I/O Systems

I/O, I/O, use s-t-d-i-o …

— apologies to *Snow White*

Input/Output

- Principles of I/O hardware
- Principles of I/O software
- I/O software layers
- Disks
- Clocks
- Character-oriented terminals
- Graphical user interfaces
- Network terminals
- Power management

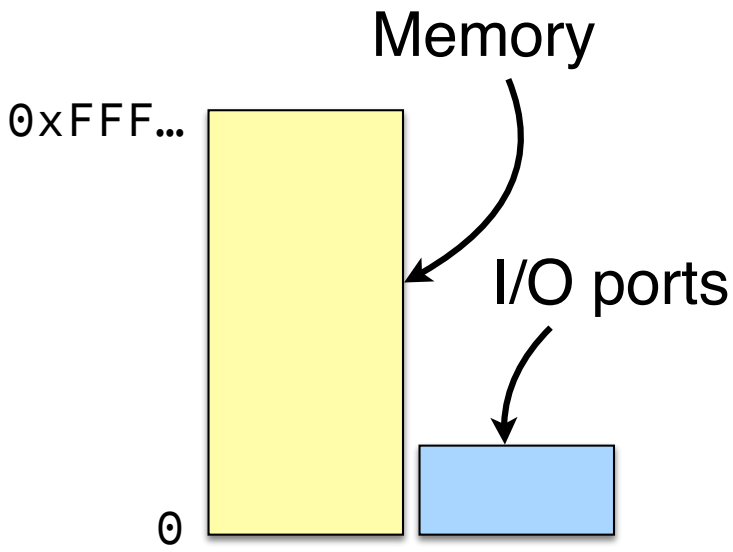
How fast is I/O hardware?

Device	Data rate (typical)
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Printer / scanner	1 MB/sec
USB	1.5 / 60 / 625 MB/sec
Digital camcorder	4 MB/sec
Gigabit Ethernet	125 MB/sec
Hard drive	125 MB/sec
Flash drive (SSD)	200 MB/sec
DisplayPort	1.62 / 2.7 / 5.4 / 8.1 Gbit/s per lane
PCIe bus	985 MB/sec per lane

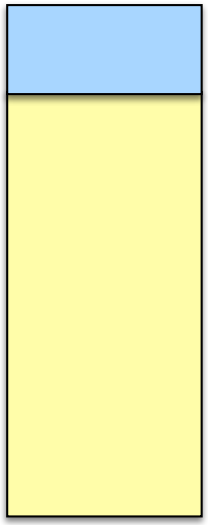
Device controllers

- I/O devices have components
  - Mechanical component
  - Electronic component
- Electronic component controls the device
  - May be able to handle multiple devices
  - May be more than one controller per mechanical component (example: hard drive)
- Controller's tasks
  - Convert serial bit stream to block of bytes
  - Perform error correction as necessary
  - Make available to main memory

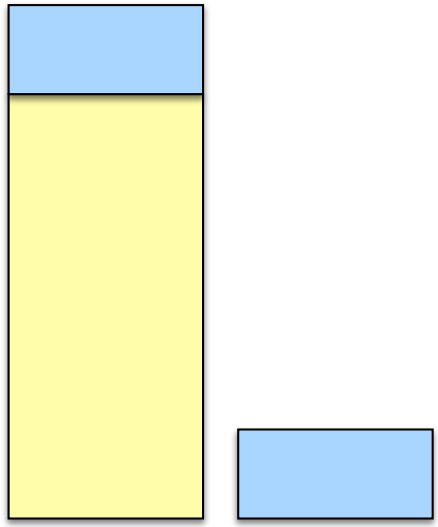
Memory-mapped I/O



Separate I/O & memory space



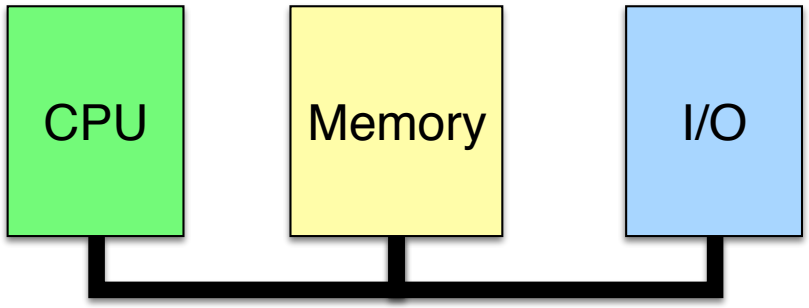
Memory-mapped I/O

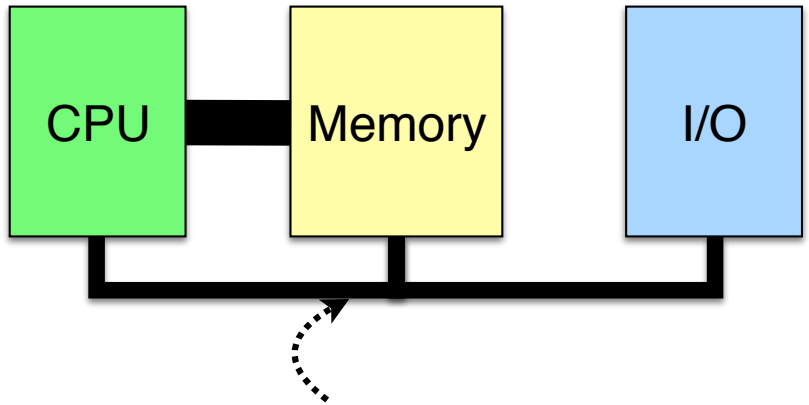


Hybrid: both memory-mapped & separate spaces

How is memory-mapped I/O done?

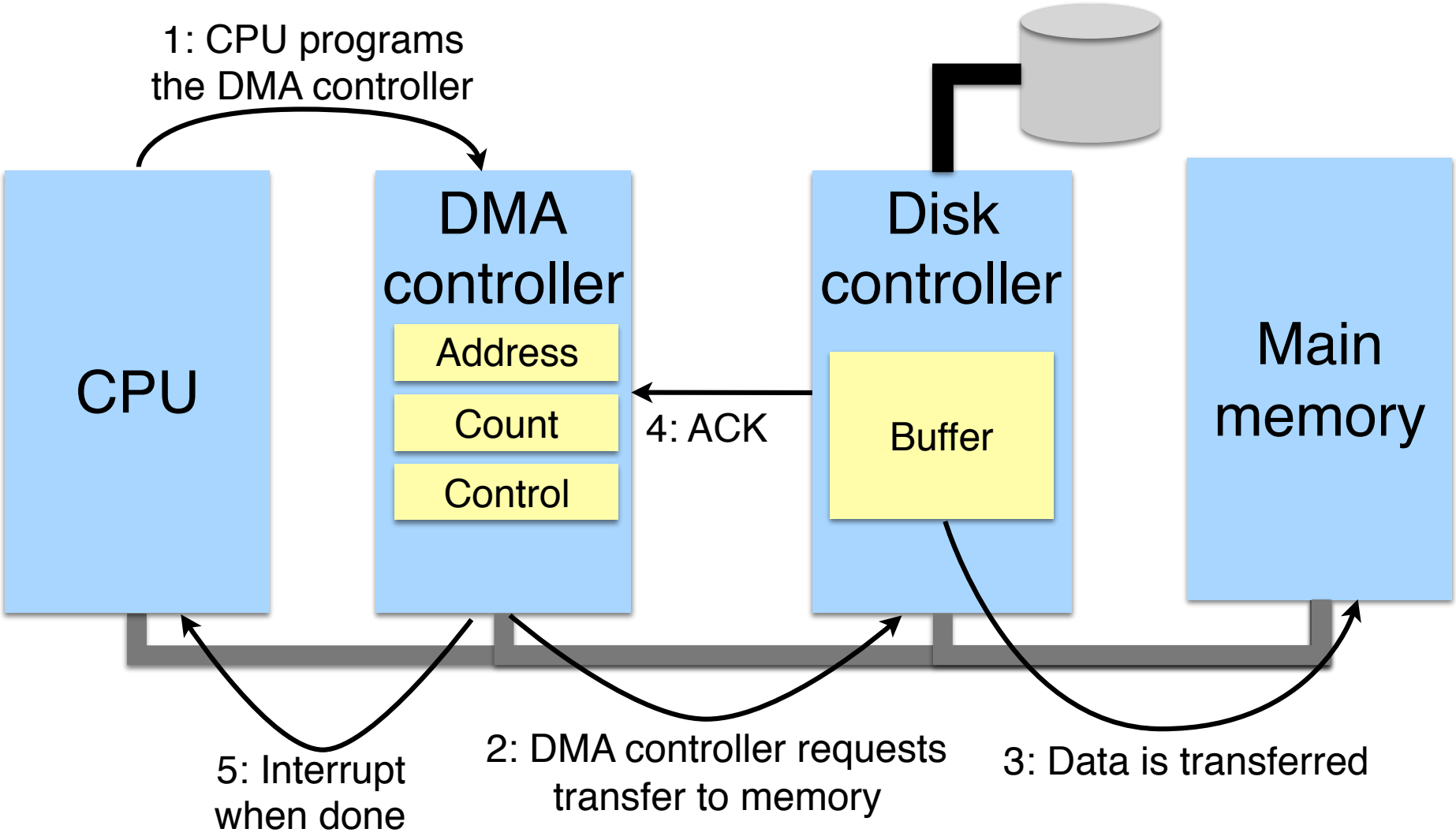
- Single-bus
  - All memory accesses go over a shared bus
  - I/O and RAM accesses compete for bandwidth
- Dual-bus
  - RAM access over high-speed bus
  - I/O access over lower-speed bus
  - Less competition for bus
  - More hardware (more expensive…)



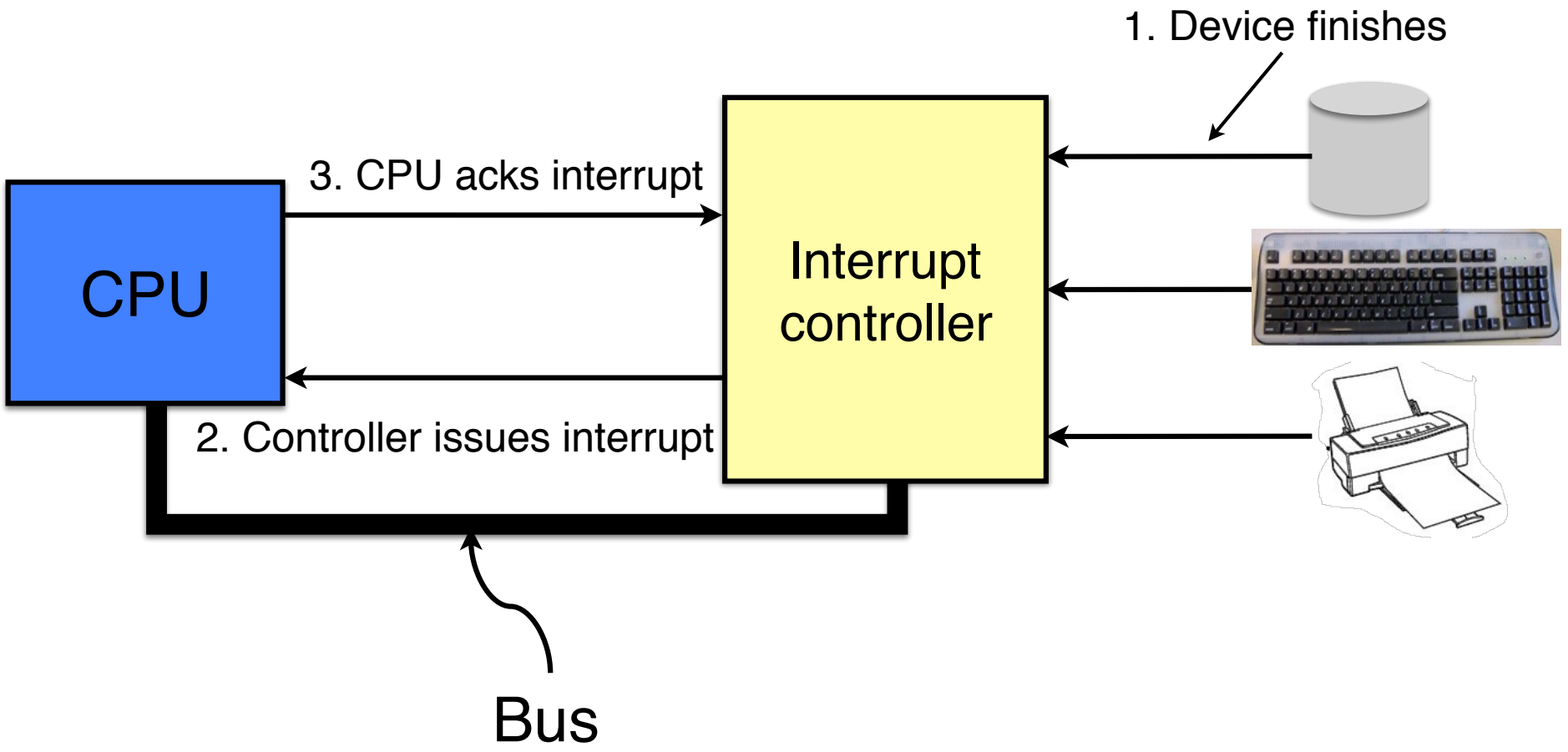


This port allows I/O devices access into memory

Direct Memory Access (DMA)



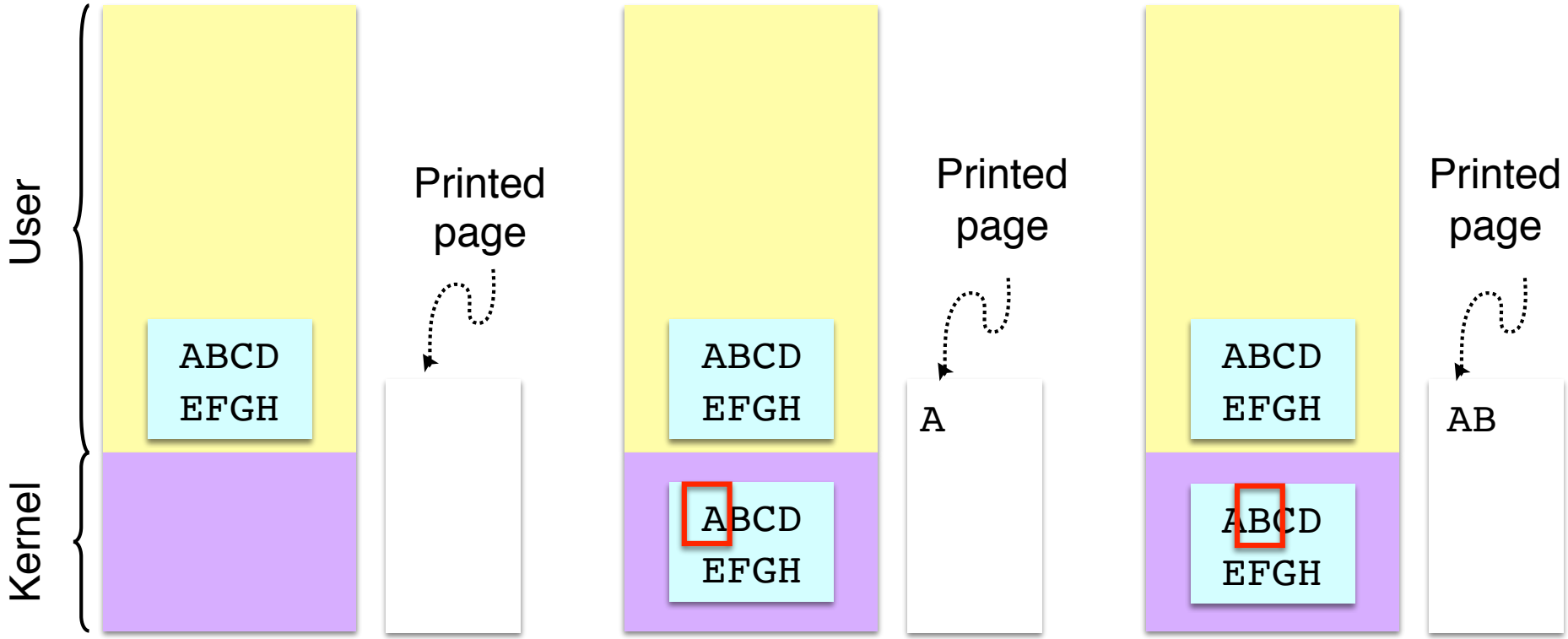
Hardware's view of interrupts



I/O software: goals

- Device independence
  - Programs can access any I/O device
  - No need to specify device in advance
- Uniform naming
  - Name of a file or device is a string or an integer
  - Doesn't depend on the machine (underlying hardware)
- Error handling
  - Done as close to the hardware as possible
  - Isolate higher-level software
- Synchronous vs. asynchronous transfers
  - Blocked transfers vs. interrupt-driven
- Buffering
  - Data coming off a device cannot be stored in final destination
- Sharable vs. dedicated devices

Programmed I/O: printing a page



Code for programmed I/O

```
copy_from_user (buffer, p, count);    // copy into kernel
buffer
for (j = 0; j < count; j++) {         // loop for each char
     // wait for printer to be ready
     while (*printer_status_reg != READY)
          ;
     // output a single character
     *printer_data_reg = p[j];
}
return_to_user();
```

Interrupt-driven I/O

```
copy_from_user (buffer, p, count);
j = 0;
enable_interrupts();
while (*printer_status_reg != READY)
     ;
*printer_data_reg = p[0];
scheduler(); // and block user
```

Code run by system call

```
if (count == 0) {
     unblock_user();
} else {
     *printer_data_reg = p[j];
     count--;
     j++;
}
acknowledge_interrupt();
return_from_interrupt();
```

Code run at interrupt time

I/O using DMA

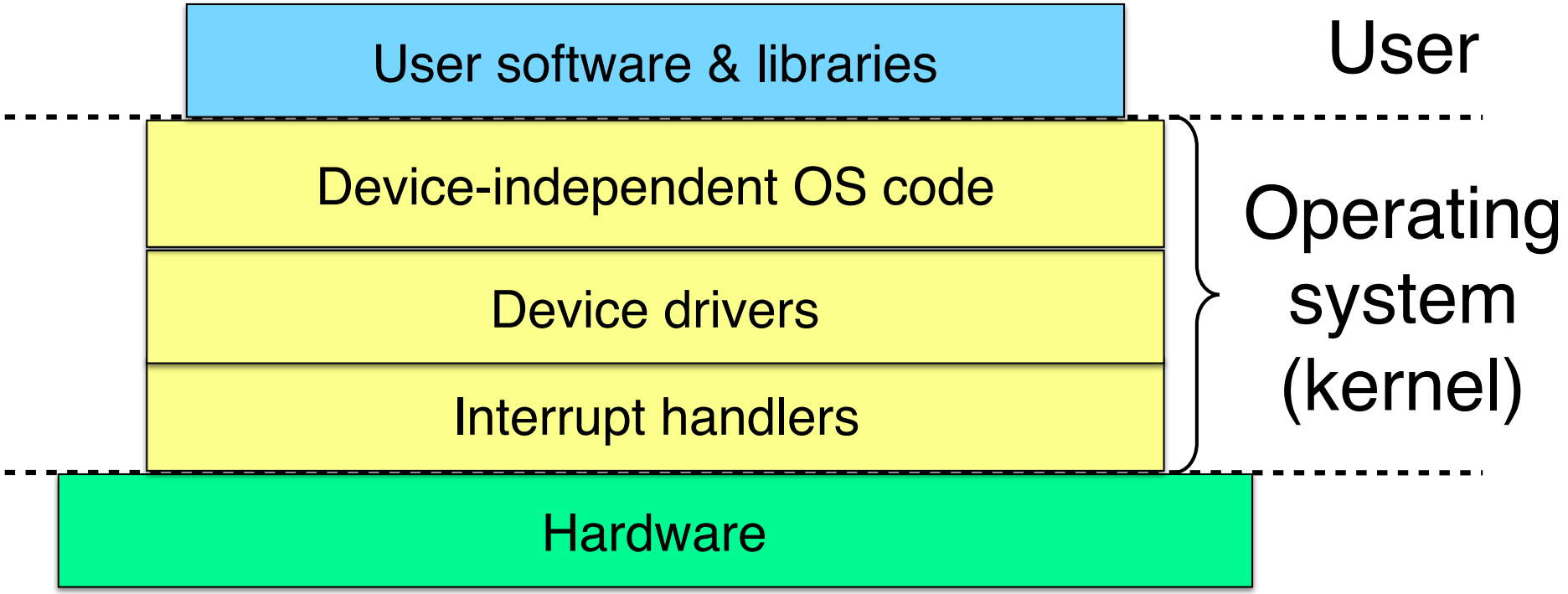
```
copy_from_user (buffer, p, count);
lock_buffer (buffer);
set_up_DMA_controller();
scheduler(); // and block user
```

Code run by system call

```
acknowledge_interrupt();
unlock_buffer (buffer);
unblock_user();
return_from_interrupt();
```

Code run at interrupt time

Layers of I/O software



Interrupt handlers

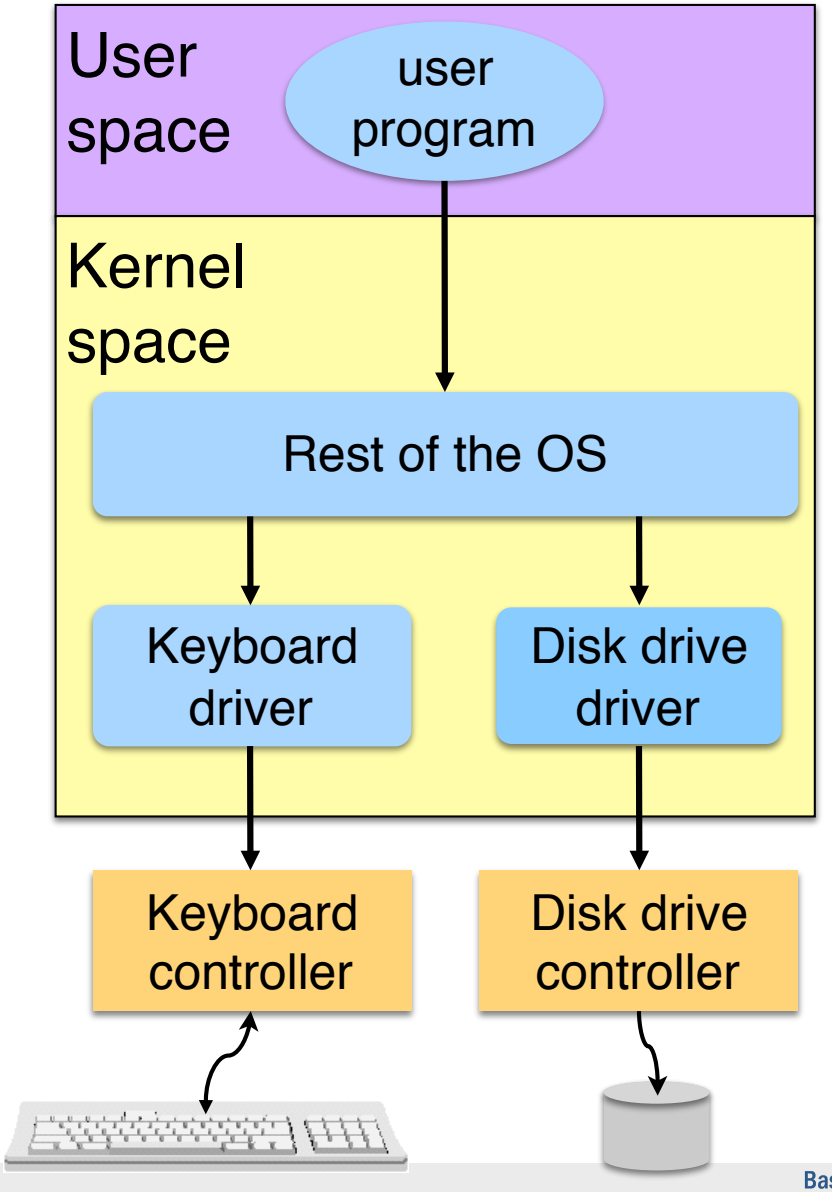
- Interrupt handlers are best hidden
  - Driver starts an I/O operation and blocks
  - Interrupt notifies of completion
- Interrupt procedure does its task
  - Then unblocks driver that started it
  - Perform minimal actions at interrupt time
    - Some of the functionality can be done by the driver after it is unblocked
- Interrupt handler must
  - Save registers not already saved by interrupt hardware
  - Set up context for interrupt service procedure

What happens on an interrupt

- Set up stack for interrupt service procedure
- Ack interrupt controller, reenable interrupts
- Copy registers from where saved
- Run service procedure
- (optional) Pick a new process to run next
- Set up MMU context for process to run next
- Load new process' registers
- Start running the new process

Device drivers

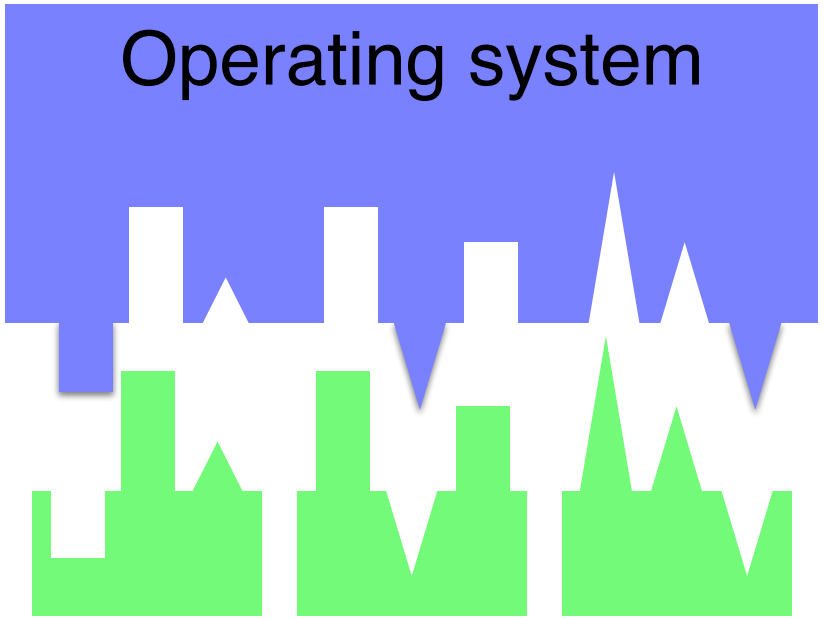
- Device drivers go between device controllers and rest of OS
  - Drivers standardize interface to widely varied devices
- Device drivers communicate with controllers over bus
  - Controllers communicate with devices themselves



Device-independent I/O software

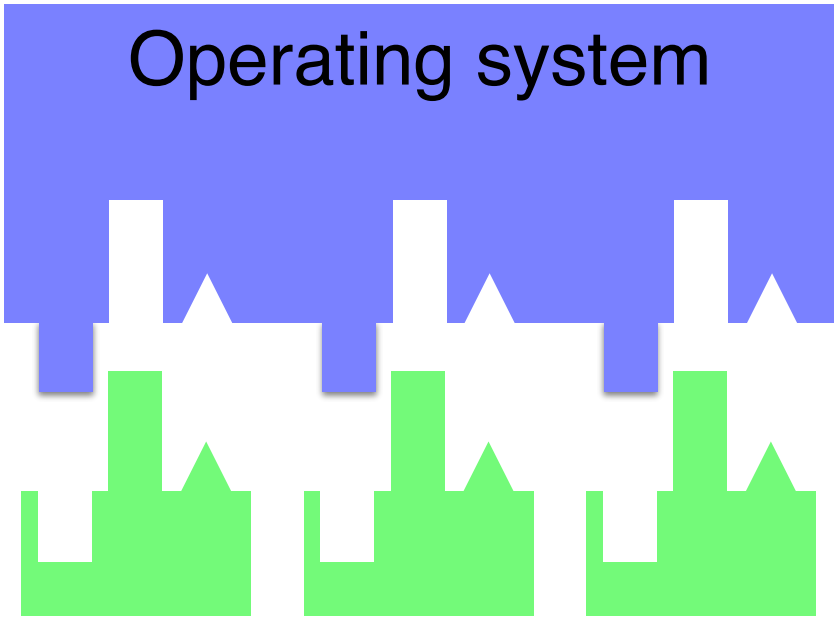
- Device-independent I/O software provides common “library” routines for I/O software
- Helps drivers maintain a standard appearance to the rest of the OS
- Uniform interface for many device drivers for
  - Buffering
  - Error reporting
  - Allocating and releasing dedicated devices
  - Suspending and resuming processes
- Common resource pool
  - Device-independent block size (keep track of blocks)
  - Other device driver resources

Why a standard driver interface?



Non-standard device driver interface

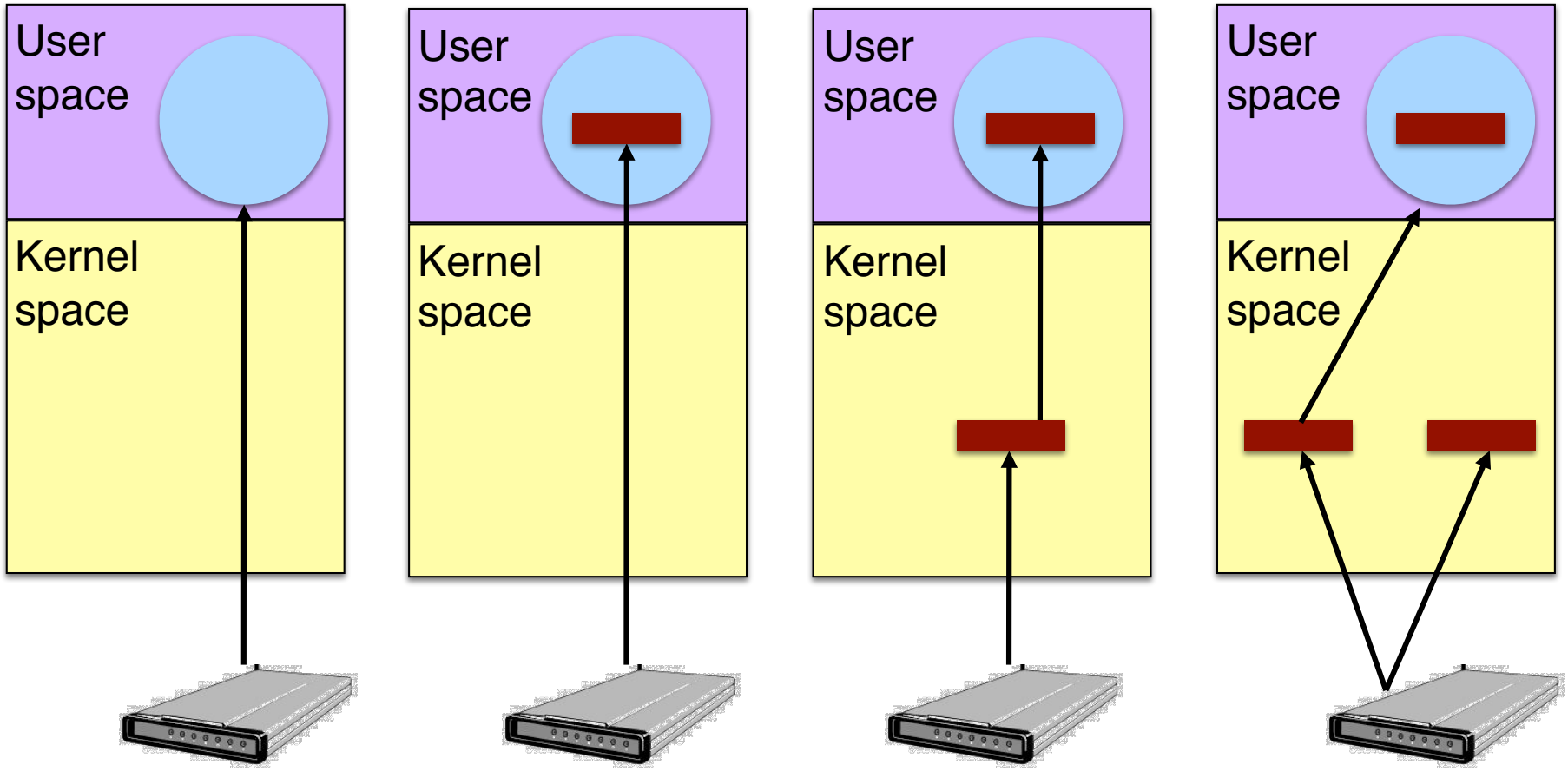
- Different interface for each driver
- High OS complexity
- Less code reuse



Standard device driver interface

+ Less OS/driver interface code
+ Lower OS complexity
+ Easy to add new drivers

Buffering device input



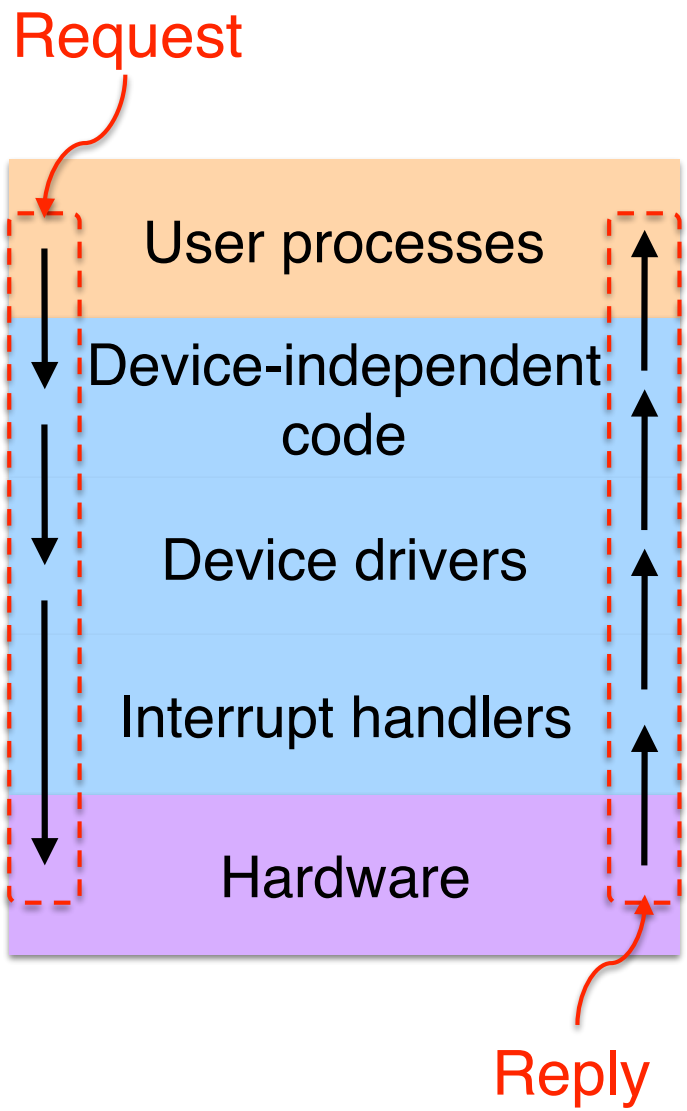
Unbuffered input

Buffering in user space

Buffer in kernel
Copy to user space

Double buffer in kernel

What happens where on an I/O request?



Make I/O call; format I/O; spooling

Naming, protection
blocking / buffering / allocation

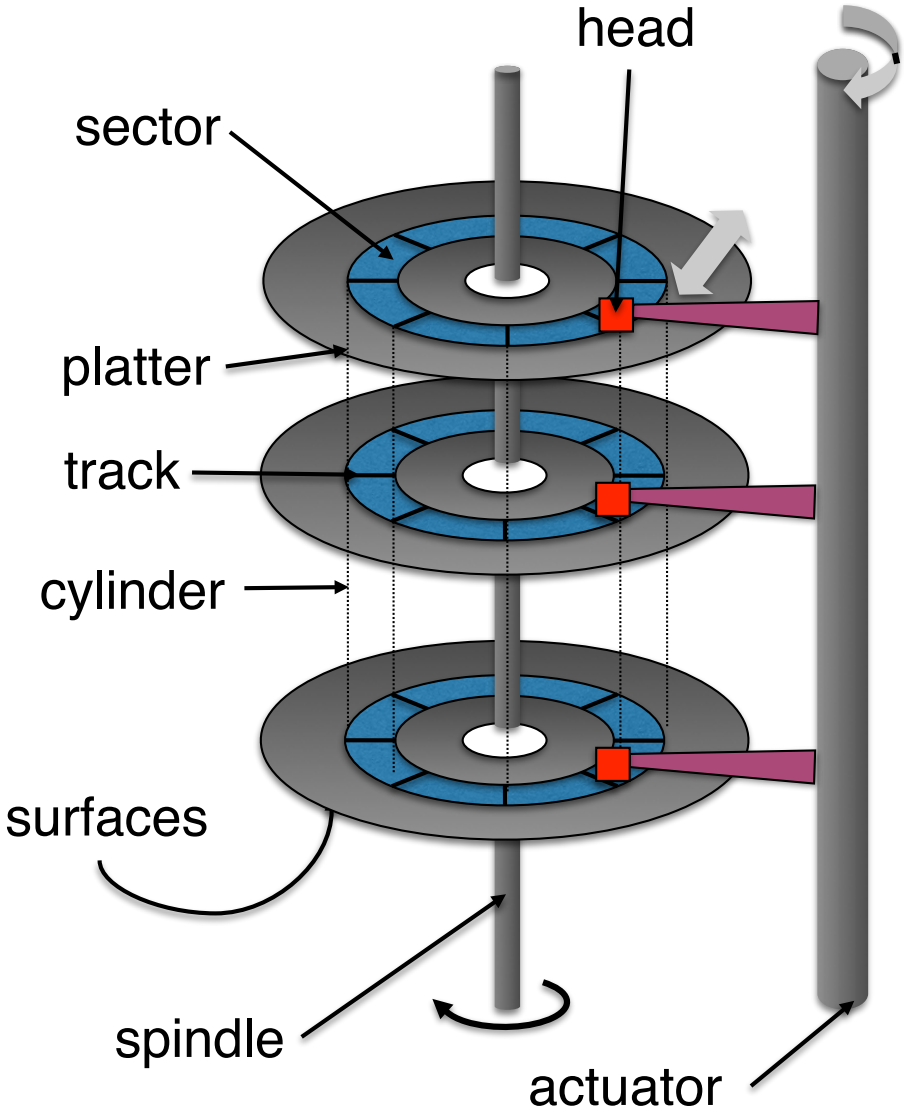
Manage device registers & status

Signal device driver on completed I/O

Actually do the I/O (in hardware)

Disk drive structure

- Data stored on surfaces
  - Up to two surfaces per platter
  - One or more platters per disk
- Data in concentric tracks
  - Tracks broken into sectors
    - 256B–4KB per sector
  - Cylinder: corresponding tracks on all surfaces
- Data read and written by heads
  - Actuator moves heads
  - Heads move in unison



Disk drive specifics

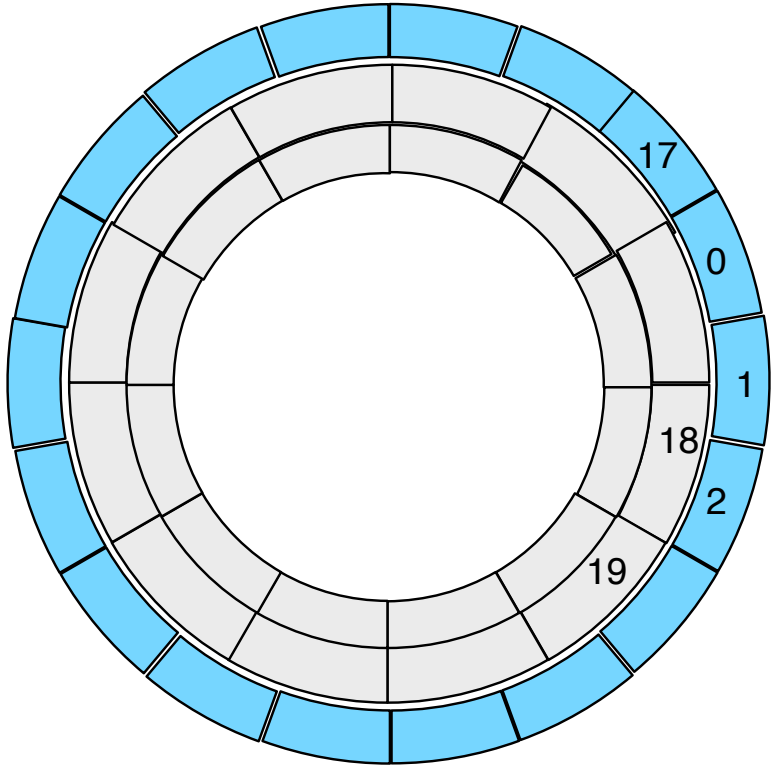
	IBM 360KB floppy	Seagate 3 TB HD
Cylinders	40	240,000 (estimate)
Tracks per cylinder	2	10
Sectors per track	9	2500 (average)
Sectors per disk	720	6×10^9
Bytes per sector	512	512
Capacity	360 KB	3 TB
Seek time (minimum)	6 ms	1.5 ms (?)
Seek time (average)	77 ms	4.16 ms
Rotation time	200 ms	8.3 ms
Spinup time	250 ms	~5 sec (?)
Sector transfer time	22 ms	3.4 µsec

Structure of a disk sector

- Preamble contains information about the sector
  - Sector number & location information
- Data is usually 512 or 1024 bytes
  - Newer disks moving to 4KB sectors
- ECC (Error Correcting Code) is used to detect & correct minor errors in the data

Sector layout on disk

- Sectors numbered sequentially on each track
- Numbering starts in different place on each track: sector skew
  - Allows time for switching head from track to track
  - Done to minimize delay in sequential transfers
- Zoning
  - Different zones have different number of sectors per track
  - Outer zones have higher transfer rate
  - Typically 8–30 zones (or more)
  - Zones set at time of manufacture

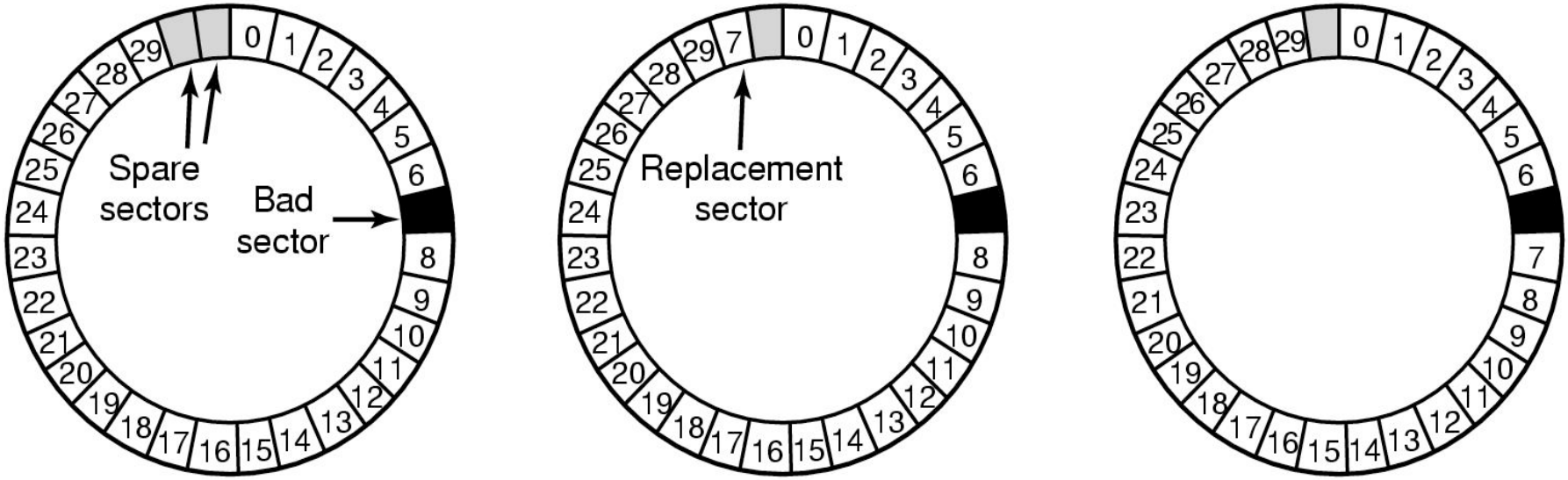


Disk "addressing"

- Millions of sectors on the disk must be labeled
- Two possibilities
  - Cylinder/track/sector
  - Sequential numbering
- Modern drives use sequential numbers
  - Disks map sequential numbers into specific location
  - Mapping may be modified by the disk
    - Remap bad sectors
    - Optimize performance
  - Hide the exact geometry, making life simpler for the OS

When good disks go bad…

- Disks have defects
  - With billions of sectors, this isn't surprising!
- ECC helps with errors, but sometimes this isn't enough
- Disks keep spare sectors (normally unused) and remap bad sectors into these spares
  - If there's time, the whole track could be reordered…
- All of this is transparent to the user (and, typically, OS)



Building a better "disk"

- Problem: CPU performance has been increasing exponentially, but disk performance hasn't
  - Disks are limited by mechanics
- Problem: disks aren't all that reliable
- Solution: distribute data across disks, and use some of the space to improve reliability
  - Data transferred in parallel
  - Data stored across drives (striping)
- Problem: more disks means less reliability
  - Generate & store additional information to allow data from a failed disk to be recovered

Calculating parity

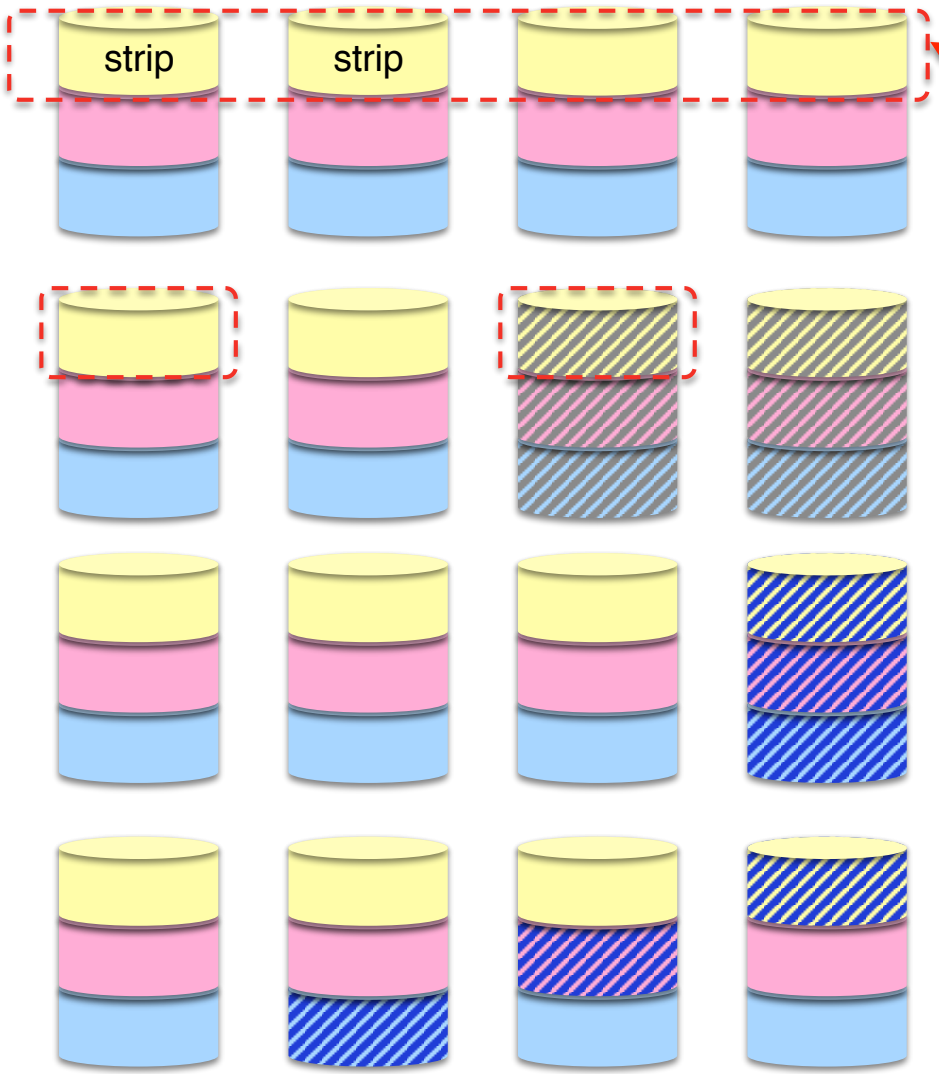
$$D0 \oplus D1 \oplus \cancel{D2} \oplus D3 = P$$

$$D0 \oplus D1 \oplus D2 \oplus D3 \oplus (D2 \oplus P) = P \oplus (D2 \oplus P)$$

$$D0 \oplus D1 \oplus D3 \oplus P = D2$$

- Goal: keep extra information to recover from device failure
- Use the extra information to rebuild the data on the missing device
- ➡ There are formulas to recover from more than one failure with additional "parity" devices

RAIDs, RAIDs, and more RAIDs



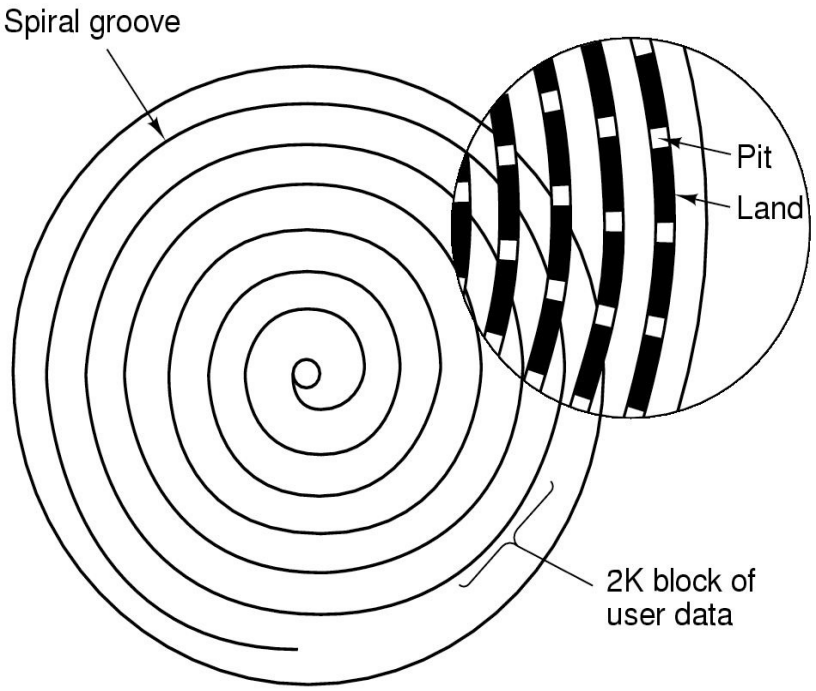
RAID 0
(Redundant Array of Inexpensive Disks)

RAID 1
(Mirrored copies)

RAID 4
(Striped with parity)

RAID 5
(Parity rotates through disks)

CD/DVD-ROM recording



- DVD-ROM has data in a spiral
  - Hard drives have concentric circles of data
- One continuous track: just like vinyl records!
  - Starts from the center
- Pits & lands “simulated” with heat-sensitive material on DVD-Rs and DVD-RWs

What's in a disk request?

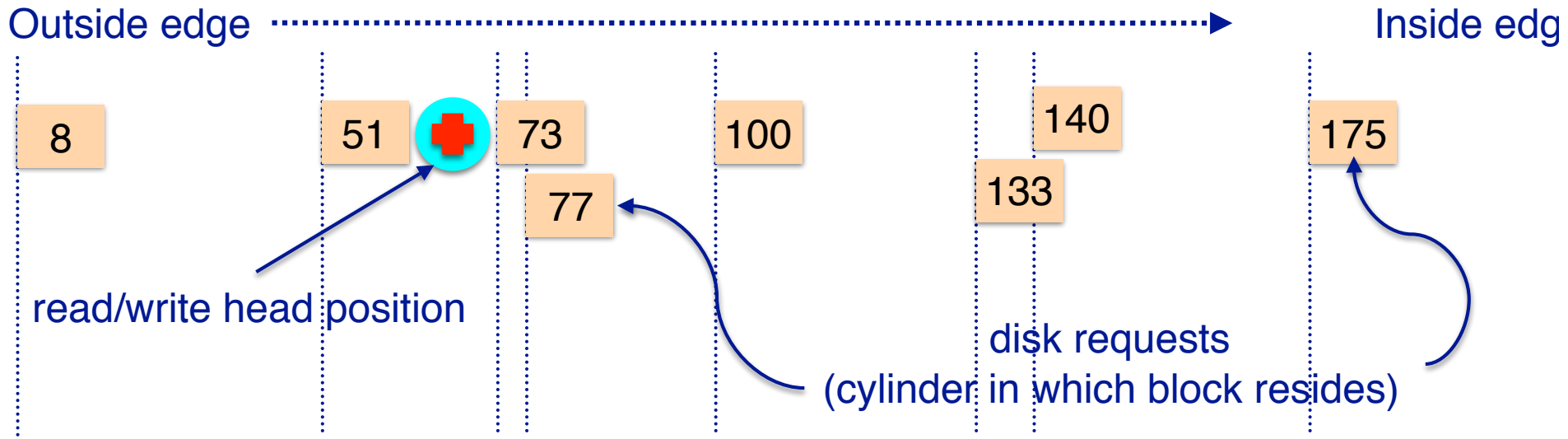
- Time required to read or write a disk block determined by 3 factors
  - Seek time
    - Typically around 2–8 ms, depending on distance
    - Longer seeks take longer, but not linear with distance
  - Rotational delay
    - Average delay = 1/2 rotation time
    - Example: rotate in 10 ms, average rotation delay = 5 ms
  - Actual transfer time
    - Transfer time = time to rotate over sector
    - Example: rotate in 10 ms, 1000 sectors/track ⇒ 10/1000 ms = 0.01 ms (10 µs) transfer time per sector
- Seek time and rotation time comparable
- Error checking is done by on-disk controller

Disk request scheduling

- Goal: use disk hardware efficiently
  - Bandwidth as high as possible
  - Disk transferring as often as possible (and not seeking)
- We want to
  - Minimize disk seek time (moving from track to track)
  - Minimize rotational latency (waiting for disk to rotate the desired sector under the read/write head)
- Calculate disk bandwidth by
  - Total bytes transferred / time to service request
  - Seek time & rotational latency are overhead (no data is transferred), and reduce disk bandwidth
- Minimize seek time & rotational latency by
  - Using algorithms to find a good sequence for servicing requests
  - Placing blocks of a given file “near” each other

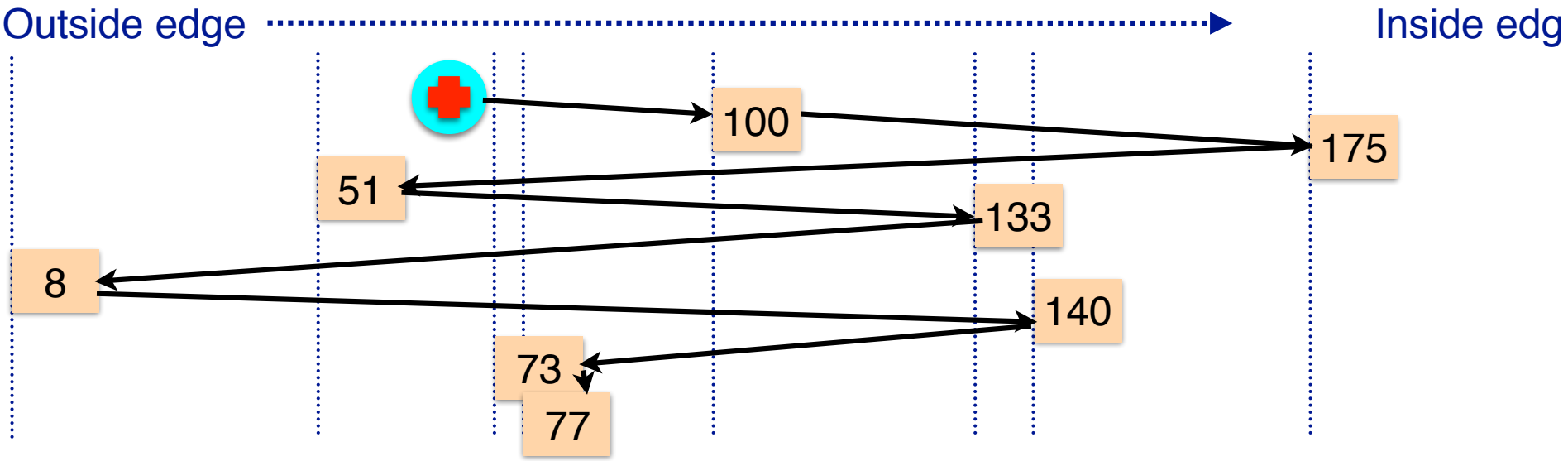
Disk scheduling algorithms

- Schedule disk requests to minimize disk seek time
  - Seek time increases as distance increases (though not linearly)
  - Minimize seek distance ➔ minimize seek time
- Disk seek algorithm examples assume a request queue & head position (disk has 200 cylinders)
  - Queue = 100, 175, 51, 133, 8, 140, 73, 77
  - Head position = 63



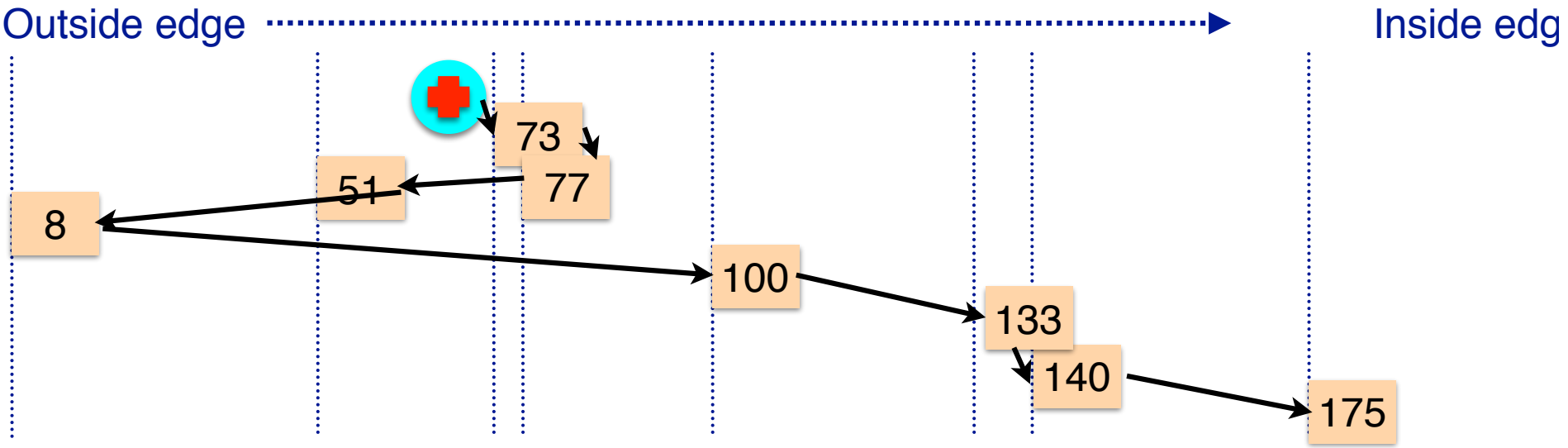
First-Come-First Served (FCFS)

- Requests serviced in the order in which they arrived
  - Easy to implement!
  - May involve lots of unnecessary seek distance
- Seek order = 100, 175, 51, 133, 8, 140, 73, 77
- Seek distance =
  (100-63) + (175-100) + (175-51) + (133-51) + (133-8) + (140-8) +
  (140-73) + (77-73) = 646 cylinders



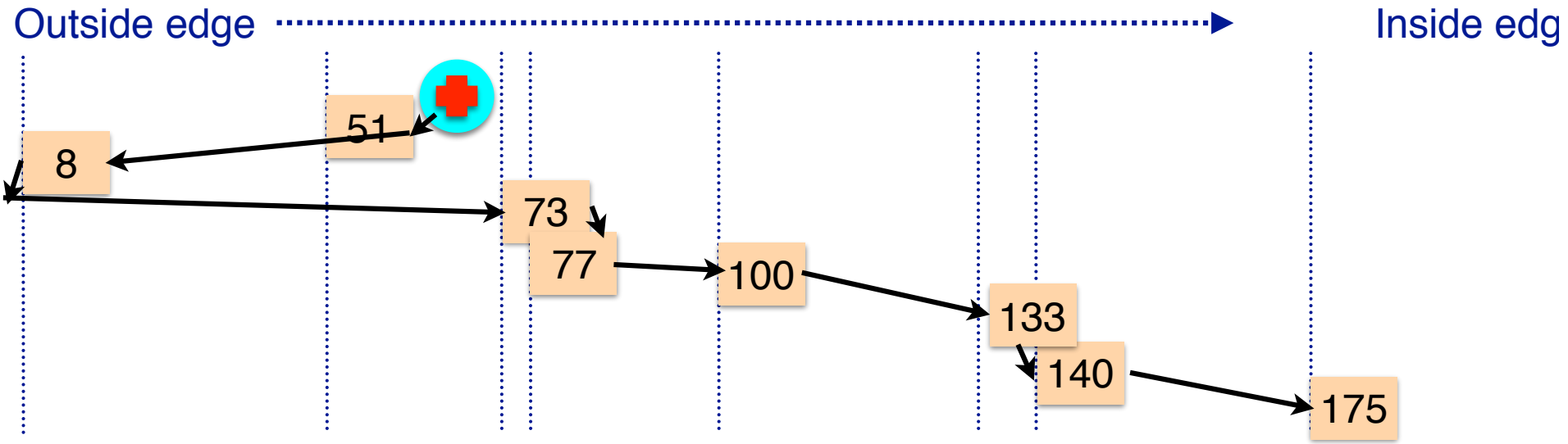
Shortest Seek Time First (SSTF)

- Service the request with the shortest seek time from the current head position
  - Form of SJF scheduling
  - May starve some requests
- Seek order = 73, 77, 51, 8, 100, 133, 140, 175
- Seek distance = 10 + 4 + 26 + 43 + 92 + 33 + 7 + 35 = 250 cylinders



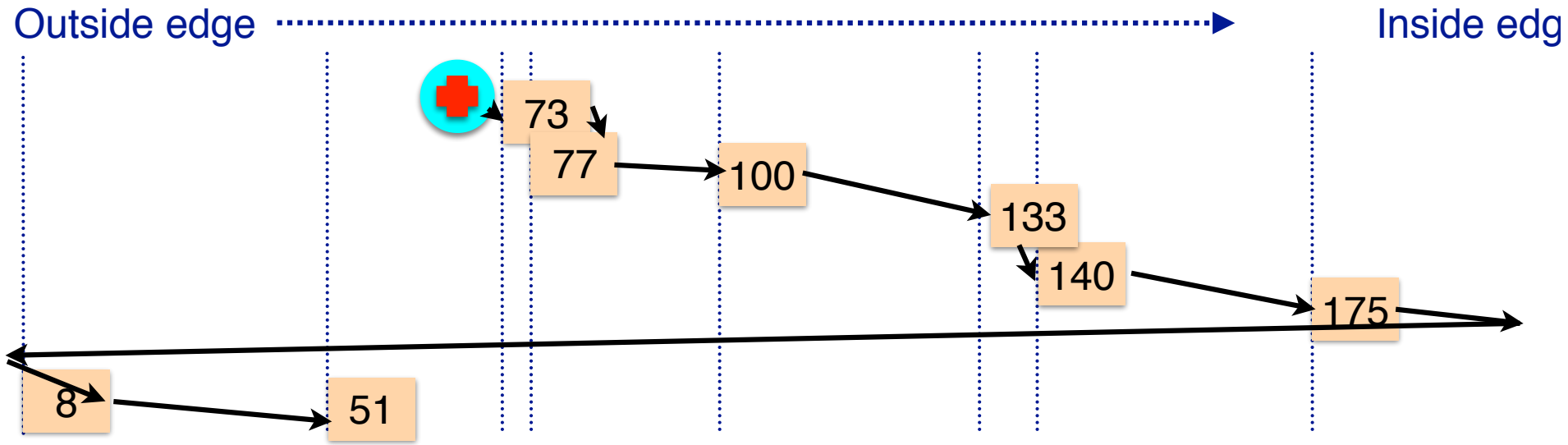
SCAN (elevator algorithm)

- Disk arm starts at one end of the disk and moves towards the other end, servicing requests as it goes
  - Reverses direction when it gets to end of the disk
  - Also known as elevator algorithm
- Seek order = 51, 8, 0 , 73, 77, 100, 133, 140, 175
- Seek distance = 12 + 43 + 8 + 73 + 4 + 23 + 33 + 7 + 35 = 238 cylinders



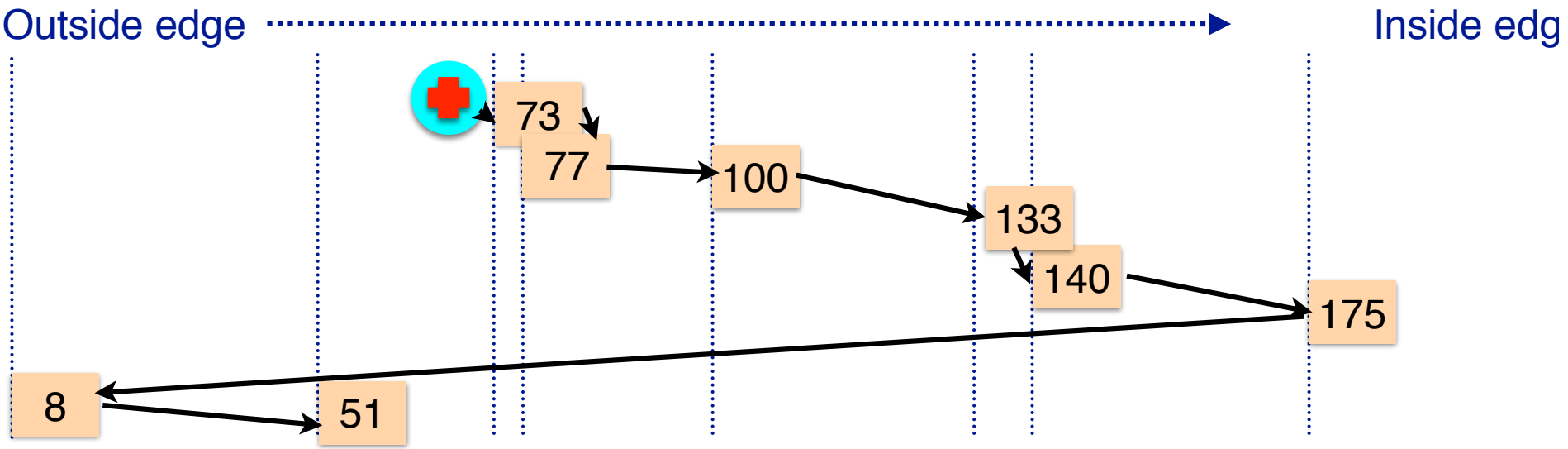
C-SCAN

- Identical to SCAN, except head returns to cylinder 0 when it reaches the end of the disk
  - Treats cylinder list as a circular list that wraps around the disk
  - Waiting time is more uniform for cylinders near the edge of the disk
- Seek order = 73, 77, 100, 133, 140, 175, 199, 0, 8, 51
- Distance = 10 + 4 + 23 + 33 + 7 + 35 + 24 + 199 + 8 + 43 = 386 cyls



C-LOOK

- Identical to C-SCAN, except head only travels as far as the last request in each direction
  - Saves seek time from last sector to end of disk
- Seek order = 73, 77, 100, 133, 140, 175, 8, 51
- Distance = 10 + 4 + 23 + 33 + 7 + 35 + 167 + 43 = 322 cylinders



Picking a disk scheduling algorithm

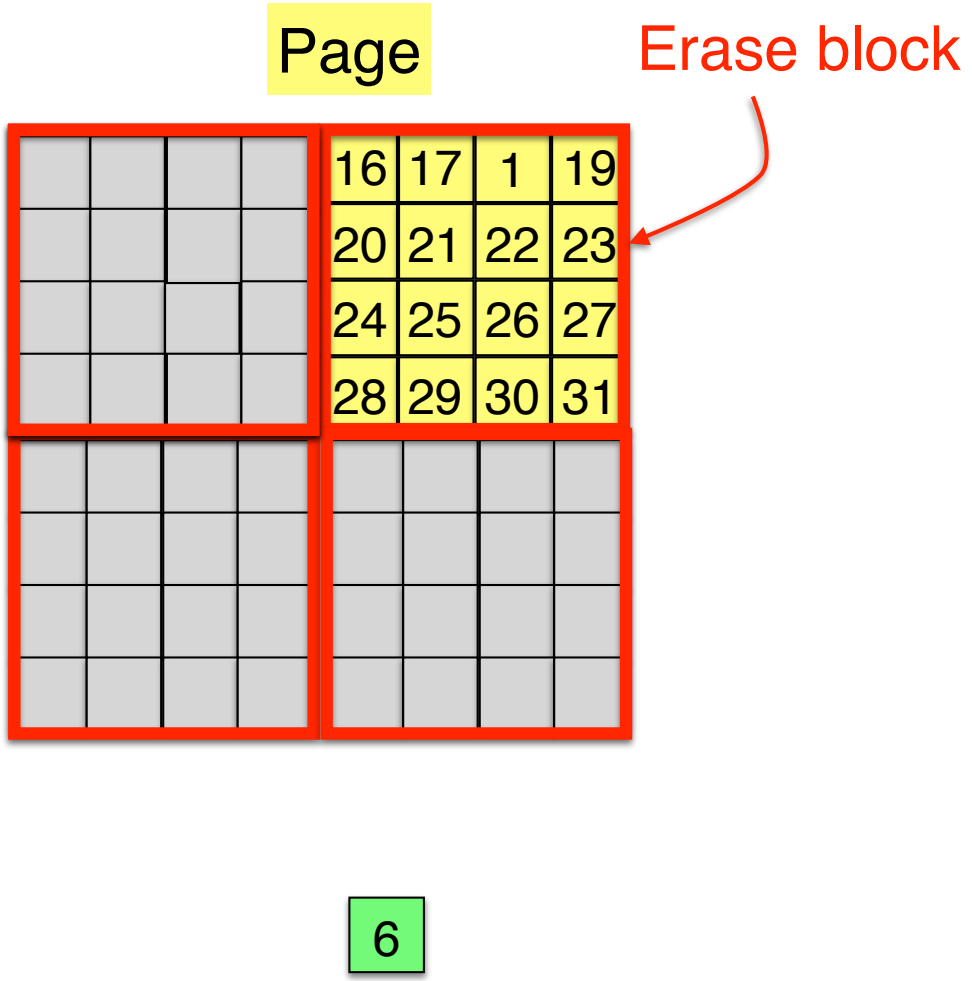
- SSTF is easy to implement and works OK if there aren't too many disk requests in the queue
- SCAN-type algorithms perform better for systems under heavy load
  - More fair than SSTF
  - Use LOOK rather than SCAN algorithms to save time
- Long seeks aren't too expensive, so choose C-LOOK over LOOK to make response time more even
- Disk request scheduling interacts with algorithms for allocating blocks to files
  - Make scheduling algorithm modular: allow it to be changed without changing the file system
- Use SSTF or FCFS for lightly loaded systems
- Use C-LOOK for heavily loaded systems

Flash memory (and SSDs)

- Compared to disk, flash is
  - Faster (shorter access time, but lower bandwidth)
  - More expensive
  - More reliable (devices)
  - Less reliable (sectors)
- Compared to DRAM, flash is
  - Cheaper (a bit)
  - Non-volatile (data survives power loss)
  - Slower
- Use flash as a level between disk and memory?
- Issues
  - Can't overwrite sectors "in place"
  - Flash wears out: can only write 3,000–100,000 times per memory cell

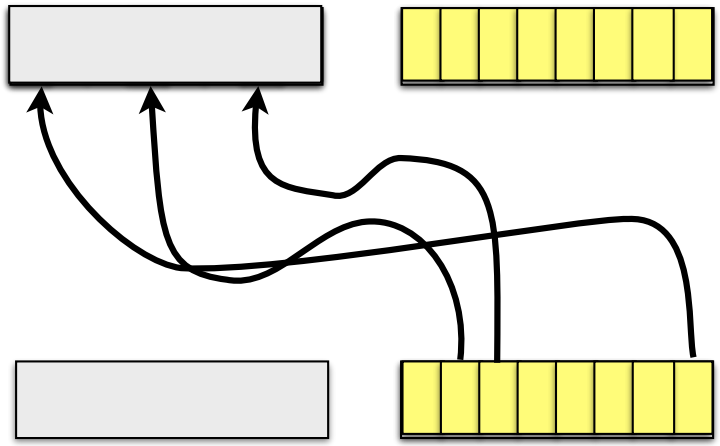
Writing to flash memory

- Can't overwrite page in place
  - Need to write to "clean" region
  - Unit of erase is "erase block"
- One solution: copy entire erase block to new location
  - Modify the page being written
- This isn't efficient!
  - Flash vendors spend a lot of time on Flash Translation Layers
  - FTLs map logical page numbers to physical pages
  - Also manage wear leveling: distribute erases among blocks



Is there a better way to manage flash?

- Log-structured data storage works well on flash
  - Append updates to a “log”
- Periodically “clean” structures
  - Rewrite live data to a new log segment
  - Obsolete data isn’t rewritten
- Need to track mapping of logical blocks ➡ physical locations
- More on this when we discuss file systems…



Handling flash in the OS

- Treat it like a disk?
  - Flash is written in blocks, just like a disk
    - Blocks have to be erased first: somewhat slow
    - Erase blocks are much larger than pages
  - Flash is random access, just like a disk

  ➡This approach is often used!
- Need to be careful about wearing out flash
  - Most flash devices do “wear leveling”: remap blocks internally when they’re erased
  - File systems for flash should take wear into account
    - Many don’t, including the standard VFAT file system

Flash to replace disk?

Clock hardware

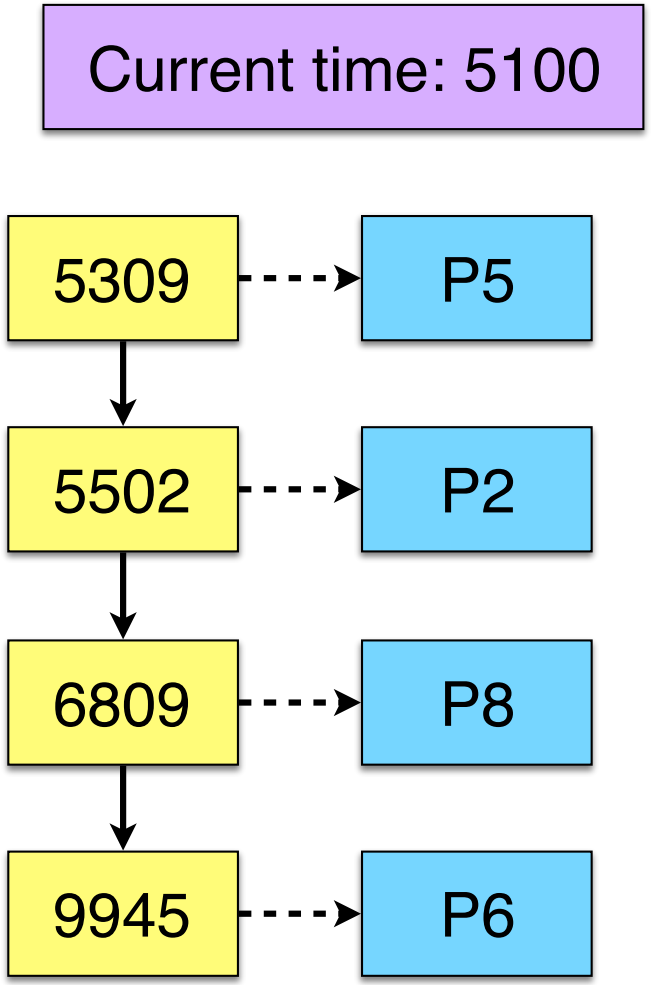
- Crystal oscillator with fixed frequency (only when computer is on)
  - Typically used to time short intervals (~ 1 second)
  - May be used to correct time-of-day clock
- Time-of-day clock (runs when system is off)
  - Keeps minutes, hours, days
  - May not be too accurate…
  - Used to load system clock at startup
- Time kept in seconds and ticks (often 10^2–10^6 per second)
  - Number of seconds since a particular time
    - For many versions of Unix, tick 0 was on January 1, 1970
  - Number of ticks this second
  - Advance ticks once per clock interrupt
  - Advance seconds when ticks "overflow"

Keeping time

- Check time via the Web
  - Protocol to get time from accurate time servers (ntp)
- What happens when system clock is slow?
  - Advance clock to the correct current time
  - May be done all at once or over a minute or two
- What happens when system clock is fast?
  - Can't have time run backwards!
  - Instead, advance time more slowly than normal until the clock is correct
- Track clock drift, do periodic updates to keep clock accurate

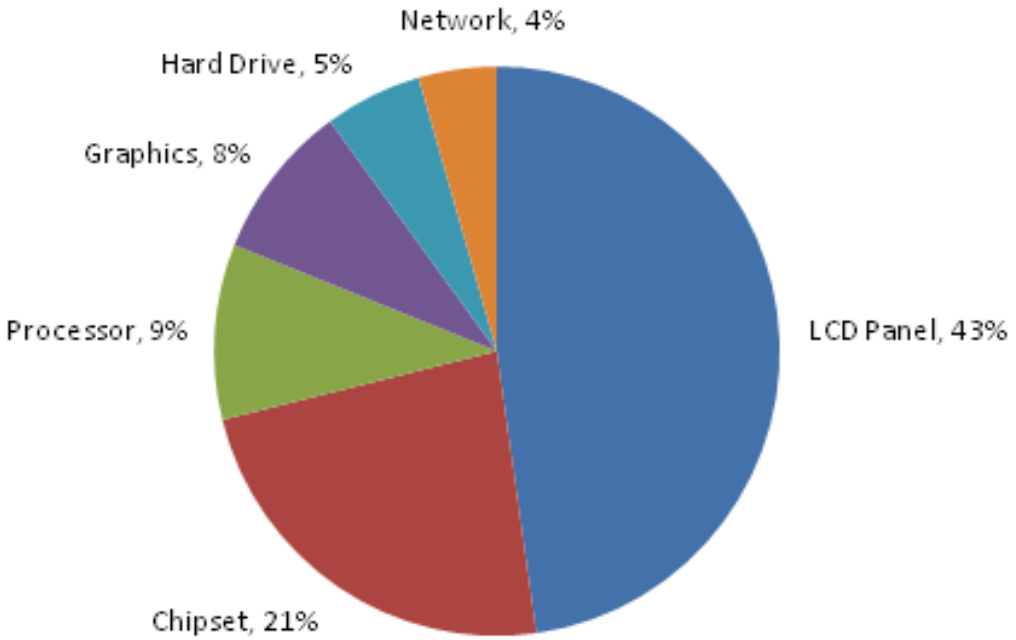
Using timers in software

- Use short interval clock for timer interrupts
  - Specified by applications
  - No problems if interrupt frequency is low
  - May have multiple timers using a single clock chip
- Use soft timers to avoid interrupts
  - Kernel checks for soft timer expiration before it exits to user mode
  - Less accurate than using a hardware timer
  - How well this works depends on rate of kernel entries



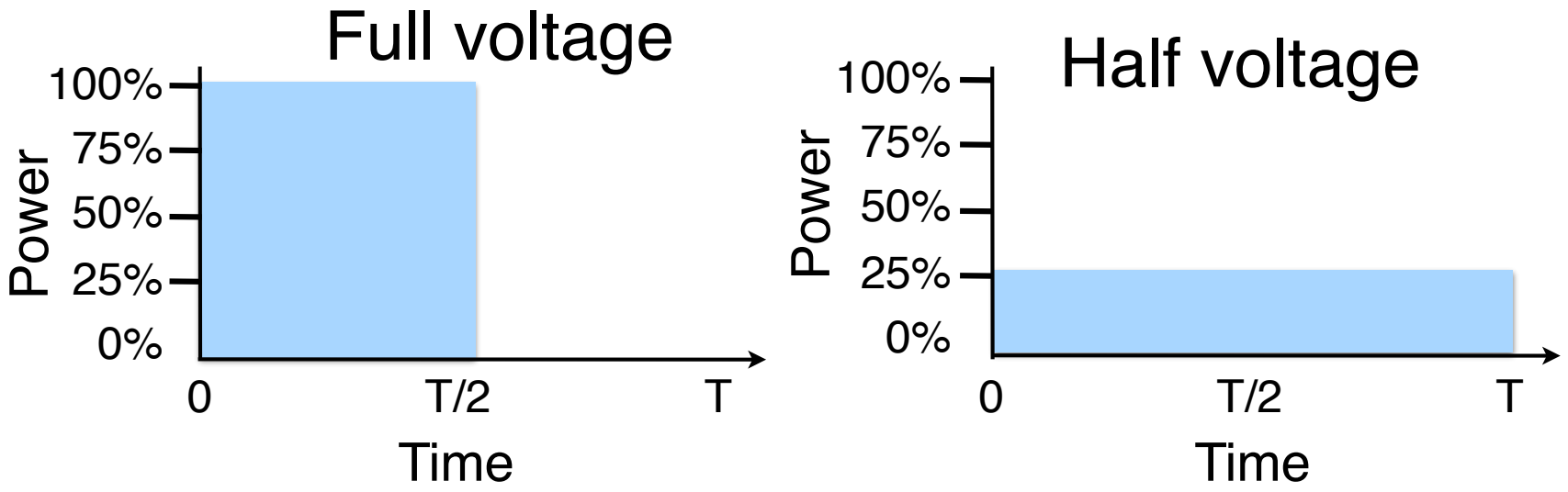
Where does the power go?

- How much power does each part of a laptop computer use?
  - Display is the biggest energy hog, by far!
  - Chipset (including memory) is next
- Save power by
  - Reducing display power
    - Turning down the backlight
  - Slowing down CPU
  - Using “less” of the CPU
  - Using an alternate GPU
  - Powering down hard drive
  - Using a flash-based drive



Source: http://blogs.msdn.com/b/e7/archive/2009/01/06/windows-7-energy-efficiency.aspx

Reducing CPU power usage



- Running at full clock speed
  - Jobs finish quickly
  - High energy consumption: proportional to shaded area
  - Intel processors may use 50–75 watts!
- Cutting voltage by two
  - Cuts clock speed by two: processes take longer
  - Cuts power by four
  - Cuts energy consumption (= power × time) in half

How can we reduce power usage?

- Tell the programs to use less energy
  - May mean poorer user experience
  - Makes batteries last longer!
- Examples
  - Change from color output to black and white (OLPC does this)
  - Dim the screen
  - Switch to lower-power processor modes
    - Dynamic voltage scaling for the CPU
    - Shut down CPU cores
    - Use a lower-power GPU
  - Fewer image updates per second (one reason why Kindle is so power efficient)
  - Use application-specific chips (Apple M7/M8 for motion capture)