




I/O and Filesystems

Based on slides by Matt Welsh, Harvard

[Announcements]

- Post your web server URL on piazza!
- Exam review postponed till special session after last class
- Research talk today: Darko Kirovski, Microsoft Research
“Making optical media impossible to counterfeit”
2405 SC, 4:00 p.m.

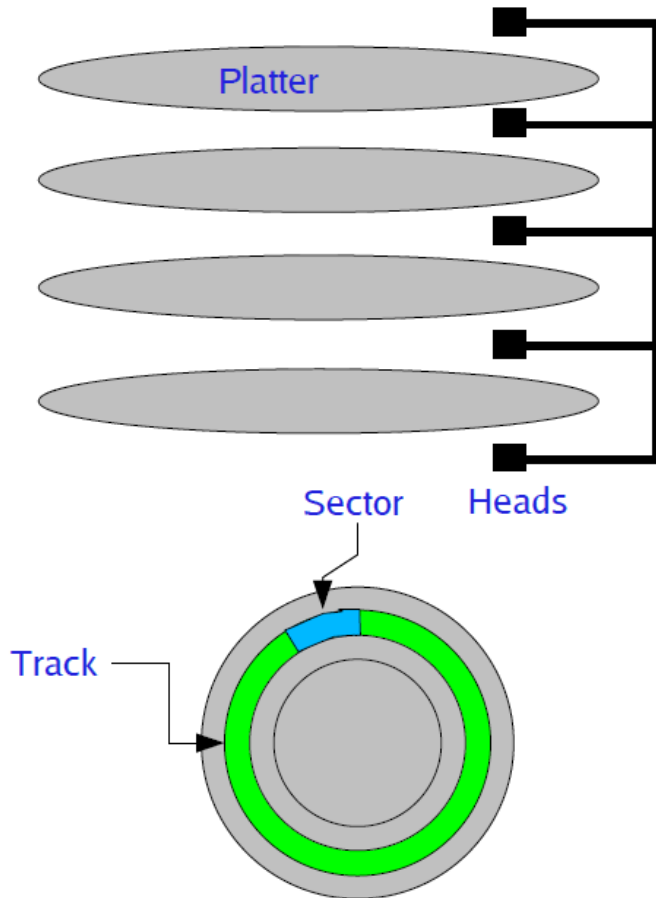




Part 1: Disks

A Disk Primer (Review)

- 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**



[Disks: messy & slow (Review)]

- 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



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]

Current track



Direction →



[SCAN example]

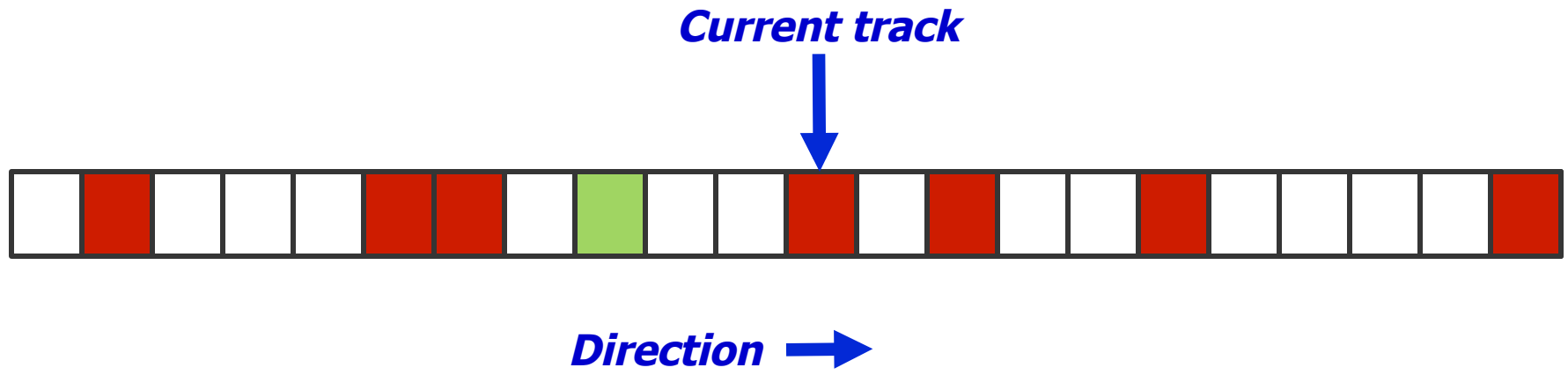
Current track



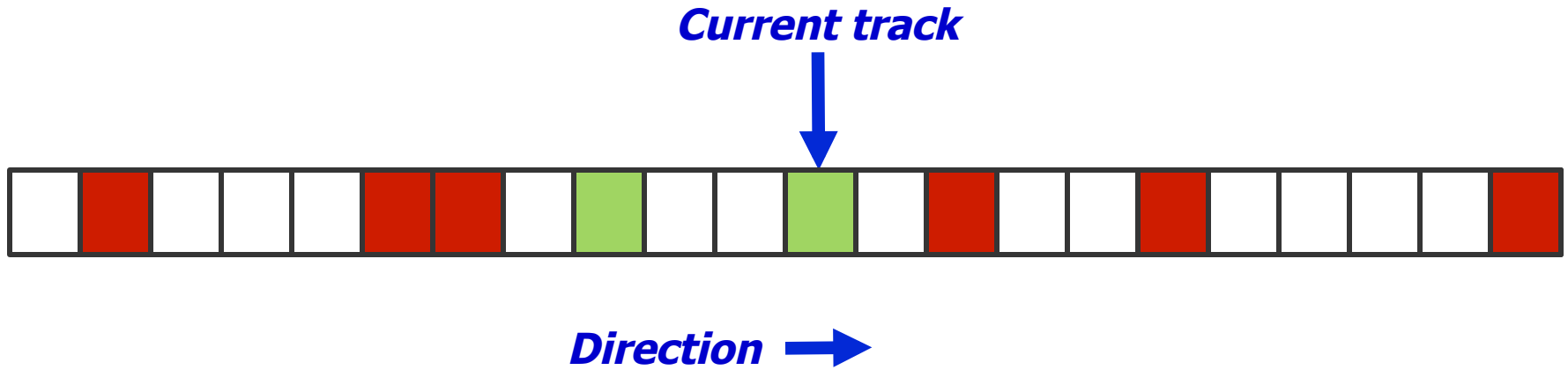
Direction →



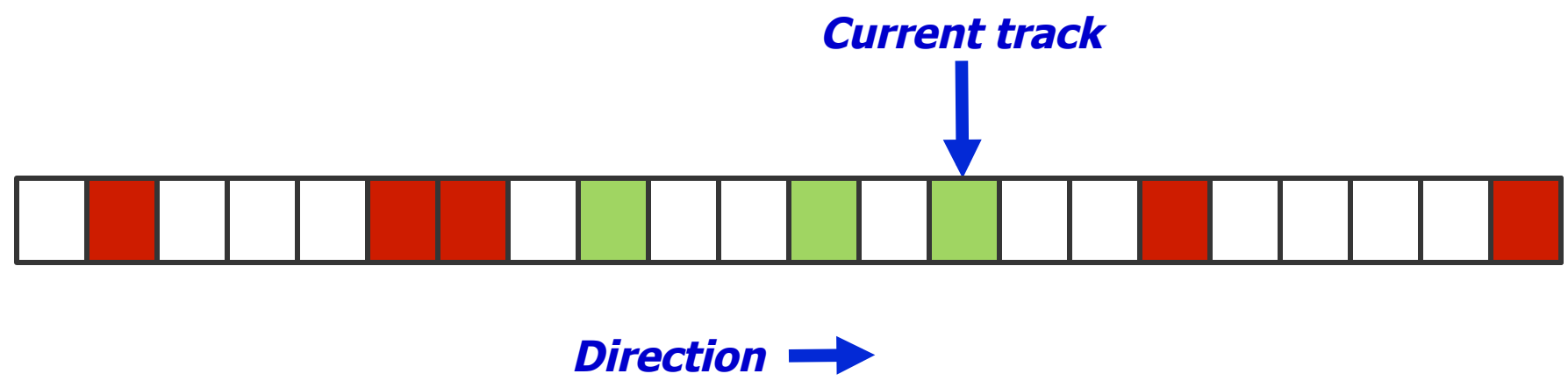
[SCAN example]



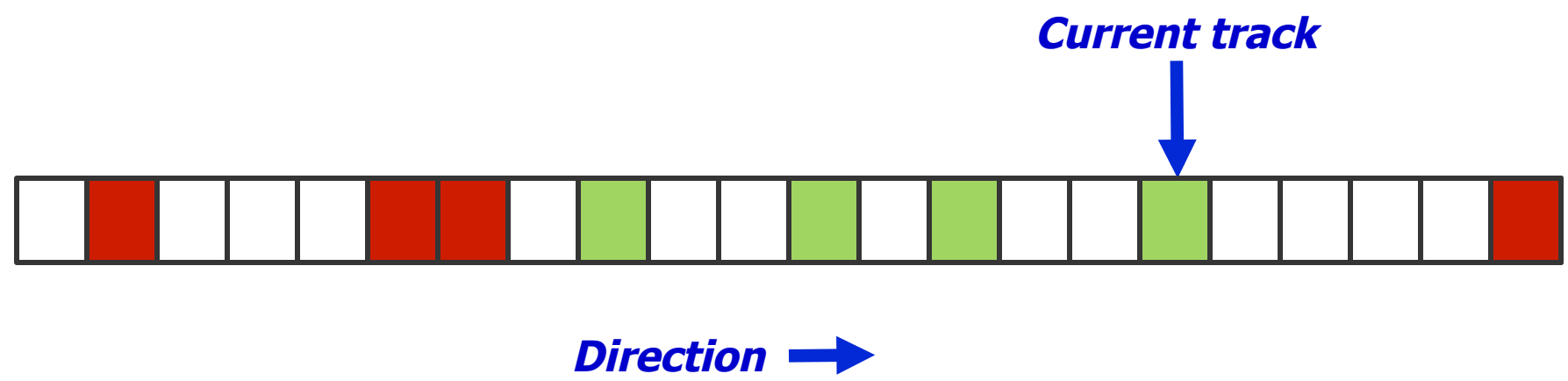
[SCAN example]



[SCAN example]



[SCAN example]



[SCAN example]

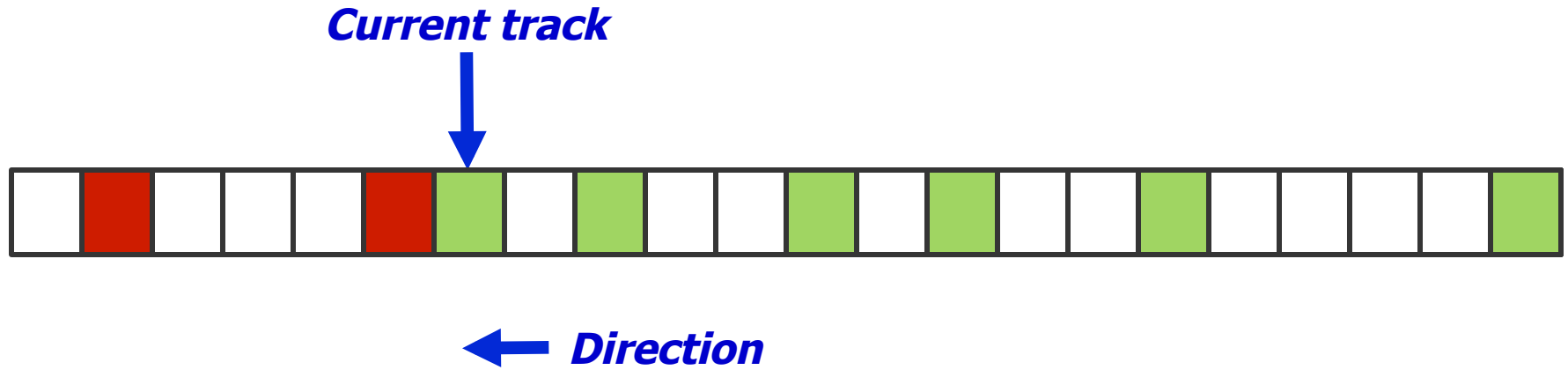
Current track



← Direction



[SCAN example]



[SCAN example]

Current track



← Direction



[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?

- **Device 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

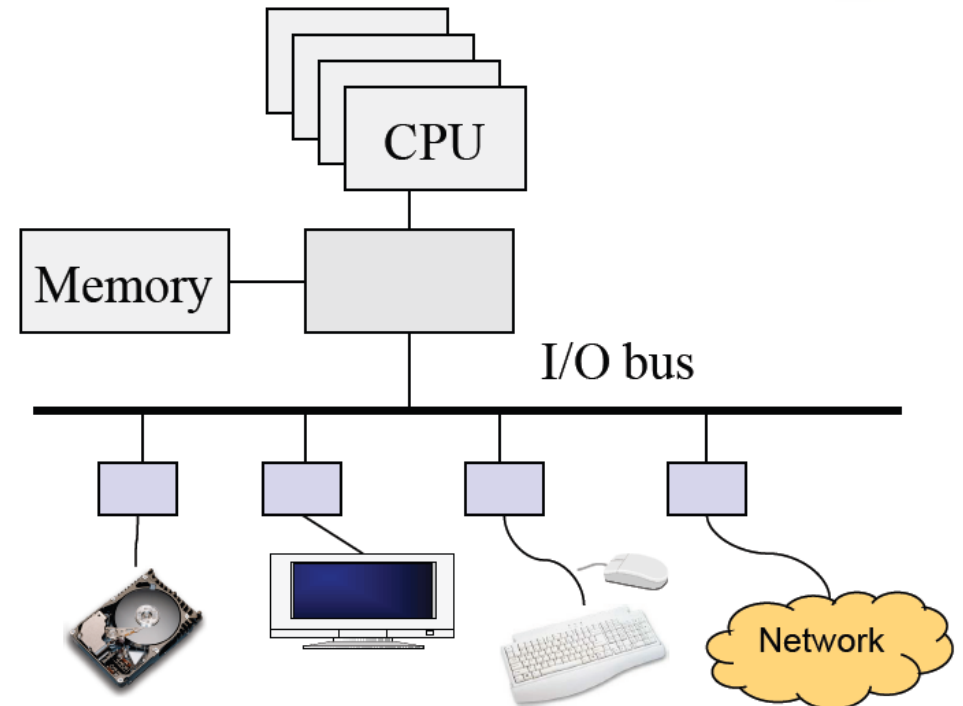
- CPU and caches
- Chipset
- Memory

- I/O Hardware

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

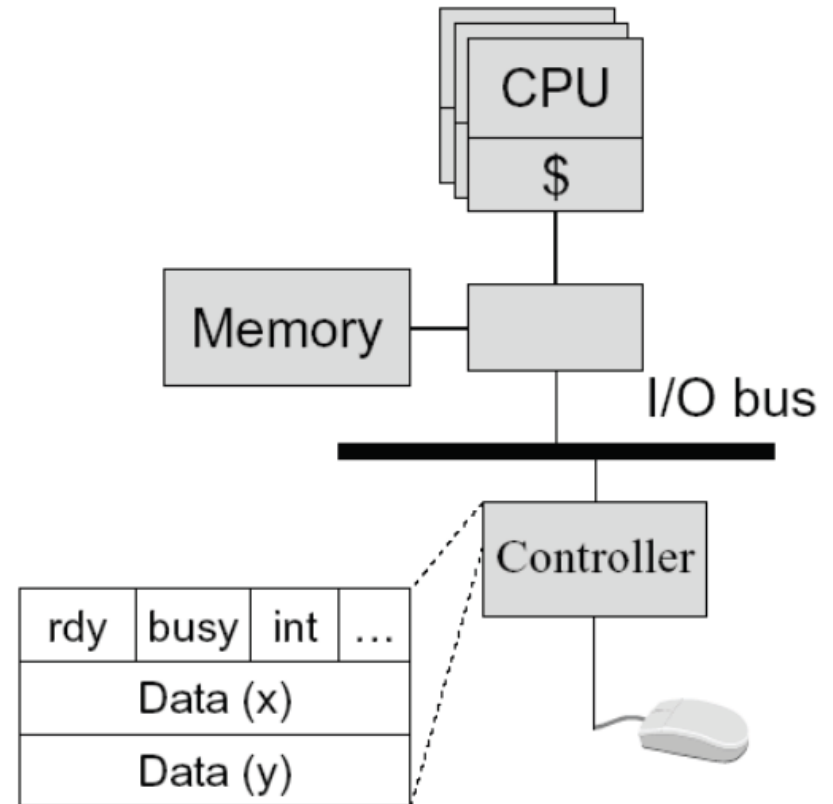
- Two types of I/O

- **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



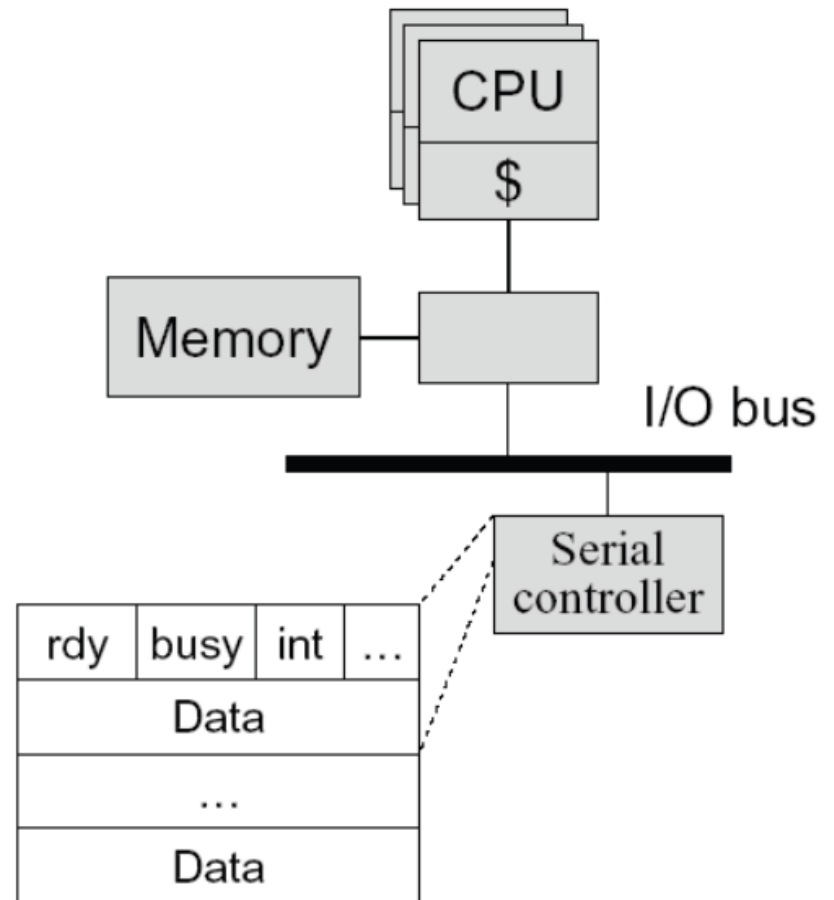
Programmed Input Device

- 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
 - When moved, put (X, Y) in mouse's device controller's data registers
 - Interrupt CPU
- 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



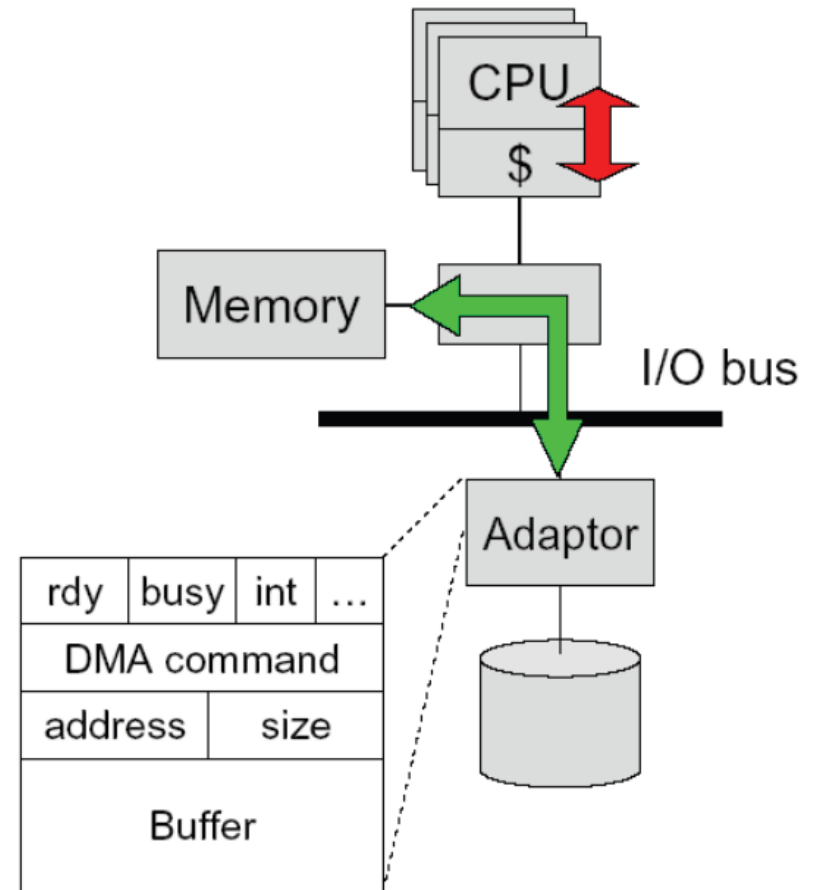
Programmed Output Device

- 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

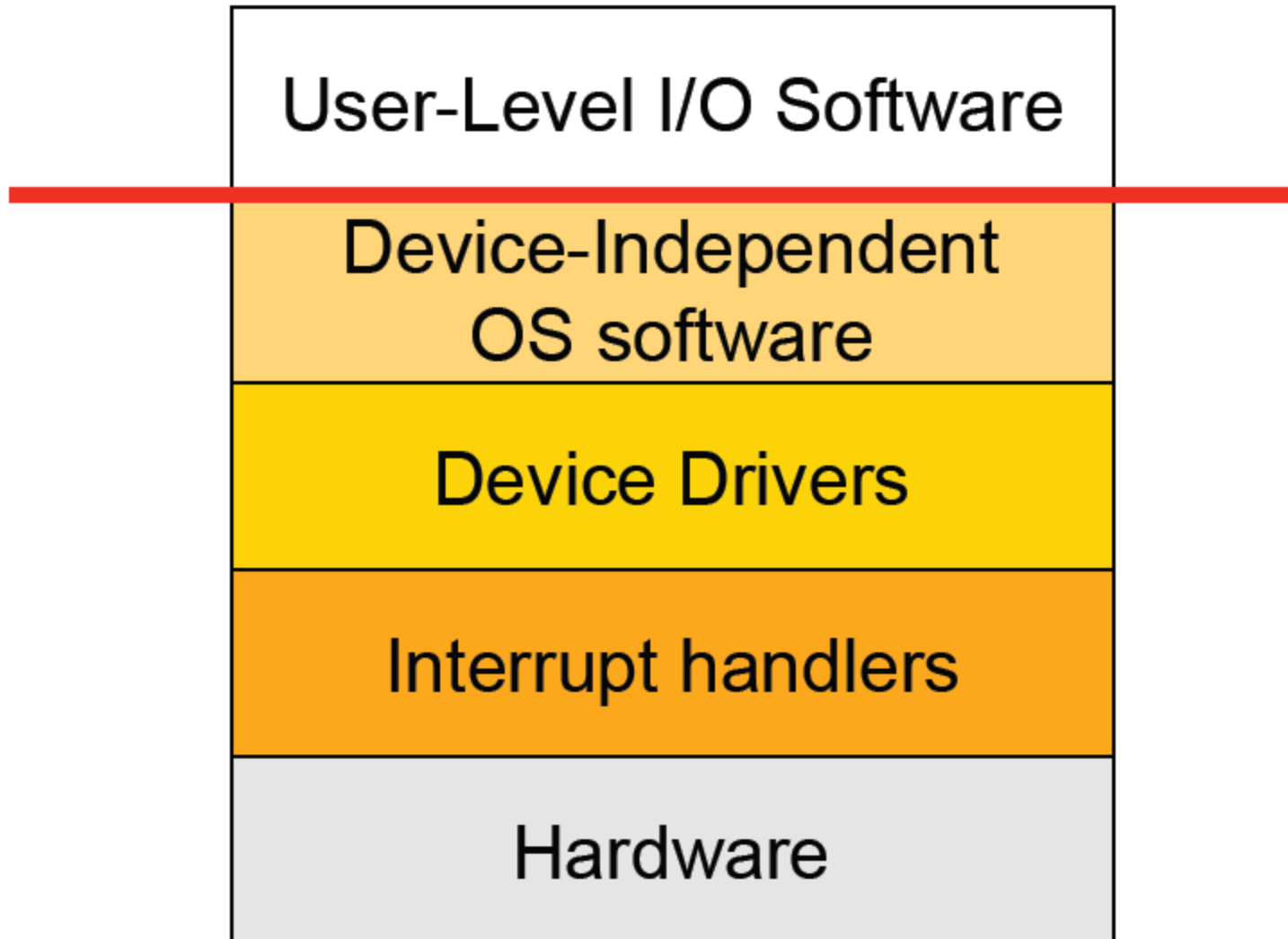


How Interrupts are implemented

- 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]

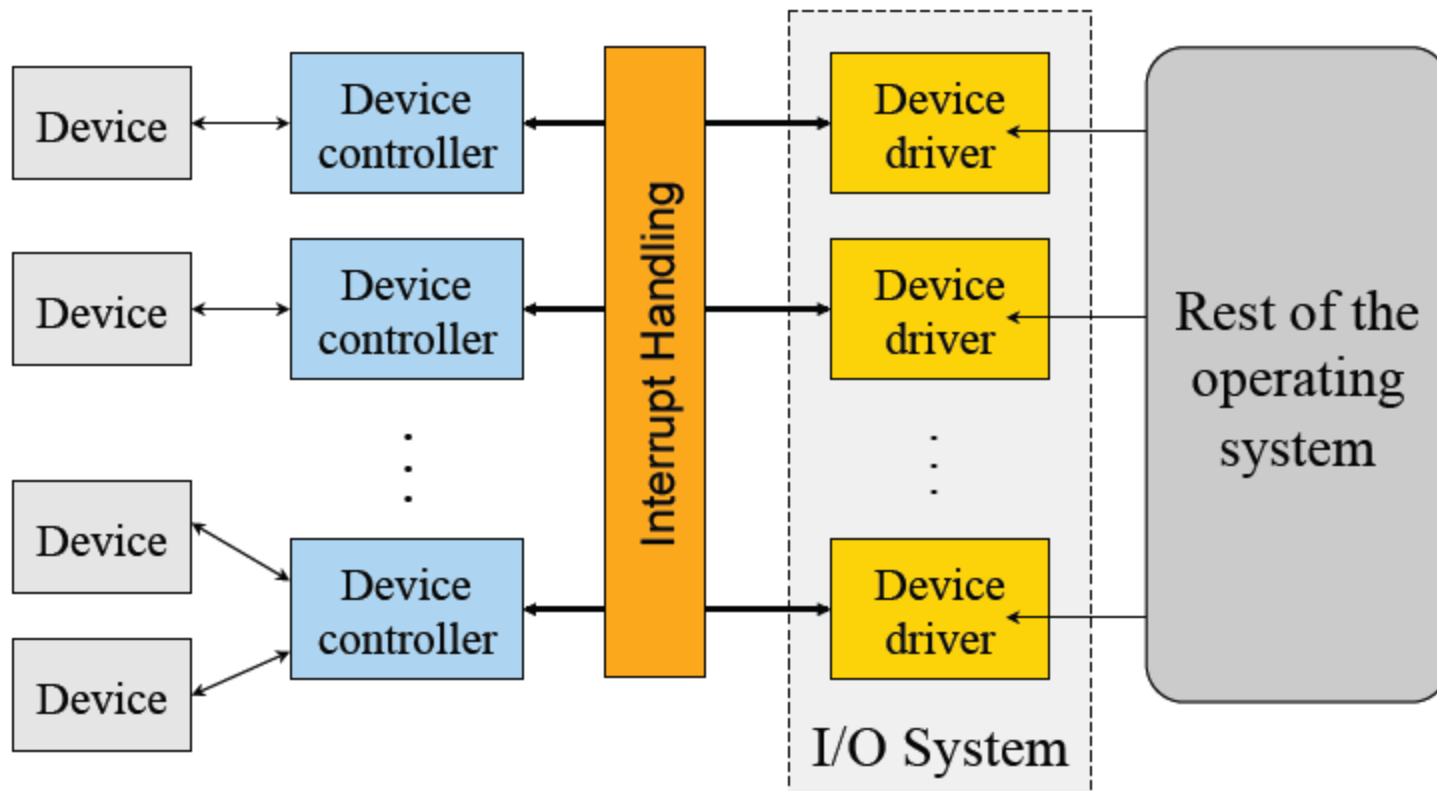


[Interrupt Handling]

- 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



[User-level interfaces: syscalls]

■ 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
 - returns -1, sets error code EAGAIN or EWOULDBLOCK
 - user process can do other things before I/O completion
 - can determine if device is ready with select() / poll()
 - Make asynchronous with O_NONBLOCK option on open() or later via fcntl()



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



[Does I/O overhead matter?]

- Many steps involved in transmitting data
- How much can this overhead slow us down?

Experiment: copy.c





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



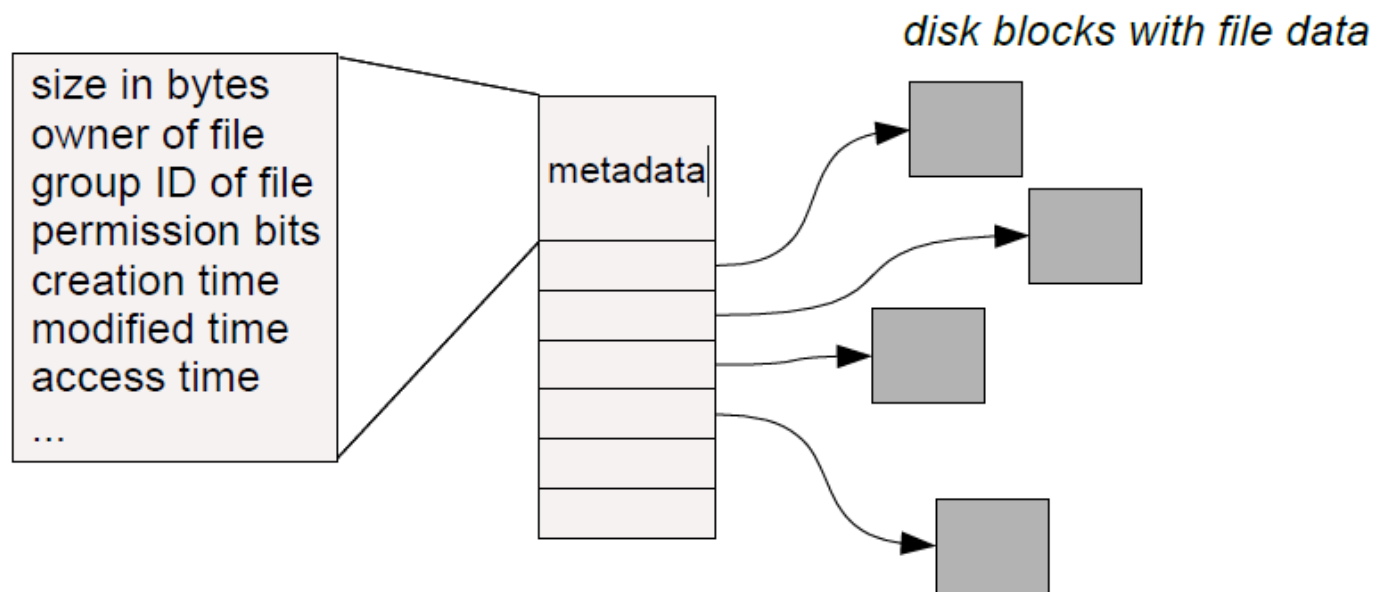
Filesystem Operations

- 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?



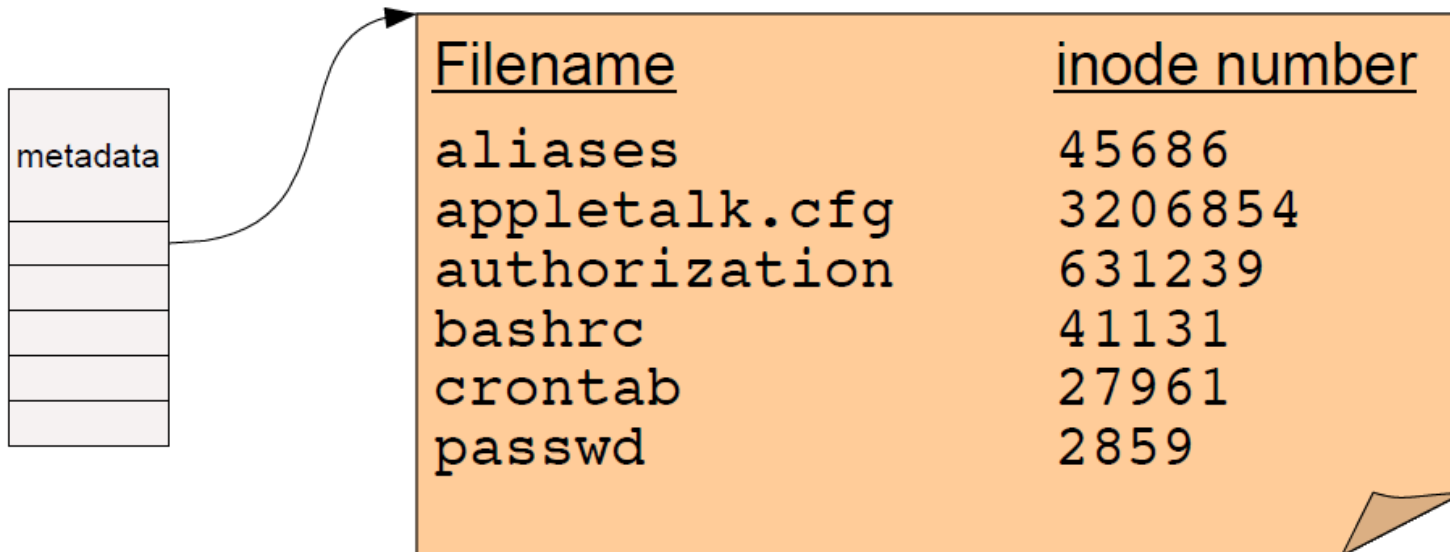
Basic Filesystem Structures

- 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

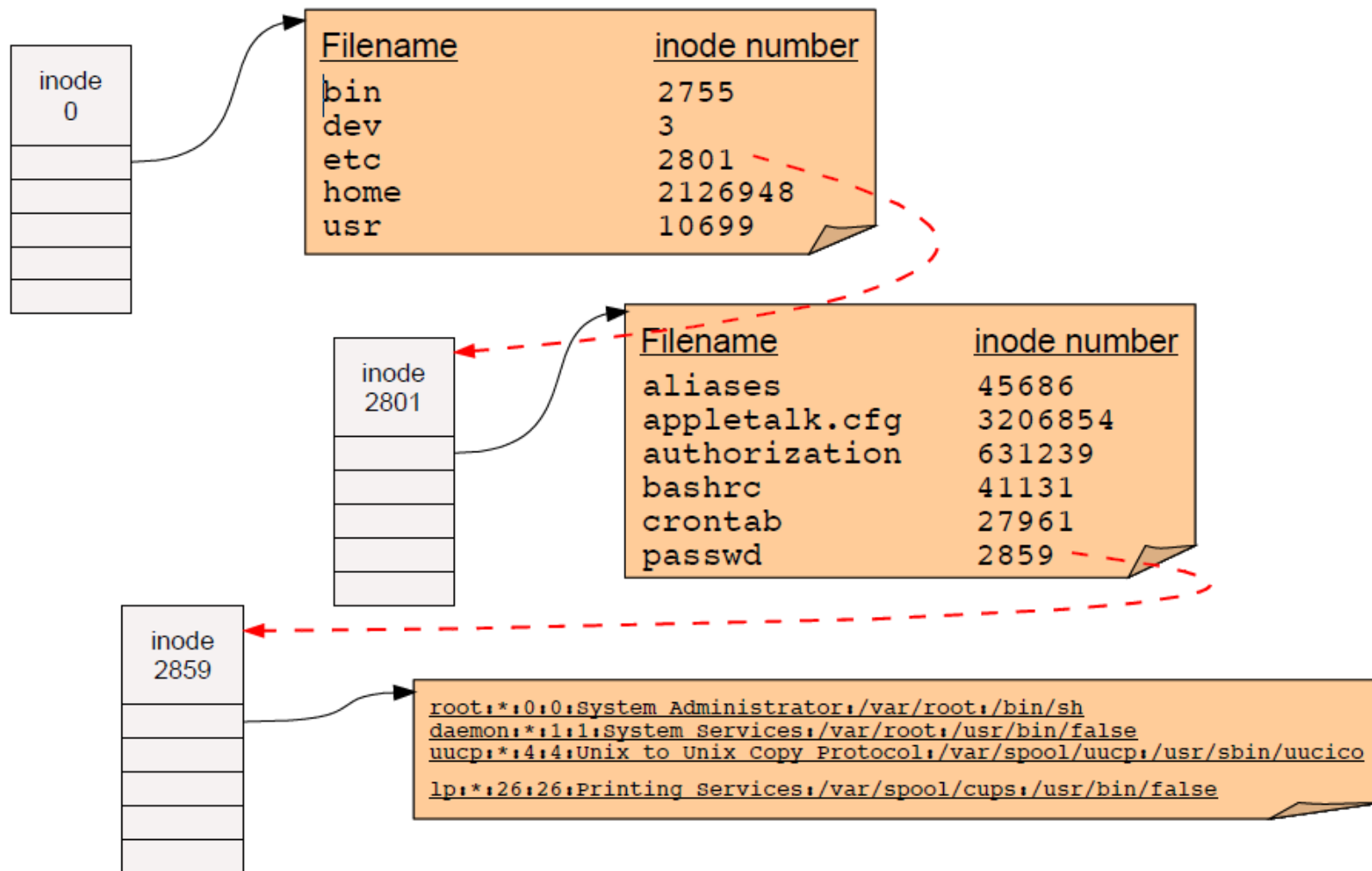


- These are the contents of the directory “file data” itself – NOT the directory's inode!
- Filenames (in UNIX) are not stored in the inode at all!
- Two open questions:
 - How do we find the root directory (“ / “ on UNIX systems)?
 - How do we get from an inode number to the location of the inode on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



superblock

inodes

File and directory data blocks

- 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”

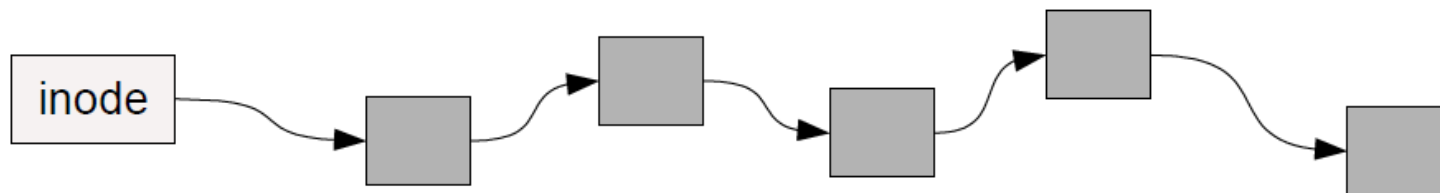
```
bash$ ls -i /home/foo
287663 /home/foo          (This is the inode number of “foo”)
bash$ ln /home/foo /tmp/foo
bash$ ls -i /home/foo /tmp/foo
287663 /home/foo
287663 /tmp/foo
```

- “/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.

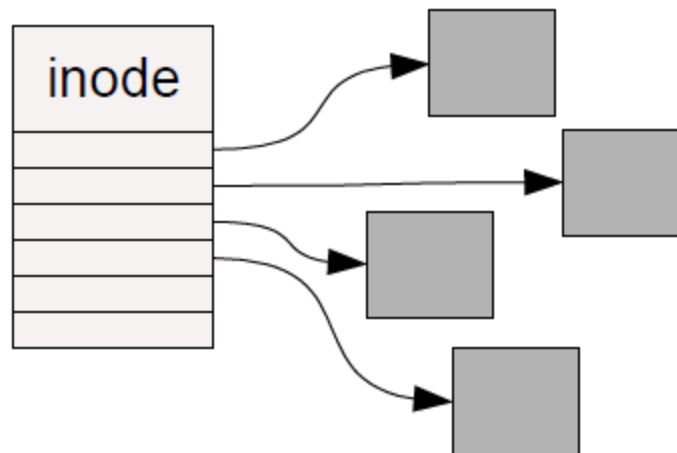


How should we organize blocks on a disk?

- 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??

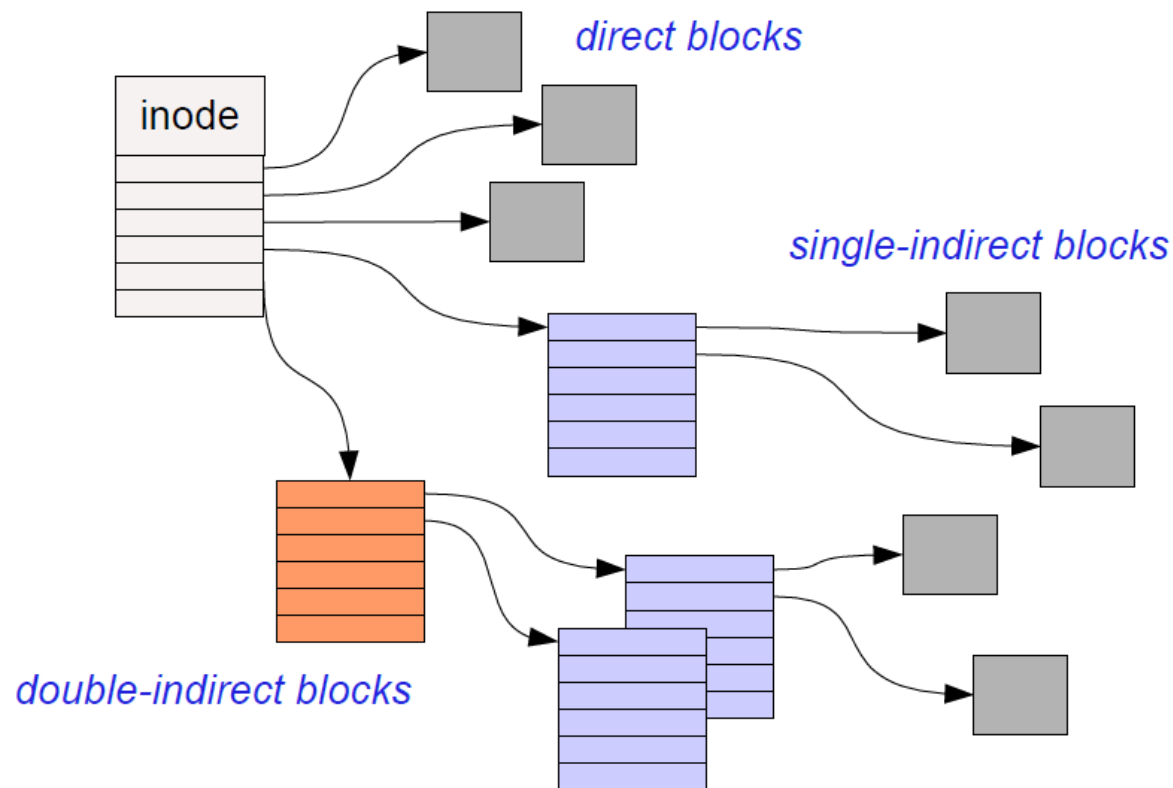


- Indexed files
 - inode contains a list of block numbers containing the file
 - Array is allocated when the file is created
 - What are the advantages and disadvantages??



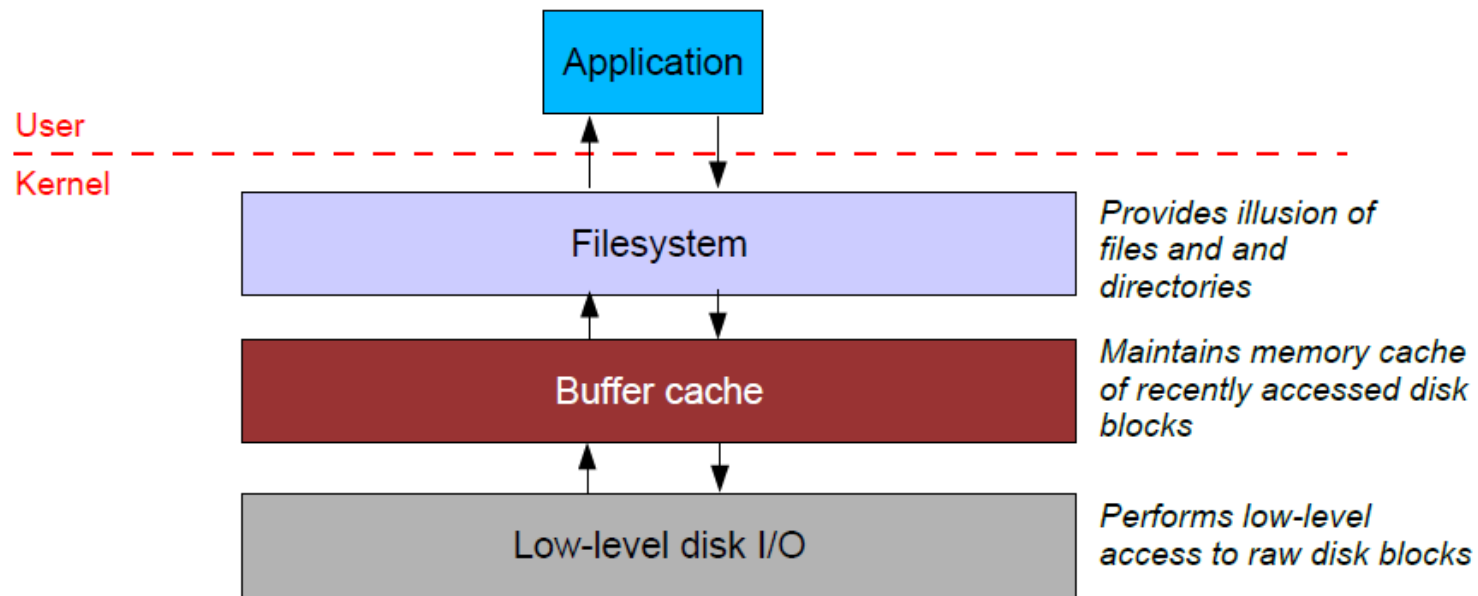
Multilevel indexed files

- 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!!!



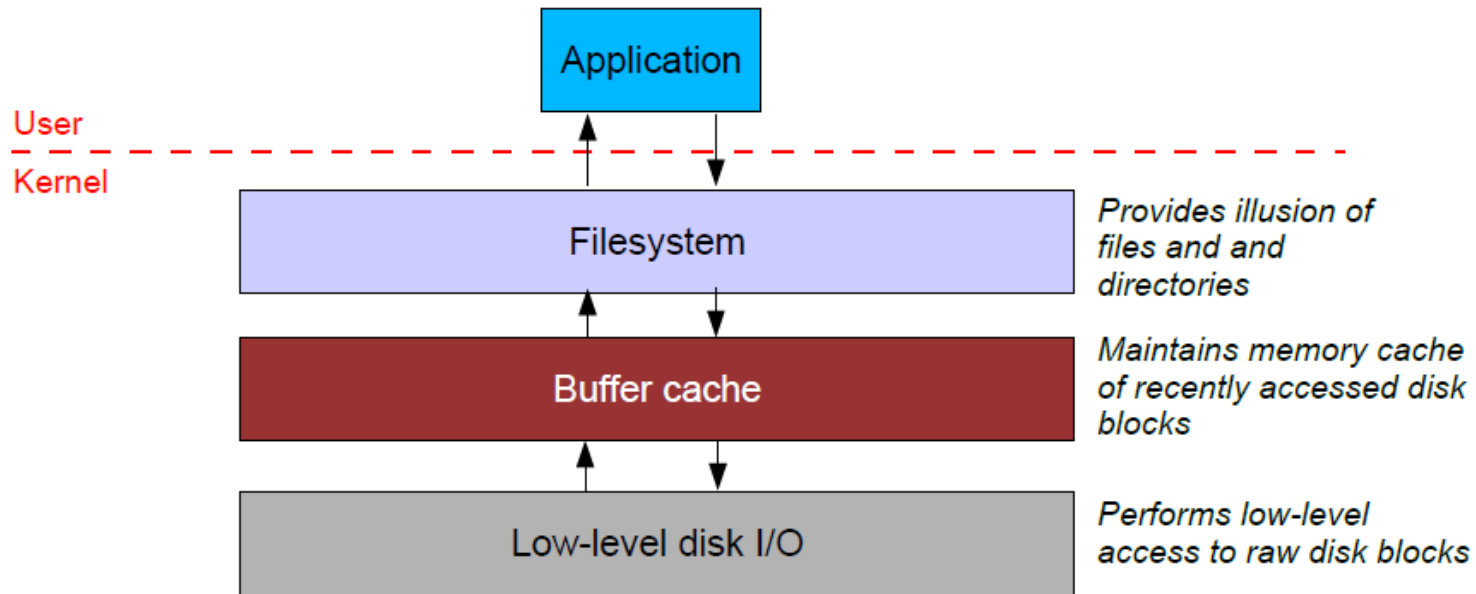
File system caching

- 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



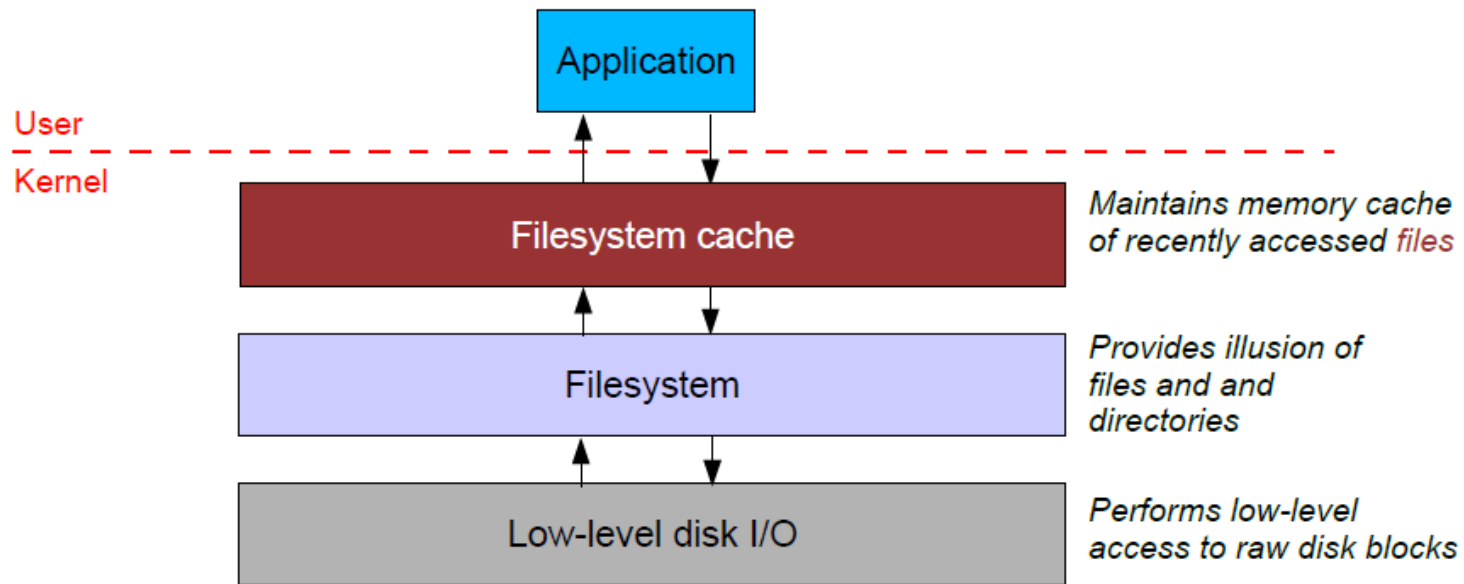
Caching issues

- 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??



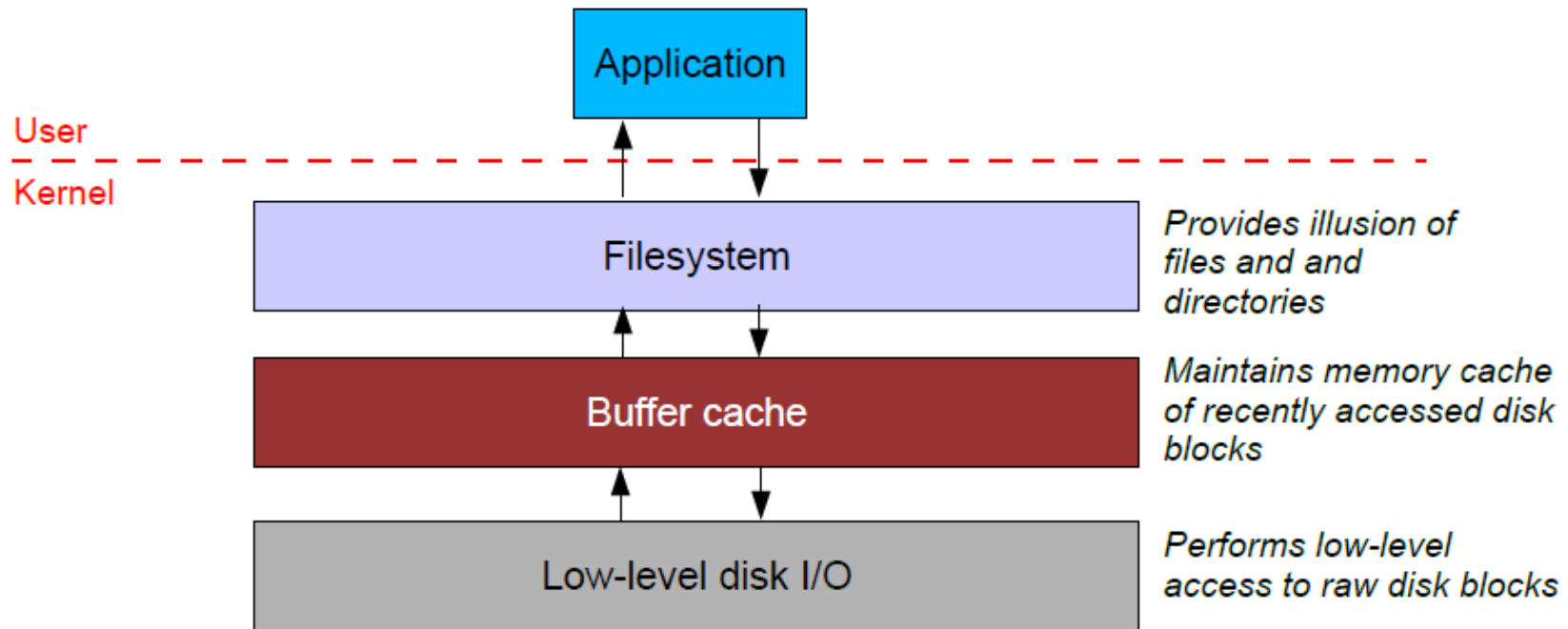
Caching issues

- 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

