




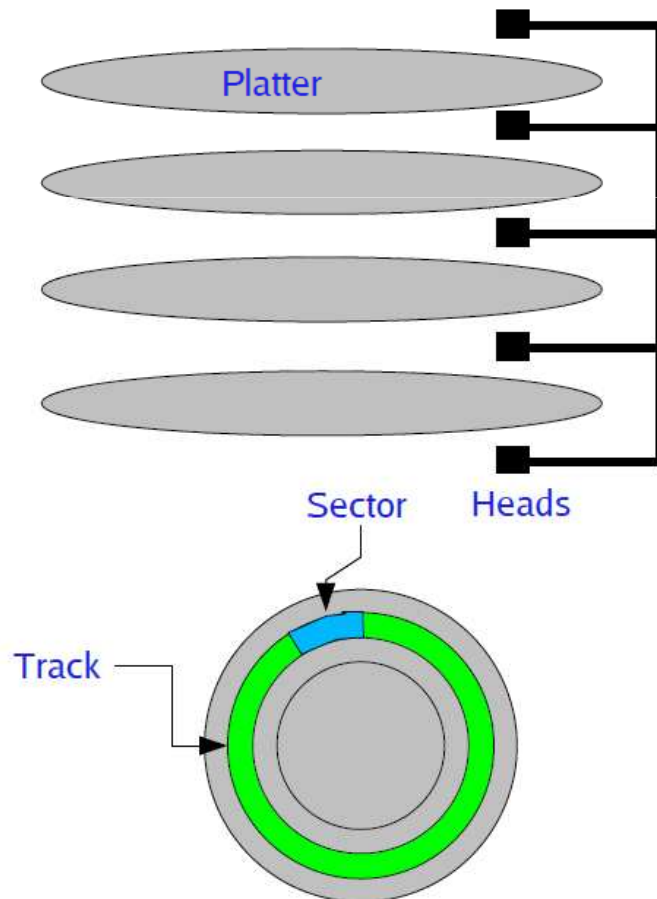
I/O and Filesystems



Part 1: Disks

A Disk Primer

- Disks consist of one or more **platters** divided into **tracks**
 - Each platter may have one or two **heads** that perform read/write operations
 - Each track consists of multiple **sectors**
 - The set of sectors across all platters is a **cylinder**

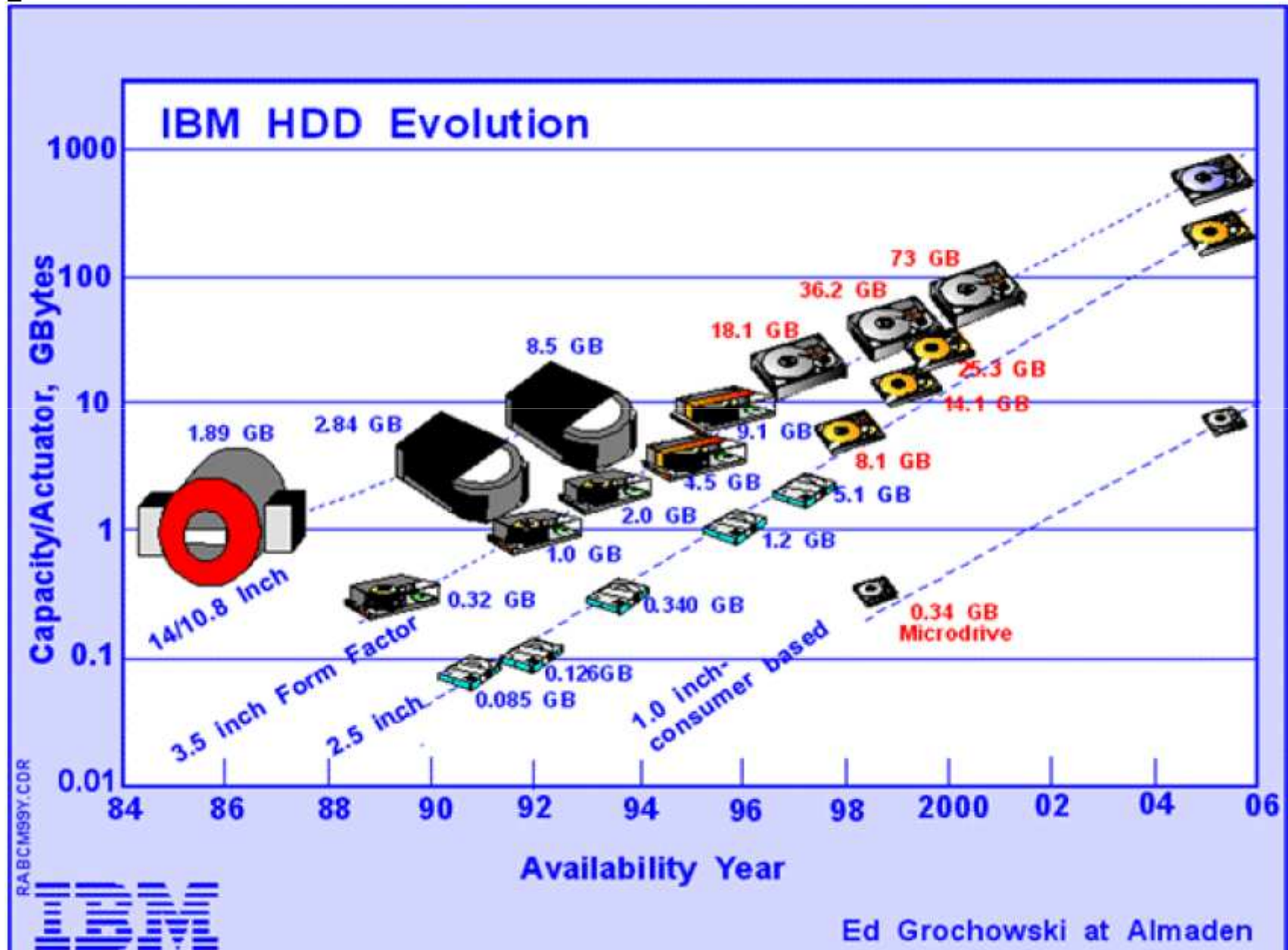


[Hard Disk Evolution]

- IBM 305 RAMAC (1956)
 - First commercially produced hard drive
 - 5 Mbyte capacity, 50 platters each 24" in diameter



Hard Drive Evolution



[Disk access time]

- **Command overhead:**
 - Time to issue I/O, get the HDD to start responding, select appropriate head
- **Seek time:**
 - Time to move disk arm to the appropriate track
 - Depends on how fast you can physically move the disk arm
 - These times are not improving rapidly!
- **Settle time:**
 - Time for head position to stabilize on the selected track
- **Rotational latency:**
 - Time for the appropriate sector to move under the disk arm
 - Depends on the rotation speed of the disk (e.g., 7200 RPM)
- **Transfer time**
 - Time to transfer a sector to/from the disk controller
 - Depends on density of bits on disk and RPM of disk rotation



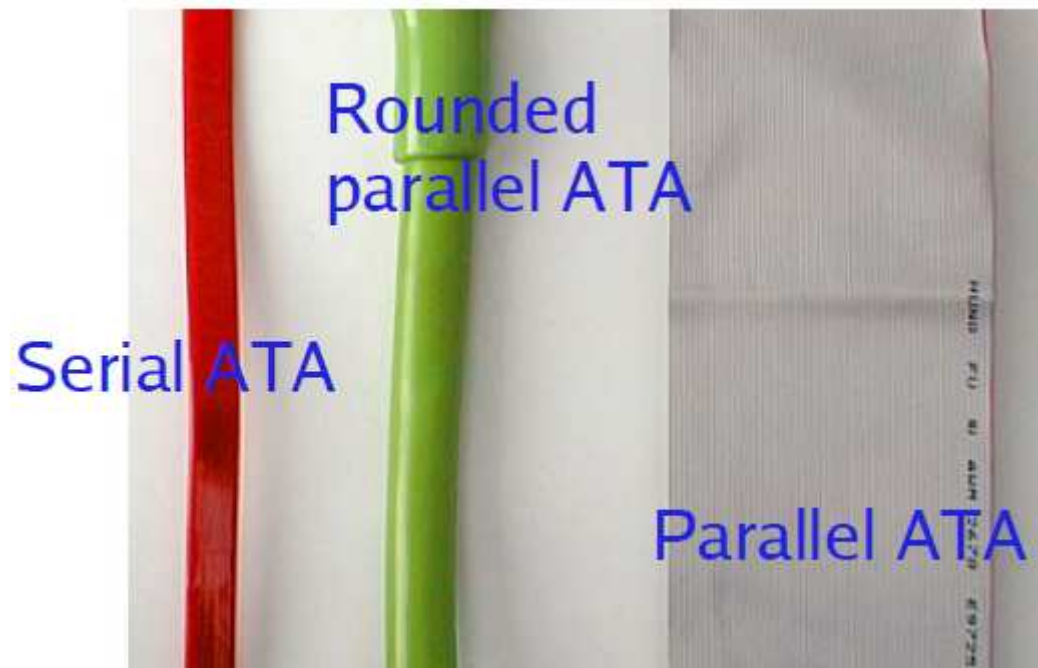
[Disks are messy and slow]

- Low-level interface for reading and writing sectors
 - Generally allow OS to read/write an entire sector at a time
 - No notion of “files” or “directories” -- just raw sectors
 - So, what do you do if you need to write a single byte to a file?
 - Disk may have numerous bad blocks – OS may need to mask this from filesystem
- Access times are still very slow
 - Disk seek times are around 10 ms
 - Although raw throughput has increased dramatically
 - Compare to several nanosec to access main memory
 - Requires careful scheduling of I/O requests



[ATA Interfaces]

- Serial ATA (SATA): Today's standard for connecting hard drives to the motherboard
 - Using a serial (not parallel) interface
 - Earlier versions used a parallel interface (PATA)
 - Speeds starting at 1.5 Gbit/sec (SATA 1.0)
 - SATA 2.0 (3.0 Gbit/sec), SATA 3.0 (6.0 Gbit/sec)
- Can drive longer cables at much higher clock speeds than parallel cable



[Disk I/O Scheduling]

- Given multiple outstanding I/O requests, what order to issue them?
- Why does it matter?
- Major goals of disk scheduling:
 - 1) Minimize **latency** for small transfers
 - Primarily: Avoid long seeks by ordering accesses according to disk head locality
 - 2) Maximize **throughput** for large transfers
 - Large databases and scientific workloads often involve enormous files and datasets
- Note that disk block layout also has a large impact on performance
 - Where we place file blocks, directories, file system metadata, etc.
 - This will be covered in future lectures

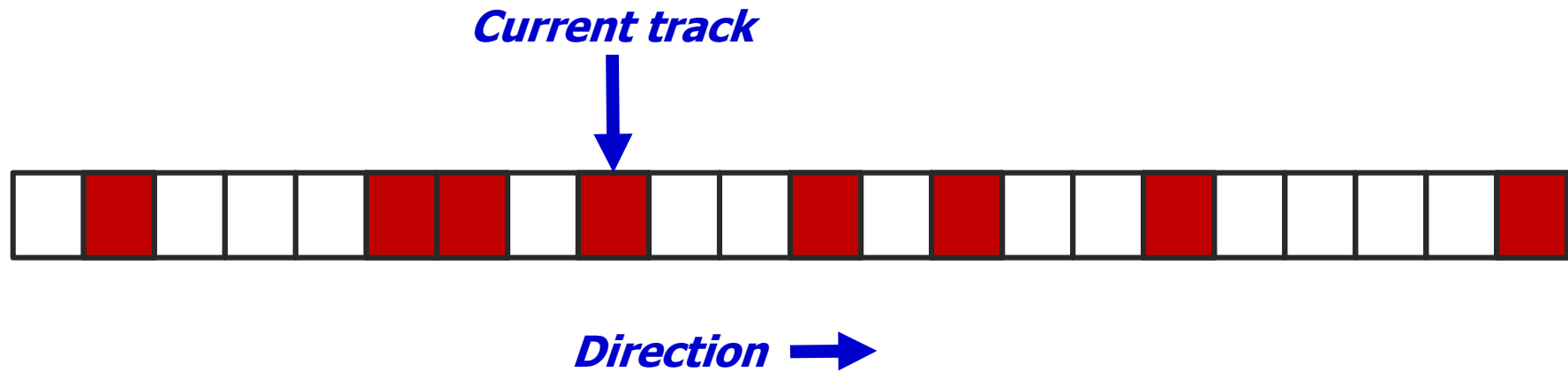


[Disk I/O Scheduling]

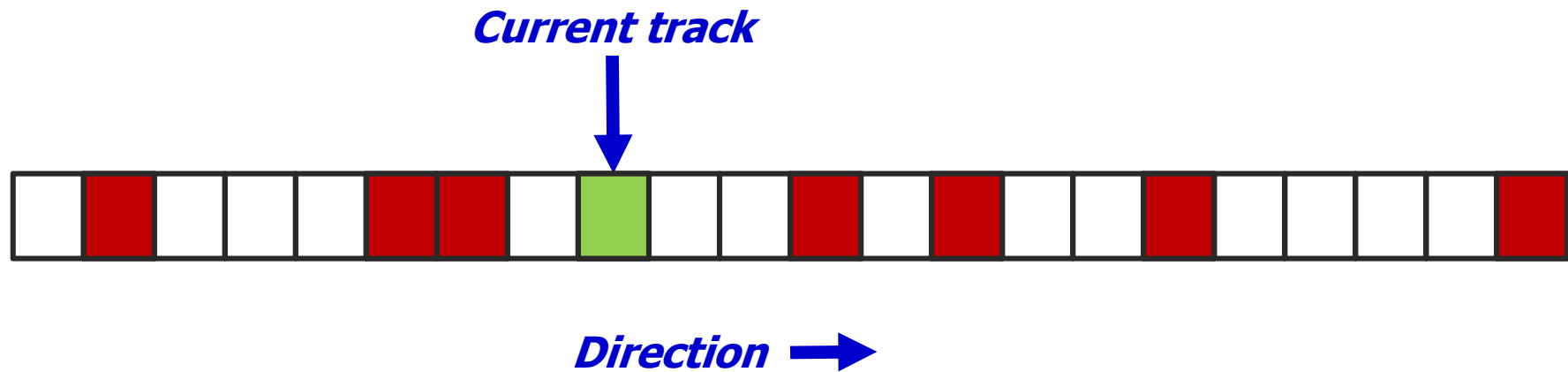
- Given multiple outstanding I/O requests, what order to issue them?
- **FIFO**: Just schedule each I/O in the order it arrives
 - What's wrong with this? **Potentially lots of seek time!**
- **SSTF**: Shortest seek time first
 - Issue I/O with the nearest cylinder to the current one
 - **Favors middle tracks: Head rarely moves to edges of disk**
- **SCAN** (or **Elevator**) Algorithm:
 - Head has a current direction and current cylinder
 - Sort I/Os according to the track # in the current direction of the head
 - If no more I/Os in the current direction, reverse direction
- **CSCAN** Algorithm:
 - Always move in one direction, “wrap around” to beginning of disk when moving off the end
 - Idea: Reduce variance in seek times, avoid discriminating against the highest and lowest tracks



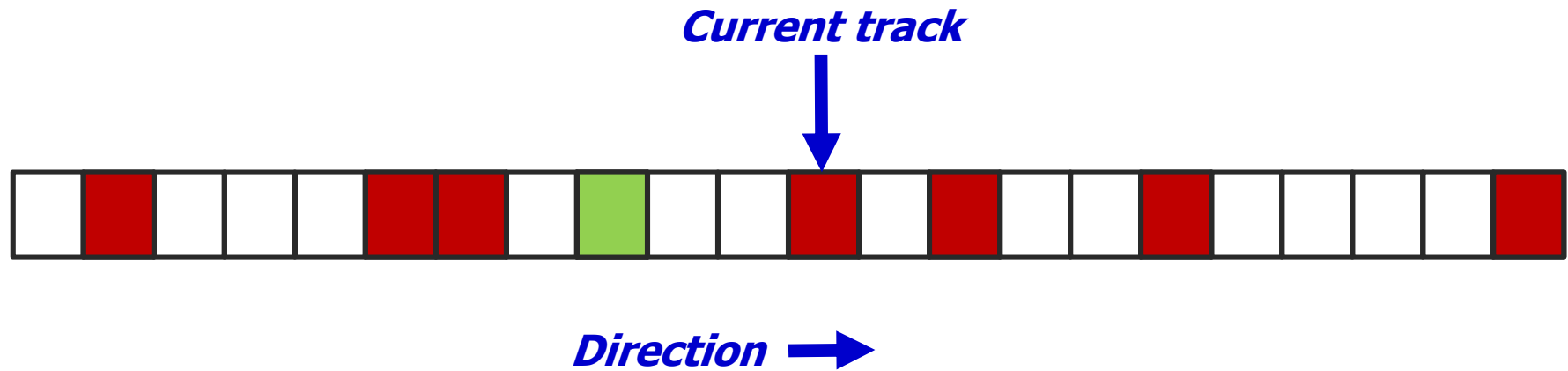
[SCAN example]



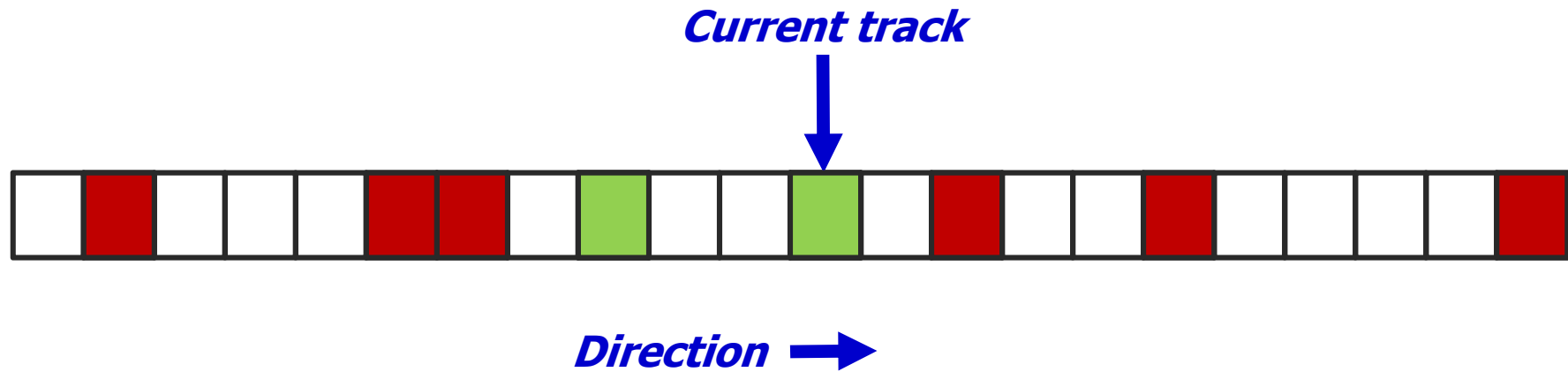
[SCAN example]



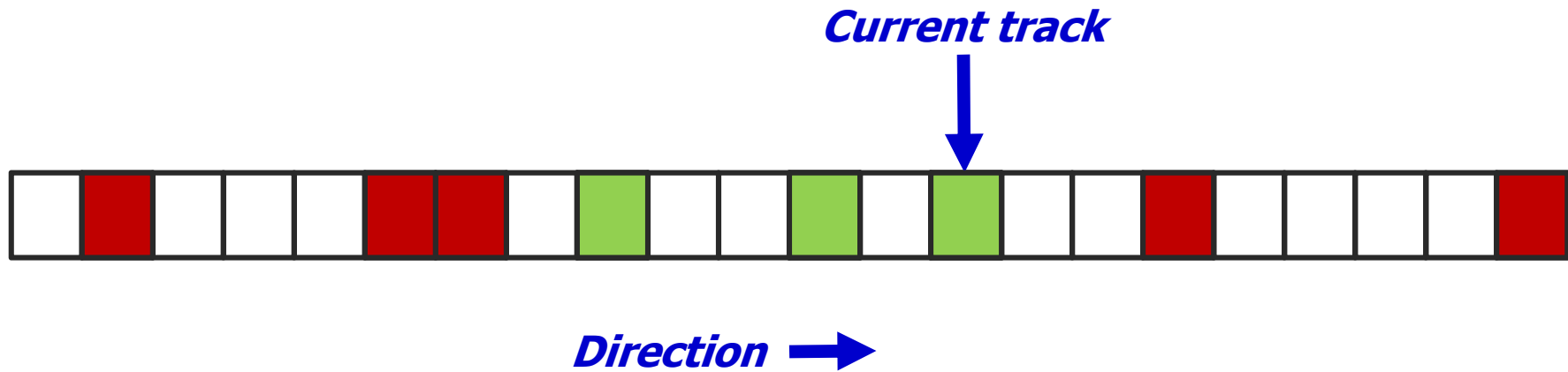
[SCAN example]



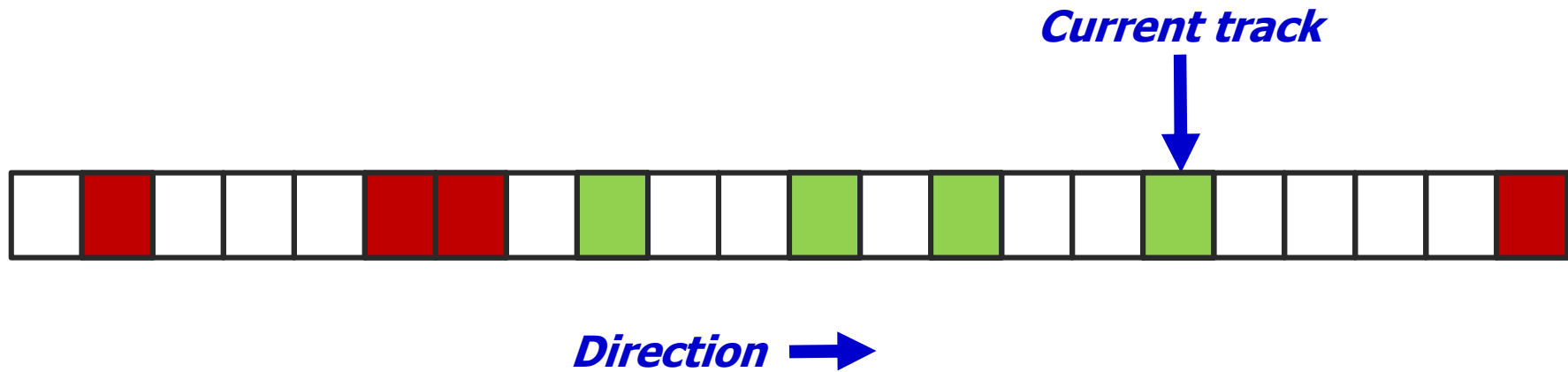
[SCAN example]



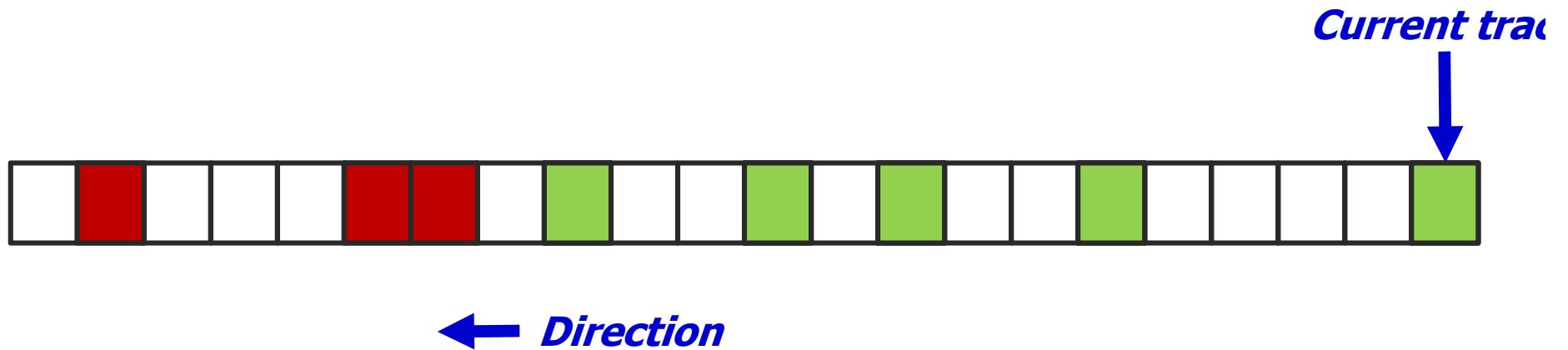
[SCAN example]



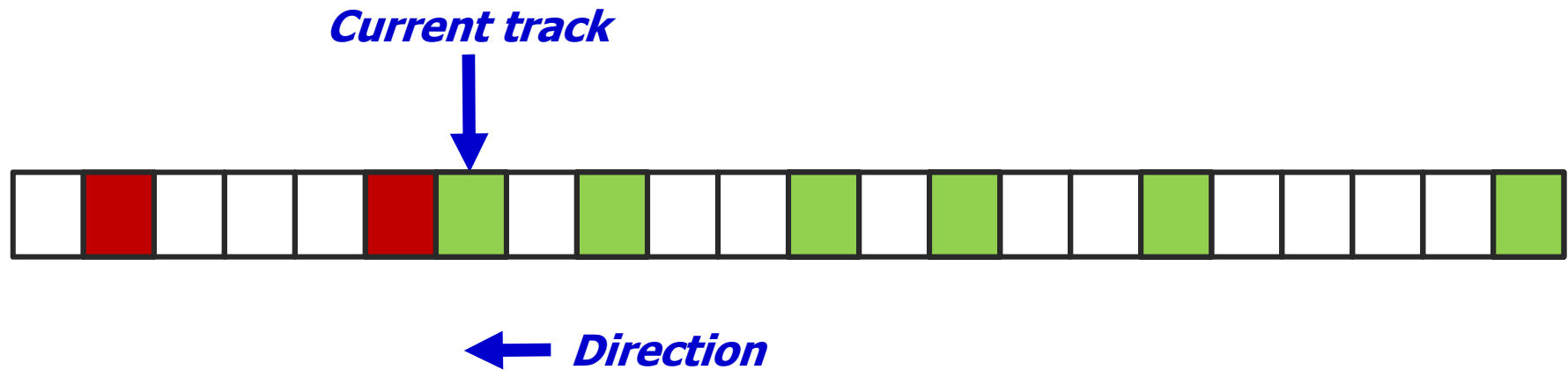
[SCAN example]



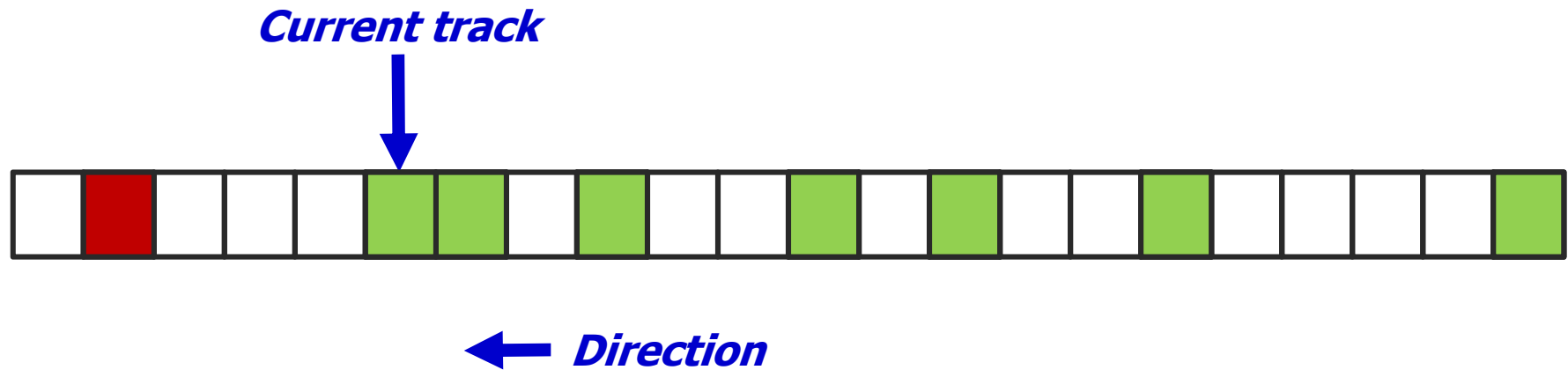
[SCAN example]



[SCAN example]



[SCAN example]



[SCAN example]

Current track

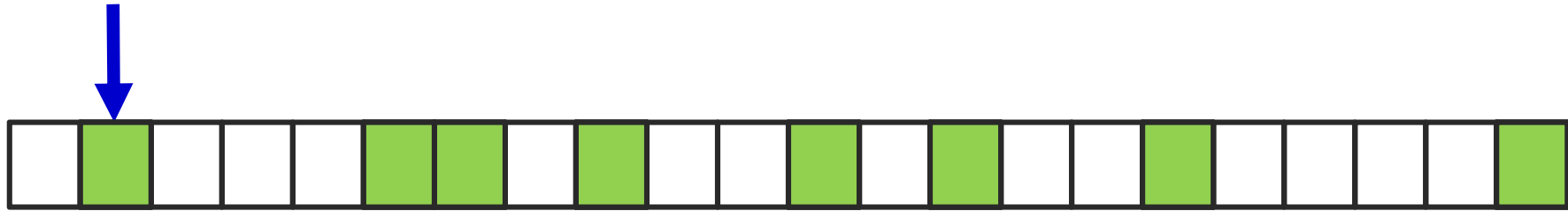


Direction



[SCAN example]

Current track



← Direction


- What is the overhead of the SCAN algorithm?
 - Count the total amount of seek time to service all I/O requests
 - I.e., count total number of track changes
 - In this case, 12 tracks in --> direction
 - 15 tracks for long seek back
 - 5 tracks in <-- direction
 - Total: $12+15+5 = 32$ tracks



[What about flash?

- Non-volatile, solid state storage
 - No moving parts!
 - Fast access times (about 0.1 msec)
 - Can read and write individual bytes at a time
- Limitations
 - Block erasure: However, must erase a whole “block” before writing to it
 - Read disturb: Reads can cause cells near the read cell to change
 - Solution: Periodically re-write blocks
 - Limited number of erase/write cycles
 - Most flash on the market today can withstand up to 1 million erase/write cycles
 - Flash Translation Layer (FTL): writes to a different cell each time to wear-level device, cache to avoid excessive writes
- How does this affect how we design filesystems???





Part 2: I/O

[Input and Output]

- A computer's job is to process data
 - Computation (CPU, cache, and memory)
 - Move data into and out of a system (between I/O devices and memory)
- Challenges with I/O devices
 - Different categories: storage, networking, displays, etc.
 - Large number of device drivers to support
 - Device drivers run in kernel mode and can crash systems
- Goals of the OS
 - Provide a generic, consistent, convenient and reliable way to
 - access I/O devices
 - As device-independent as possible
 - Don't hurt the performance capability of the I/O system too much



[How does the CPU talk to devices?]

- **Devices controller:** Circuit that enables devices to talk to the peripheral bus
- **Host adapter:** Circuit that enables the computer to talk to the peripheral bus
- **Bus:** Wires that transfer data between components inside computer
- Device controller allows OS to specify simpler instructions to access data
- Example: a disk controller
 - Translates “access sector 23” to “move head reader 1.672725272 cm from edge of platter”
 - Disk controller “advertises” disk parameters to OS, hides internal disk geometry
 - Most modern hard drives have disk controller embedded as a chip on the physical device



Review: Computer Architecture

- Compute hardware

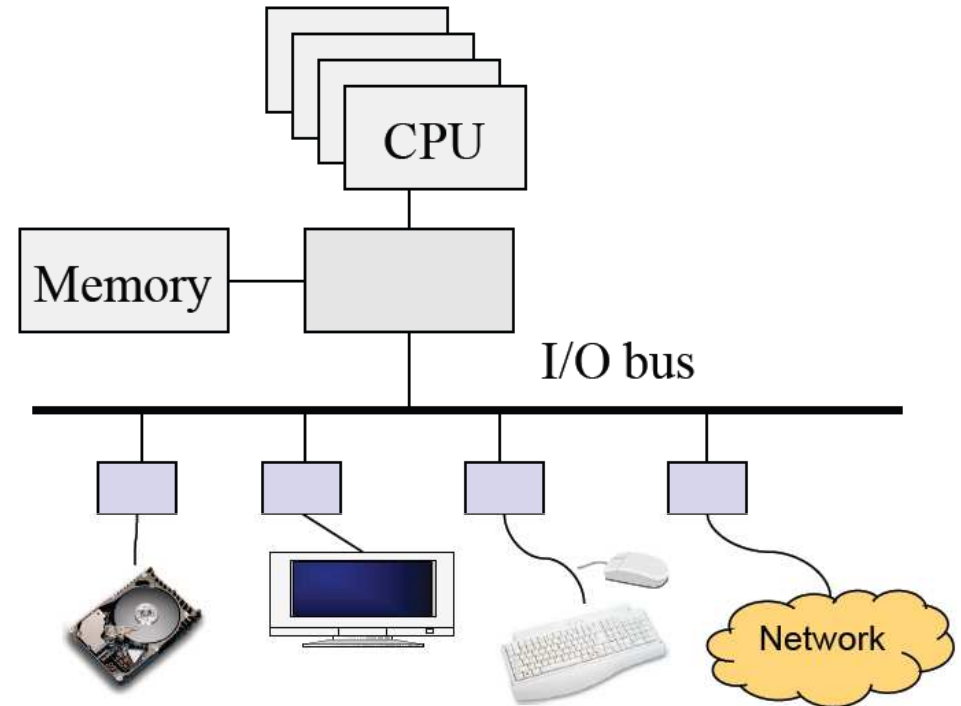
- CPU and caches
- Chipset
- Memory

- I/O Hardware

- I/O bus or interconnect
- I/O controller or adaptor
- I/O device

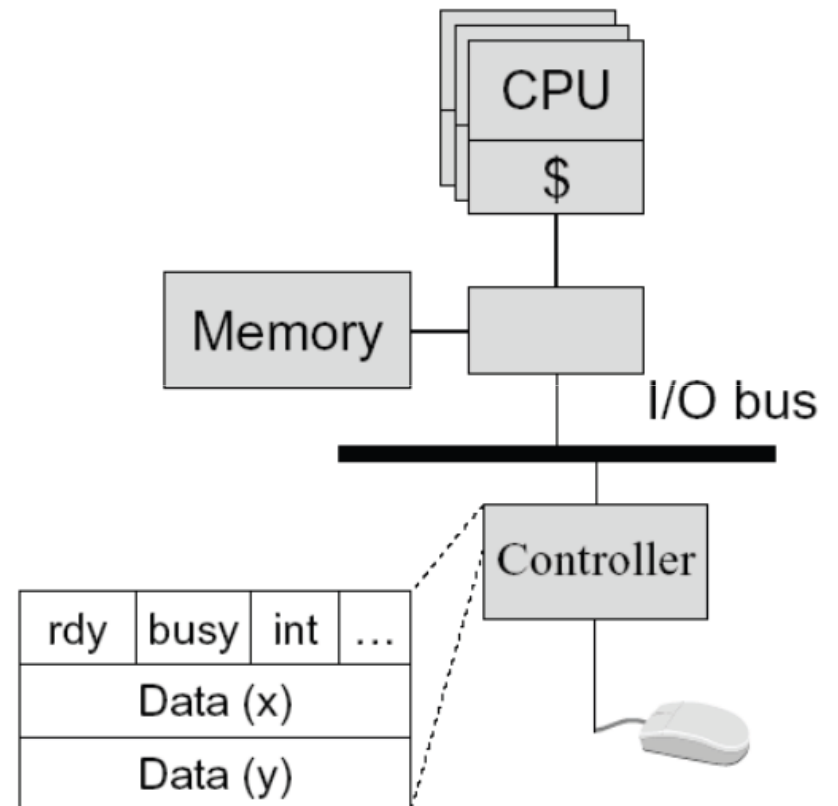
- Two types of I/O

- **Programmed I/O (PIO)**
 - CPU does the work of moving data
- **Direct Memory Access (DMA)**
 - CPU offloads the work of moving data to DMA controller



[Programmed Input Device]

- Device controller
 - Status register
 - ready: tells if the host is done
 - busy: tells if the controller is done
 - int: interrupt
 - ...
 - Data registers
- A simple mouse design
 - Put (X, Y) in mouse's device controller's data registers on a move
 - Interrupt
- Input on an interrupt
 - CPU saves state of currently-executing program
 - Reads values in X, Y registers
 - Sets ready bit
 - Wakes up a process/thread or execute a piece of code to handle interrupt



[Programmed Output Device]

■ Device

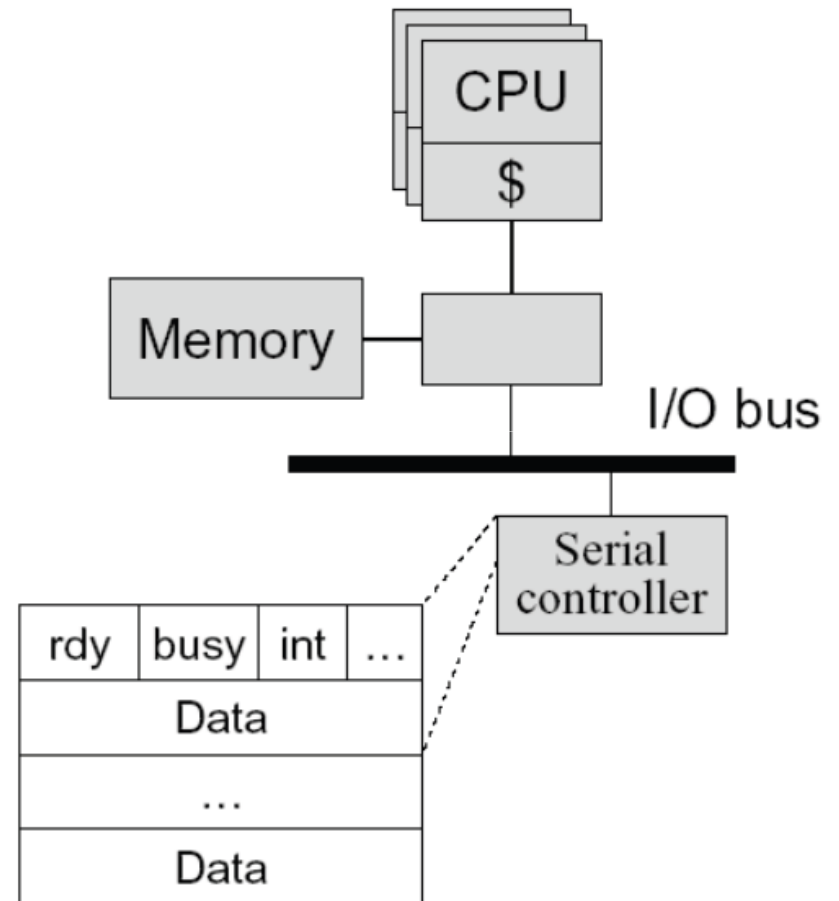
- Status registers (ready, busy, ...)
- Data registers

■ Example

- A serial output device

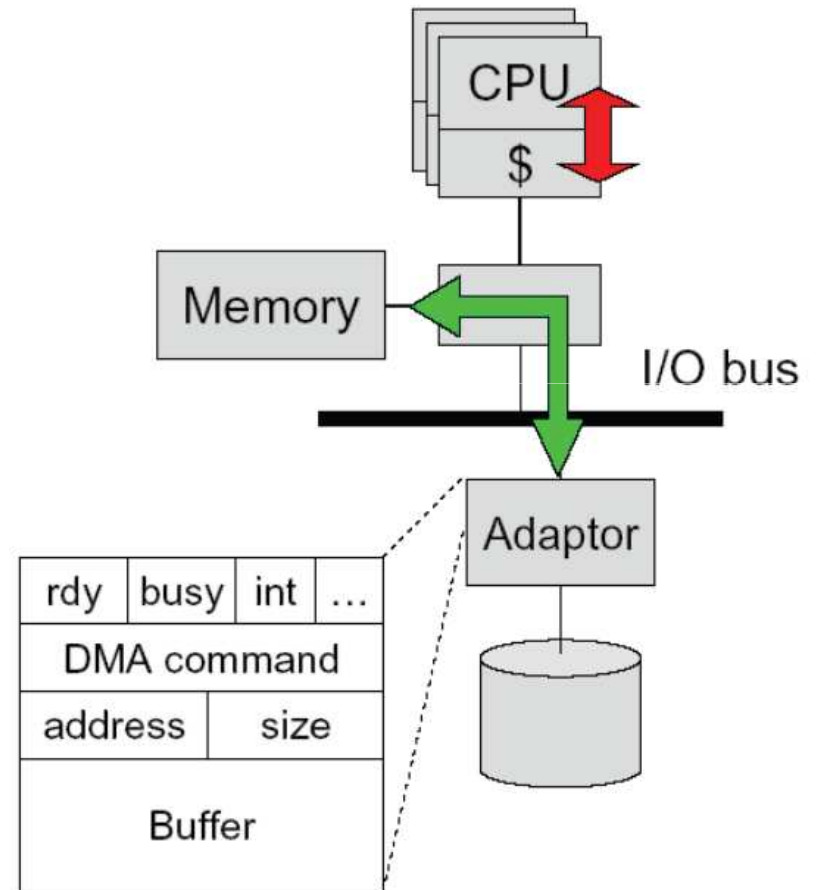
■ Perform an output

- CPU: Poll the busy bit
- Writes the data to data register(s)
- Set ready bit
- Controller sets busy bit and transfers data
- Controller clears the busy bit



Direct Memory Access (DMA)

- DMA controller or adaptor
 - Status register (ready, busy, interrupt, ...)
 - DMA command register
 - DMA register (address, size)
 - DMA buffer
- Host CPU initiates DMA
 - Device driver call (kernel mode)
 - Wait until DMA device is free
 - Initiate a DMA transaction
 - (command, memory address, size)
 - Block
- Controller performs DMA
 - DMA data to device (size--; address++)
 - Interrupt on completion (size == 0)
- Interrupt handler (on completion)
 - Wakeup the blocked process



[Memory-mapped I/O]

- Use the same address bus to address both memory and I/O devices
 - The memory and registers of I/O devices are mapped to address values
 - Allows same CPU instructions to be used with regular memory and devices
- I/O devices, memory controller, monitor address bus
 - Each responds to addresses they own
- Orthogonal to DMA
 - May be used with, or without, DMA



[Polling- vs. Interrupt-driven I/O]

- Polling
 - CPU issues I/O command
 - CPU directly writes instructions into device's registers
 - CPU busy waits for completion
- Interrupt-driven I/O
 - CPU issues I/O command
 - CPU directly writes instructions into device's registers
 - CPU continues operation until interrupt
- Direct Memory Access (DMA)
 - Typically done with Interrupt-driven I/O
 - CPU asks DMA controller to perform device-to-memory transfer
 - DMA issues I/O command and transfers new item into memory
 - CPU module is interrupted after completion
- Which is better, polling or interrupt-driven I/O?



[Polling- vs. Interrupt-driven I/O]

- Polling
 - Expensive for large transfers
 - Better for small, dedicated systems with infrequent I/O

- Interrupt-driven
 - Overcomes CPU busy waiting
 - I/O module interrupts when ready: event driven

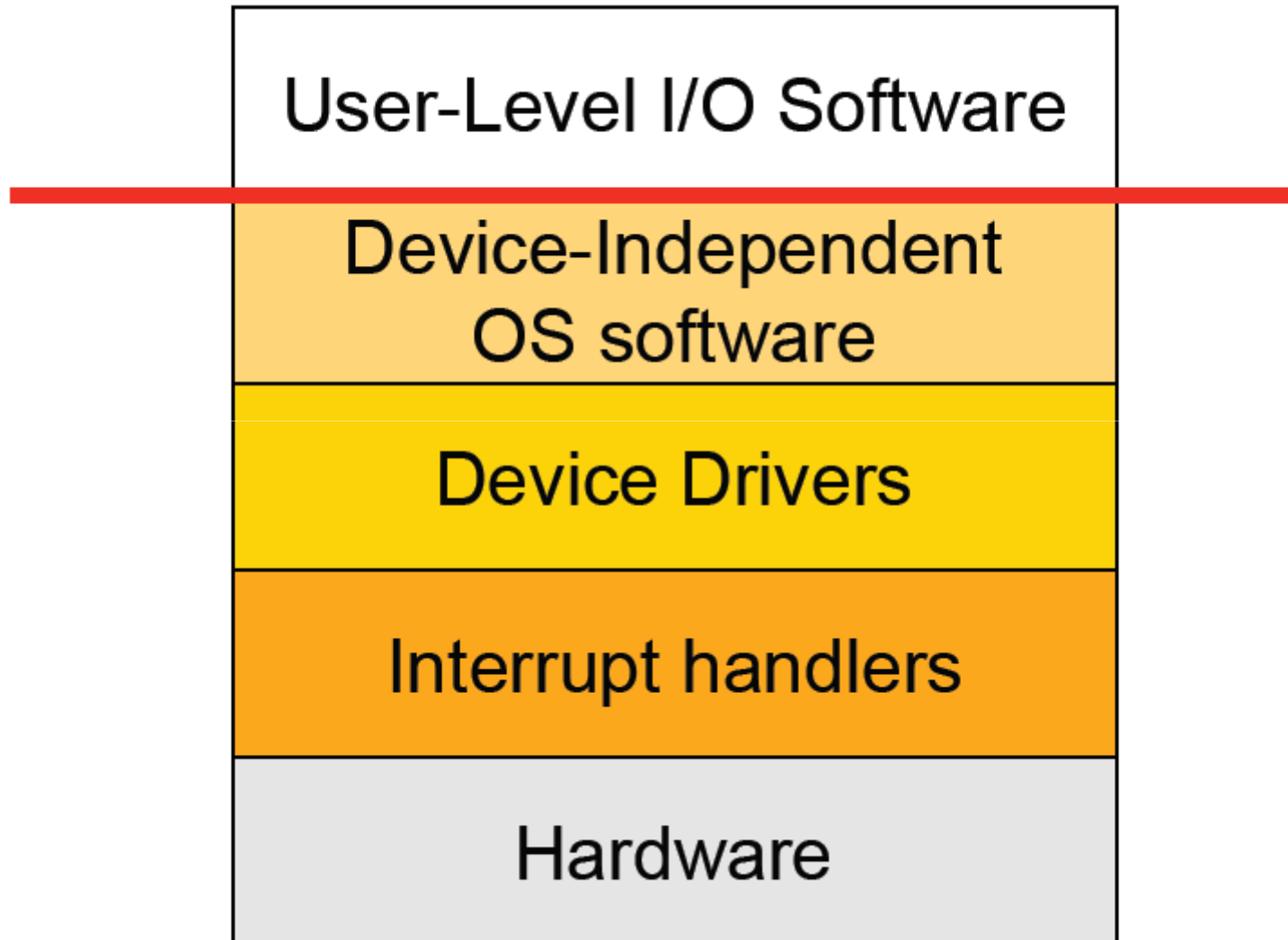


[How Interrupts are implemented]

- CPU hardware has an interrupt report line that the CPU tests after executing every instruction
 - If a(ny) device raises an interrupt by setting interrupt report line
 - CPU catches the interrupt and saves the state of current running process into PCB
 - CPU dispatches/starts the interrupt handler
 - Interrupt handler determines cause, services the device and clears the interrupt report line
- Other uses of interrupts: exceptions
 - Division by zero, wrong address
 - System calls (software interrupts/signals, trap)
 - Virtual memory paging



[I/O Software Stack]

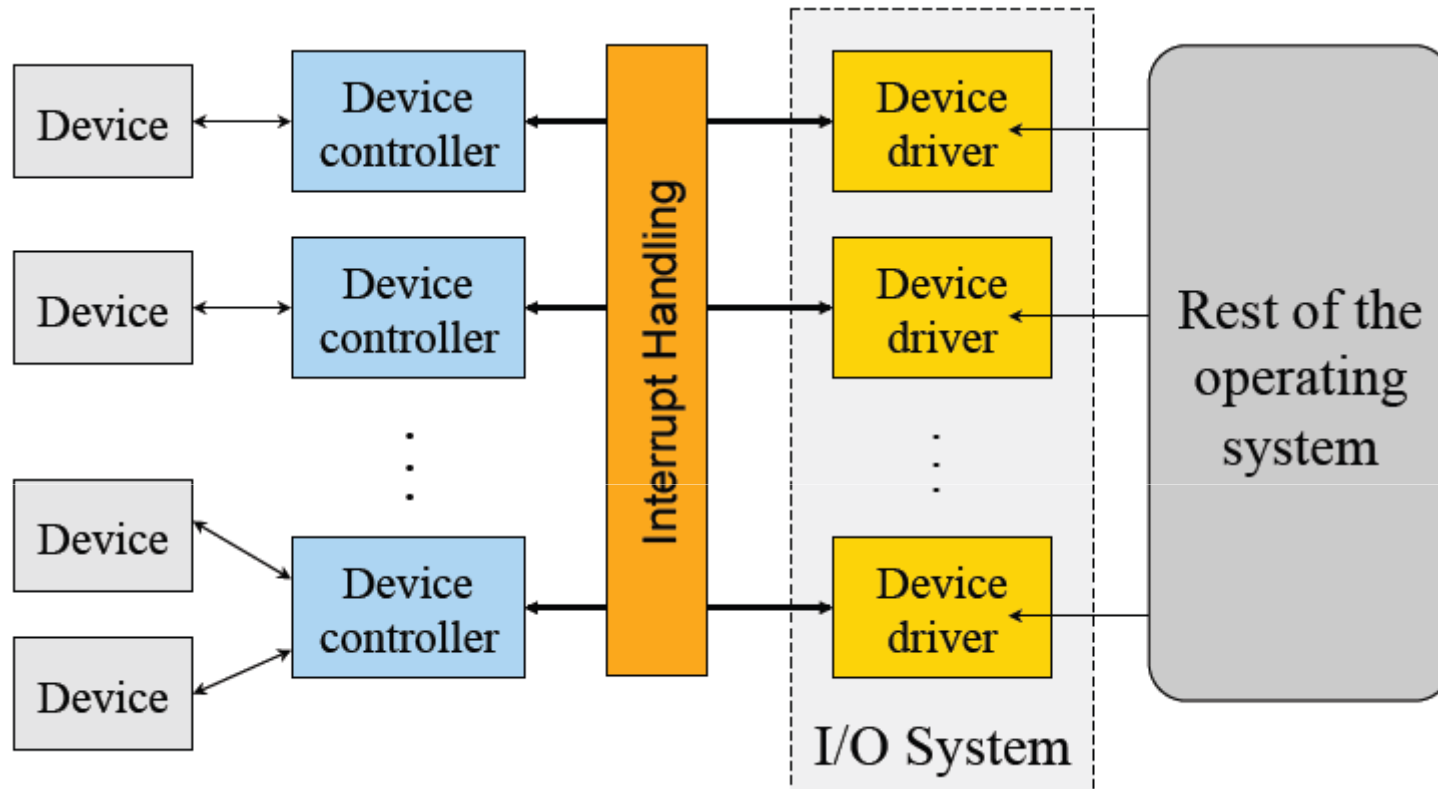


[Interrupt Handling]

- Save context (registers that hw hasn't saved, PSW etc)
- Mask interrupts if needed
- Set up a context for interrupt service
- Set up a stack for interrupt service
- Acknowledge interrupt controller, perhaps enable it
- Save entire context to PCB
- **Run the interrupt service**
- Unmask interrupts if needed
- Possibly change the priority of the process
- Run the scheduler
- Then OS will set up context for next process, load registers and PSW, start running process ...



[Device Drivers]



- Manage the complexity and differences among specific types of devices (disk vs. mouse, different types of disks ...)
- Each handles one type of device or small class of them (eg SCSI)

[Typical Device Driver Design]

- Operating system and driver communication
 - Commands and data between OS and device drivers
- Driver and hardware communication
 - Commands and data between driver and hardware
- Driver responsibilities
 - Initialize devices
 - Interpreting commands from OS
 - Schedule multiple outstanding requests
 - Manage data transfers
 - Accept and process interrupts
 - Maintain the integrity of driver and kernel data structures



[Device Driver Behavior]

- Check input parameters for validity, and translate them to device specific language
- Check if device is free (wait or block if not)
- Issue commands to control device
 - Write them into device controller's registers
 - Check after each if device is ready for next (wait or block if not)
- Block or wait for controller to finish work
- Check for errors, and pass data to device-independent software
- Return status information
- Process next queued request, or block waiting for next
- Challenges:
 - Must be reentrant (can be called by an interrupt while running)
 - Handle hot-pluggable devices and device removal while running
 - Complex and many of them; bugs in them can crash system



[Types of I/O Devices]

- Block devices
 - Organize data in fixed-size blocks
 - Transfers are in units of blocks
 - Blocks have addresses and data are therefore addressable
 - E.g. hard disks, USB disks, CD-ROMs
- Character devices
 - Delivers or accepts a stream of characters, no block structure
 - Not addressable, no seeks
 - Can read from stream or write to stream
 - Printers, network interfaces, terminals
- Like everything, not a perfect classification
 - E.g. tape drives have blocks but not randomly accessed
 - Clocks are I/O devices that just generate interrupts



[Char/Block Device Interfaces]

■ Character device interface

- read(deviceNumber, bufferAddr, size)
 - Reads “size” bytes from a byte stream device to “bufferAddr”
- write(deviceNumber, bufferAddr, size)
 - Write “size” bytes from “bufferAddr” to a byte stream device

■ Block device interface

- read(deviceNumber, deviceAddr, bufferAddr)
 - Transfer a block of data from “deviceAddr” to “bufferAddr”
- write(deviceNumber, deviceAddr, bufferAddr)
 - Transfer a block of data from “bufferAddr” to “deviceAddr”
- seek(deviceNumber, deviceAddress)
 - Move the head to the correct position
 - Usually not necessary



[Sync vs Asynchronous I/O]

- Synchronous I/O
 - read() or write() will block a user process until its completion
 - OS overlaps synchronous I/O with another process
- Asynchronous I/O
 - read() or write() will not block a user process
 - user process can do other things before I/O completion
 - I/O completion will notify the user process



[Example: Blocked Read]

- A process issues a read call which executes a system call
- System call code checks for correctness
- If it needs to perform I/O, it will issues a device driver call
- Device driver allocates a buffer for read and schedules I/O
- Controller performs DMA data transfer
- Block the current process and schedule a ready process
- Device generates an interrupt on completion
- Interrupt handler stores any data and notifies completion
- Move data from kernel buffer to user buffer
- Wakeup blocked process (make it ready)
- User process continues when it is scheduled to run





Part 2: Filesystems

[Filesystems]

- A filesystem provides a high-level application access to disk
 - As well as CD, DVD, tape, floppy, etc...
 - Masks the details of low-level sector-based I/O operations
 - Provides structured access to data (files and directories)
 - Caches recently-accessed data in memory
- Hierarchical filesystems: Most common type
 - Organized as a tree of directories and files
- Byte-oriented vs. record-oriented files
 - UNIX, Windows, etc. all provide byte-oriented file access
 - May read and write files a byte at a time
 - Many older OS's provided only record-oriented files
 - File composed of a set of records; may only read and write a record at a time
- Versioning filesystems
 - Keep track of older versions of files
 - e.g., VMS filesystem: Could refer to specific file versions:foo.txt;1, foo.txt;2



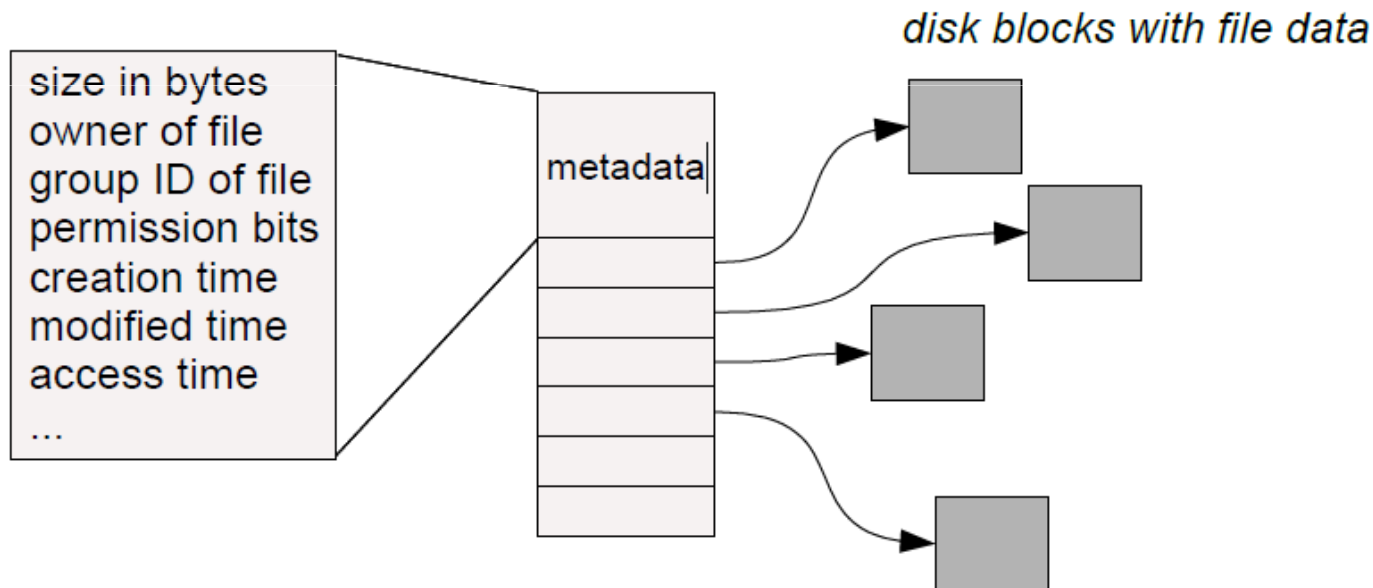
[Filesystem Operations]

- Filesystems provide a standard interface to files and directories:
 - Create a file or directory
 - Delete a file or directory
 - Open a file or directory – allows subsequent access
 - Read, write, append to file contents
 - Add or remove directory entries
 - Close a file or directory – terminates access
- What other features do filesystems provide?
 - Accounting and quotas – prevent your classmates from hogging the disks
 - Backup – some filesystems have a “\$HOME/.backup” containing automatic snapshots
 - Indexing and search capabilities
 - File versioning
 - Encryption
 - Automatic compression of infrequently-used files
- Should this functionality be part of the filesystem or built on top?
- Classic OS community debate: Where is the best place to put functionality?



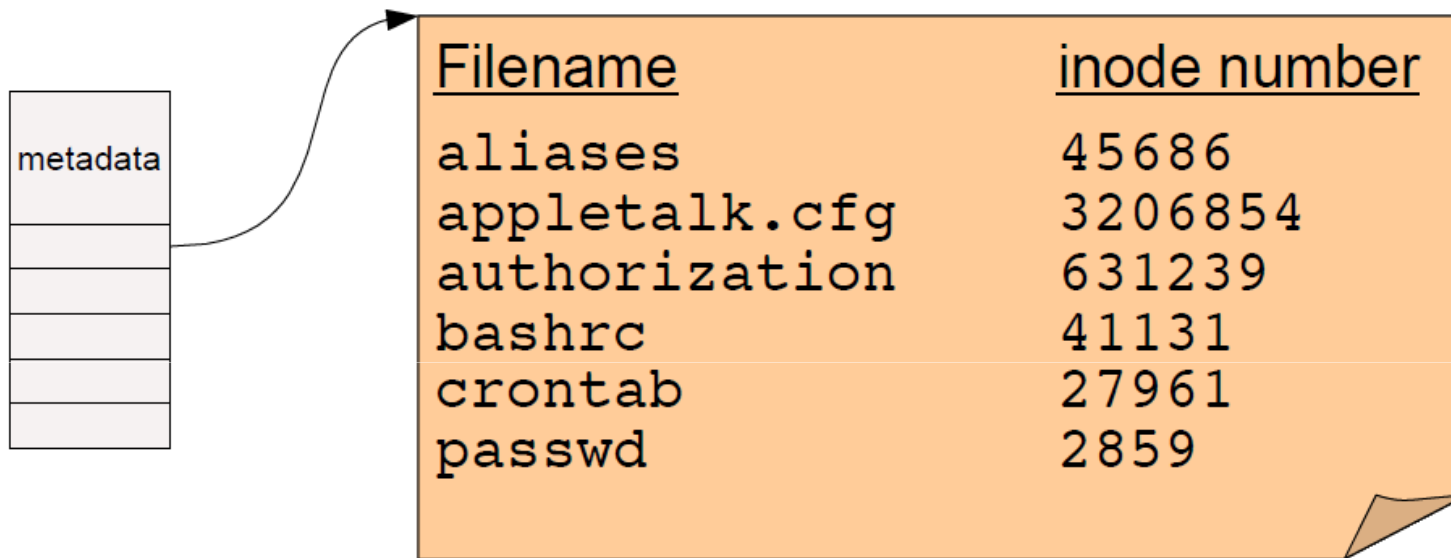
[Basic Filesystem Structures]

- Every file and directory is represented by an inode
 - Stands for “index node”
- Contains two kinds of information:
 - 1) Metadata describing the file's owner, access rights, etc.
 - 2) Location of the file's blocks on disk



[Directories]

- A directory is a special kind of file that contains a list of (filename, inode number) pairs

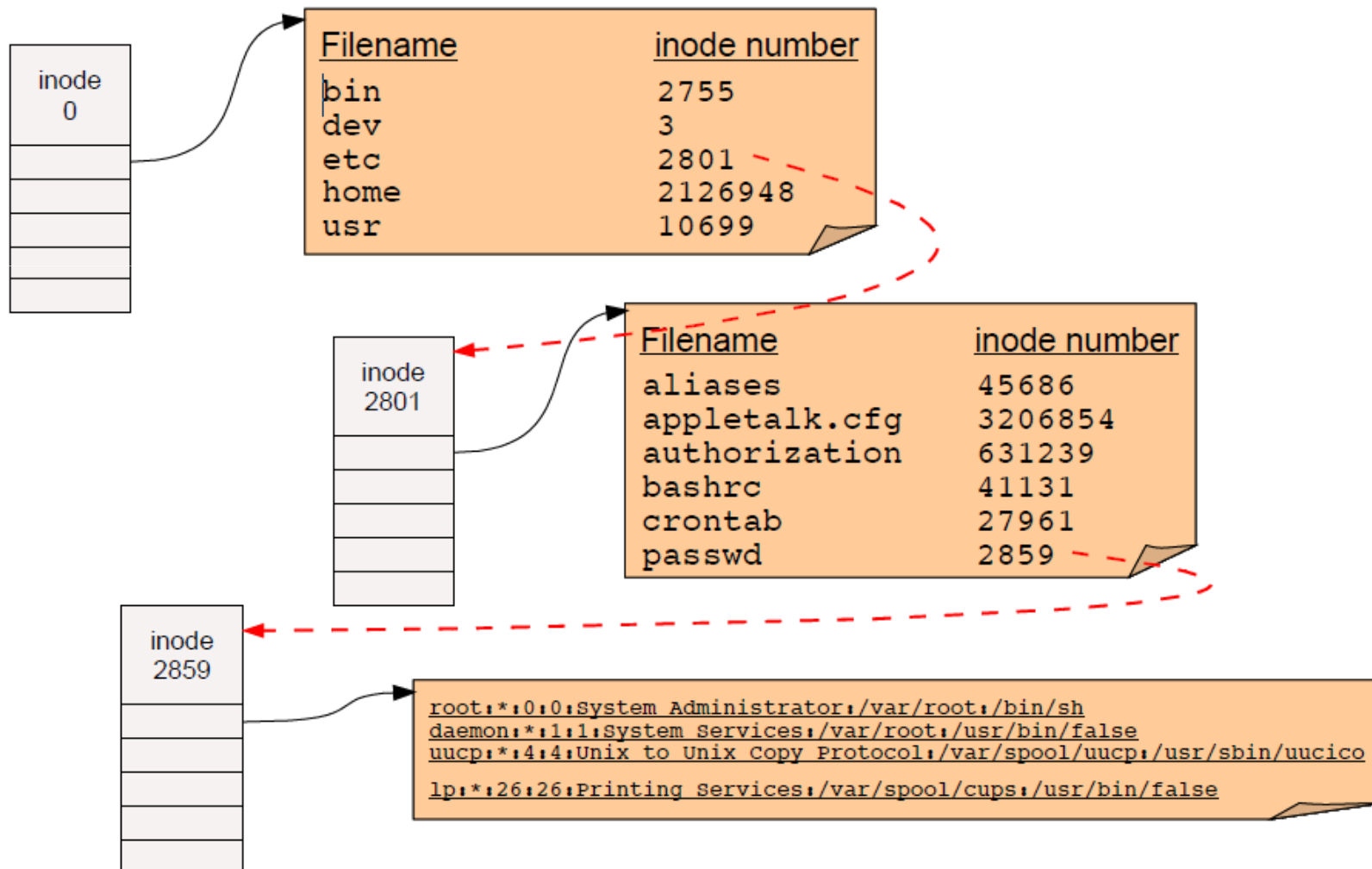


- These are the contents of the directory “file data” itself – NOT the directory's inode!
- Filenames (in UNIX) are not stored in the inode at all!
- Two open questions:
 - How do we find the root directory (“ / “ on UNIX systems)?
 - How do we get from an inode number to the location of the inode on disk?



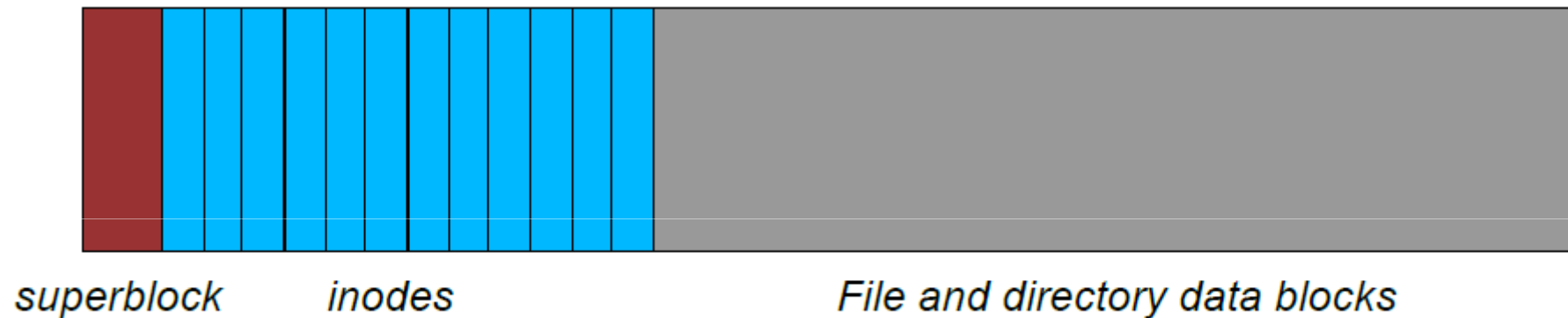
Pathname resolution

- To look up a pathname “/etc/passwd”, start at root directory and walk down chain of inodes...



[Locating inodes on disk]

- All right, so directories tell us the **inode number** of a file.
 - How the heck do we find the inode itself on disk?
- Basic idea: Top part of filesystem contains **all** of the inodes!



- inode number is just the “index” of the inode
- Easy to compute the block address of a given inode:
 - $\text{block_addr}(\text{inode_num}) = \text{block_offset_of_first_inode} + (\text{inode_num} * \text{inode_size})$
- This implies that a filesystem has a fixed number of potential inodes
 - This number is generally set when the filesystem is created
- The superblock stores important metadata on filesystem layout, list of free blocks, etc.



[Stupid directory tricks]

- Directories map filenames to inode numbers. What does this imply?
- We can create multiple pointers to the same inode in different directories
 - Or even the same directory with different filenames
- In UNIX this is called a “hard link” and can be done using “ln”

```
bash$ ls -i /home/foo
```

```
287663 /home/foo           (This is the inode number of “foo”)
```

```
bash$ ln /home/foo /tmp/foo
```

```
bash$ ls -i /home/foo /tmp/foo
```

```
287663 /home/foo
```

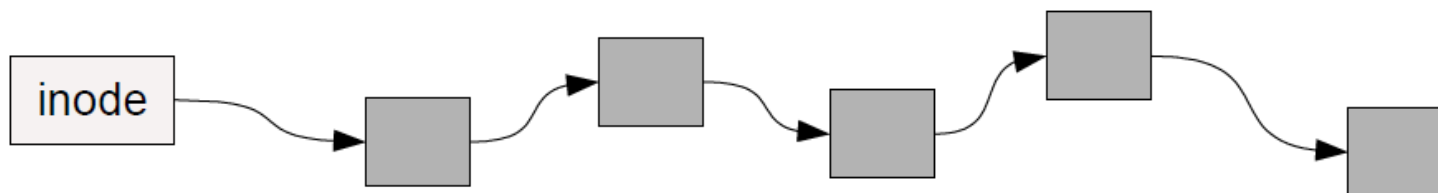
```
287663 /tmp/foo
```

- “/home/foo” and “/tmp/foo” now refer to the same file on disk
 - Not a copy! You will always see identical data no matter which filename you use to read or write the file.
- Note: This is not the same as a “symbolic link”, which only links one filename to another.

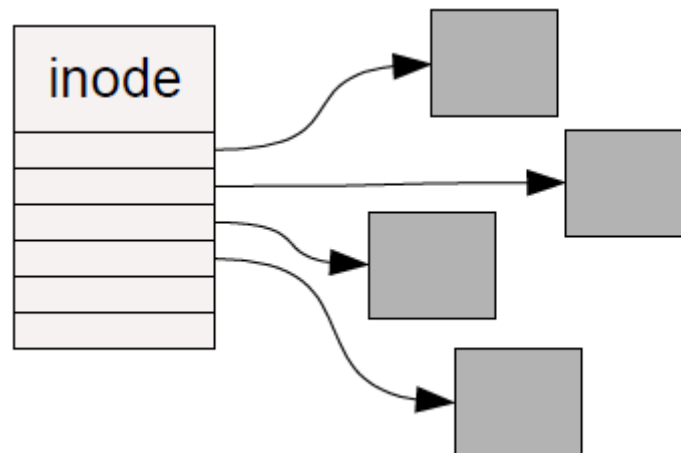


How should we organize blocks on a disk?

- Very simple policy: A file consists of linked blocks
 - inode points to the first block of the file
 - Each block points to the next block in the file (just a linked list on disk)
 - What are the advantages and disadvantages??

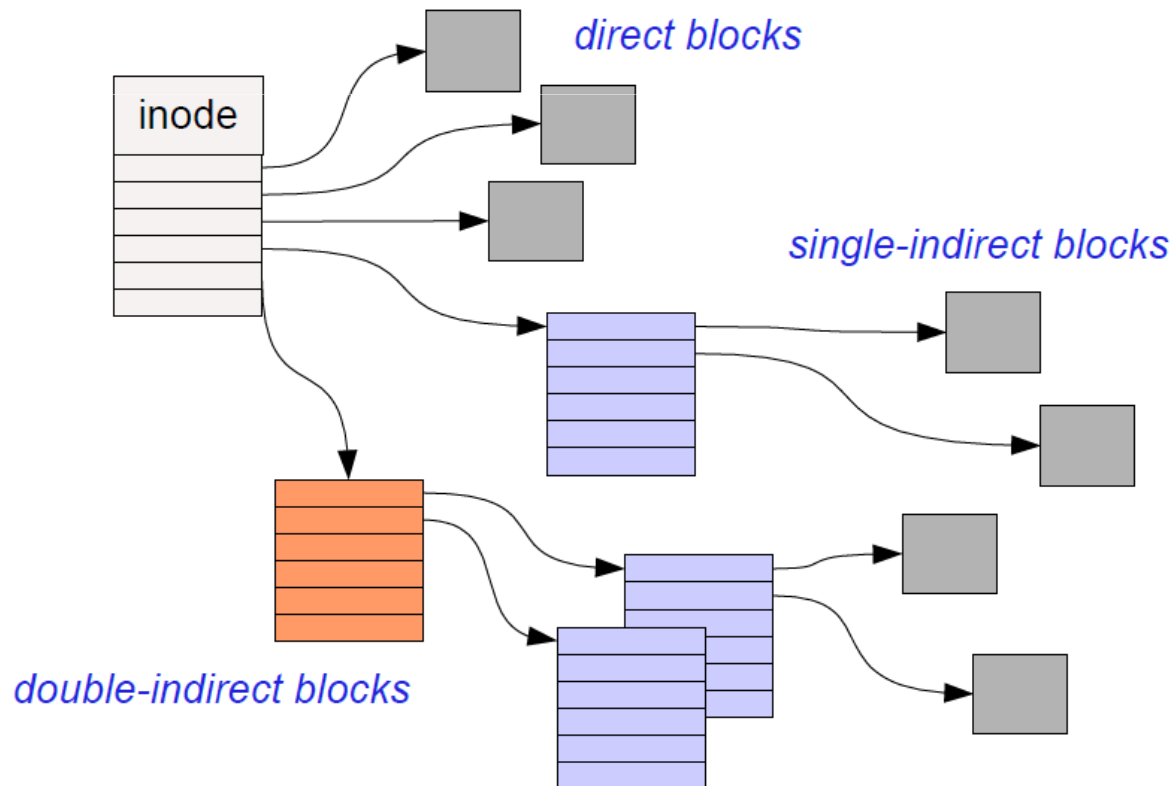


- Indexed files
 - inode contains a list of block numbers containing the file
 - Array is allocated when the file is created
 - What are the advantages and disadvantages??



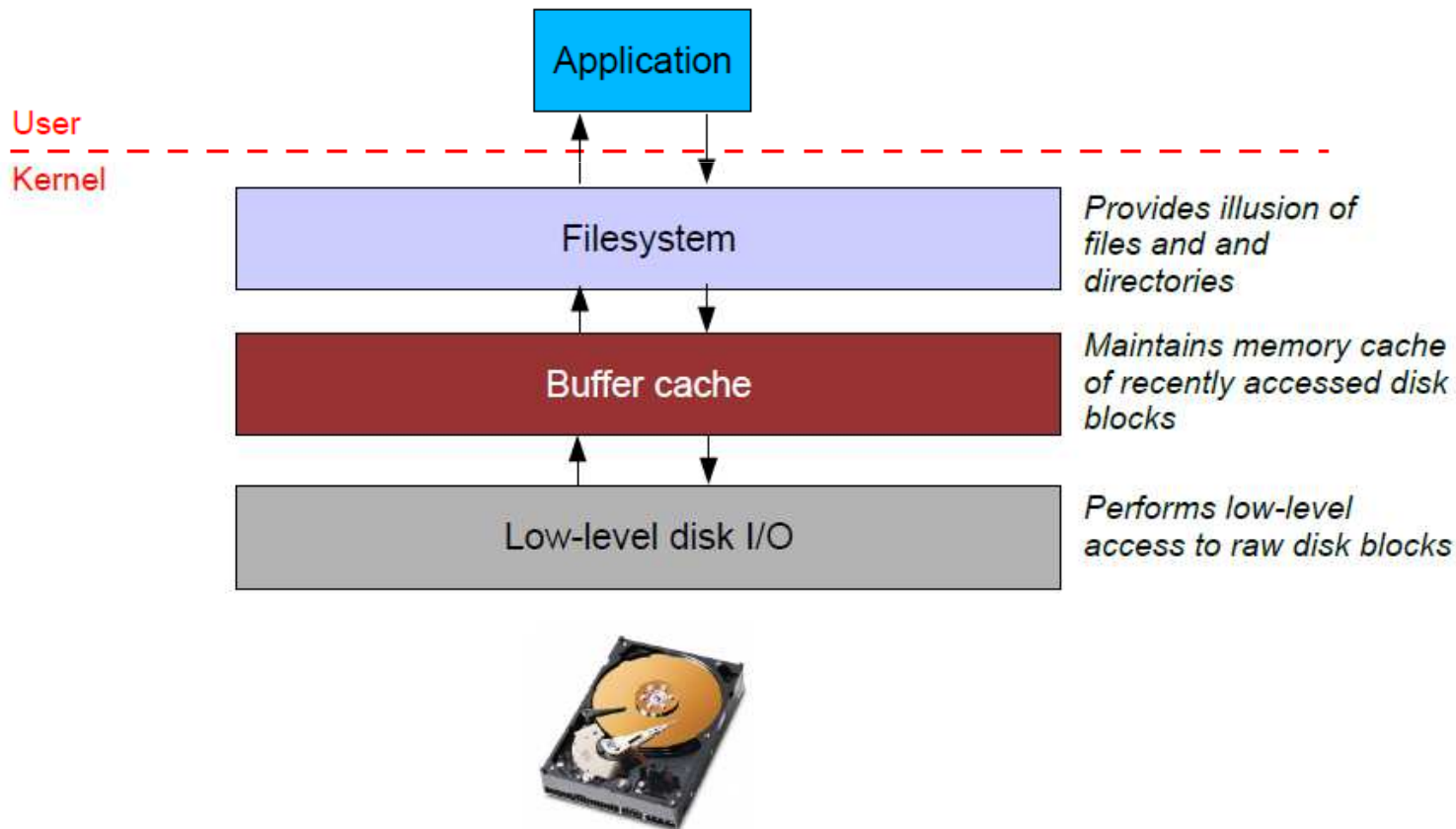
[Multilevel indexed files]

- inode contains a list of 10-15 **direct block pointers**
 - First few blocks of file can be referred to by the inode itself
- inode also contains a pointer to a **single indirect, double indirect, and triple indirect blocks**
 - Allows file to grow to be incredibly large!!!



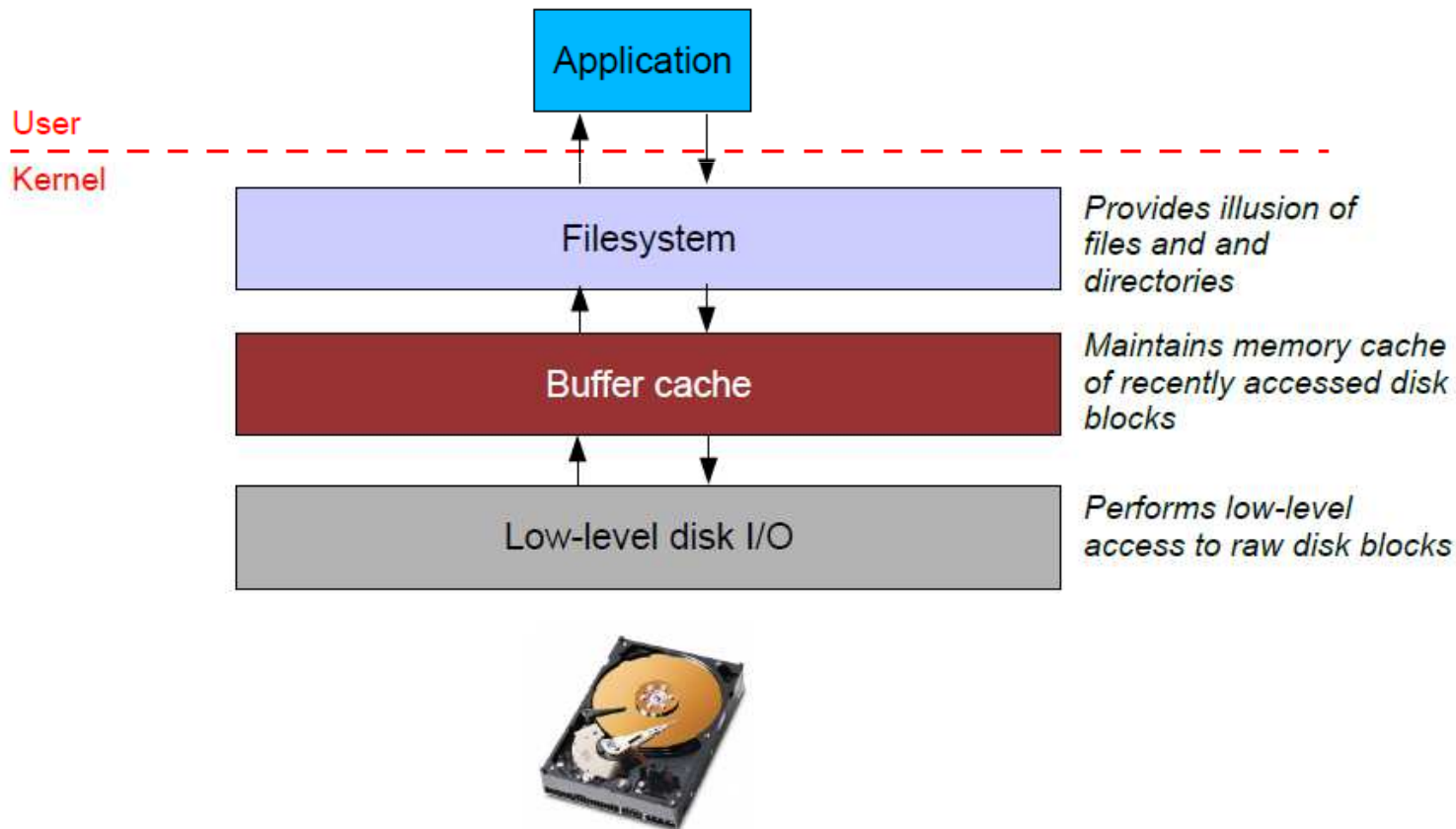
[File system caching]

- Most filesystems cache significant amounts of disk in memory
 - e.g., Linux tries to use all “free” physical memory as a giant cache
 - Avoids huge overhead for going to disk for every I/O



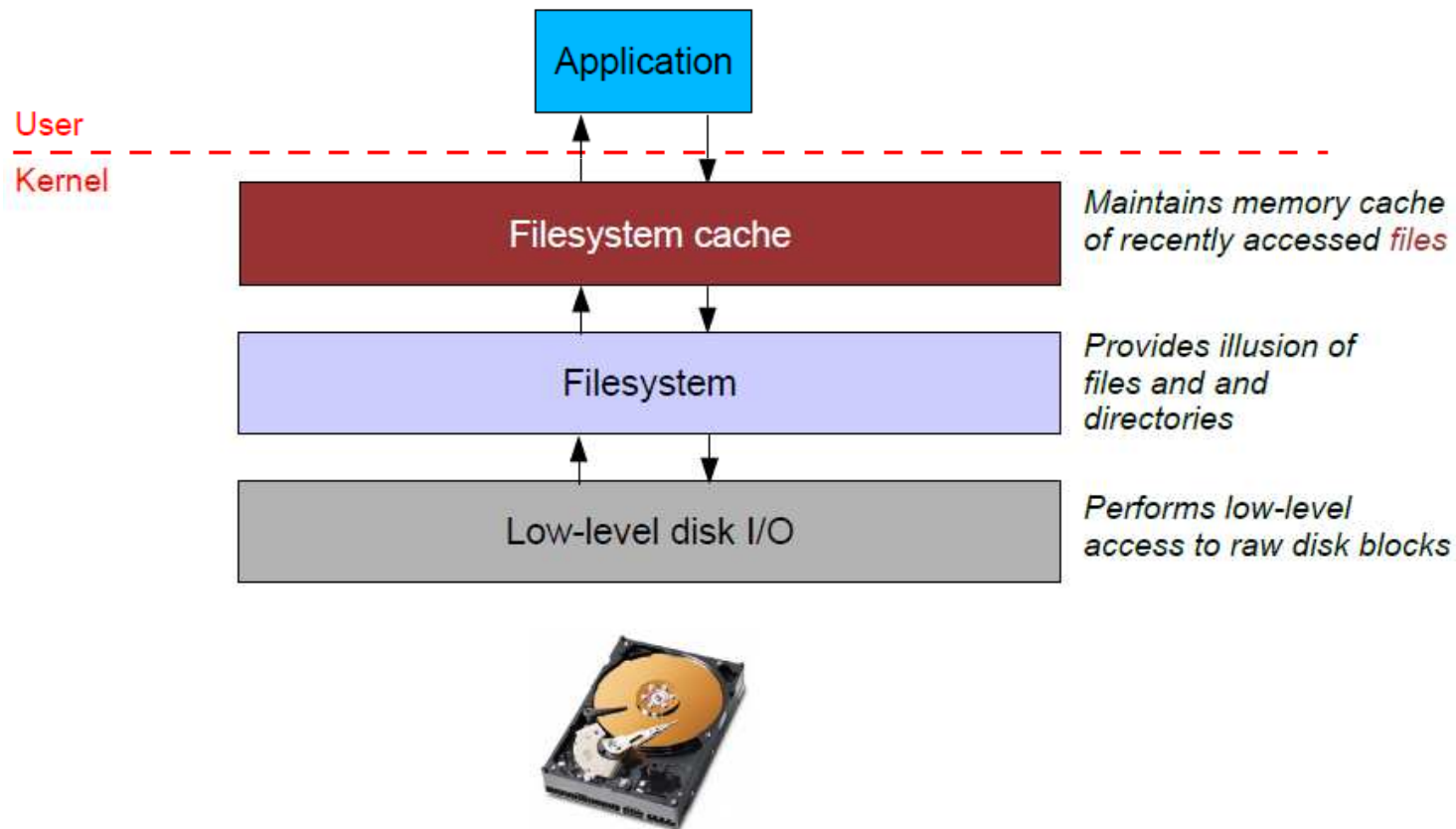
Caching issues

- Where should the cache go?
 - Below the filesystem layer: Cache individual disk blocks
 - Above the filesystem layer: Cache entire files and directories
 - Which is better??



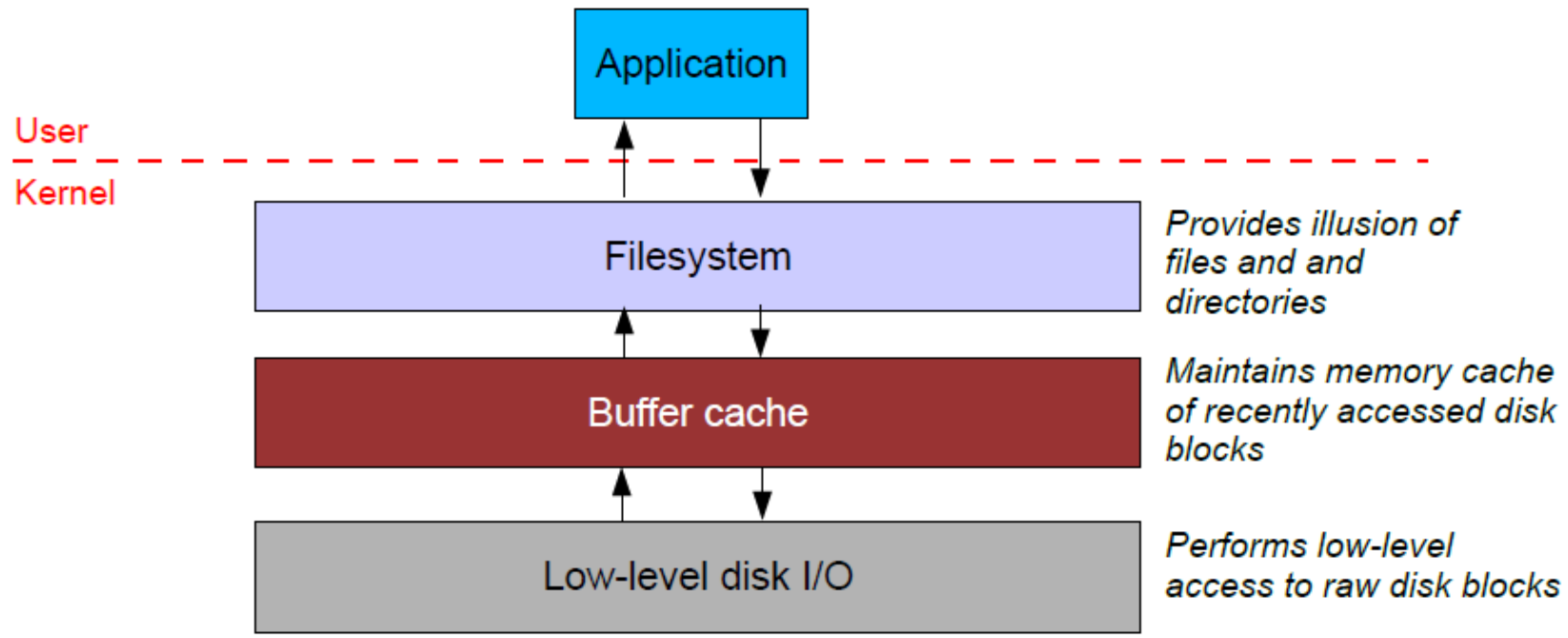
Caching issues

- Where should the cache go?
 - Below the filesystem layer: Cache individual disk blocks
 - Above the filesystem layer: Cache entire files and directories
 - Which is better??



Caching issues (2)

- Reliability issues
 - What happens when you write to the cache but the system crashes?
 - What if you update some of the blocks on disk but not others?
 - Example: Update the inode on disk but not the data blocks?
 - **Write-through cache:** All writes immediately sent to disk
 - **Write-back cache:** Cache writes stored in memory until evicted (then written to disk)
 - Which is better for performance? For reliability?



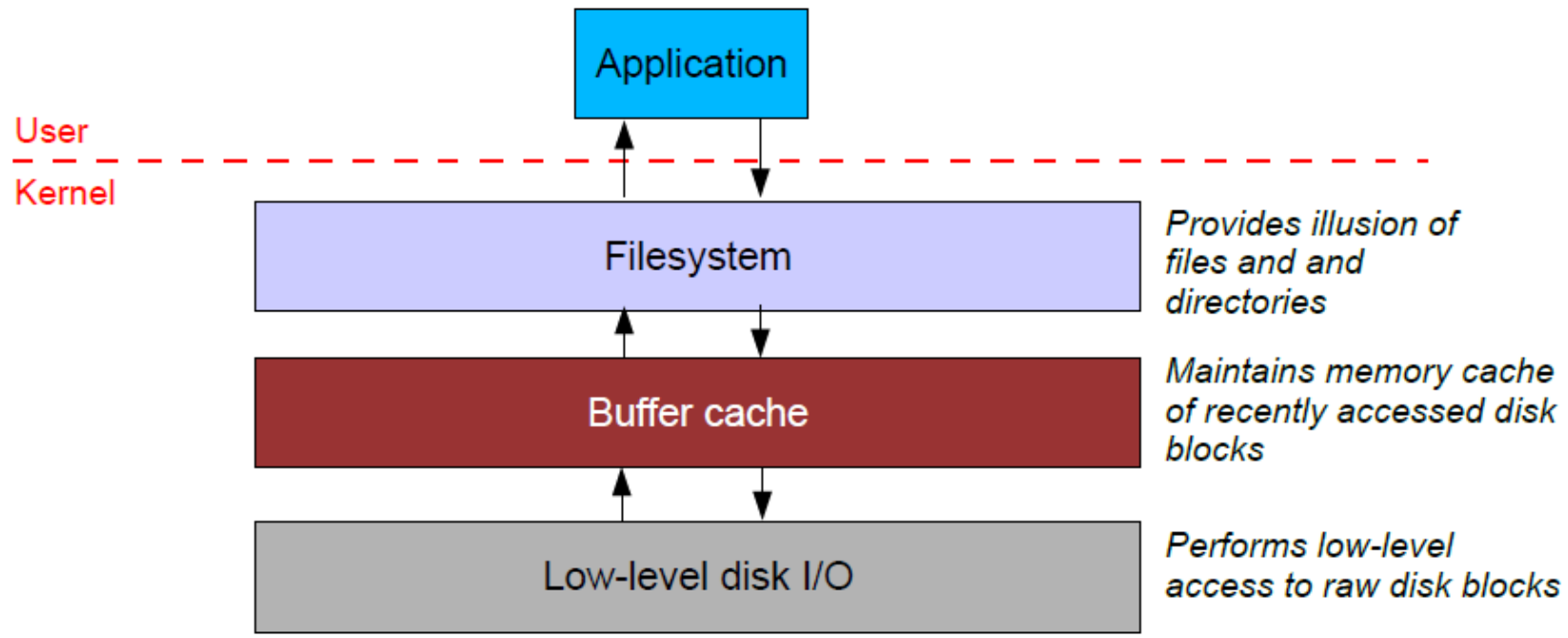
[Caching issues (2)]

- “Syncing” a filesystem writes back any dirty cache blocks to disk
 - UNIX “sync” command achieves this.
 - Can also use fsync() system call to sync any blocks for a given file.
 - Warning – not all UNIX systems guarantee that after sync returns that the data has really been written to the disk!
 - This is also complicated by memory caching on the disk itself.
- Crash recovery
 - If system crashes before sync occurs, “fsck” checks the filesystem for errors
 - Example: an inode pointing to a block that is marked as free in the free block list
 - Another example: An inode with no directory entry pointing to it
 - These usually get linked into a “lost+found” directory
 - inode does not contain the filename so need the sysadmin to look at the file data and guess where it might belong!



Caching issues (3)

- Read ahead
 - Recall: Seek time dominates overhead of disk I/O
 - So, would ideally like to read multiple blocks into memory when you have a cache miss
 - Amortize the cost of the seek for multiple reads
 - Useful if file data is laid out in contiguous blocks on disk
 - Especially if the application is performing sequential access to the file





Part 3: Modern Filesystems

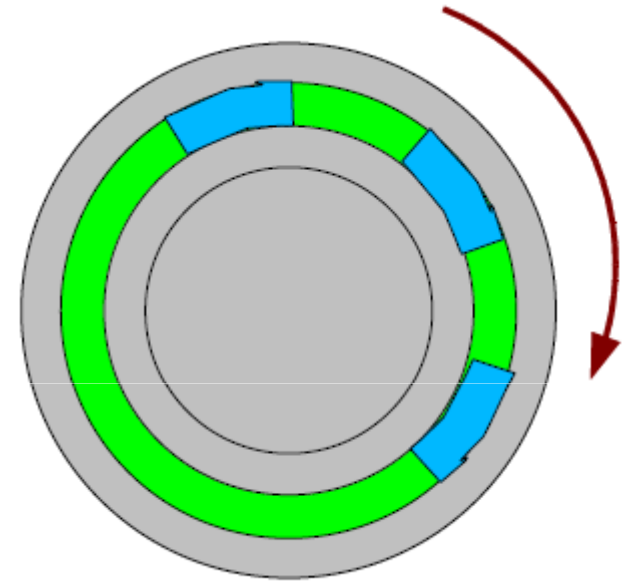
[Modern Filesystem Tricks]

- Extents
- Pre-allocation
- Delayed allocation (Block remapping)
- Colocating inodes and directories
- Soft metadata updates
- Journaling
- These tricks are used by many modern filesystems
 - E.g., ext3 and ext4



[Extent-based transfers]

- One idea: a gap between sectors on a track
 - Try to take advantage of rotational latency for performing next read or write operation
 - Problem: Hurts performance for multi-sector I/O!
 - Cannot achieve the full transfer rate of the disk for large, contiguous reads or writes.
- Possible fix: Just get rid of the gap between sectors
 - Problem: “Dropped rotation” between consecutive reads or writes: have to wait for next sector to come around under the heads.
- Hybrid approach - “extents” [McVoy, USENIX'91]
 - Group blocks into “extents” or clusters of contiguous blocks
 - Try to do all I/O on extents rather than individual blocks
 - To avoid wasting I/O bandwidth, only do this when FS detects sequential access
 - Kind of like just increasing the block size...



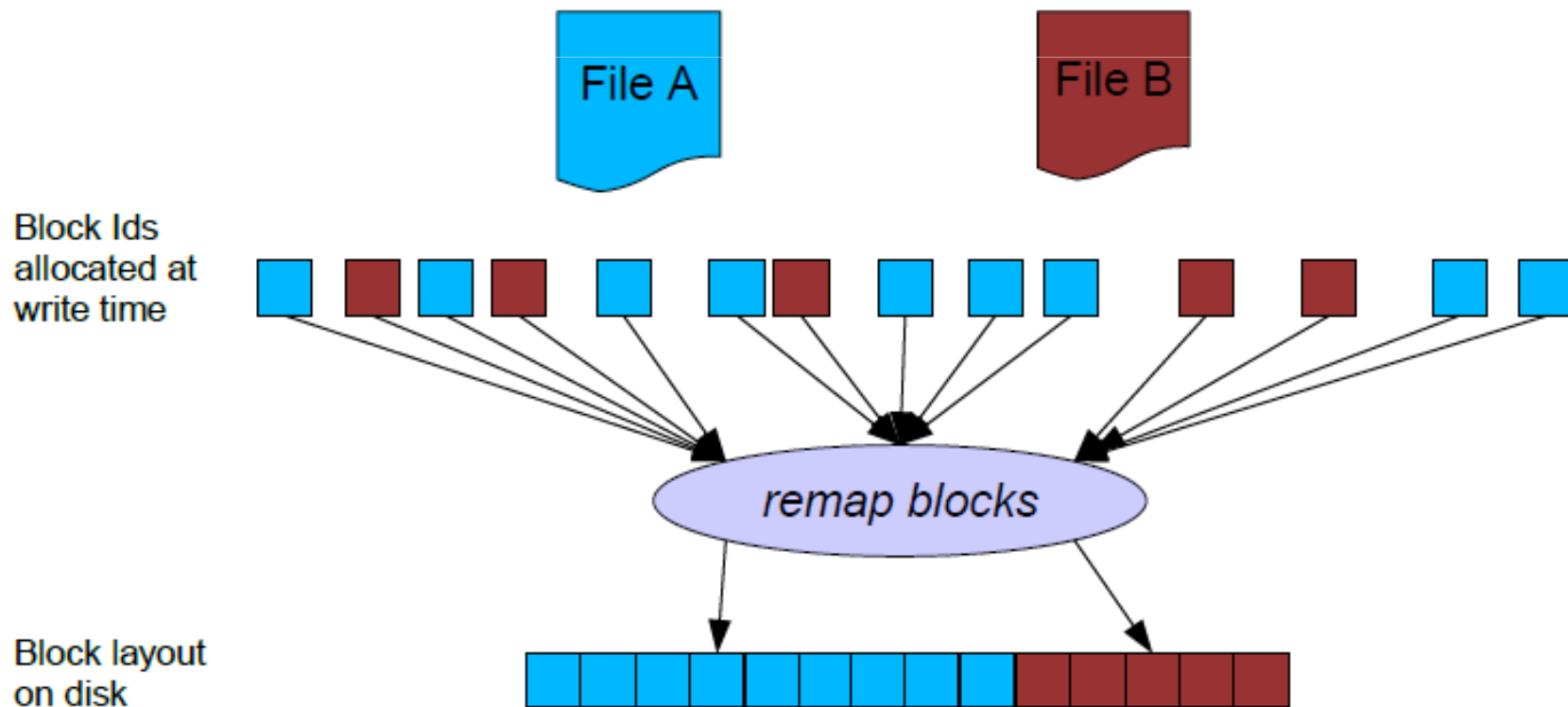
[Block remapping]

- Problem: Block numbers are allocated when they are first written
 - FS maintains a free list of blocks and simply picks the first block off the list
 - No guarantee that these blocks will be contiguous for a large write!
 - A single file may end up with blocks scattered across the disk
- Why can't we maintain the free list in some sorted order?
 - Problem: Interleaved writes to multiple files may end up causing each file to be discontinuous.



[Block remapping]

- Idea: Delay determination of block address until cache is flushed
 - Hope that multiple block writes will accumulate in the cache
 - Can remap the block addresses for each file's writes to a contiguous set
 - This is kind of a hack, introduced “underneath” the FFS block allocation layer.
 - Meant fewer changes to the rest of the FFS code.
 - Sometimes building real systems means making these kinds of tradeoffs!



[Colocating inodes and directories]

- Problem: Reading small files is slow. Why?
 - What happens when you try to read all files in a directory (e.g., “ls -l” or “grep foo *”) ?
 - Must first read directory.
 - Then read inode for each file.
 - Then read data pointed to by inode.
- Solution: Embed the inodes in the directory itself!
 - Recall: Directory just a set of <name, inode #> values
 - Why not stuff inode contents in the directory file itself?
- Problem #2: Must still seek to read contents of each file in the directory.
 - Solution: Pack all files in a directory in a contiguous set of blocks.



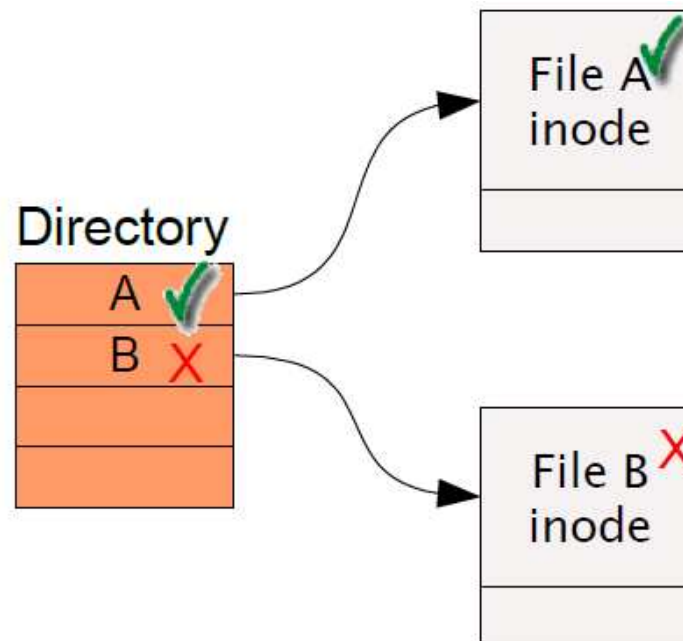
[Synchronous metadata updates]

- Problem: Some updates to metadata require synchronous writes
 - Means the data has to “hit the disk” before anything else can be done.
- Example #1: Creating a file
 - Must write the new file's inode to disk before the corresponding directory entry.
 - Why???
- Example #2: Deleting a file
 - Must clear out the directory entry before marking the inode as “free”
 - Why???



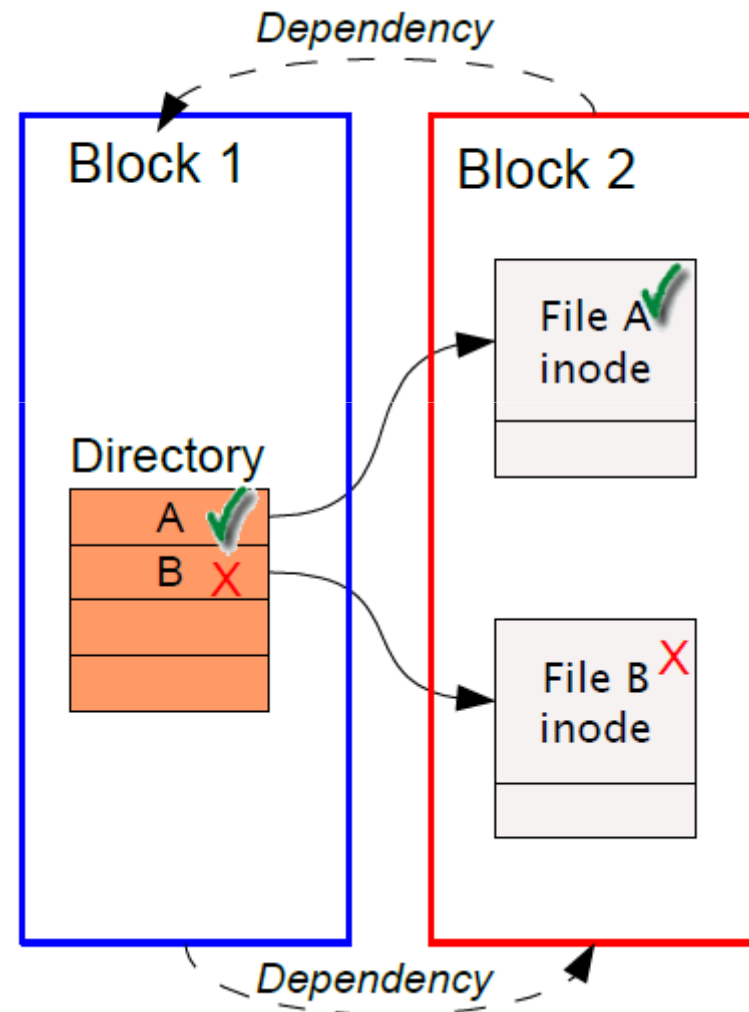
[Synchronous metadata updates]

- Problem: Some updates to metadata require synchronous writes
 - Means the data has to “hit the disk” before anything else can be done.
- Example #1: Creating a file
 - Must write the new file's inode to disk before the corresponding directory entry.
 - Why???
- Example #2: Deleting a file
 - Must clear out the directory entry before marking the inode as “free”
 - Why???



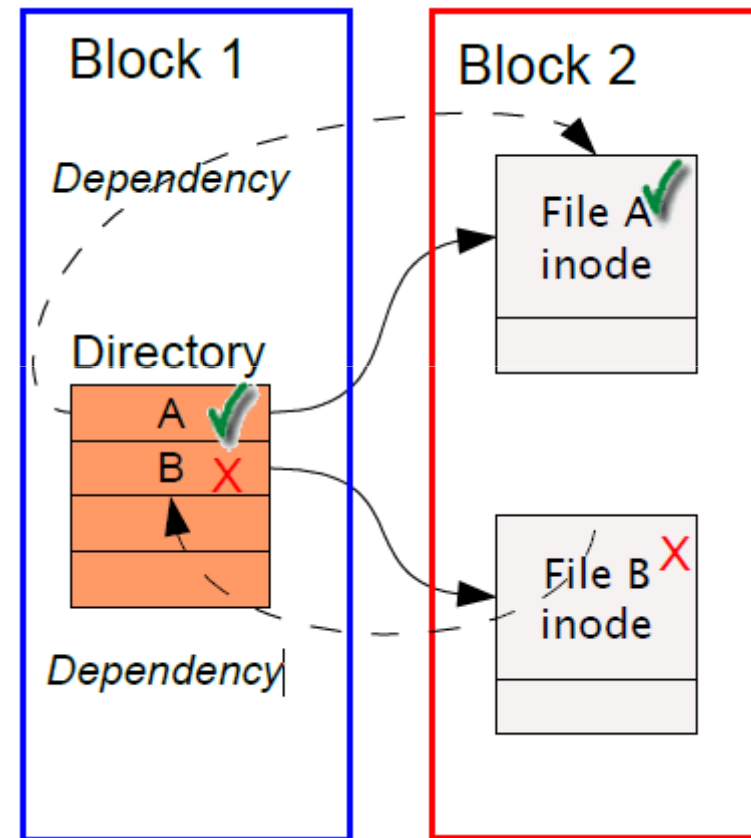
[Synchronous metadata updates]

- Say that ...
 - 1) Both inodes are in the same disk block.
 - 2) Both the file create and file delete have happened in the cache, but neither has hit the disk yet.
 - Given this, what order are we allowed to write the disk blocks out?
 - *We have a cyclic dependency here!!!
Argggghhhh*



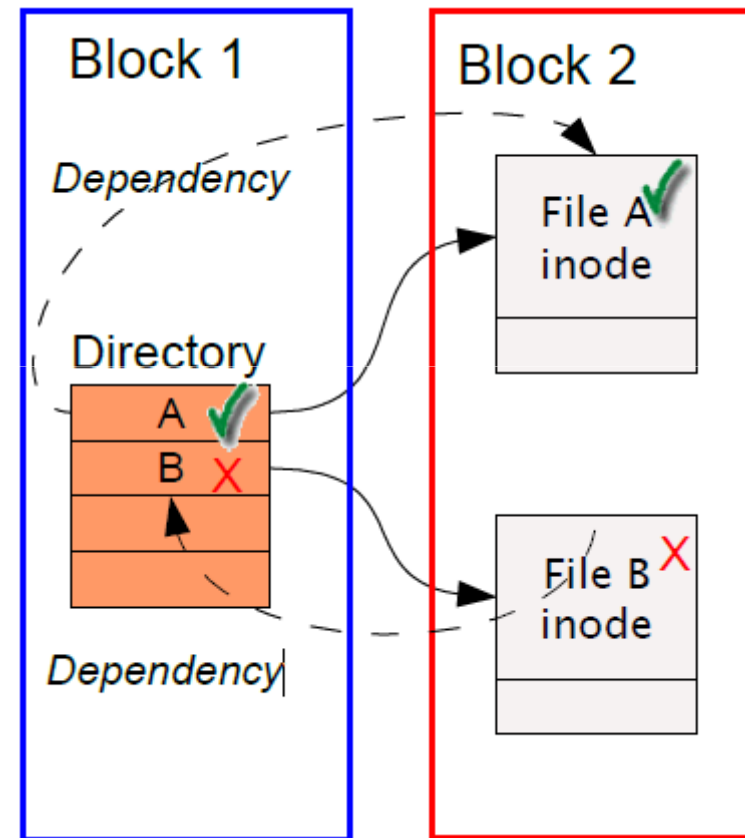
[Solution: Soft Updates]

- Idea: Keep track of dependencies on a finer granularity
 - Rather than at a block level, do this at a “data structure level”
 - Example: Track dependencies on individual inodes or directory entries.



[Soft Updates - Example]

- How to break the cyclic dependency?
 - “Roll back” one of the changes before writing the data out to disk!
- When flushing inode block (Block 2) to disk...
 - **Undo** the file delete operation (as if it never happened!)
 - Write out the inode block (Block 2) – still contains B!
 - Then write out the directory block (Block 1) – still contains entry for B!
 - Then **redo** the file delete operation ... can now proceed.



[Log-structured Filesystems (LFS)]

- Around '91, two trends in disk technology were emerging:
 - Disk bandwidth was increasing rapidly (over 40% a year)
 - Seek latency not improving much at all
 - Machines had increasingly large main memories
 - Large buffer caches absorb a large fraction of read I/Os
 - Can use for writes as well!
 - Coalesce several small writes into one larger write
- Some lingering problems with earlier filesystems...
 - Writing to file metadata (inodes) was required to be synchronous
 - Couldn't buffer metadata writes in memory
 - Lots of small writes to file metadata means lots of seeks!
- LFS takes advantage of both to increase FS performance
 - Started as a grad-school research project at Berkeley
 - Mendel Rosenblum and John Ousterhout

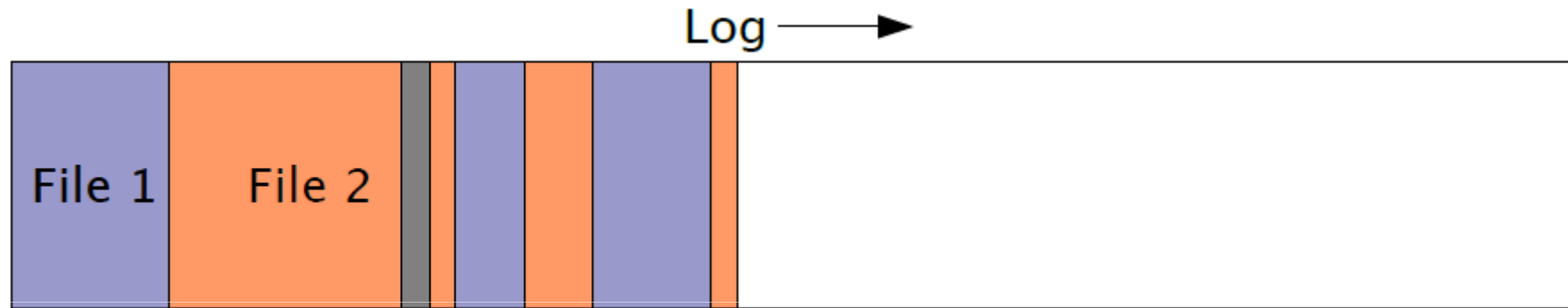


[LFS: The basic idea]

- Treat the entire disk as *one big append-only log* for writes!
 - Don't try to lay out blocks on disk in some predetermined order
 - Whenever a file write occurs, append it to the *end of the log*
 - Whenever file metadata changes, append it to the *end of the log*
- Collect pending writes in memory and stream out in one big write
 - Maximizes disk bandwidth
 - No “extra” seeks required (only those to move the end of the log)
- When do writes to the actual disk happen?
 - When a user calls `sync()` -- synchronize data on disk for whole filesystem
 - When a user calls `fsync()` -- synchronize data on disk for one file
 - When OS needs to reclaim dirty buffer cache pages
 - *Note that this can often be avoided, eg., by preferring clean pages*
- Sounds simple ...
 - But lots of hairy details to deal with!



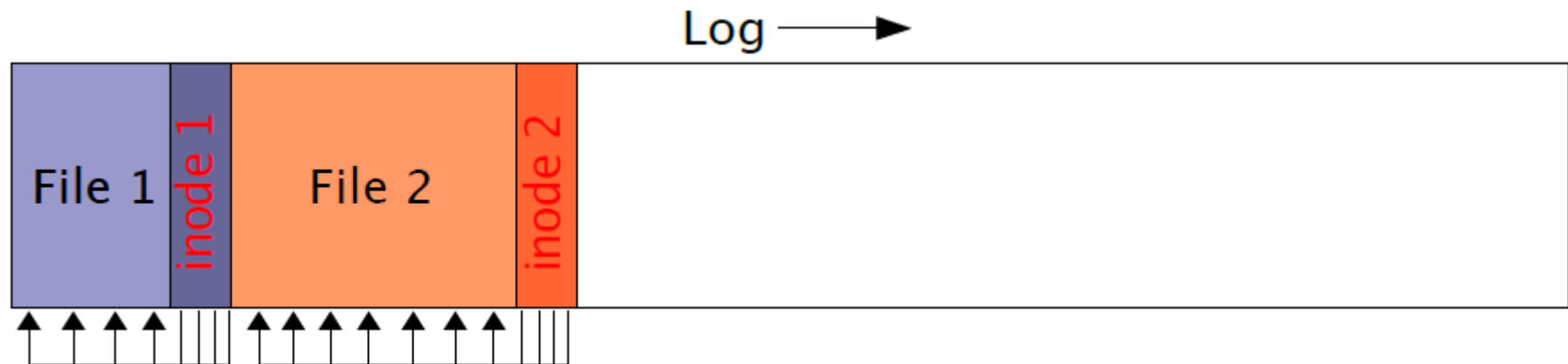
[LFS Example]



- Just append every new write that happens to the end of the log
 - Writing a block in the middle of the file just appends that block to the end of the log

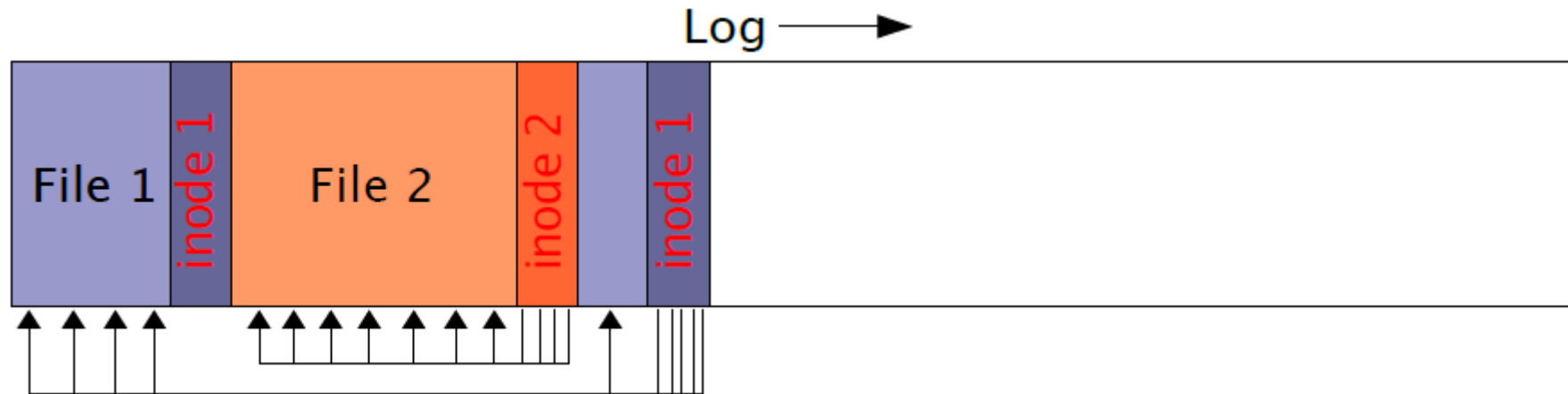
[LFS and inodes]

- How do you locate file data?
 - Sequential scan of the log is probably a bad idea ...
- Solution: Write the inodes to the tail of the log! (just like regular data)



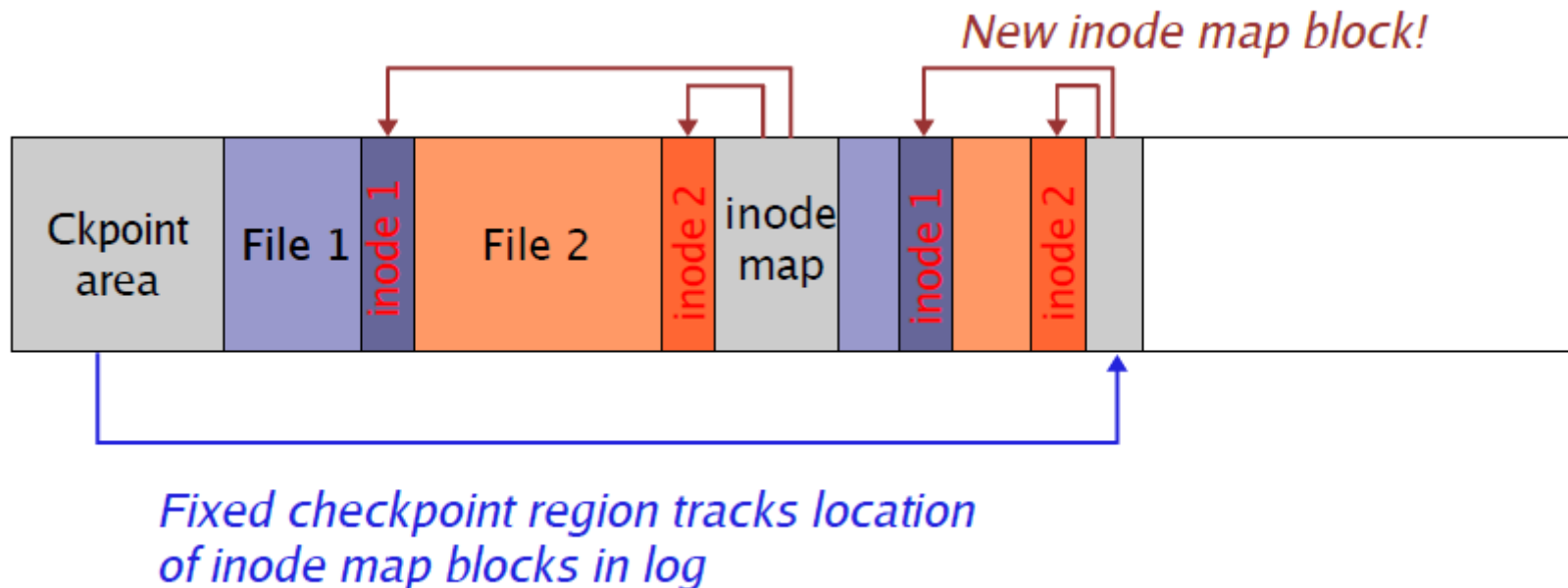
[LFS and inodes]

- How do you locate file data?
 - Sequential scan of the log is probably a bad idea ...
- Solution: Use FFS-style inodes!



[inode map (this is getting fun)]

- Well, now, how do you find the inodes??
 - Could also be anywhere in the log!
- Solution: *inode maps*
 - Maps “file number” to the location of its inode in the log
 - Note that inode map is *also* written to the log!!!!
 - Cache inode maps in memory for performance



[Reading from LFS]

- But wait ... now file data is scattered all over the disk!
 - Seems to obviate all of the benefits of grouping data on common cylinders
- Basic assumption: Buffer cache will handle most read traffic
 - Or at least, reads will happen to data roughly in the order in which it was written
 - Take advantage of huge system memories to cache the heck out of the FS!



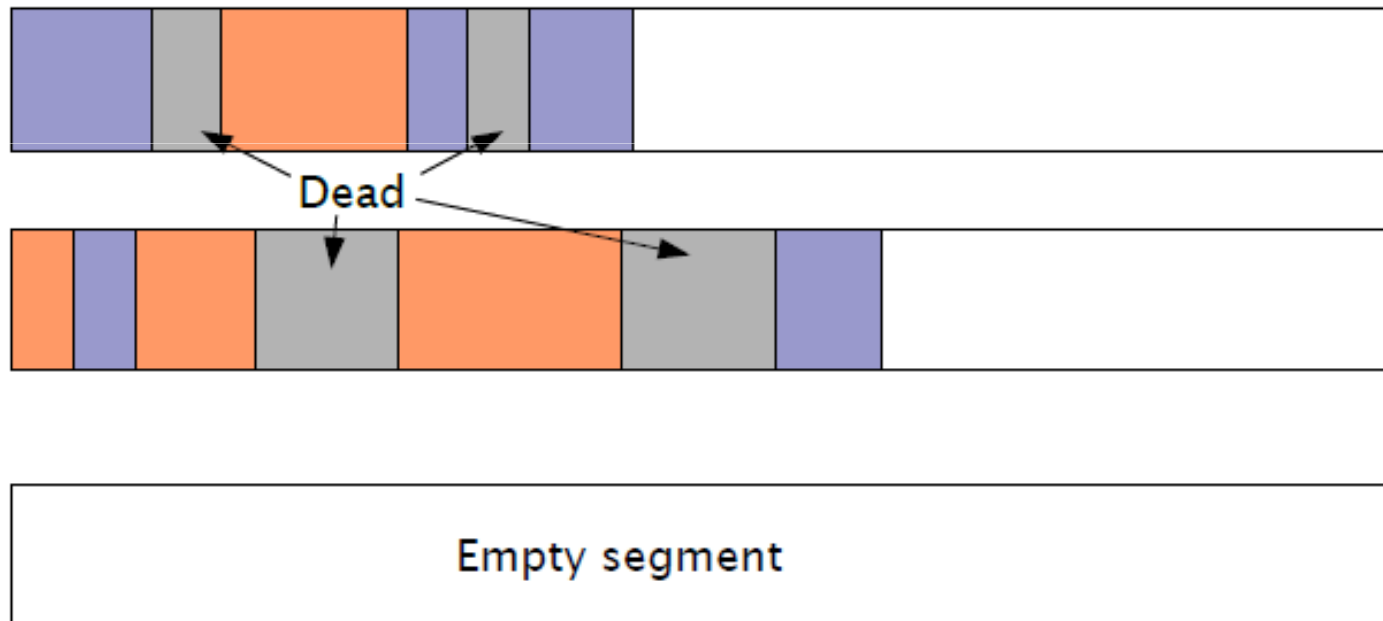
[Log cleaner]

- With LFS, eventually the disk will fill up!
 - Need some way to reclaim “dead space”
- What constitutes “dead space?”
 - Deleted files
 - File blocks that have been “overwritten”
- Solution: Periodic “log cleaning”
- Scan the log and look for deleted or overwritten blocks
 - Effectively, clear out stale log entries
- Copy *live data to the end of the log*
 - The rest of the log (at the beginning) can now be reused!



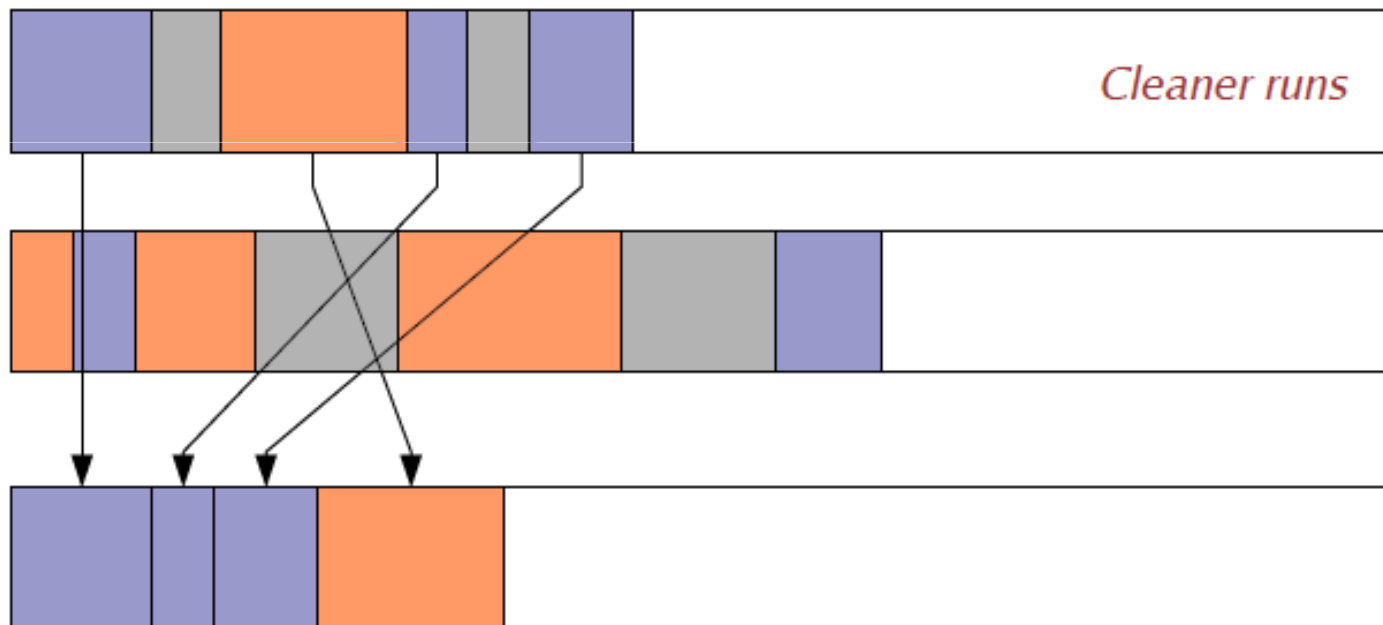
[Log cleaning example]

- LFS cleaner breaks log into *segments*
 - Each segment is scanned by the cleaner
 - Live blocks from a segment are copied into a new segment
 - The entire scanned segment can then be reclaimed



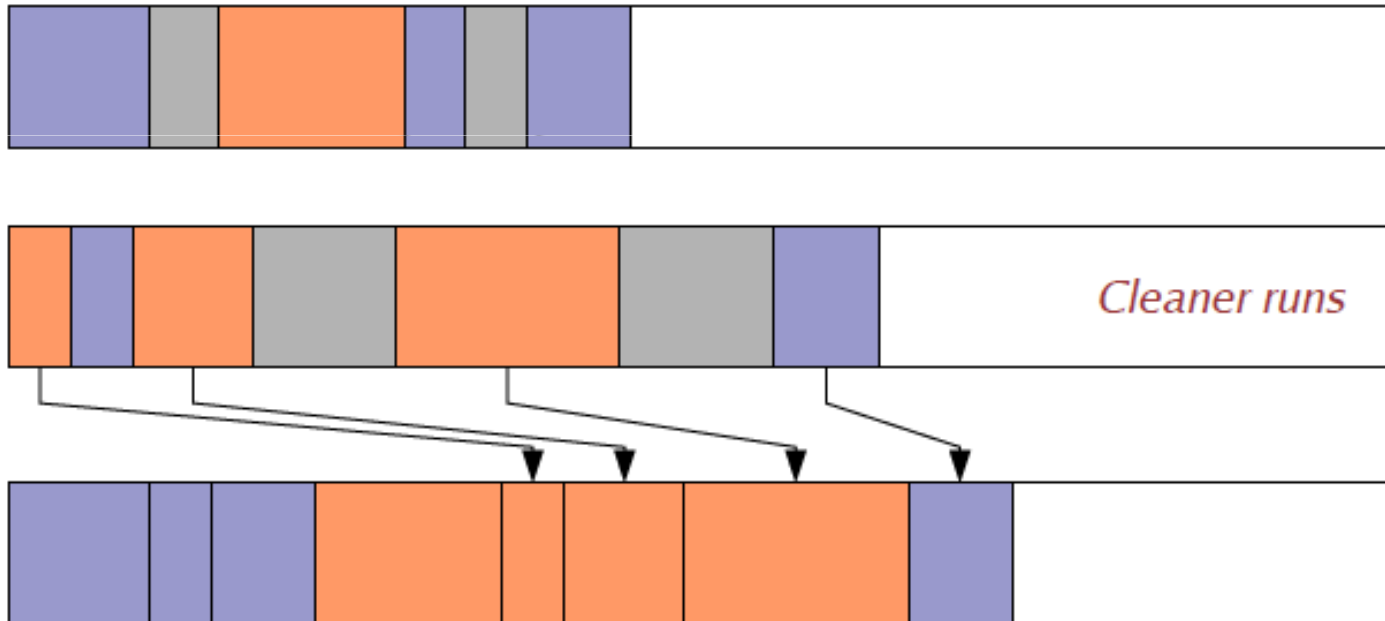
[Log cleaning example]

- LFS cleaner breaks log into *segments*
 - Each segment is scanned by the cleaner
 - Live blocks from a segment are copied into a new segment
 - The entire scanned segment can then be reclaimed



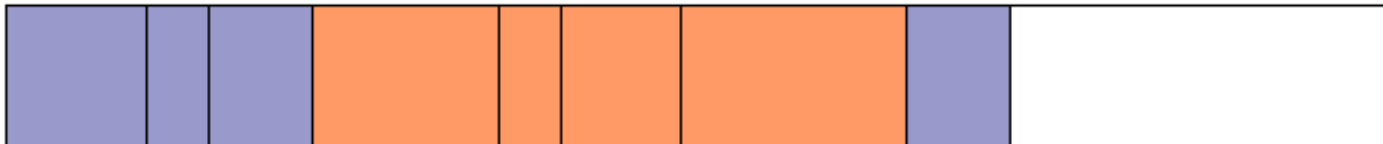
[Log cleaning example]

- LFS cleaner breaks log into *segments*
 - Each segment is scanned by the cleaner
 - Live blocks from a segment are copied into a new segment
 - The entire scanned segment can then be reclaimed



[Log cleaning example]

- LFS cleaner breaks log into *segments*
 - Each segment is scanned by the cleaner
 - Live blocks from a segment are copied into a new segment
 - The entire scanned segment can then be reclaimed



[Properties of LFS]

■ Advantages

- High write throughput
- Few in-place writes
 - Some kinds of storage media have limited write/erase cycles per location (e.g., flash memory, CD-RW)
 - LFS prolongs life of media through [write-leveling](#)

■ Disadvantages

- Increases file fragmentation, can harm performance on systems with high seek times
- Less throughputs on flash memory, where write fragmentation has much less of an impact on write throughput

■ “Lies, damn lies, and benchmarks”

- It is very difficult to come up with definitive benchmarks proving that one system is better than another
- Can always find a scenario where one system design outperforms another



[Filesystem corruption]

- What happens when you are making changes to a filesystem and the system crashes?
 - Example: Modifying block 5 of a large directory, adding lots of new file entries
 - System crashes while the block is being written
 - The new files are “lost!”
- System runs `fsck` program on reboot
 - Scans through the entire filesystem and locates corrupted inodes and directories
 - Can typically find the bad directory, but may not be able to repair it!
 - *The directory could have been left in any state during the write*
- `fsck` can take a very long time on large filesystems
 - And, no guarantees that it fixes the problems anyway



[Journaling filesystems]

- Ensure that changes to the filesystem are made *atomically*
 - That is, a group of changes are made all together, or not at all
- Example: creating a new file
 - Need to write both the inode for the new file and the directory entry “together”
 - Otherwise, if a crash happens between the two writes, either..
 - 1) Directory points to a file that does not exist
 - 2) Or, file is on disk but not included in any directory



[Journaling filesystems]

- Goal: Make updates to filesystems appear to be atomic
 - The directory either looks exactly as it did **before the file was created**
 - Or the directory looks exactly as it did **after the file was created**
 - Cannot leave an FS entity (data block, inode, directory, etc.) in an intermediate state!
- Idea: Maintain a **log of all changes to the filesystem**
 - Log contains information on any operations performed to the filesystem state
 - e.g., “Directory 2841 had inodes 404, 407, and 408 added to it”
- To make a filesystem change:
 - 1. Write an *intent-to-commit* record to the log
 - 2. Write the appropriate **changes to the log**
 - *Do not modify the filesystem data directly!!!*
 - 3. Write a *commit record* to the log
- This is very similar to the notion of database *transactions*⁸⁵

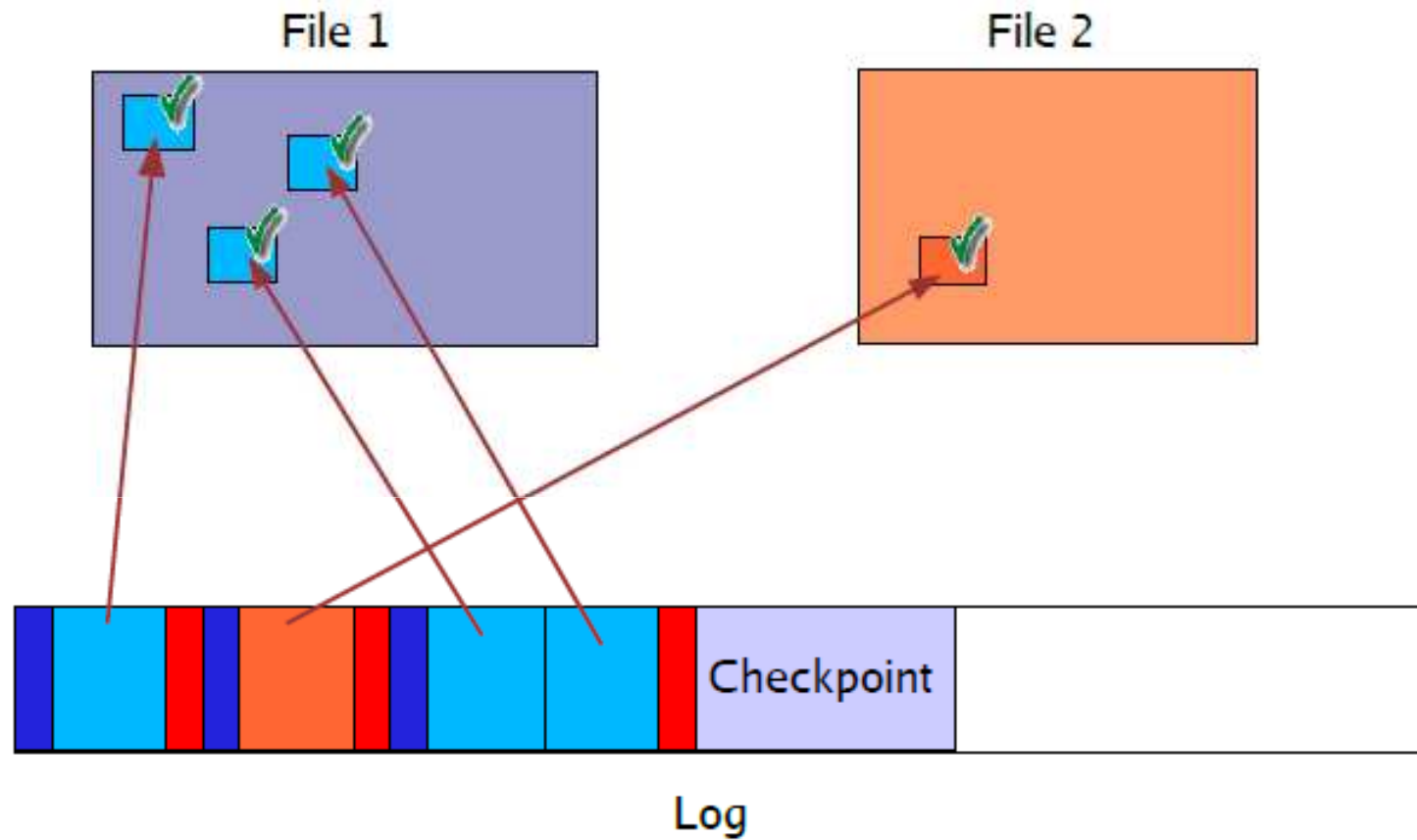


[Journaling FS Recovery]

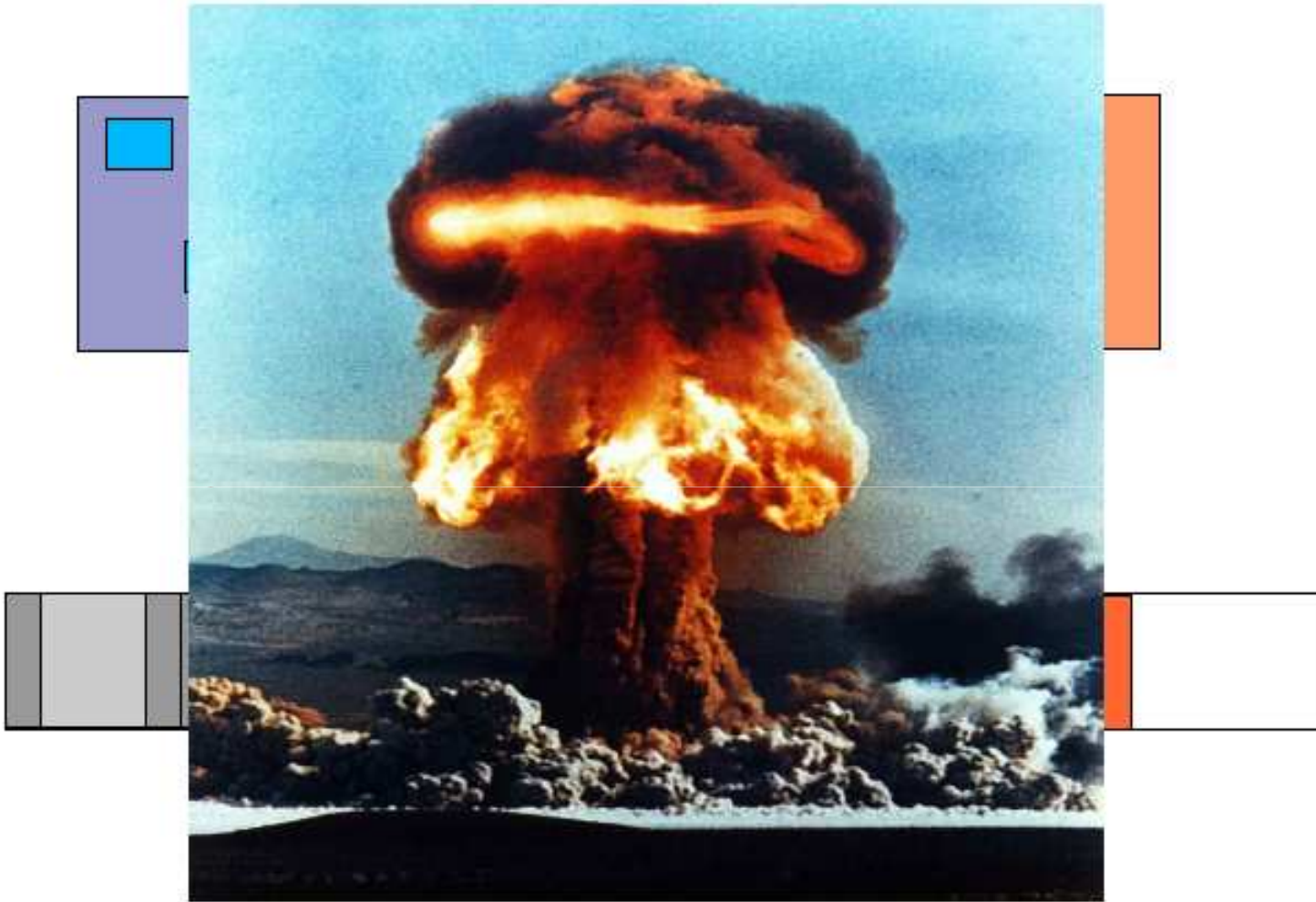
- What happens when the system crashes?
 - Filesystem data has not actually been modified, just the log!
 - So, the FS itself reflects only what happened *before the crash*
- Periodically synchronize the log with the filesystem data
 - Called a *checkpoint*
 - Ensures that the FS data reflects all of the changes in the log
- No need to scan the entire filesystem after a crash...
 - Only need to look at the log entries **since the last checkpoint!**
- For each log entry, see if the commit record is there
 - If not, consider the changes incomplete, and don't try to make them



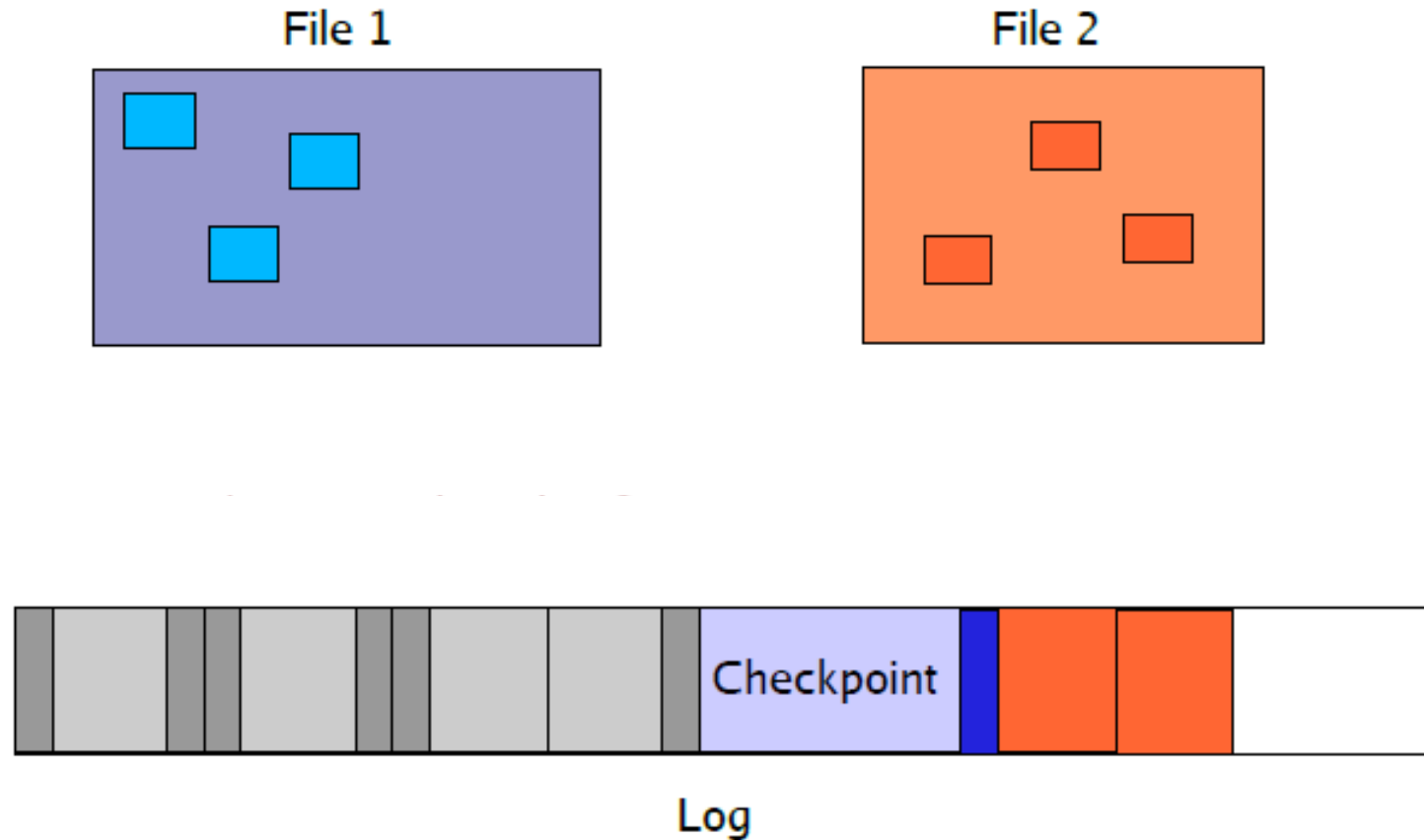
[Journaling FS Example]



[Journaling FS Example]



[Journaling FS Example]



- Filesystem reflects changes up to last checkpoint
- Fsync scans changelog from last checkpoint forward
- Doesn't find a commit record ... changes are simply ignored



Part 5: Advanced filesystems

[More recent filesystems]

- How can we share filesystems over a network?
 - NFS, SAN, NAS
- How can we make a filesystem resilient to failures?
 - RAID



[Networked File System (NFS)]

- NFS allows a system to access files over a network
 - One of many distributed file systems
 - Extremely successful and widely used
 - You use NFS on all your shared files in the lab machines



[Networked File System (NFS)]

- Development of LANs made it really attractive to provide shared file systems to all machines on a network
 - Login to any machine and see the same set of files
 - Install software on a single server that all machines can run
 - Let users collaborate on shared set of files (before CVS)

- Why might this be hard to do?
 - Clients and servers might be running different OS
 - Clients and servers might be using different CPU architecture with differing byte ordering (**endianess**)
 - Client or server might crash independently of each other
 - Must be easy to recover from crashes
 - Potentially very large number of client machines on a network
 - Different users might be trying to modify a shared file at the same time
 - Transparency: Allow user programs to access remote files just like local files
 - No special libraries, recompilation, etc.

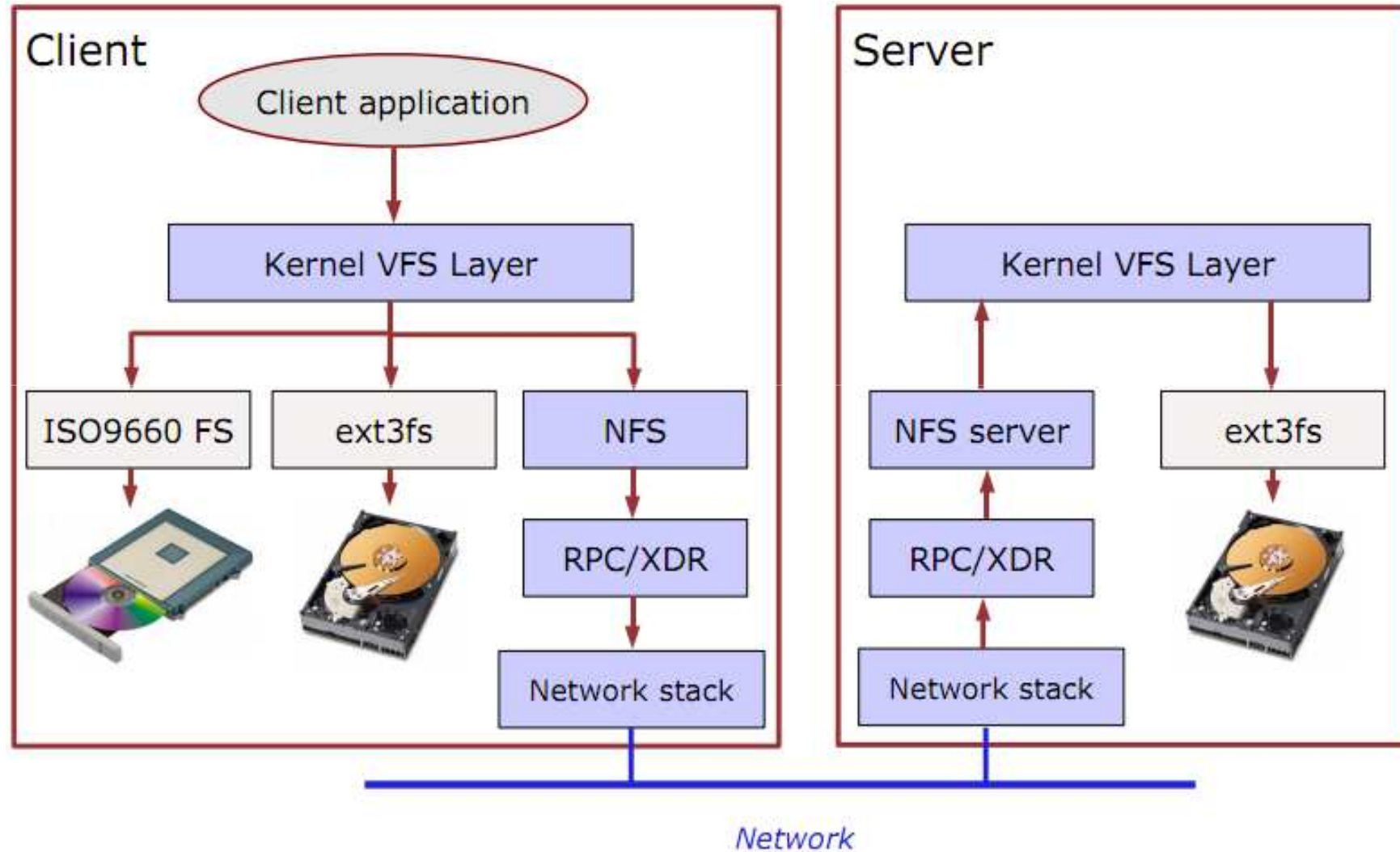


[NFS Overview]

- NFS was developed by Sun Microsystems in the mid-80s
 - Networked machines at the time were predominantly UNIX-based workstations
 - Various vendors: Sun, DEC, IBM, etc.
 - Different CPU architectures and OS implementations
 - But, all used UNIX filesystem structure and semantics
- NFS is based on Remote Procedure Call (RPC)
 - Allows a client machine to invoke a function on a server machine, over a network
 - Client sends a message with the function arguments
 - Server replies with a message with the return value.
- External Data Representation (XDR) to represent data types
 - Canonical network representation for ints, longs, byte arrays, etc.
 - Clients and servers must translate parameters and return values of RPC calls into XDR before shipping on the network
 - Otherwise, a little-endian machine and a big-endian machine would disagree on what is meant by the stream of bytes “fe 07 89 da” interpreted as an “int”



[NFS Design]



[Stateless Protocol]

- The NFS protocol is *stateless*
 - The server maintains no information about individual clients!
 - This means that NFS does not support any notion of “opening” or “closing” files
 - Each client simply issues read and write requests specifying the file, offset in the file, and the requested size
- Advantages:
 - Server doesn't need to keep track of open/close status of files
 - Server doesn't need to keep track of “file offset” for each client's open files
 - Clients do this themselves
 - Server doesn't have to do anything to recover from a crash!
 - Clients simply retry NFS operations until the server comes back up
- Disadvantages:
 - Server doesn't keep track of concurrent access to same file
 - Multiple clients might be modifying a file at the same time
 - NFS does not provide any consistency guarantees!!!
 - However, there is a separate *locking protocol* – discussed later



[NFS Protocol Overview]

- `mount()` returns filehandle for root of filesystem
 - Actually a separate protocol from NFS...
- `lookup(dir-handle, filename)` returns filehandle, attribs
 - Returns unique file handle for a given file
 - File handle used in subsequent read/write/etc. calls
- `create(dir-handle, filename, attributes)` returns filehandle
- `remove(dir-handle, filename)` returns status
- `getattr(filehandle)` returns attribs
 - Returns attributes of the file, e.g., permissions, owner, group ID, size, access time, last-modified time
- `setattr(filehandle, attribs)` returns attribs
- `read(filehandle, offset, size)` returns attribs, data
- `write(filehandle, offset, count, data)` returns attribs



[NFS Caching]

- NFS clients are responsible for caching recently-accessed data
 - Remember: the server is stateless!
- The NFS protocol does not *require* that clients cache data ...
 - But, it provides support allowing a range of client-side caching techniques
- This is accomplished through the `getattr()` call
 - Returns size, permissions, and last-modified time of file
 - This can tell a client whether a file has changed since it last read it
 - Read/write calls also return attributes so client can tell if object was modified since the last `getattr()` call
- How often should the client use `getattr()`?
 - Whenever the file is accessed?
 - Could lead to a lot of `getattr` calls!
 - Only if the file has not been accessed for some time?
 - e.g., If the file has not been accessed in 30 sec?
 - Different OSs implement this differently!



[NFS Locking]

- NFS does not prevent multiple clients from modifying a file simultaneously
 - Clearly, this can be a Bad Thing for some applications...
- Solution: Network Lock Manager (NLM) protocol
 - Works alongside NFS to provide file locking
 - NFS itself does not know anything about locks
 - Clients have to use NLM “voluntarily” to avoid stomping on each other
 - NLM has to be **stateful**
 - Why?

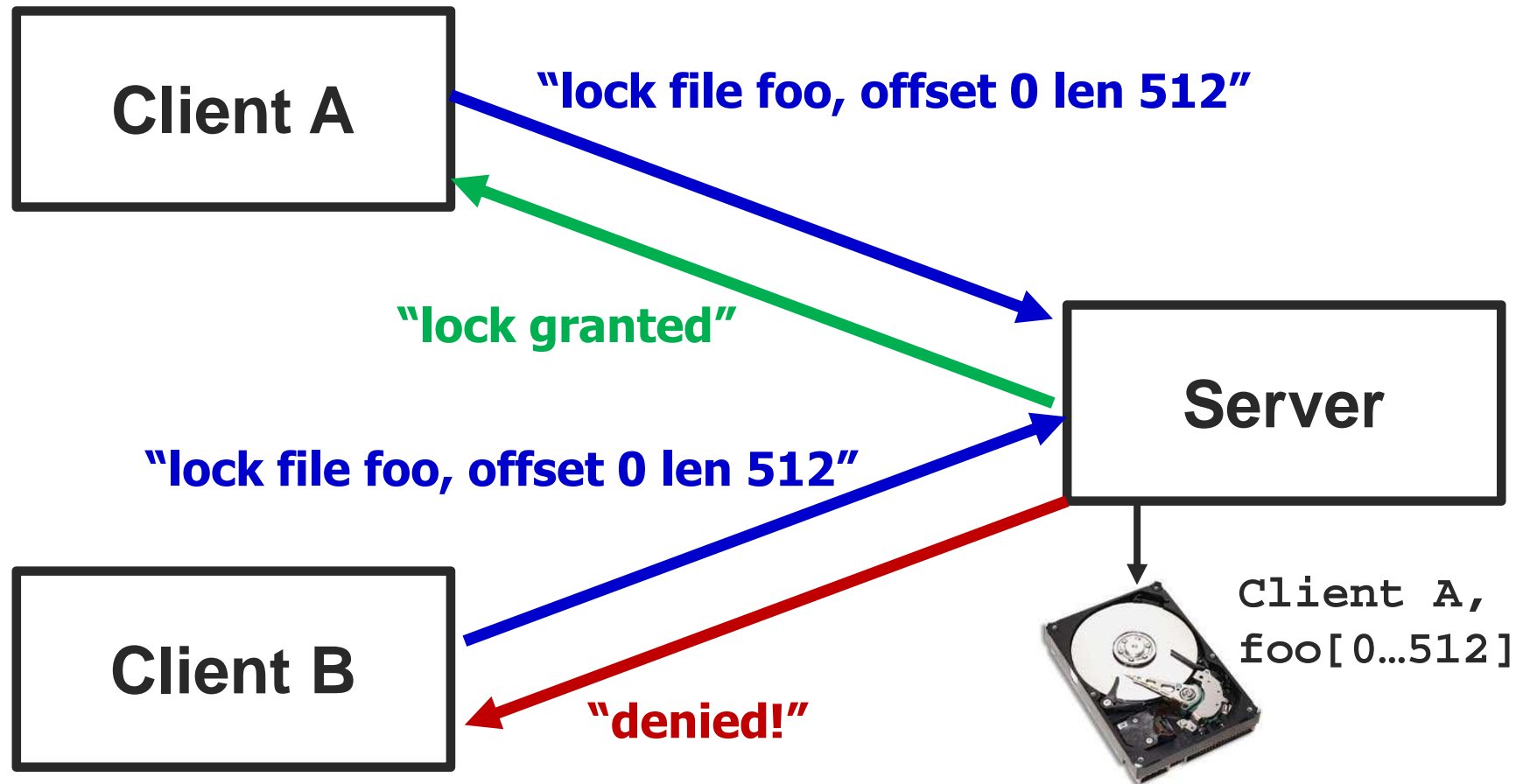


[NLM Protocol]

- NLM server has to keep track of locks held by clients
- If the NLM server crashes...
 - All locks are released!
 - BUT ... clients can reestablish locks during a “grace period” after the server recovers
 - No new locks are granted during the grace period
 - Server has to remember which locks were previously held by clients
- If an NLM client crashes...
 - The server is notified when the client recovers and releases all of its locks
 - What happens if a client crashes and does not come back up for a while?
- Servers and clients must be notified when they crash and recover
 - This is done with the simple “Network Status Monitor” (NSM) protocol
 - Essentially, send a notification to the other host when you reboot



[NLM Example]



[NLM Example]

Client A

Client B

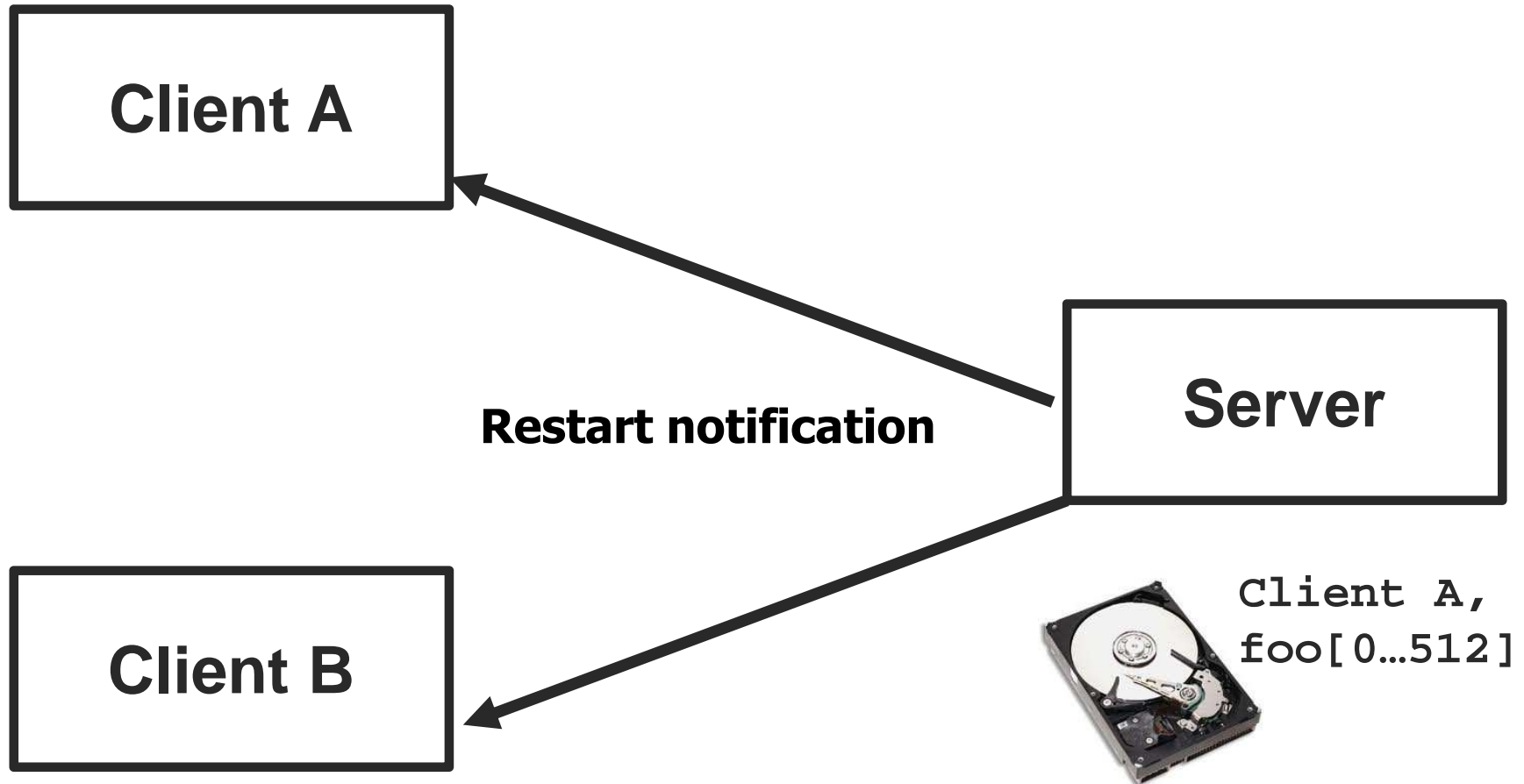


er

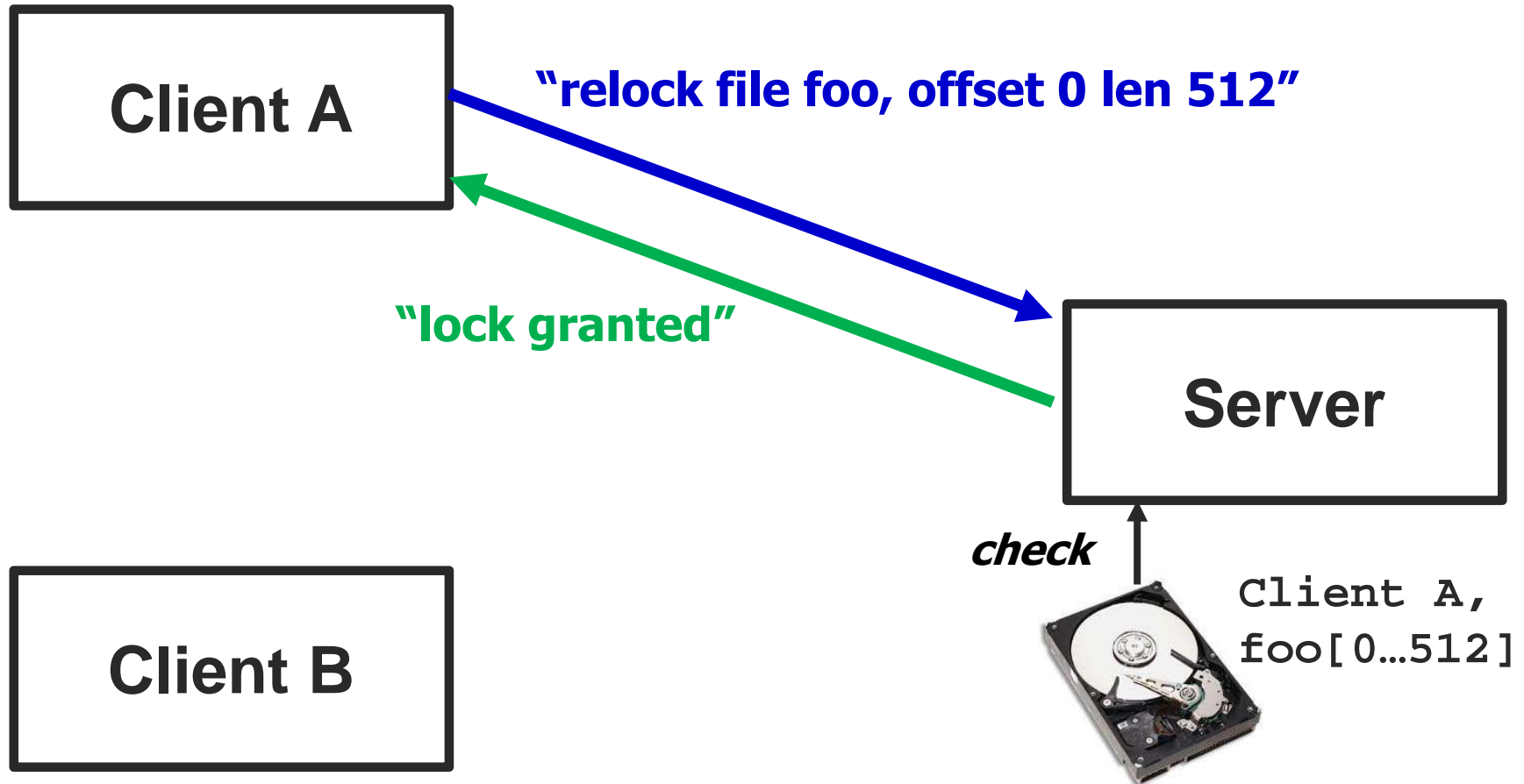


Client A,
foo[0...512]

[NLM Example]

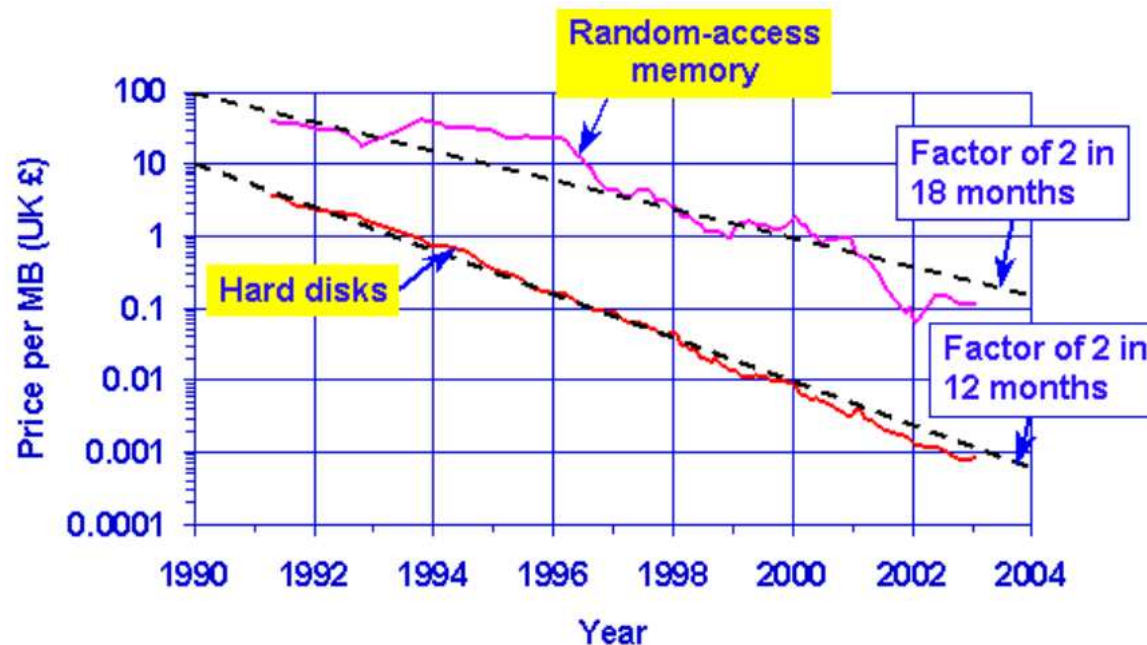


[NLM Example]



[RAID Motivation]

- Speed of disks not matching other components
 - Moore's law: CPU speed doubles every 18 months
 - SRAM speeds increasing by 40-100% a year
 - In contrast, disk seek time only improving 7% a year
 - Although greater density leads to improved transfer times once seek is done
- Emergence of PCs starting to drive down costs of disks
 - (This is 1988 after all)
 - PC-class disks were smaller, cheaper, and only marginally slower



[RAID Motivation]

- Basic idea: Build I/O systems as arrays of cheap disks
 - Allow data to be **striped** across multiple disks
 - Means you can read/write multiple disks in parallel – greatly improve performance
- Problem: disks are extremely unreliable
- Mean Time to Failure (MTTF)
 - $\text{MTTF (disk array)} = \text{MTTF (single disk)} / \# \text{ disks}$
 - Adding more disks means that failures happen more frequently..
 - **An array of 100 disks with an MTTF of 30,000 hours = just under 2 weeks!**



[Increasing reliability]

- Idea: Replicate data across multiple disks
 - When a disk fails, lost information can be regenerated from the redundant data
- Simplest form: Mirroring (also called “RAID 1”)
 - All data is mirrored across two disks
- Advantages:
 - Reads are faster, since both disks can be read in parallel
 - Higher reliability (of course)
- Disadvantages:
 - Writes are slightly slower, since OS must wait for both disks to do write
 - This approach also doubles the cost of the storage system!

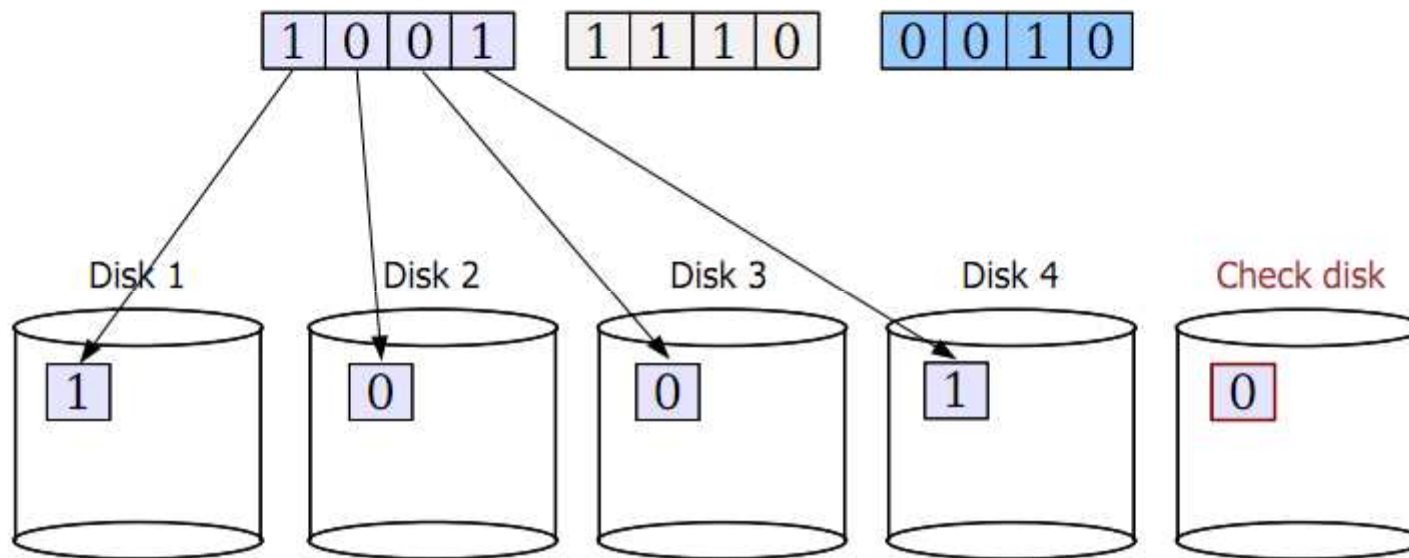


[RAID 3]

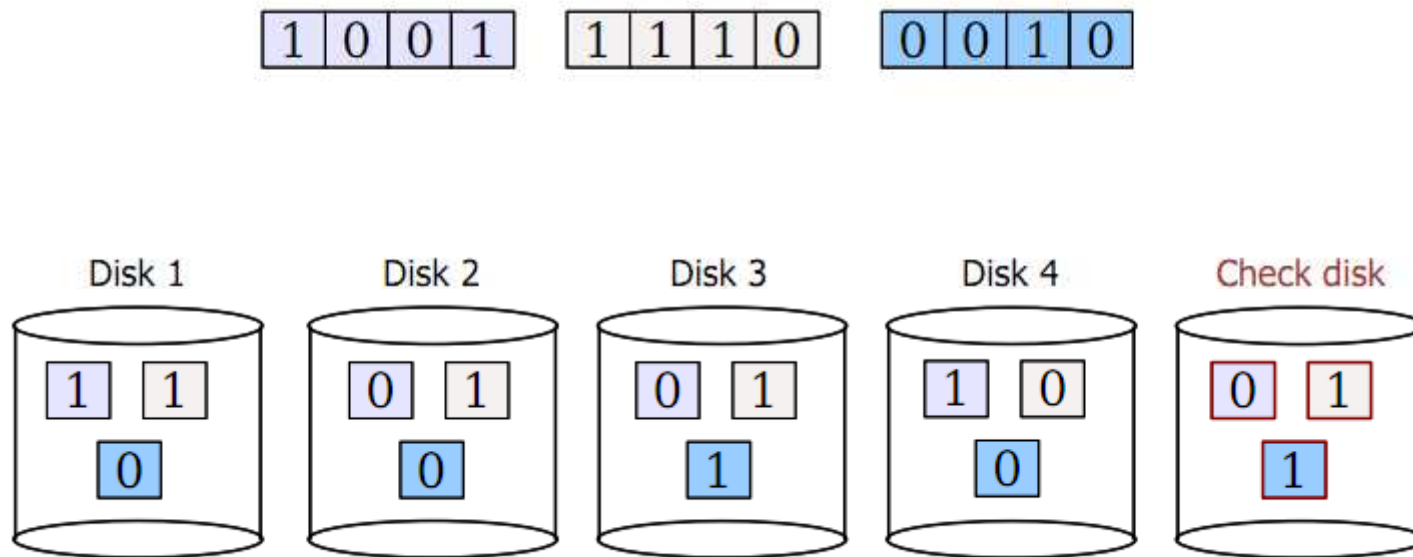
- Rather than mirroring, use **parity codes**
 - Given N bits $\{b_1, b_2, \dots, b_N\}$, the **parity bit** P is the bit $\{0,1\}$ that yields an even number of “1” bits in the set $\{b_1, b_2, \dots, b_N, P\}$
 - Idea: If any bit in $\{b_1, b_2, \dots, b_N\}$ is lost, can use the remaining bits (plus P) to recover it.
- Where to store the parity codes?
 - Add an extra “check disk” that stores parity bits for the data stored on the rest of the N
- disks
- Advantages:
 - If a single disk fails, can easily recompute the lost data from the parity code
 - Can use one parity disk for **several** data disks (reduces cost)
- Disadvantages:
 - Each write to a block must update the corresponding parity block as well



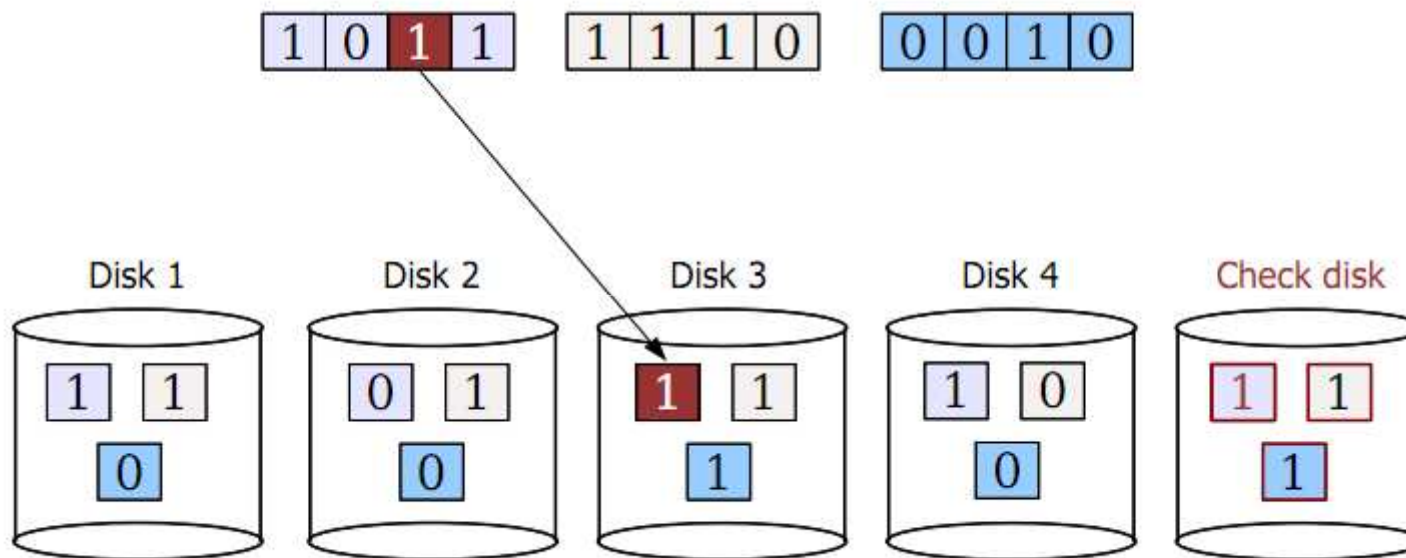
[RAID 3 example]



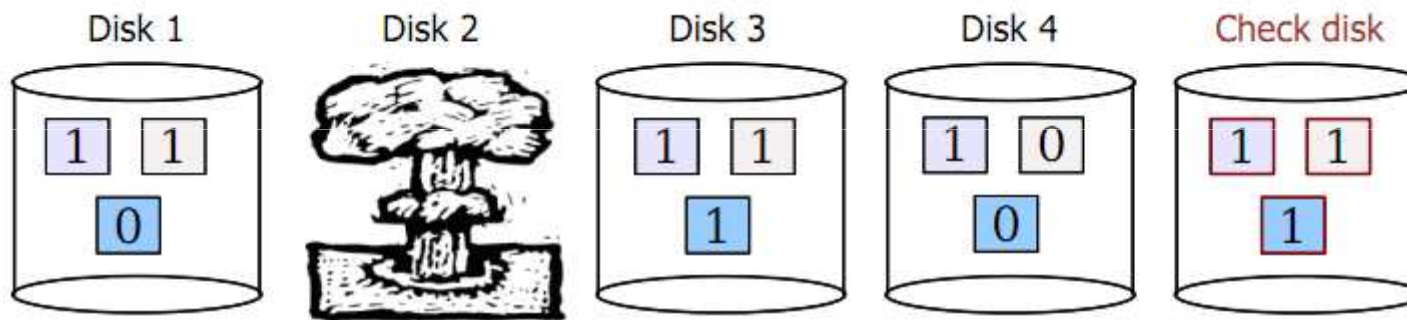
[RAID 3 example]



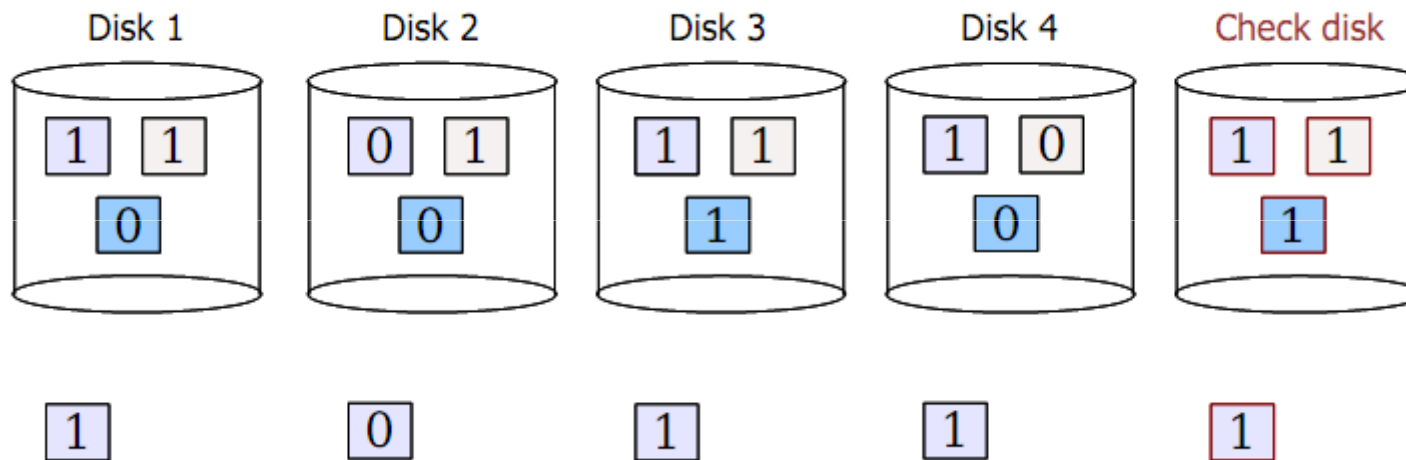
[RAID 3 example]



[RAID 3 example]



[RAID 3 example]



- 1. Read back data from other disks
- 2. Recalculate lost data from parity code
- 3. Rebuild data on lost disk



[RAID 3 issues]

- What is the MTTF of RAID?
 - Both RAID 1 and RAID 3 tolerate the failure of a single disk
 - As long as a second disk does not die while we are repairing the first failure, we are in good shape!
- So, what is the probability of a second disk failure?
- $P(\text{2nd failure}) = \text{MTTR} / (\text{MTTF of one disk} / \# \text{ disks} - 1)$
 - This can be derived from independent and exponential failure rates
 - See Patterson RAID paper for details
 - 10 disks, MTTF (disk) = 1000 days, MTTR = 1 day
 - $P(\text{2nd failure}) = 1 \text{ day} / (1000 / 9) = 0.009$
- What is the performance of RAID 3?
 - Well, the check disk must be updated each time there is a write
 - Problem: The check disk is then a performance bottleneck
 - Only a single read/write can be done at once on the whole system!



[RAID 5]

- Another approach: Interleaved check blocks (“RAID 5”)
 - Rotate the assignment of data blocks and check blocks across disks
 - Avoids the bottleneck of a single disk for storing check data
 - Allows multiple reads/writes to occur in parallel (since different disks affected)

