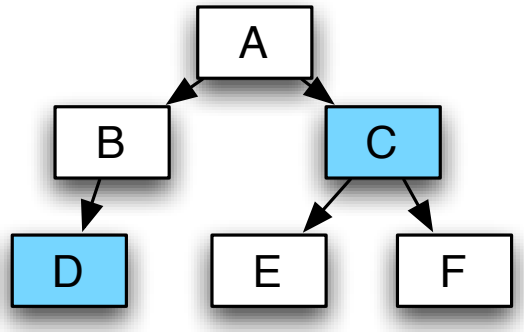
Finding files to back up

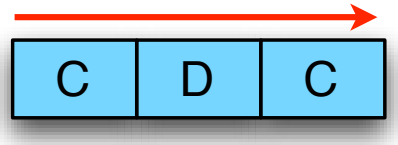
- "Walk" the file system tree looking for files modified since the last backup
  - Slow
  - Works on any file system
- Scan the list of inodes looking for modified files
  - Faster (sequential access)
  - FS-specific code (may require raw disk access)
- Keep a log of changes the FS makes
  - Can be very fast
  - Requires help from the FS

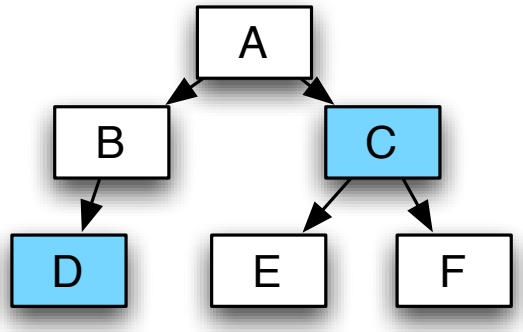
Files changed:
C (*t*=2)
D (*t*=4)
C (*t*=5)





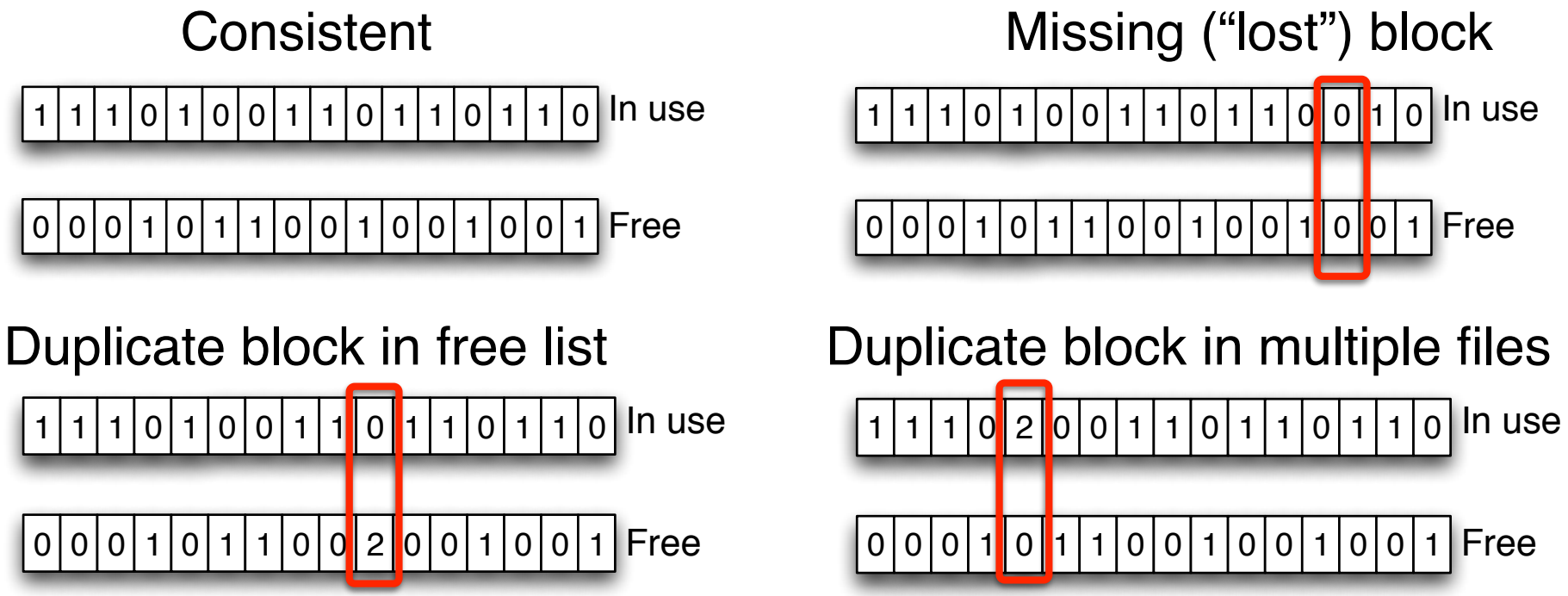






Baskin Engineering
UC SANTA CRUZ

Checking a file system for consistency



- Build “in use” map by traversing all file inodes
  - Add 1 to block entry if mentioned in block list
- Free map built from free list (or just use free block bitmap if it exists)

Baskin Engineering UC SANTA CRUZ

Fixing inconsistencies

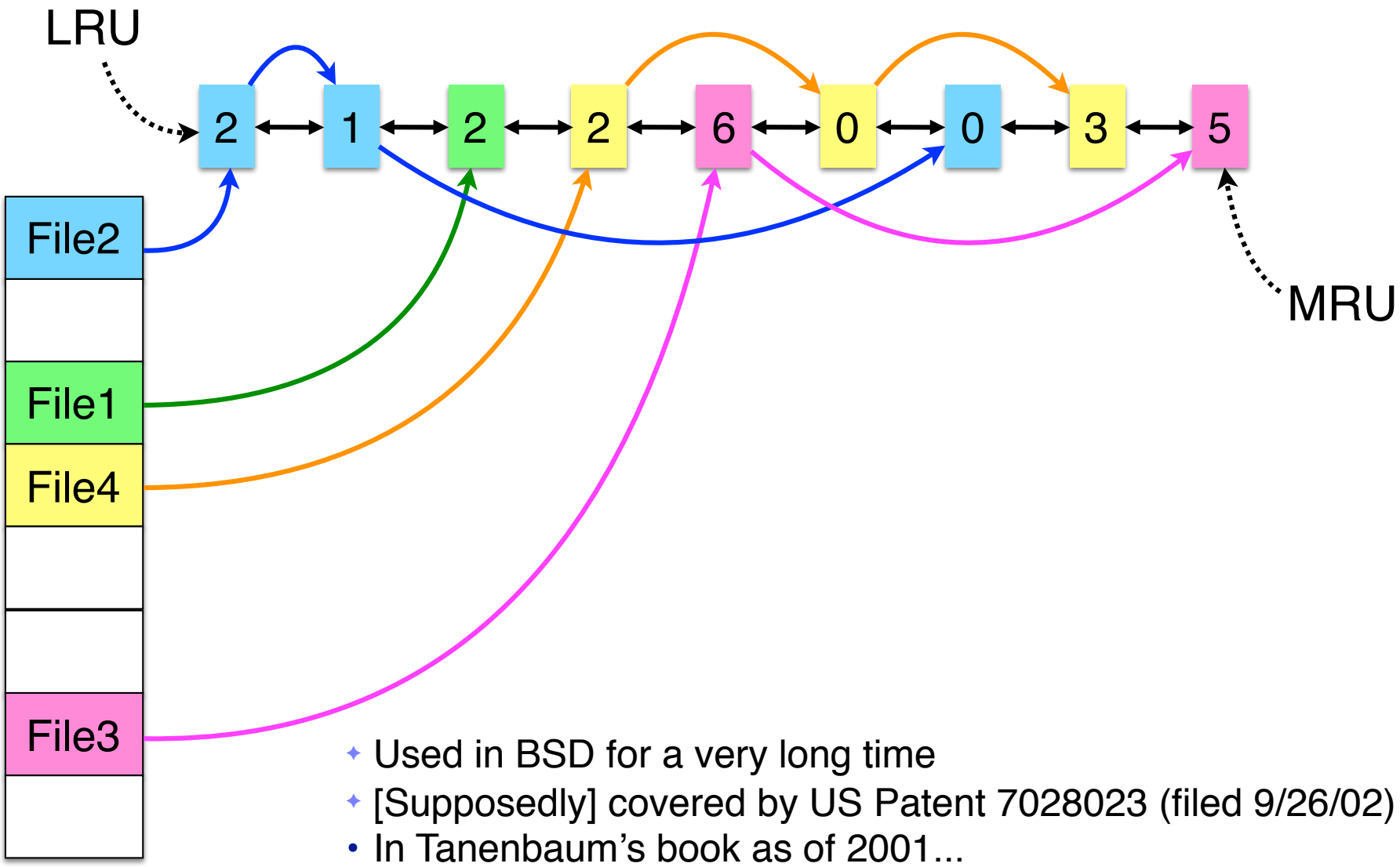
- If file system check reveals inconsistencies, what should be done?
- Block marked neither free nor in use: mark free
  - No file is using it, so mark it free
- Block marked both free and in use: mark in use
  - Best to give it to the file that's using it
- Block marked free more than once: mark free
  - Ensure that it's only on the free list once!
- Block in use in two files: ask user?
  - Difficult to decide what to do in this case
  - Consistency checking is designed to avoid having this happen: other cases are easy to handle!
- This is the reasoning behind ordered updates of data and metadata

Baskin Engineering UC SANTA CRUZ

File system cache

- Many files are used repeatedly
  - Option: read it each time from disk
  - Better: keep a copy in memory
- File system cache
  - Set of recently used file blocks
  - Keep blocks just referenced
  - Throw out old, unused blocks
    - Same kinds of algorithms as for virtual memory
    - More effort per reference is OK: file references are a lot less frequent than memory references
- Goal: eliminate as many disk accesses as possible!
  - Repeated reads & writes
  - Files deleted before they're ever written to disk

File block cache data structures



- Used in BSD for a very long time
- [Supposedly] covered by US Patent 7028023 (filed 9/26/02)
- In Tanenbaum's book as of 2001...

Managing the file cache: reads

- Update block lists on every read (and write)
  - Affordable: update is fast even compared to read of cached data
  - Evict blocks from cache using LRU, LFU, aging, or other VM-like algorithms
- Readahead: fetch blocks that might be needed soon
  - Example: read block *n*+1 after reading block *n* of a file
  - May be very inexpensive: disk controller has the block in its own cache
  - Particularly helpful if file is accessed sequentially: keep track of sequentiality to decide whether to read ahead
  - Not very detrimental if the OS guesses wrong: only waste a bit of disk bandwidth, and often no seek

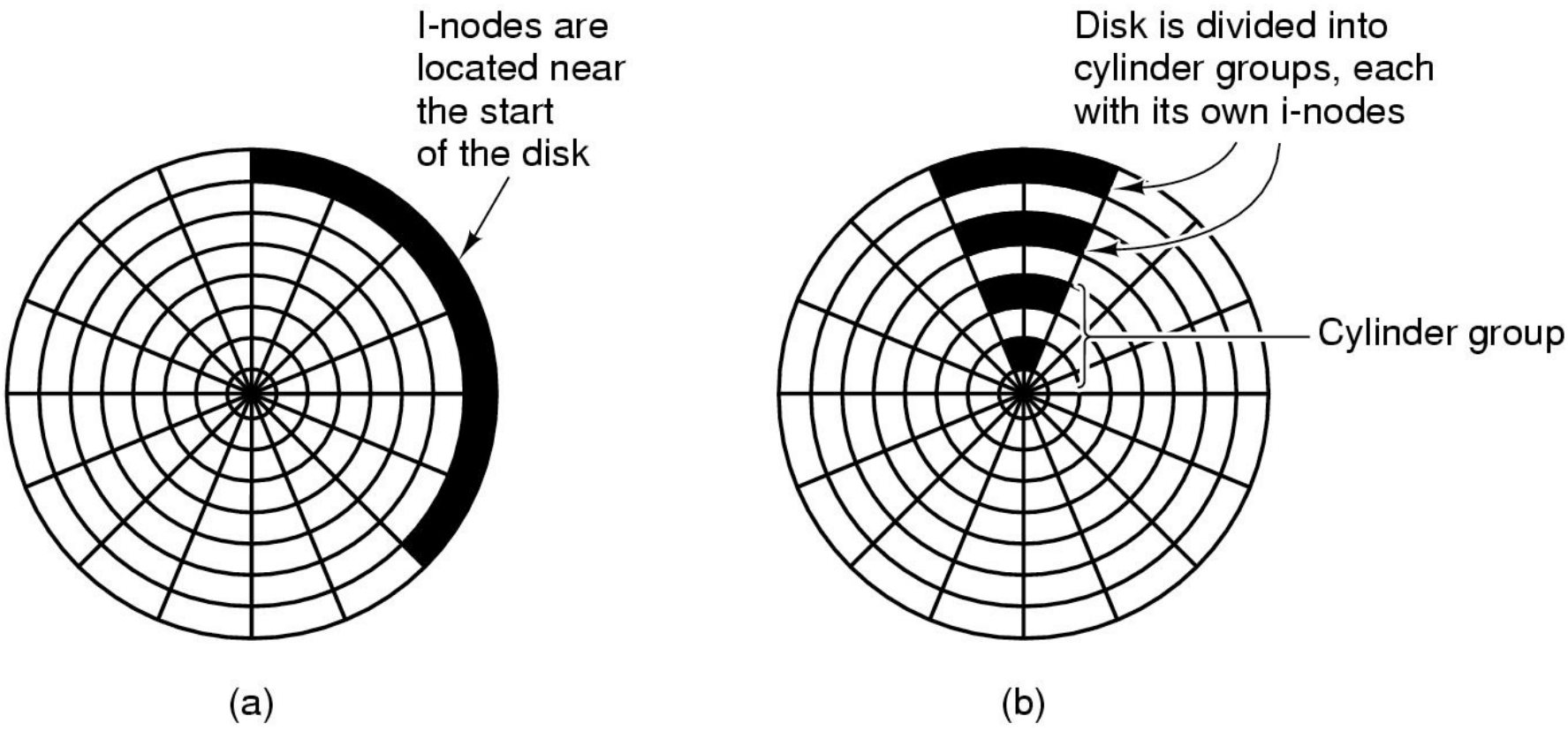
Baskin Engineering UC SANTA CRUZ

Managing the file cache: writes

- Buffer writes in cache (writeback cache)
- Periodically go through cache writing back dirty blocks
  - Typically about every 30 seconds on Unix
- Benefits
  - Multiple writes to a single file block are coalesced
  - Files that are created and deleted quickly are never written to disk!
- Cache can be flushed manually if needed: sync()
- Issue: should disk blocks be allocated on write to cache or on write to disk?

Baskin Engineering UC SANTA CRUZ

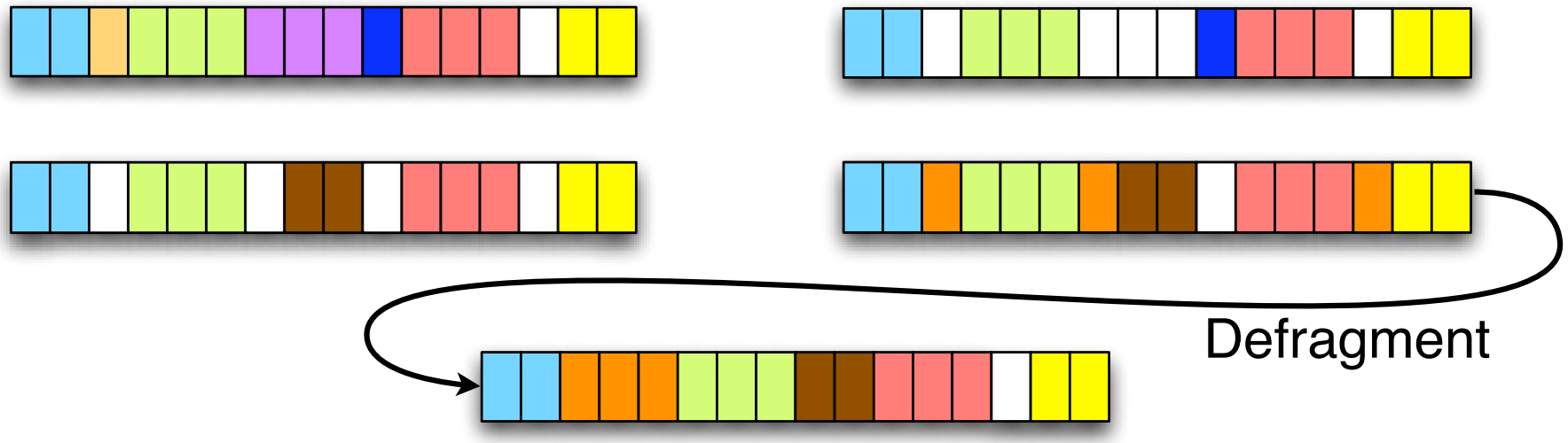
Grouping data on disk



Baskin
Engineering
UC SANTA CRUZ

Defragmenting disks

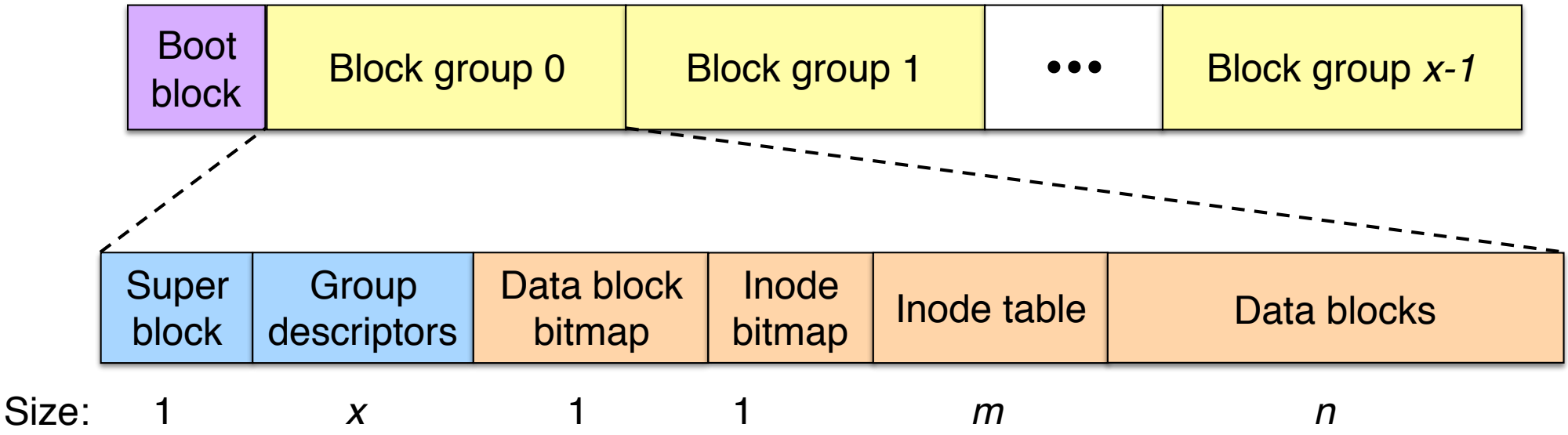
- Files on disk are stored where there's space
- On a full disk, files can be fragmented: stored non-contiguously
  - "Older" disks can have lots of fragmented files
- Solution: defragment the disk
  - Collect blocks of each file together and store them contiguously
  - Some file systems do this automatically



Ext2 & Ext3: the Linux file system

- Ext2/3 were the standard Linux file system
  - Similar to BSD's Fast File System (FFS)
- Ext3 is like Ext2, but with journaling
  - More on that in a bit…
- Ext2/3 is designed to be
  - Flexible across a wide range of workloads
  - (Relatively) simple to implement
  - Reliable
  - Usable across a wide range of file system sizes from hundreds of megabytes to terabytes
- Basic design hasn't changed much in 20 years!
  - Ext4 is now in use...

Large-scale structure in Ext3



- Disk broken into *x* block groups
- Each block group is a (mostly) self-contained unit
  - Files can span block groups, but doesn't happen often
  - Most information is local to the block group
- Bitmaps must fit into a single file system block, limiting the number of data blocks
- Copies of the superblock and group descriptors are stored in ***every*** block group for reliability

Baskin Engineering UC SANTA CRUZ

Superblock

- Information about the file system as a whole
  - File block size & number of blocks
  - Fragment size & number of fragments
  - Number of blocks / fragments / inodes per group
- Updated when the file system is modified
  - Free block & free inode counters
  - Time of last mount and write
- Superblock cached in memory, updated on disk frequently
- Superblock copy kept in each block group
  - Guards against loss of the superblock to disk corruption
  - Makes it more convenient to write out a copy in the nearest location

Group descriptor

- One descriptor for each group
- Block numbers for
  - Block bitmap
  - Inode bitmap
  - First inode table block
- Counts (limited to 16 bit integers!)
  - Free blocks in the group
  - Free inodes in the group
  - Directories in the group
- Each block group has a copy of all group descriptors

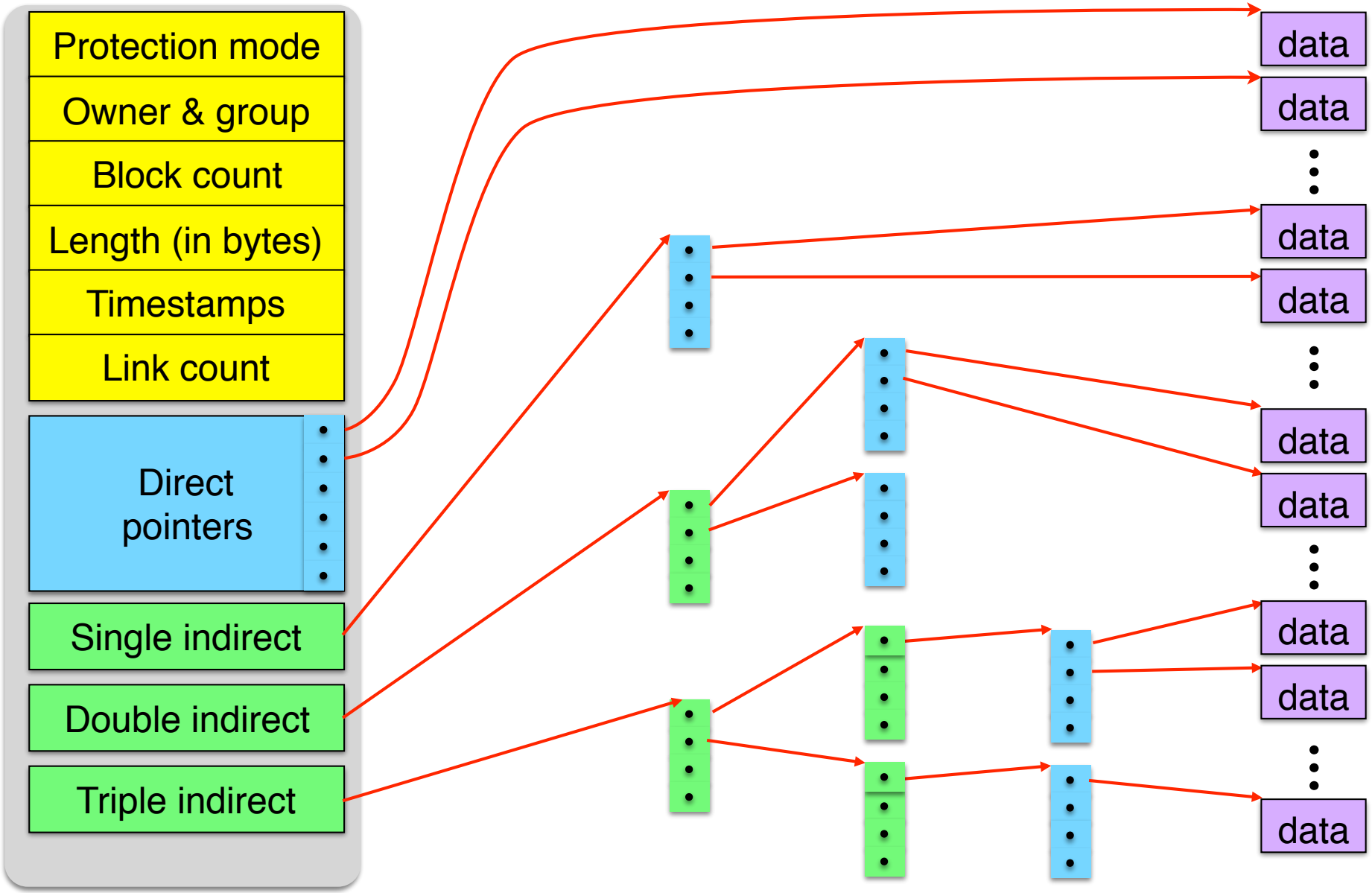
Baskin Engineering UC SANTA CRUZ

More on block groups

- Block group limited to maximum of *block_size* × 8 blocks (so block bitmap fits in a single block)
  - Maximum size (8 KB blocks) is 512 MB
  - May have any number of block groups per file system
- Maximum of 64K inodes per block group
- Size of inode region (in blocks) calculated by *num_inodes* / (*block_size*/128)
  - This determines the number of data blocks as well
- All block groups must be the same size

Baskin Engineering UC SANTA CRUZ

Linux inode: file indexing



Inode contents

- Inode contains a file's metadata
  - Ownership & permissions
  - Time stamps
  - Block pointers
- Inodes stored in the inode region in the block group
  - Inode number maps to a particular group / offset for the inode
- Soft link stored in inode if possible (using block pointers)
- File "hole": missing blocks in the middle of a large file

Field	Meaning
Mode	Type & access rights
Owner	File owner
Size	Length in bytes
Atime	Last access time
Ctime	Last time inode modified
Mtime	Last time file contents modified
Gid	Group identifier
Link count	Number of hard links
Blocks	Number of data blocks
Flags	File flags
Pointers	Pointers to data blocks

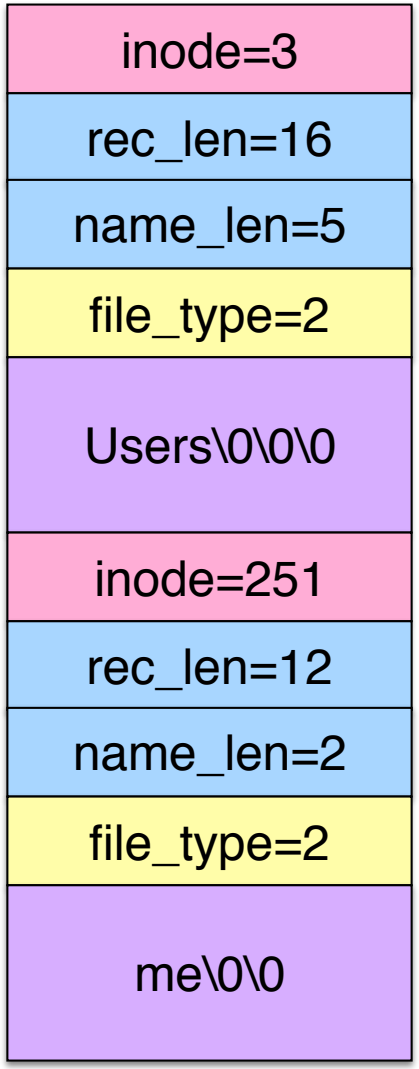
More on Unix FFS and Linux ext3

- First few block pointers kept in inode
  - Small files have no extra overhead for index blocks
  - Reading & writing small files is very fast!
- Indirect structures only allocated if needed
- For 4 KB file blocks (common in Unix), max file sizes are:
  - 48 KB in directory (usually 12 direct blocks)
  - 1024 × 4 KB = 4 MB of additional file data for single indirect
  - 1024 × 1024 × 4 KB = 4 GB of additional file data for double indirect
  - 1024 × 1024 × 1024 × 4 KB = 4 TB for triple indirect
- Maximum of 5 accesses for any file block on disk
  - 1 access to read inode & 1 to read file block
  - Maximum of 3 accesses to index blocks
  - Usually much fewer (1–2) because inode in memory

Baskin Engineering UC SANTA CRUZ

Linux directories

- Linux supports
  - Plain text directories
  - Hashed directories
  - Plain text more common, but slower
- Directories just like regular files
- Name length limited to 255 bytes
- Hashed directories
  - Use extendible hashing to make lookup faster
  - May be “sparse”



Major issue: consistency

- Creating a file can require 5 disk writes
  - A crash in the middle can leave the file system in an inconsistent state
  - Example: inode allocated, but not inserted into directory
- This is unacceptable!
- Solution: use journaling, as provided in Ext3

- Problem: what do you journal?
  - Metadata and data?
  - Metadata only?
    - This introduces an ordering constraint
    - Data must be written before metadata is written

Journaling in Ext3

- Ext3 uses a journal to record operations before they are written to disk
- Three journaling modes
  - Journal: record everything to the journal
    - Slowest
    - Records data and metadata as soon as they're written
  - Ordered: record only metadata, but order data and metadata writes so there's no inconsistency
    - Faster
    - Less safe: metadata updates not committed immediately
    - Default mode for Ext3
  - Metadata only: record only metadata
    - Not as safe as other two
    - Fastest of the three

Baskin Engineering UC SANTA CRUZ

Ext3 journal details

- Log record: basic entry in journal
  - One per disk request
  - Span of bytes, or entire block
- Atomic operation handle: used to group log records together
  - Typically, one handle corresponds to a single system call
  - Ensures that high-level operation either completes or never happens (no partial operations)
- Transaction: consists of multiple atomic operations
  - Done for efficiency reasons
  - Entire transaction written out at once
  - Only one transaction “open” at any time

FreeBSD vs. ext3

- FreeBSD inodes have very similar structures
  - Set of direct block pointers, and one each single, double, triple indirect pointers
  - Similar flags
  - No extents, but supports per-file specification of “block size”
    - Larger sizes require more blocks allocated consecutively
  - There are some small differences…
- FreeBSD allocates inodes dynamically
  - Each inode block holds 128 inodes
  - Two allocated initially
  - Additional inode blocks allocated as needed
    - Potential inode blocks held as free until all other space is used
- FreeBSD uses *soft updates* to ensure that updates are written in an order that preserves dependencies
  - Never point to a structure before it has been initialized
  - Never reuse a resource before nullifying all previous pointers to it
  - Never reset the old pointer to a live resource before the new pointer has been set
- FreeBSD supports journaling of metadata operations
- FreeBSD supports snapshots by creating a “snapshot file” to save information that may be overwritten by ongoing activity

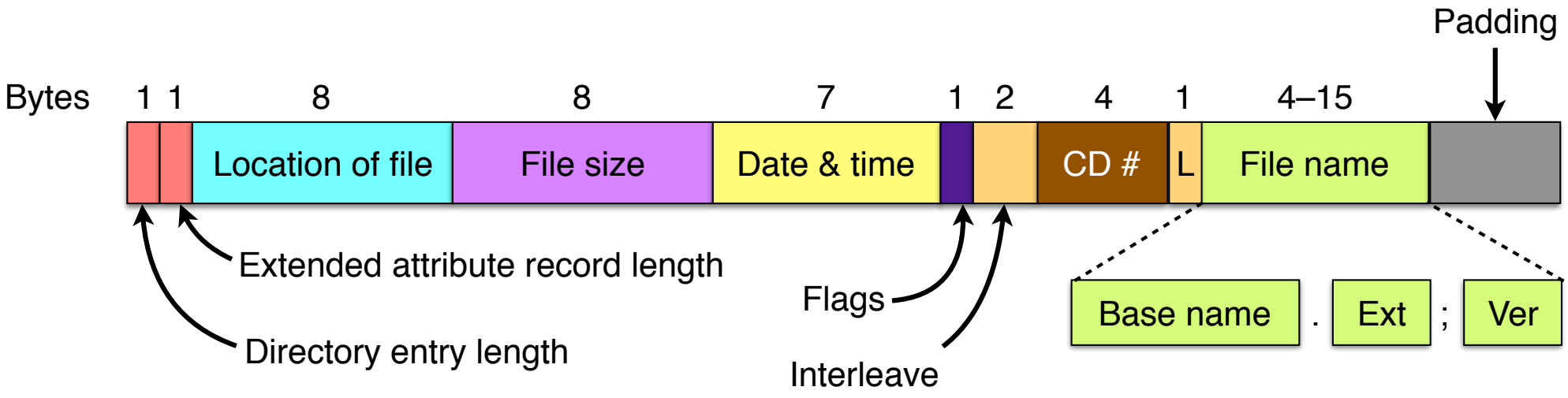
What's new in ext4?

- New features
  - Extents
    - Handled by a separate two-level extent tree: pointer to it is in the inode
    - Need to ensure backwards compatibility...
  - Option for larger inodes
  - More scalable
  - Faster extended attributes

- Ext4 somewhat compatible with ext2/3
  - Mounts old file systems
  - Mountable under ext2/3 if no new features (*i.e.*, extents) used

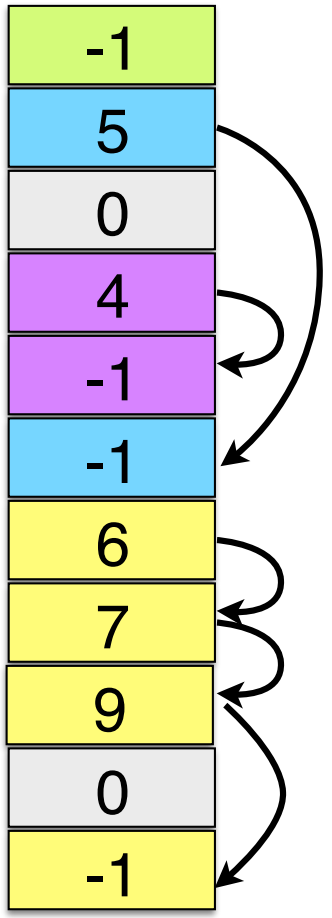
CD-ROM file system

- File catalog in first few blocks of disk
  - Lists all files on the CD
  - Hierarchical structure can be built by treating files as directories (ISO 9660)
- Files all allocated contiguously
  - No need to handle overwrites…
- Current standard (UDF) is a superset of this



MS-DOS (FAT) file system

- Originally designed for floppy disks
  - Originally designed for small (5 MB) hard drives
  - Commonly used for thumb drives and SD cards
- Single list of blocks in use (FAT ➡ File Allocation Table)
  - Fixed overhead per block
  - Easy to find all blocks in a file
- Root directory “file” starts in block 0
  - Other files have first block number in directory entry



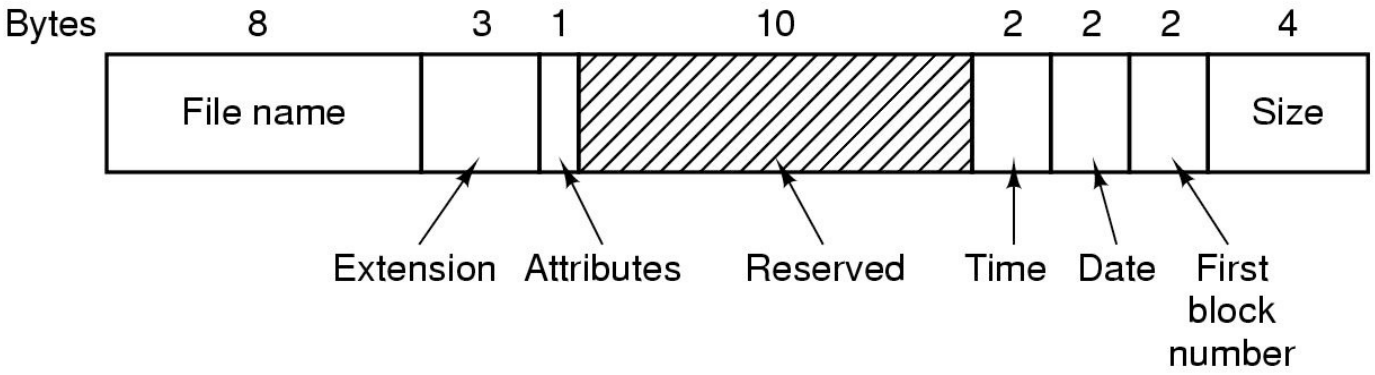
MS-DOS File Allocation Table sizes

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

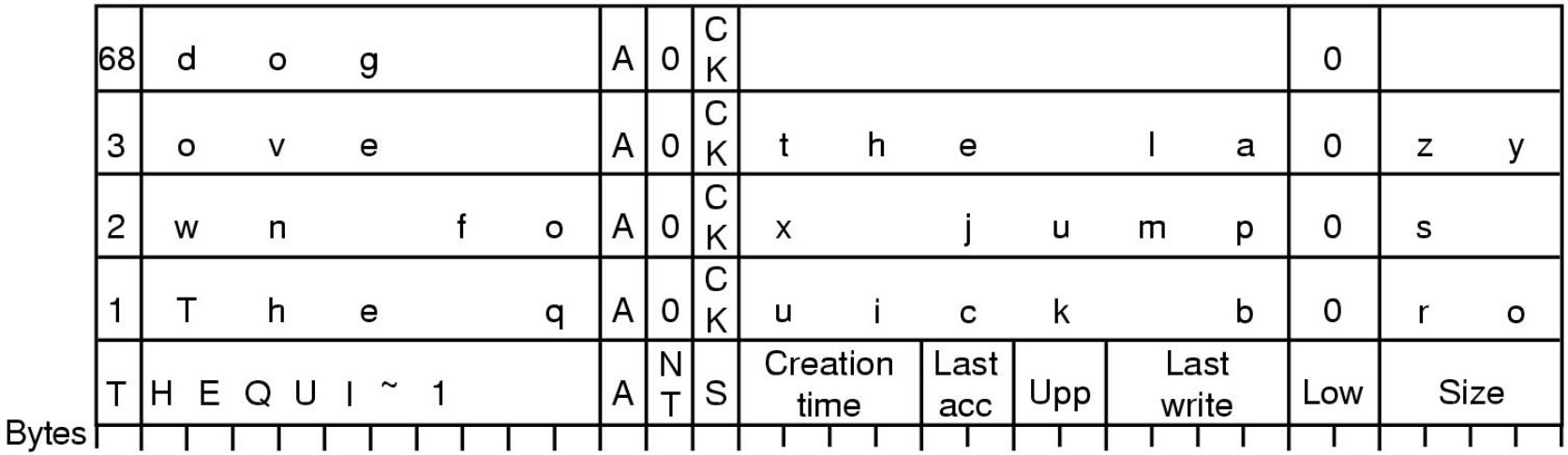
Baskin Engineering
UC SANTA CRUZ

Directory entry in FAT

FAT



Storing a long name in VFAT



- Long name stored in Windows 98 so that it's backwards compatible with short names
  - Short name in "real" directory entry
  - Long name in "fake" directory entries: ignored by older systems
- OS designers will go to great lengths to make new systems work with older systems…

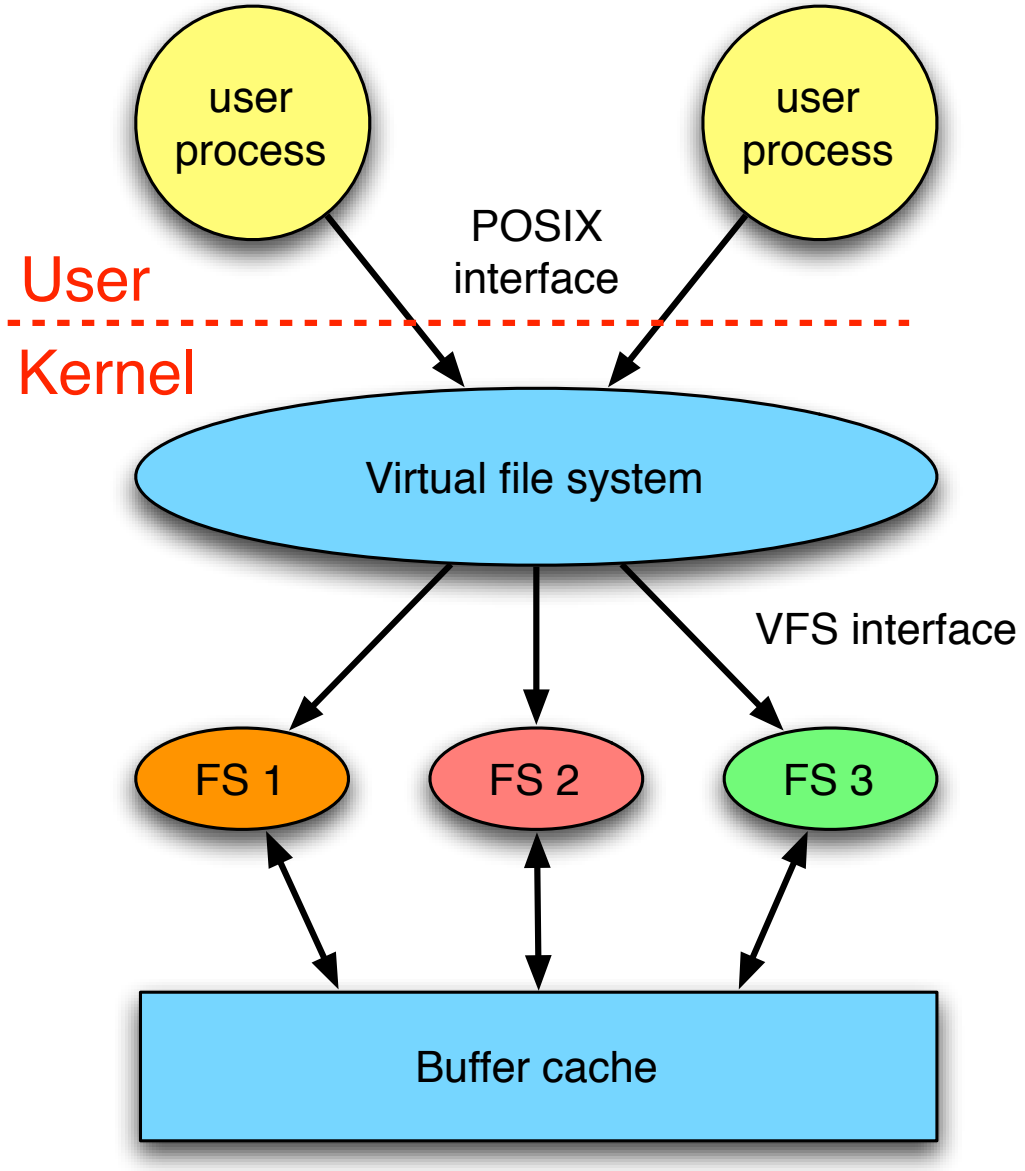
Baskin Engineering UC SANTA CRUZ

Zettabyte File System (ZFS)

- Designed to handle very large disk partitions
- Extensive error checking and recovery
  - Checksums on individual blocks
  - RAID at the file level (RAIDZ)
- Never overwrites data in place: no chance of inconsistent state
  - Write new blocks to disk into previously-free blocks
  - Write a new *uberblock* to reference the new file system state
    - Transition from one consistent state to another is done with a single block write!
- Some properties similar to log-structured file systems
  - No overwrite-in-place
  - Needs cleaning!
  - But can write anywhere on disk: no need for segments
- Very complex—read Chapter 10 in FreeBSD book for details

Virtual file system switch

- How can the OS use multiple file systems?
  - HFS+ / ext3 / NTFS
  - VFAT
  - ISO 9660 / UDF
- Much code is common between them
  - Directory lookups
  - Caching
- Solution: virtual file system (VFS) switch
  - Abstracts out common functionality
  - Provides default routines for many common functions



Baskin Engineering UC SANTA CRUZ

Layering file systems

- Often useful to layer one file system on top of another one
  - Use the underlying file system for many operations
  - Add (or enhance) functionality in the layered file system
- Example: stacking file systems in FreeBSD
  - nullfs: stacked file system does nothing
  - FUSE: file system intercepts calls and lets a user-space program handle them
    - This makes it easier to implement “interesting” functionality with a file system interface
    - Assignment 4 is going to involve using FUSE to implement a different way of naming files…

Baskin Engineering UC SANTA CRUZ