University of Pittsburgh

Introduction to Operating Systems
CS 1550

Spring 2023

Sherif Khattab

ksm73@pitt.edu

(Some slides are from **Silberschatz, Galvin and Gagne ©2013)**

Announcements

- Upcoming deadlines
  - **All deadlines moved to Monday May 1st at 11:59 pm**
    - But please don't wait to last minute!
  - Homework 11, 12, Bonus Homework
  - Lab 4 and Lab 5
  - Quiz 3 and Quiz 4
  - Project 4 **(no late deadline)**
  - Post-Course Quiz (1 bonus point)
  - Bonus point for all class when OMET response rate >= 80%
    - Currently at 5%
    - Deadline: 4/23

Previous Lecture …

- How does a file system handle errors?

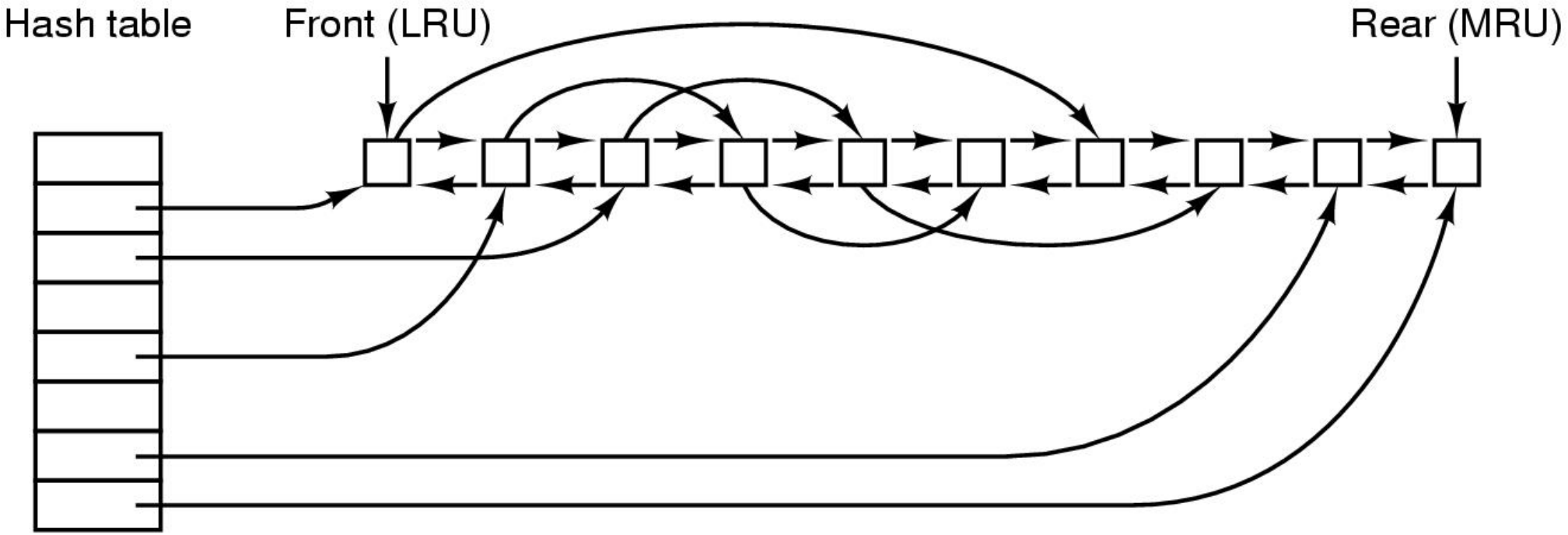
This lecture …

- How does a file system hide disk access delays?
- How do device drivers program I/O devices?

Question of the Day – Part 1

- How does a file system hide disk access delays?
- Answer:
  - Optimize disk reads
    - Caching
  - Optimize disk writes
    - log structured file system

Disk block cache: data structures



Side effects of caching

- Larger memory sizes → disk block caches are larger
- Increasing number of read requests come from cache
- Thus, **most disk accesses will be writes**

Optimize Disk Writes: Log-structured file systems

- Log-structured file system (LFS)
  - structures entire disk **as a log**
    - writes are always **appends**
  - writes initially **buffered** in memory
  - Periodically write these to **the end of the disk**
  - **What if a block is updated?**
    - Ignore the old **version** of the block
    - write the new version to the end of the disk
    - write a new i-node to point to the new block versions
- Issue: what happens when blocks are deleted?

Conventional File System

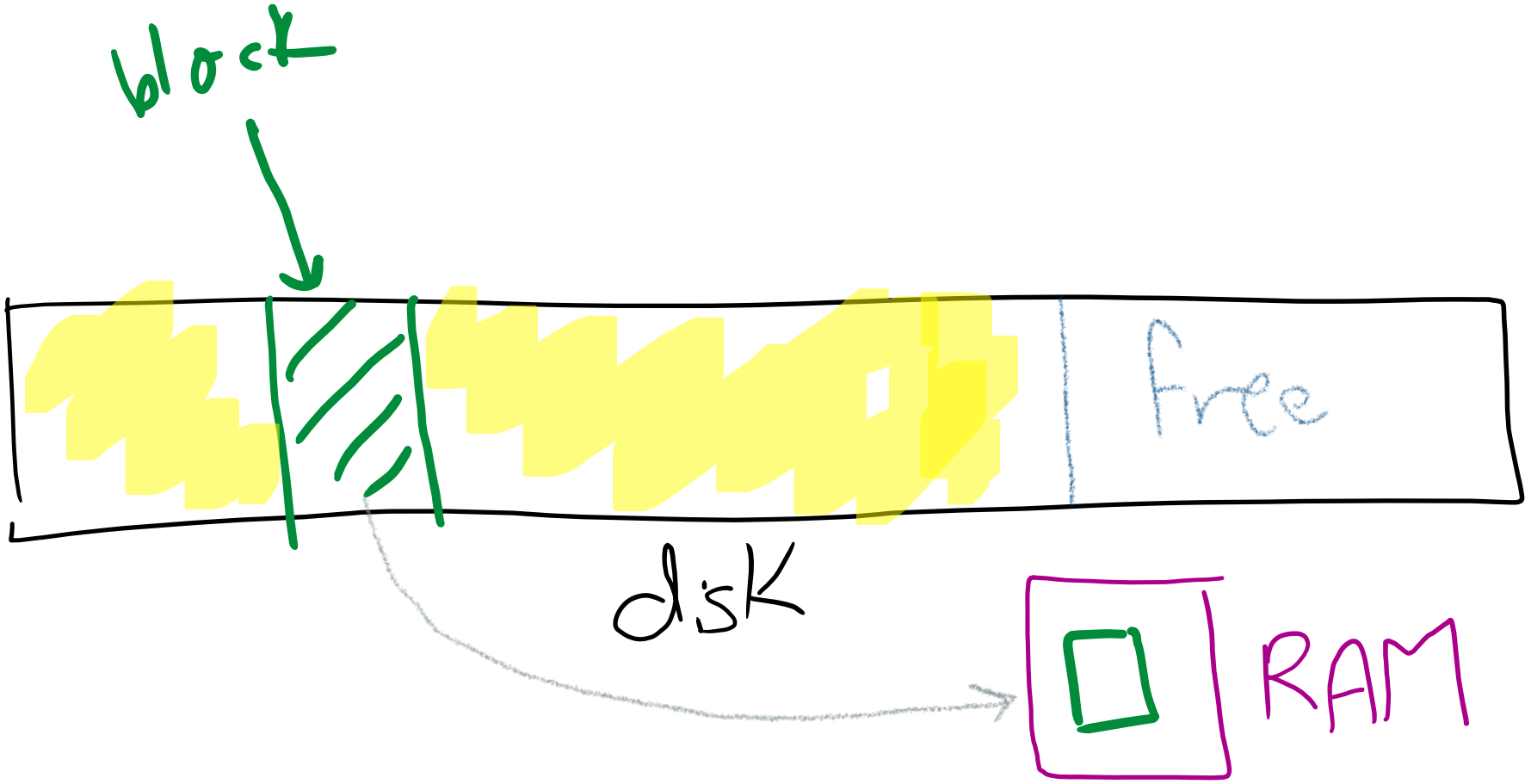
- When a block is updated
  - Read block to memory (if not already in cache)
  - Modify block in memory
  - Write block back to same location on disk

Log-structured File System

- When a block is updated

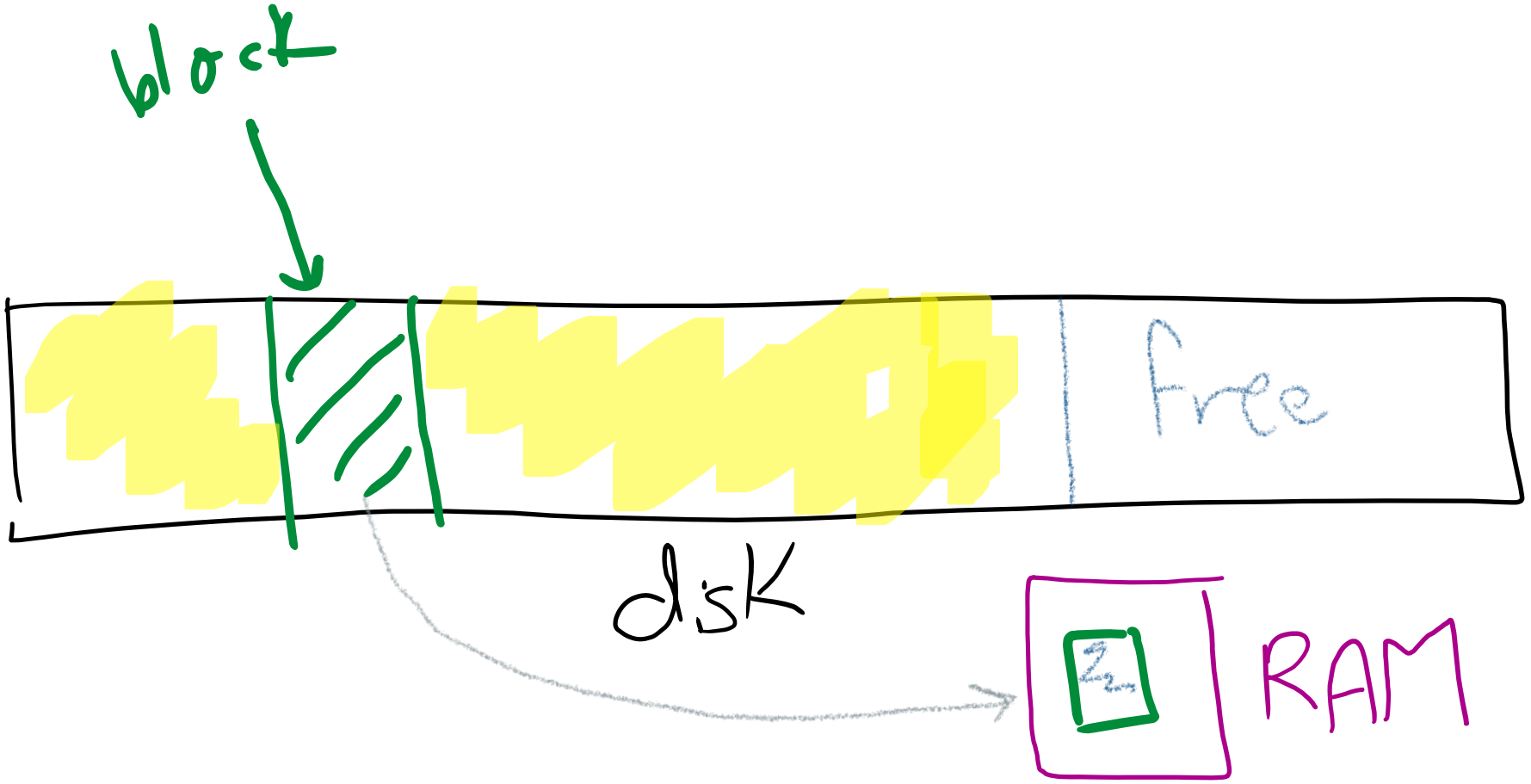
Log-structured File System

- When a block is updated
- Read block to memory (if not already in cache)



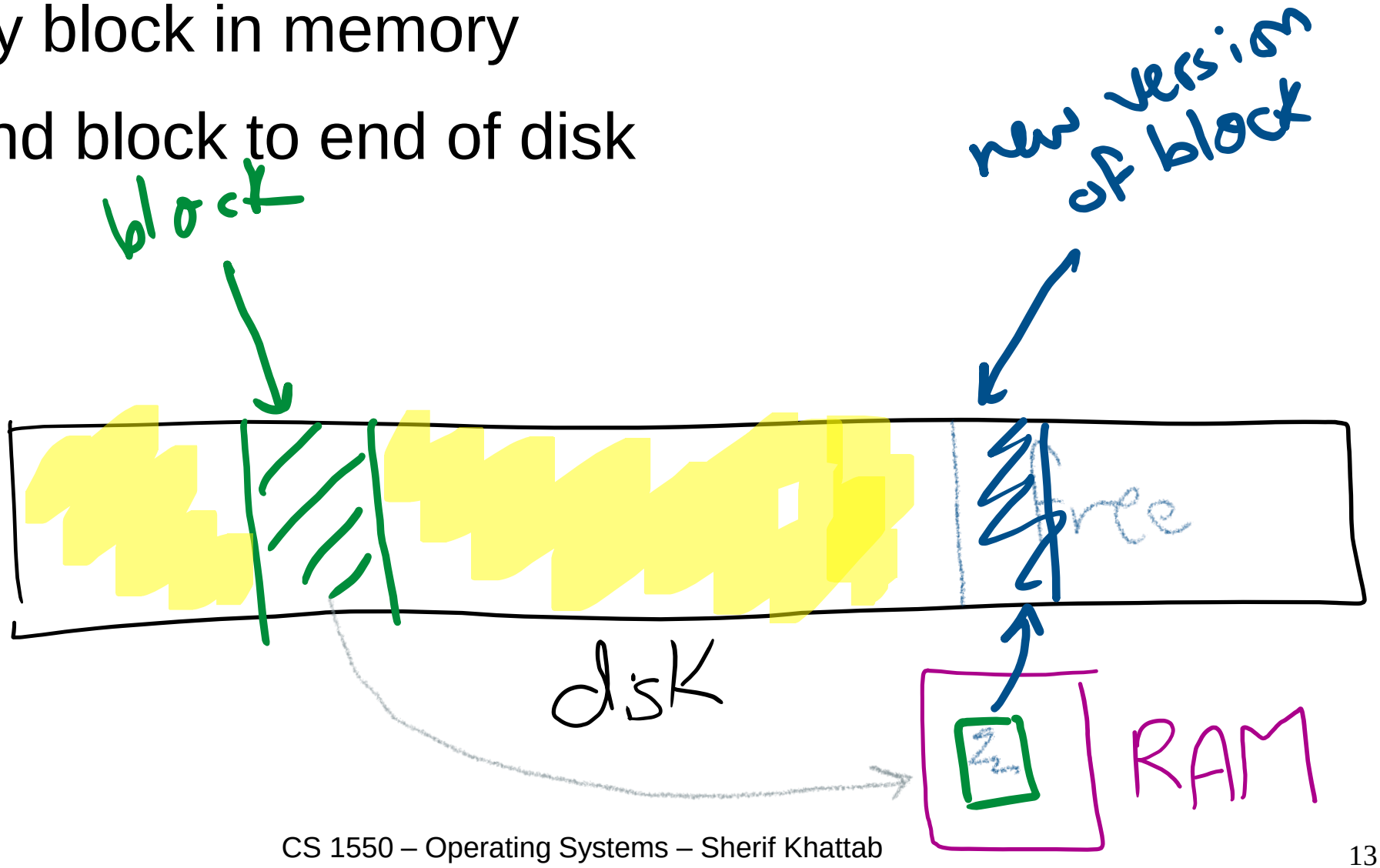
Log-structured File System

- When a block is updated
- Read block to memory (if not already in cache)
- Modify block in memory



Log-structured File System

- When a block is updated
- Read block to memory (if not already in cache)
- Modify block in memory
- Append block to end of disk



Log-structured File System

- When a block is updated
- Read block to memory (if not already in cache)
- Modify block in memory
- Append block to end of disk
- Create a new i-node pointing to the new block versions
- The process is process for all modified disk blocks
  - including the blocks for the i-nodes and
  - directories to point to the new i-nodes

Log-structured File System

- Disks will get full faster
- Solution:
  - Allow users to go back in time to older versions of files and directories
    - since they all already exist on disk
  - **Turn a bug into a feature!**

Log-structured File System

- Idea of LFS turns to work well for Solid State Disks!

Solid State Disks in a nutshell

- SSD divided into **erasure blocks**
  - Erasure blocks divided into **pages**
  - pages contain **cells**
- pages are read and written individually
  - fast operation
- The entire erasure block **erased before any page can be written**
  - erasing (aka flashing) is a **slow** operation
  - affects **durability** of the disk

Updating data on SSDs

- To modify a flash-disk page:
  - read block into memory
  - modify page in memory
  - erase block on disk
  - write modified block

Wear Leveling in SSDs

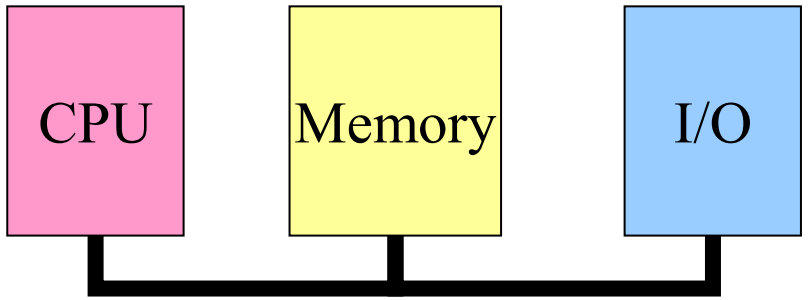
- **Wear Leveling**
  - balance number of flash operations across disk blocks

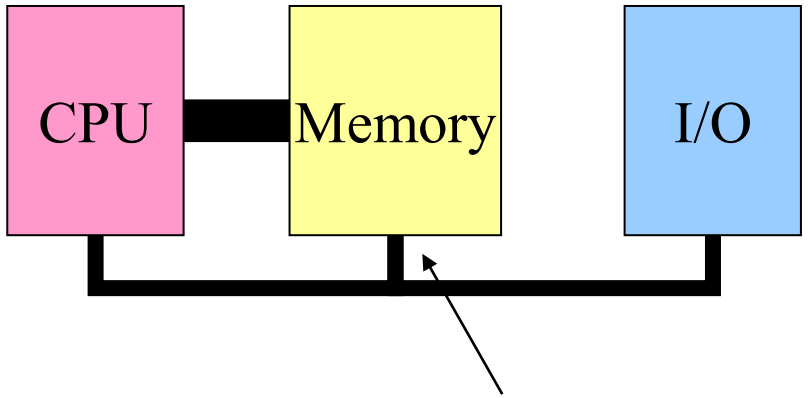
Problem of the Day – Part 2

- How do device drivers program I/O devices?
- Answer: three methods
  - polling
  - Interrupts
  - DMA

How are I/O devices connected?

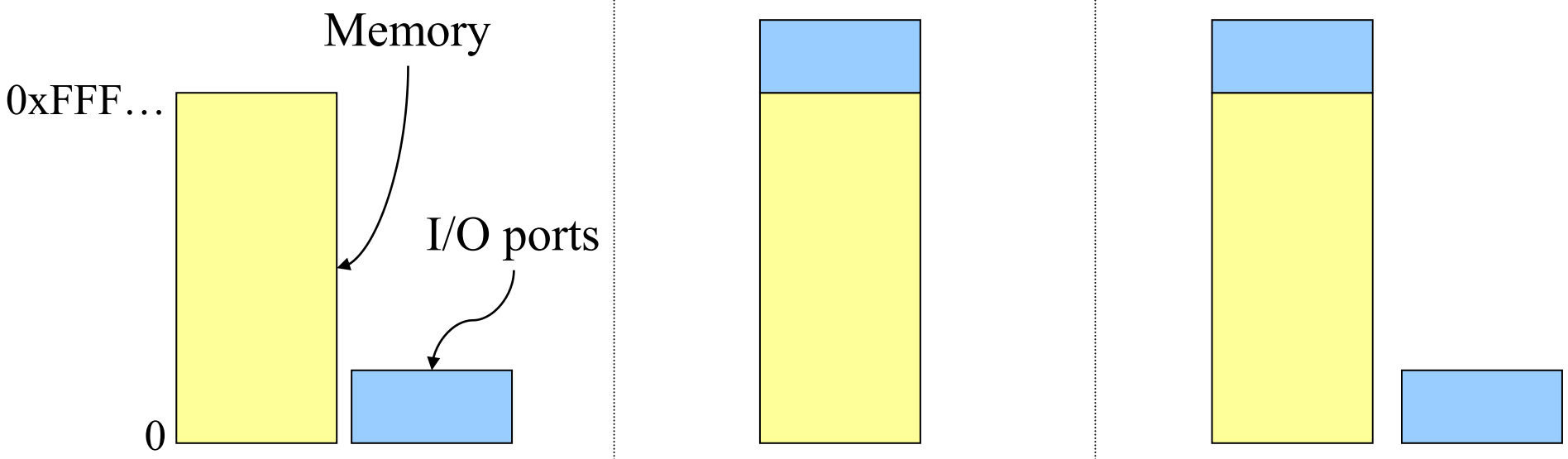
- Single-bus
  - All memory accesses go over a shared bus
  - I/O and RAM accesses compete for bandwidth
- Dual-bus
  - RAM access over high-speed bus
  - I/O access over lower-speed bus
  - Less competition
  - More hardware (more expensive…)





This port allows I/O devices access into memory

Memory-Mapped vs. separate I/O Space



Separate I/O & memory space

Memory-mapped I/O

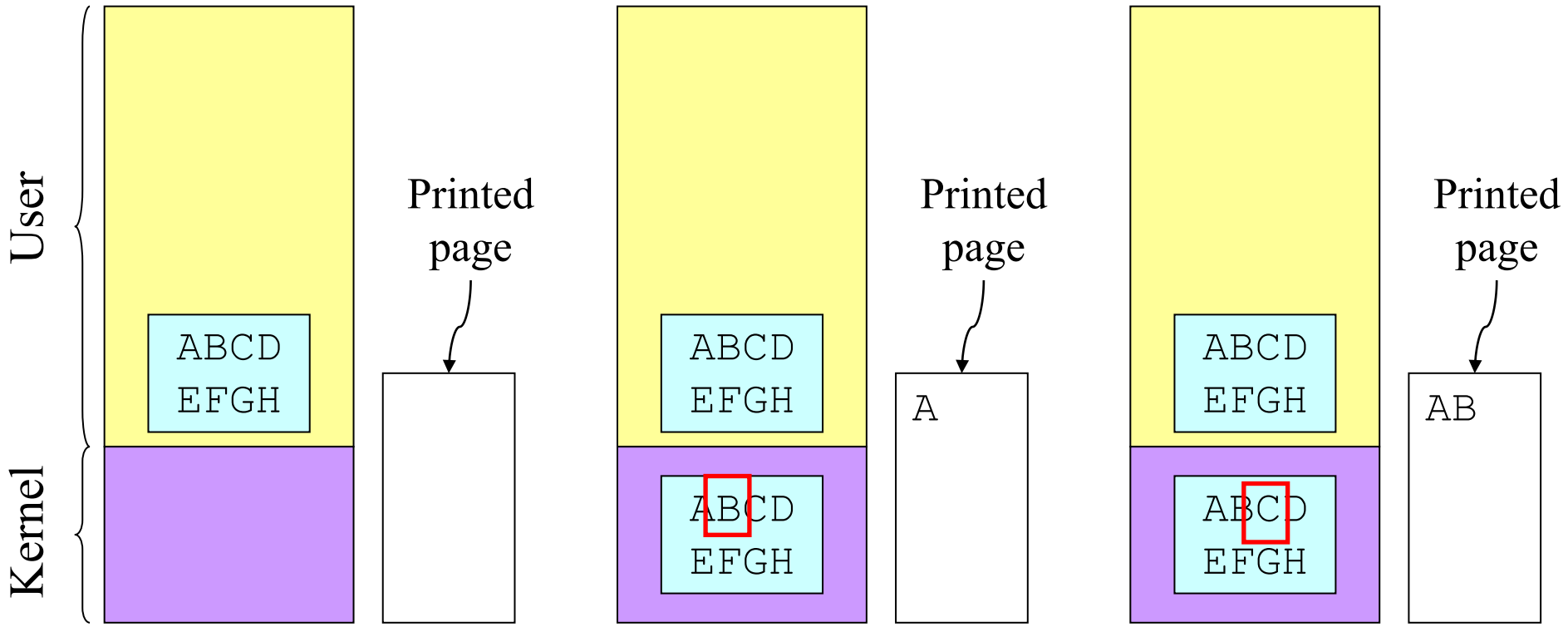
Hybrid: both memory-mapped & separate spaces

Example: Dynamic Frequency on XScale

```
// Allocate some space for the virtual reference to CCCR
LPVOID virtCCCR = VirtualAlloc(0, sizeof(DWORD), MEM_RESERVE, PAGE_NOACCESS);

//0x41300000 is the memory-mapped location of the CCCR register
LPVOID CCCR = (LPVOID)(0x41300000 / 256); // shift by 8 bits for ability to address 2^40 bytes

// Map writing the virtual pointer to the physical address of the CCCR register
VirtualCopy((LPVOID)virtCCCR, CCCR, sizeof(DWORD), PAGE_READWRITE | PAGE_NOCACHE | PAGE_PHYSICAL);

// Set the CCCR register with the new speed
*(int *)virtCCCR = new_speed;

// Call the assembly function to actually perform the switch
doSwitch(0x02 | 0x01); //0x02 means turbo mode, 0x01 means the clock is being switched

// Clean up memory by freeing the virtual register.
VirtualFree(virtCCCR, 0, MEM_RELEASE);
virtCCCR = NULL;
```

```
; Coprocessor 14, register C6 (CLKCFG) initiates the changes programmed in CCCR
; when CLKCFG is written.

doSwitch
      MOV r3, r0                ; Move r0, the argument to doSwitch, into register r3
      MCR p14, 0, r3, c6, c0, 0 ; Copy the contents of r3 into register c6 on coprocessor 14.
      MOV pc, lr                ; return execution to where it last left off
```

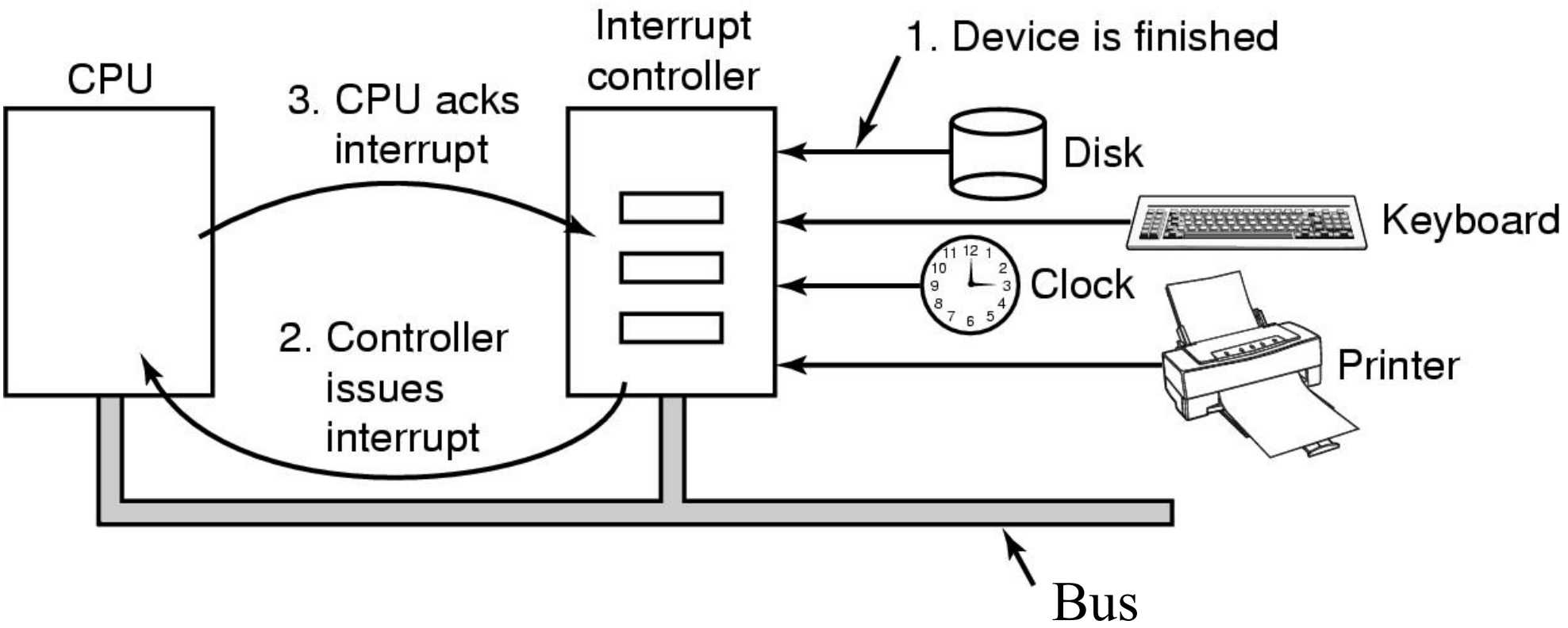
Programmed I/O example: printing a page



Polling

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

Hardware's view of interrupts



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

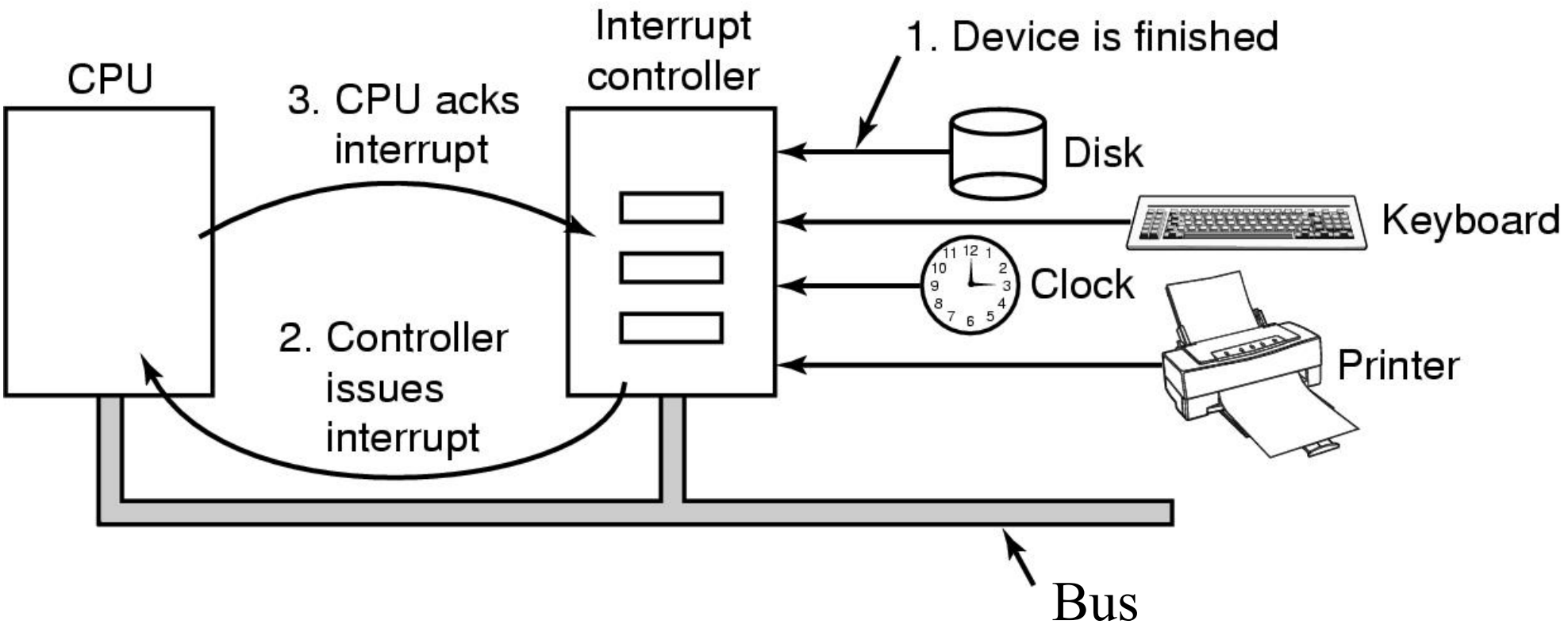
```
if (count == 0) {
  unblock_user();
} else {
  j++;
  *printer_data_reg = p[j];
  count--;
}
acknowledge_interrupt();
return_from_interrupt();
```

Code run at interrupt time
(Interrupt handler)

Polling

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

Hardware's view of interrupts



Interrupt-driven I/O

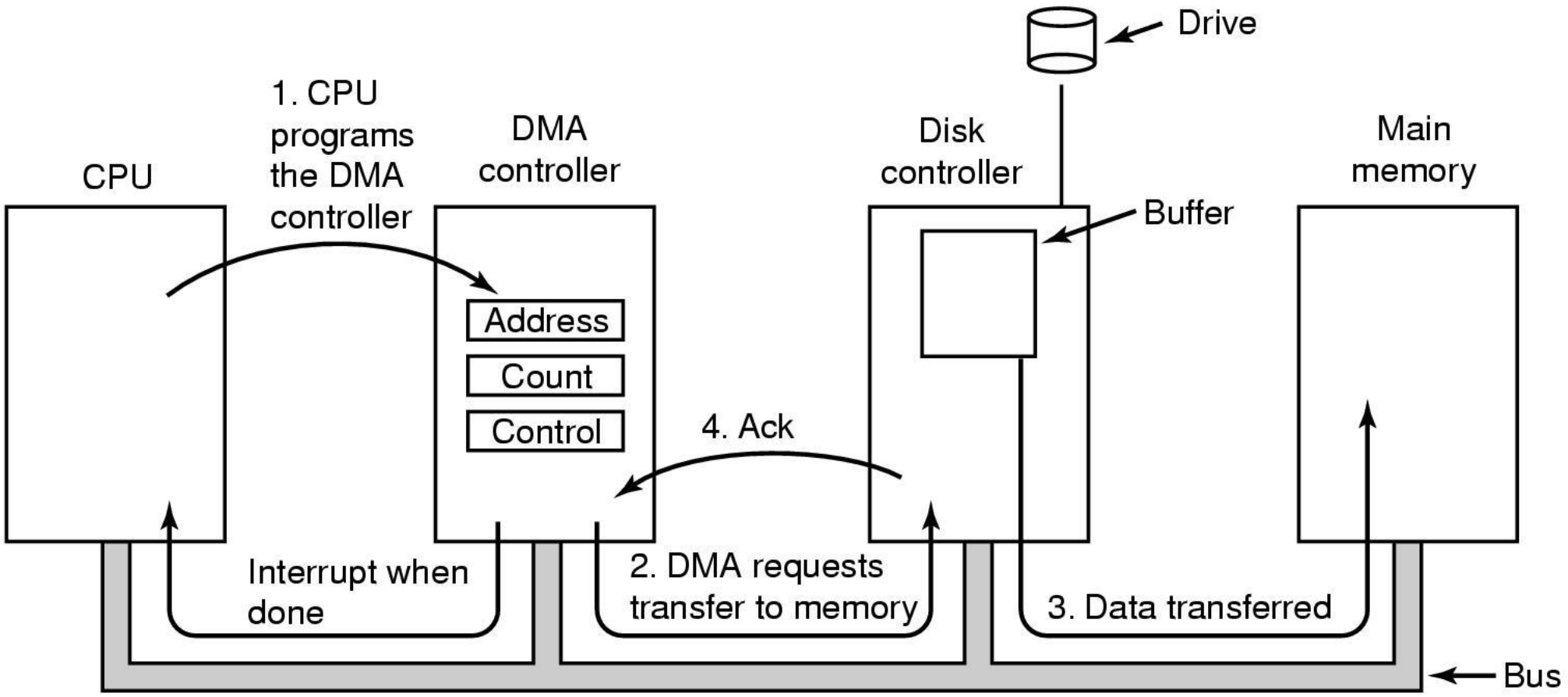
```
copy_from_user (buffer, p, count);
j = 0;
enable_interrupts();
while (*printer_status_reg != READY)
  ;
*printer_data_reg = p[0];
scheduler(); // and block user
```

Code run by system call

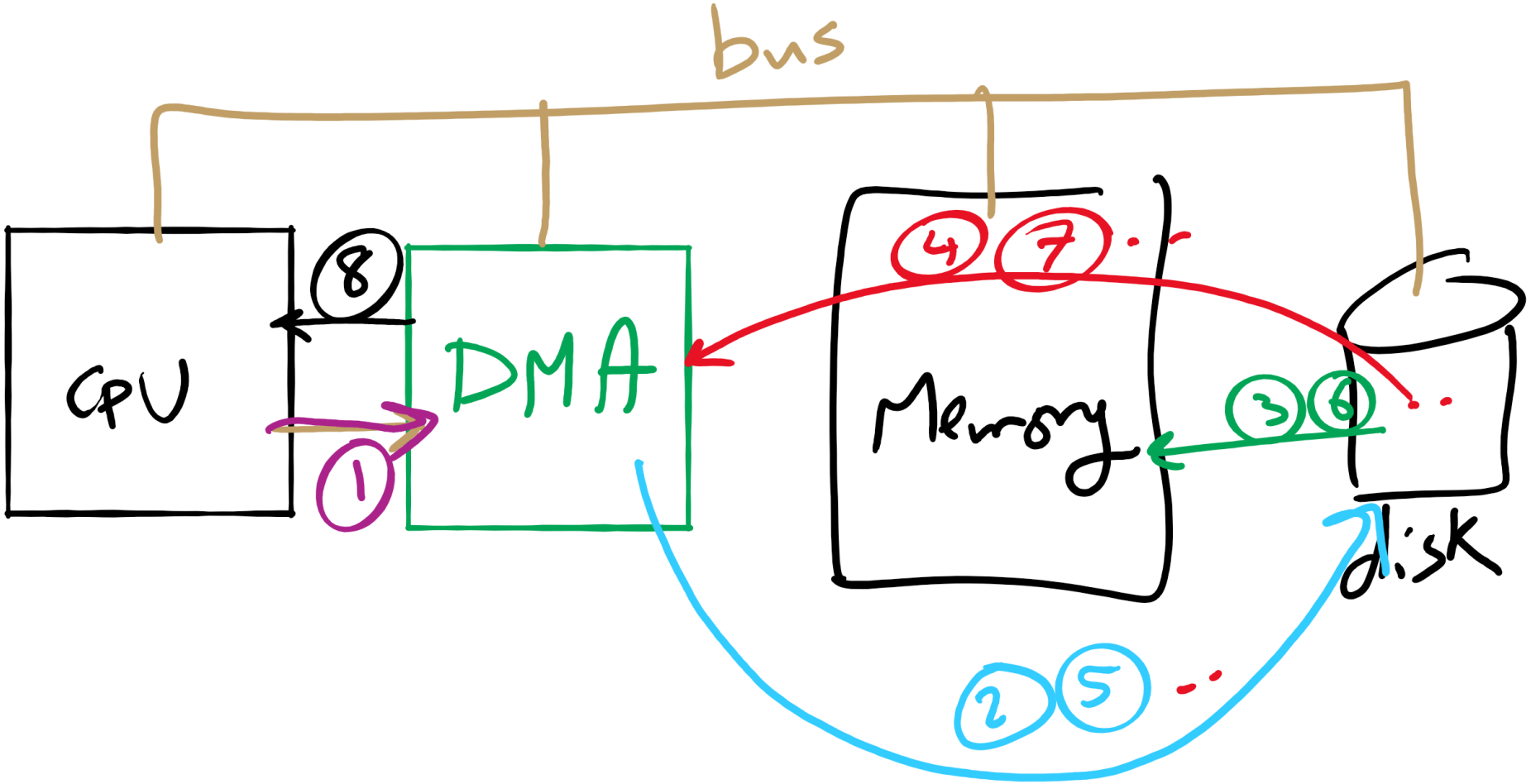
```
if (count == 0) {
  unblock_user();
} else {
  j++;
  *printer_data_reg = p[j];
  count--;
}
acknowledge_interrupt();
return_from_interrupt();
```

Code run at interrupt time
(Interrupt handler)

Direct Memory Access (DMA) operation



DMA



I/O using DMA

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

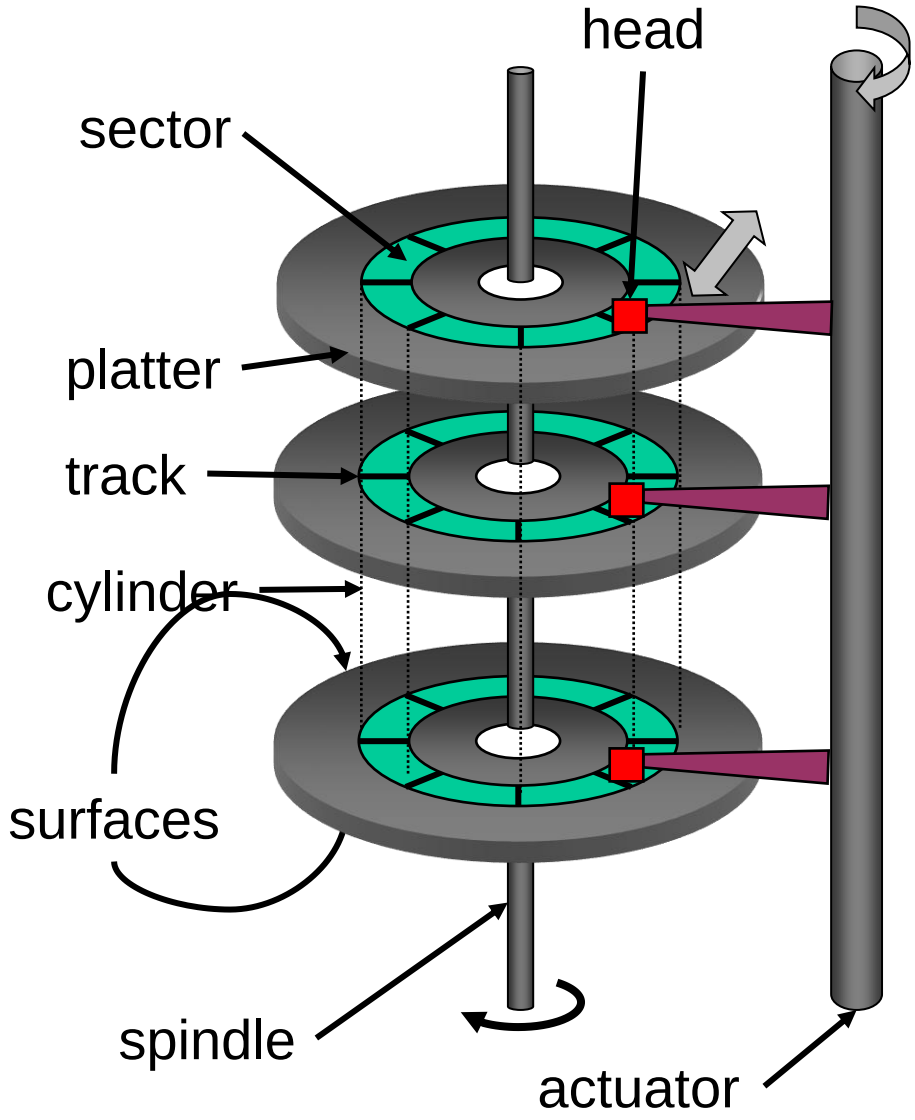
Code run by system call

```
acknowledge_interrupt();
unblock_user();
return_from_interrupt();
```

Code run at interrupt time

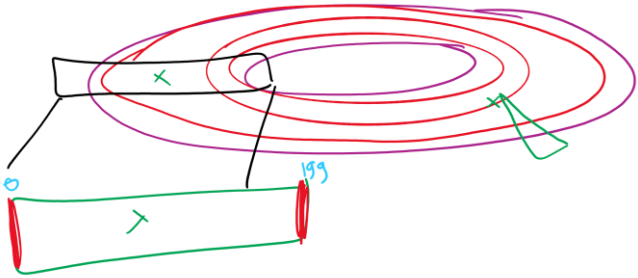
Disk drive structure

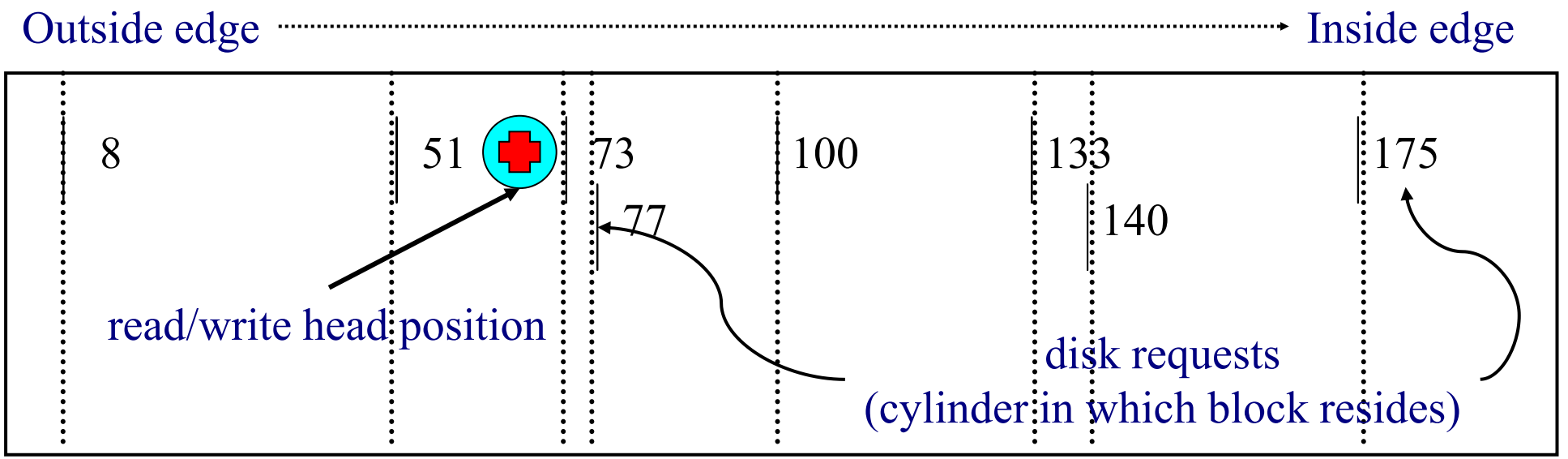
- Data stored on surfaces
  - Up to two surfaces per platter
  - One or more platters per disk
- Data in concentric tracks
  - Tracks broken into sectors
    - 256B-1KB per sector
  - Cylinder: corresponding tracks on all surfaces
- Data read and written by heads
  - Actuator moves heads
  - Heads move in unison



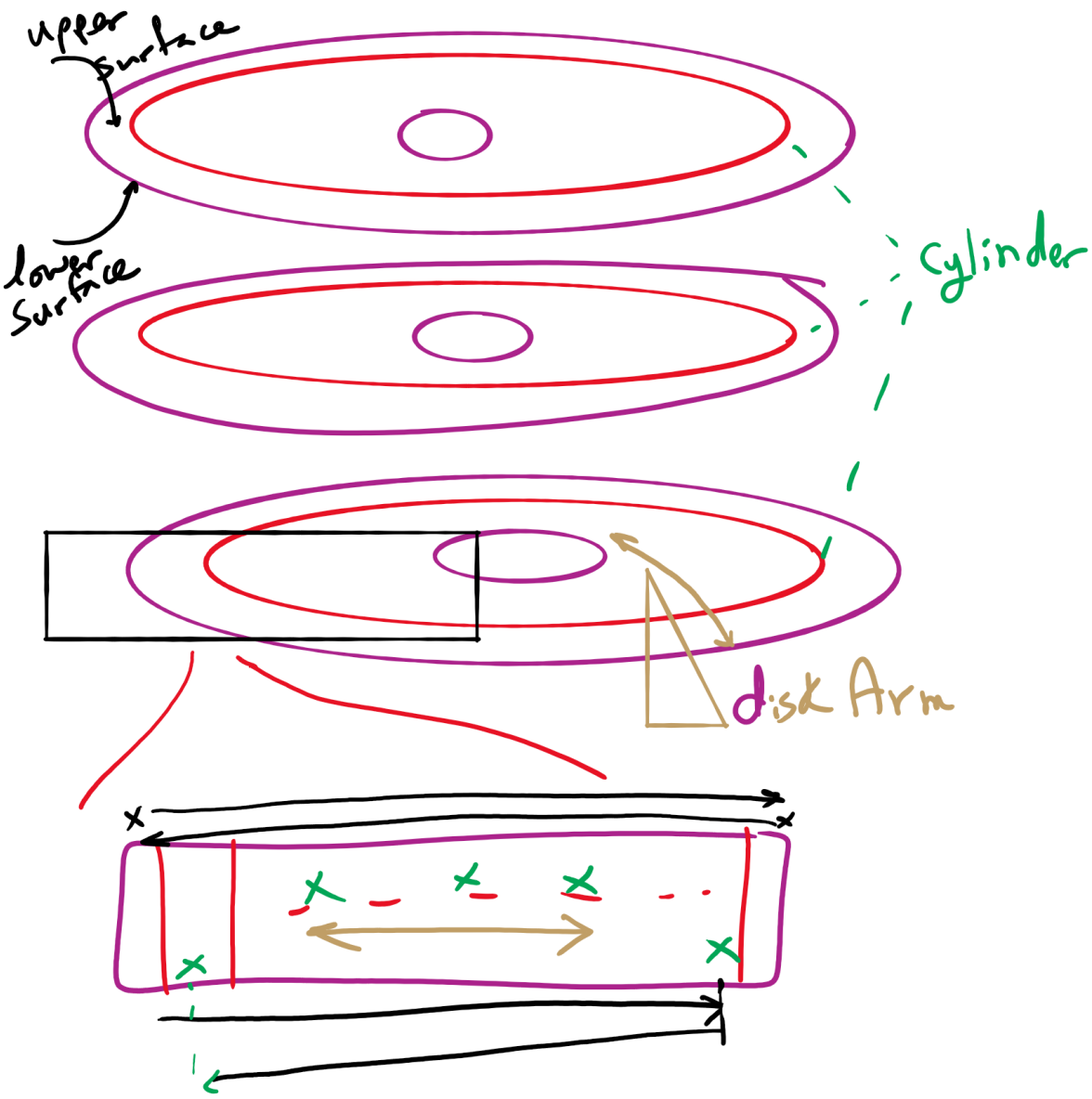
Disk scheduling algorithms

- 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



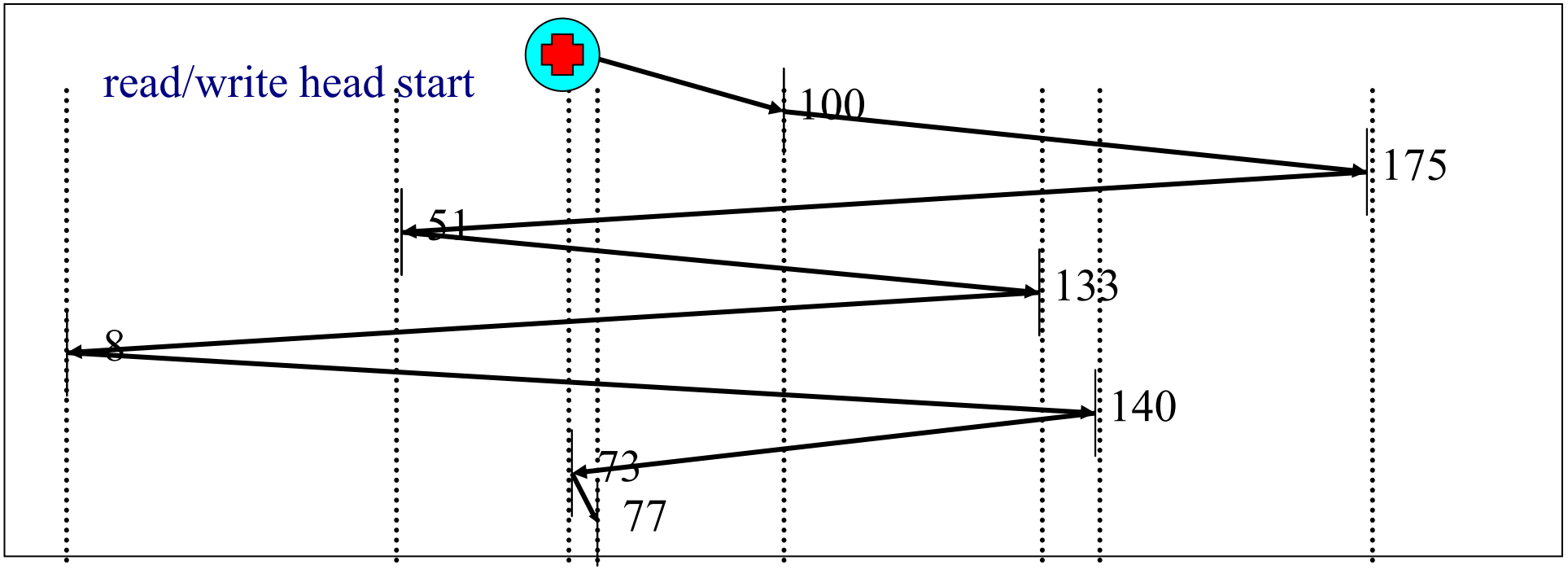


Disk Arm Scheduling

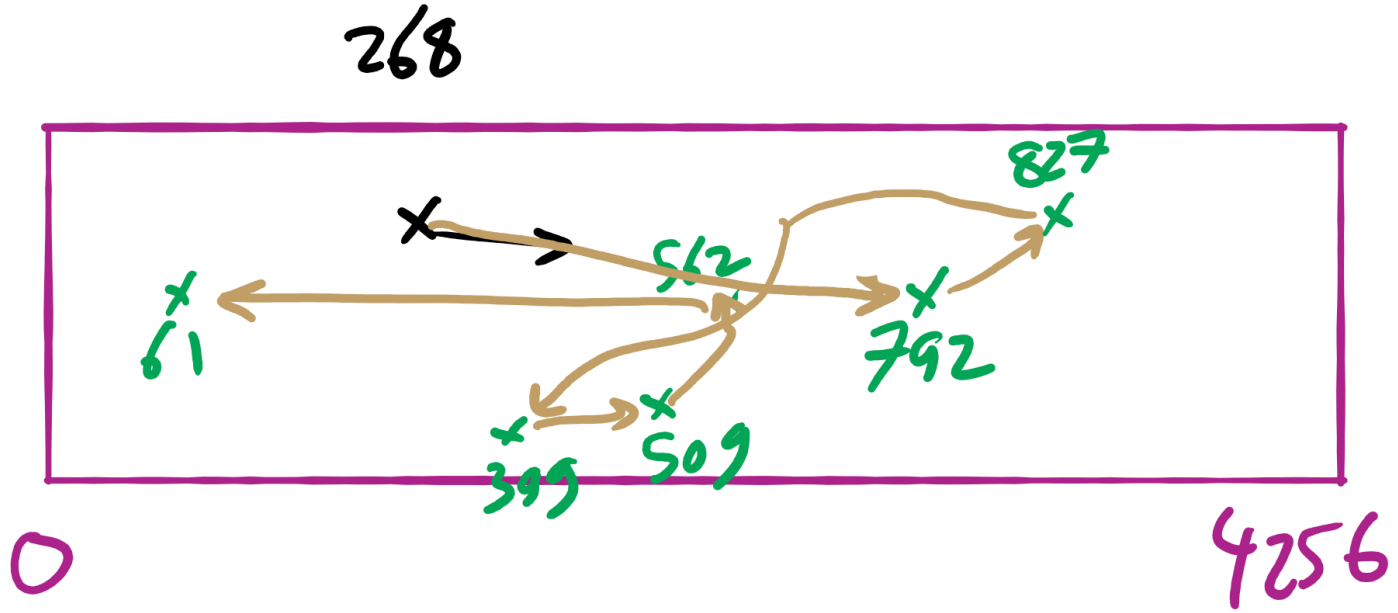


First-Come-First Served (FCFS)

- 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



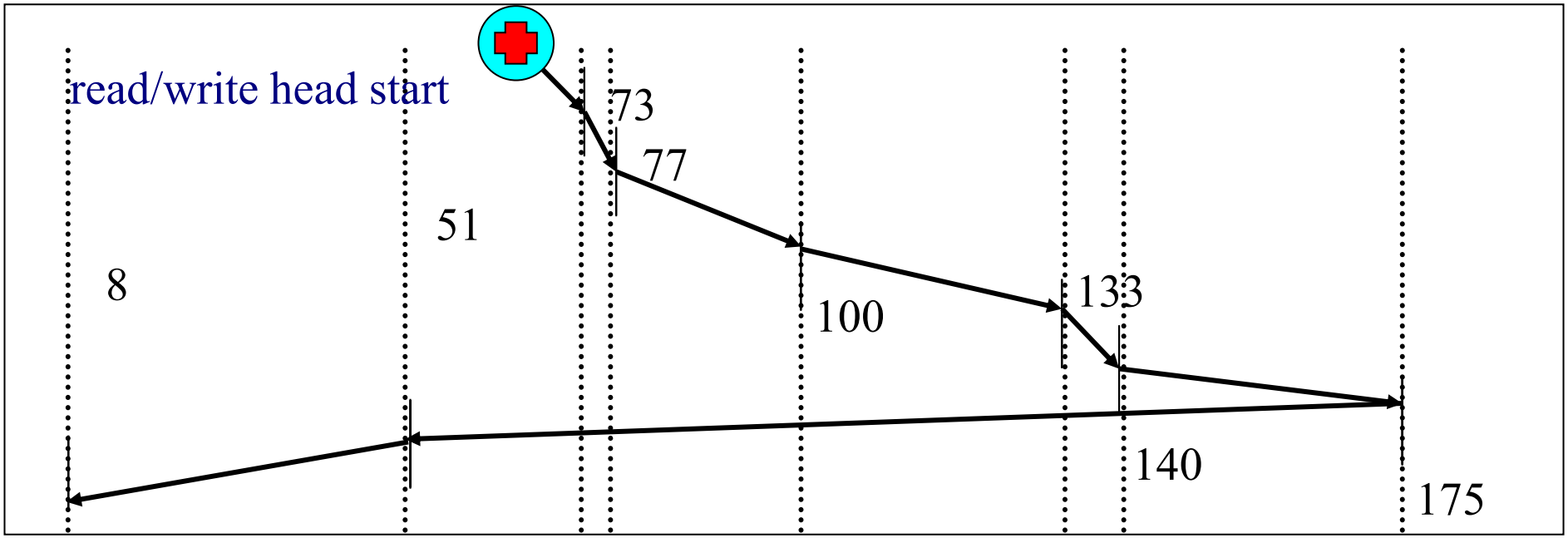
FCFS Example 2



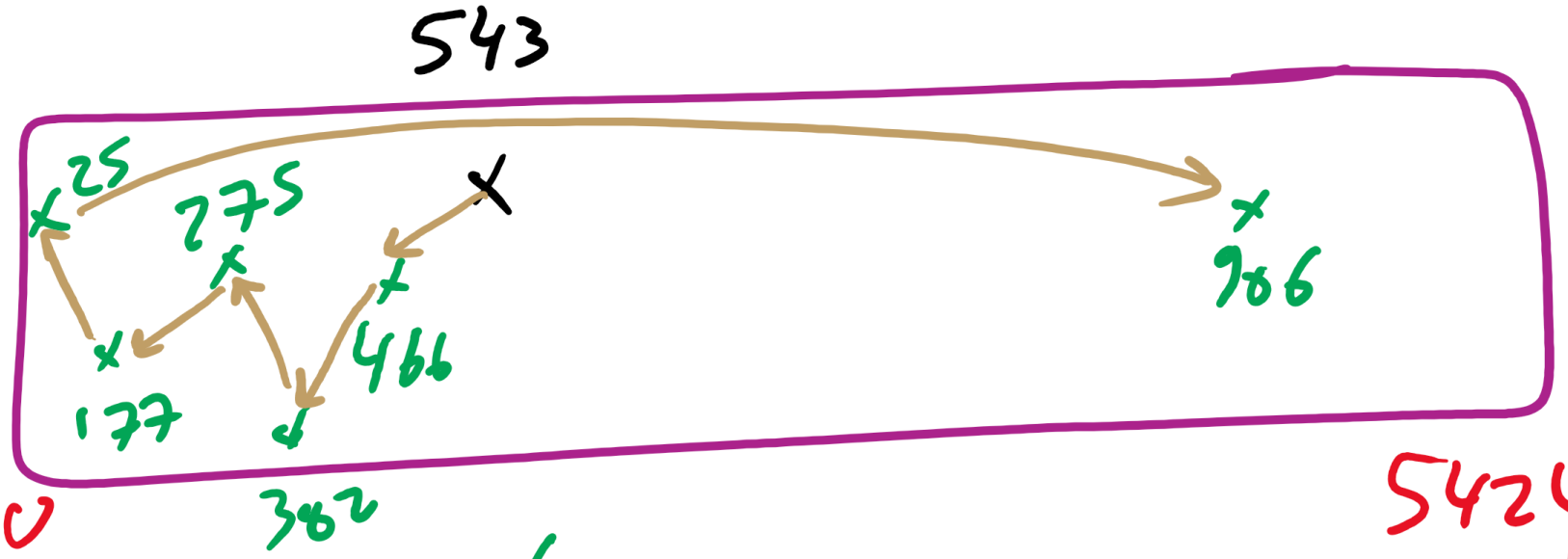
Total Seek distance = (792-268) + (827-792)
+ (827-399) + (509-399) +
(562-509) + (562-61)

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, 100, 133, 140, 175, 51, 8
- Seek distance = 10 + 4 + 23 + 33 + 7 + 35 + 124 + 43 = 279 cylinders



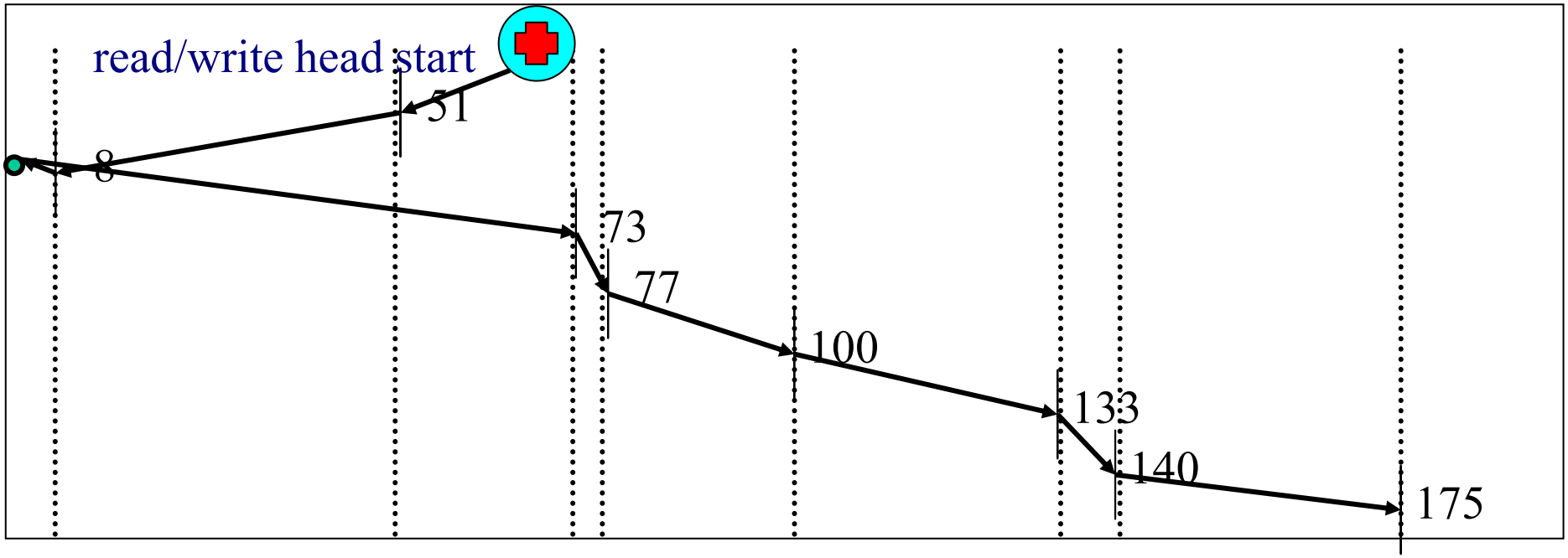
SSTF Example 2



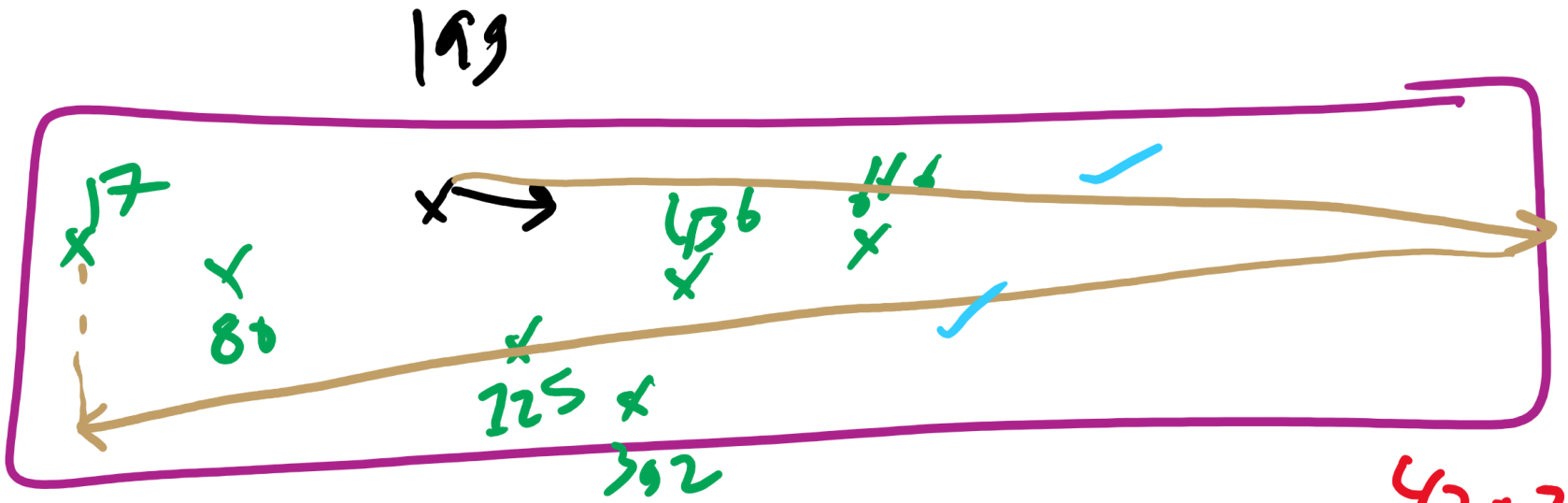
Total Seek distance = (543 − 466) + (466 − 382) + (382 − 275) + (275 − 177) + (177 − 25) + (906 − 25)

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 cyls



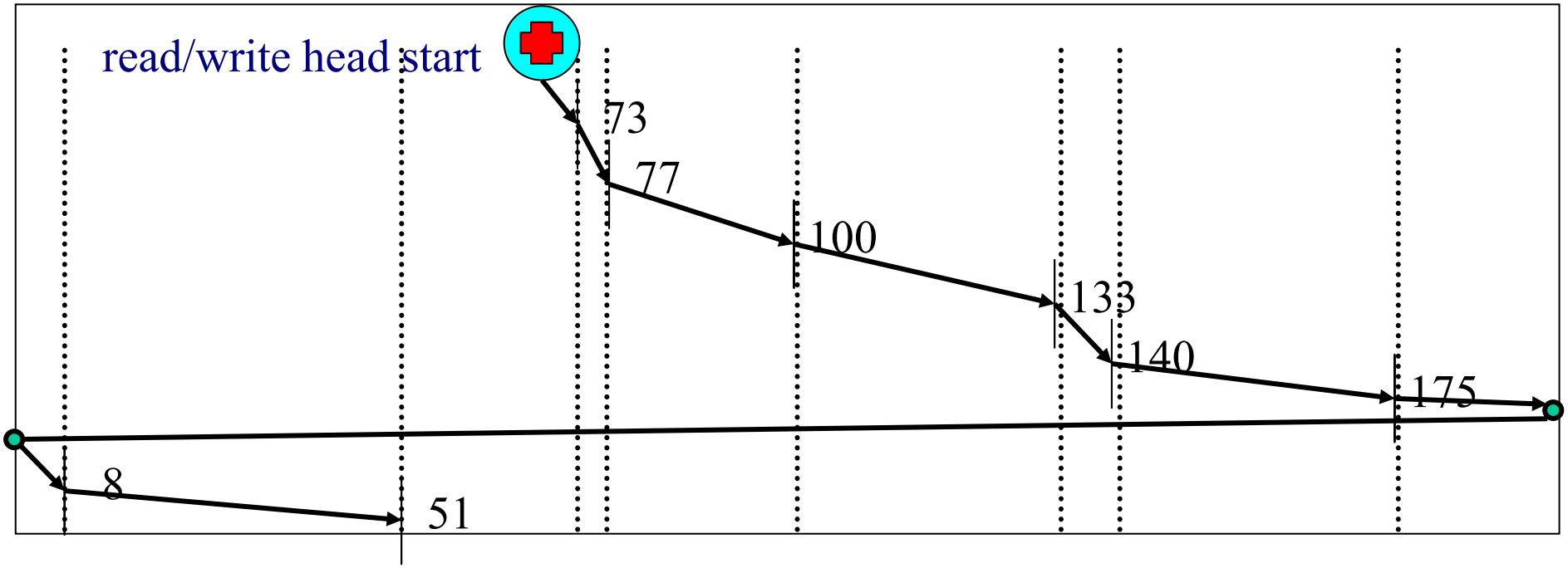
SCAN Example 2



Total Seek distance = (4303 − 199) + (4303 − 0)

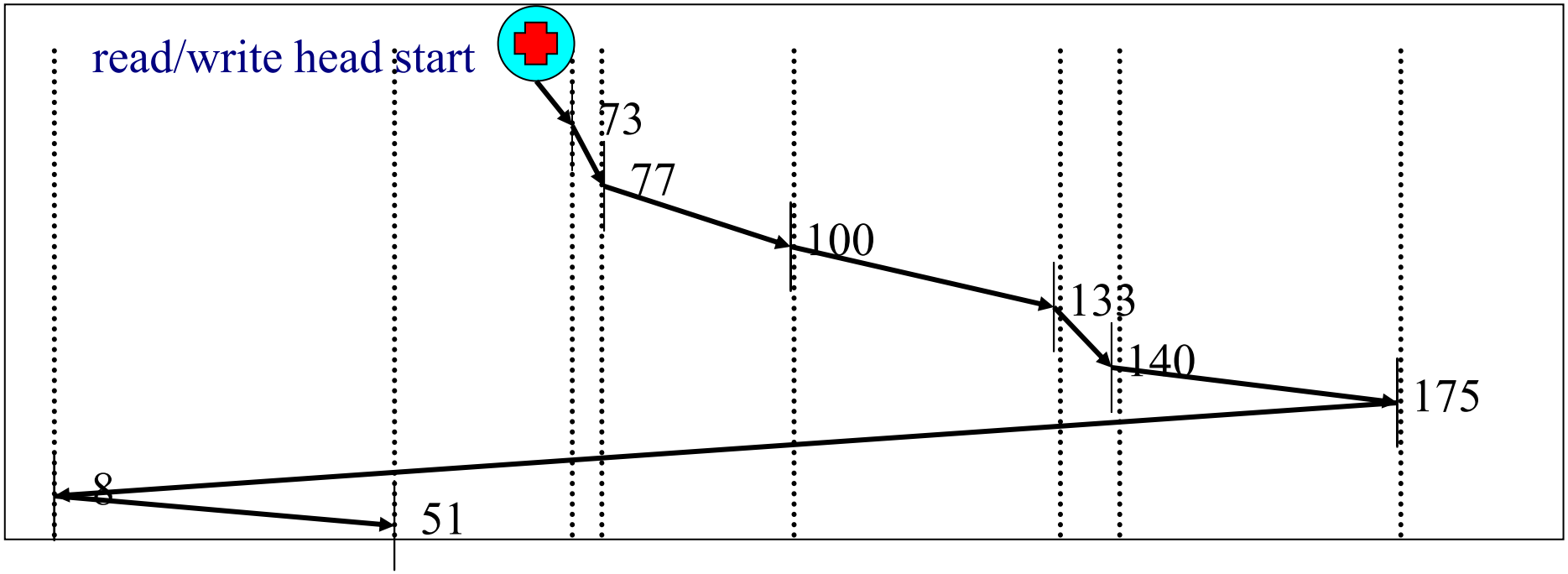
C-SCAN

- 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

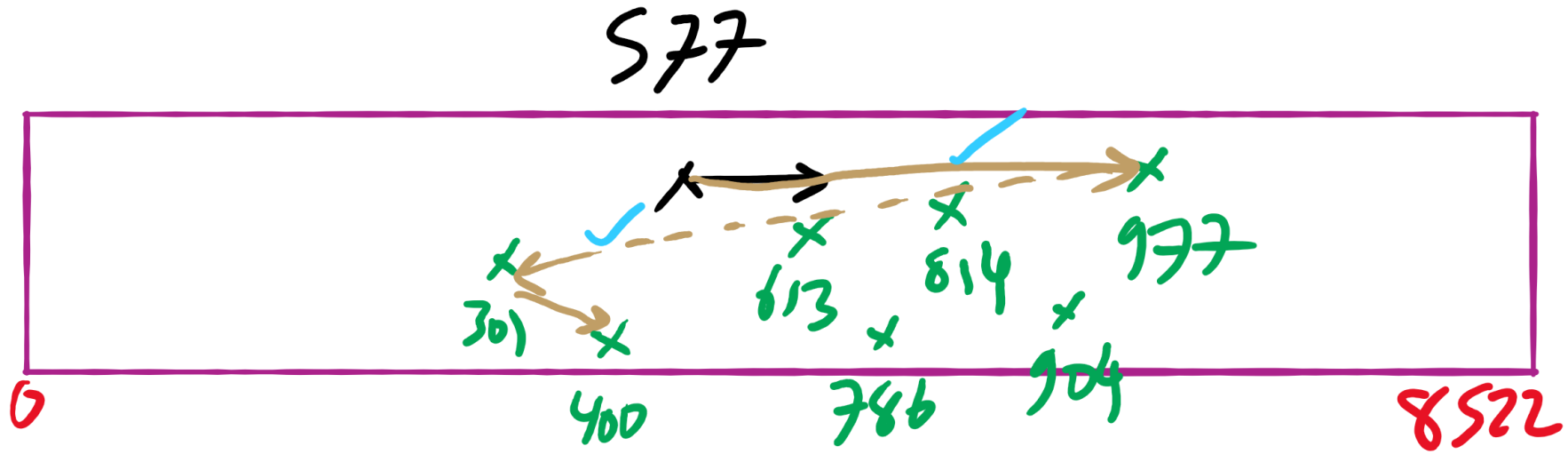


C-LOOK

- 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



C-LOOK Example 2

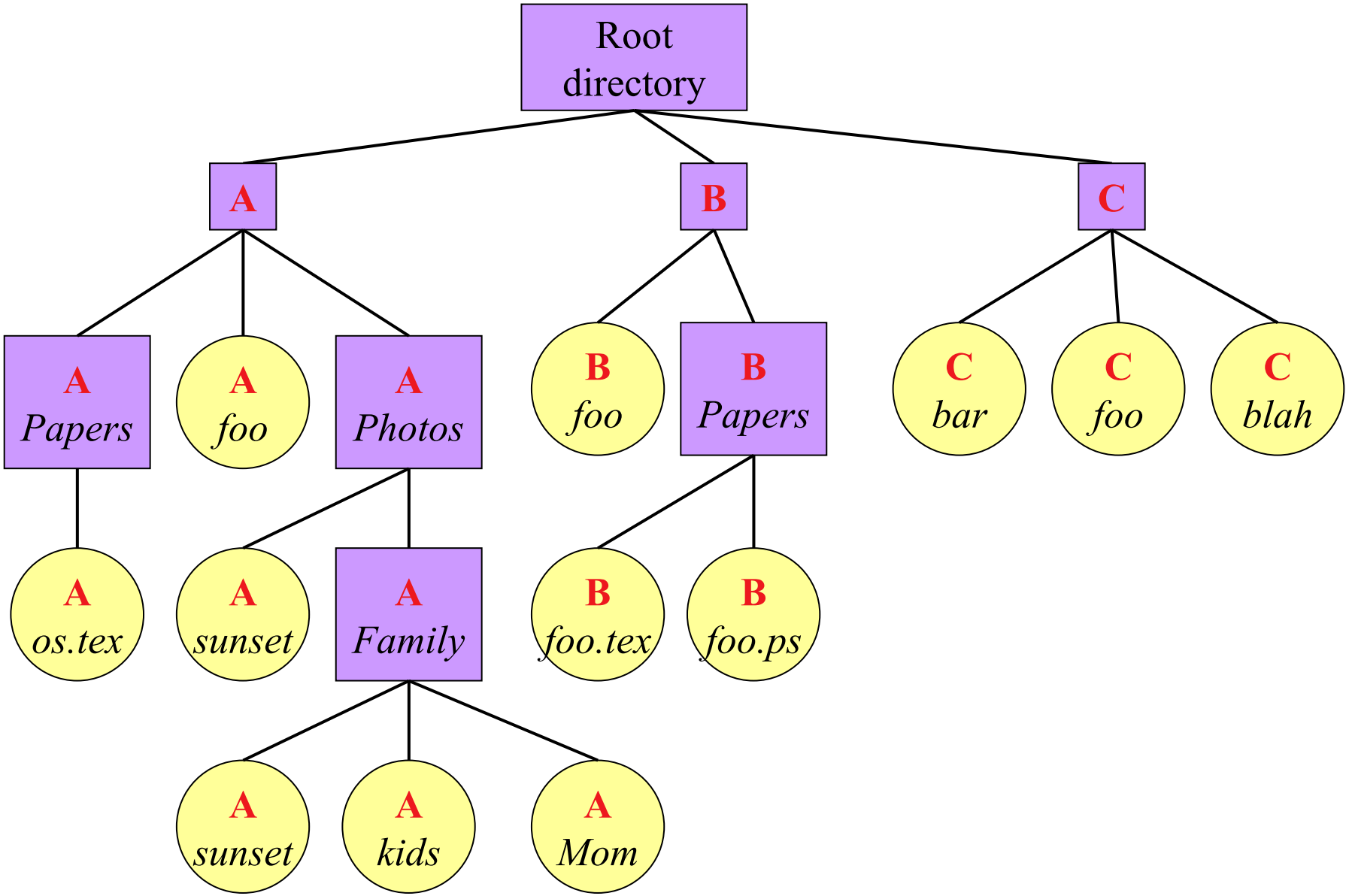


Total Seek distance = (977 − 577) +
(977 − 301) +
(400 − 301)

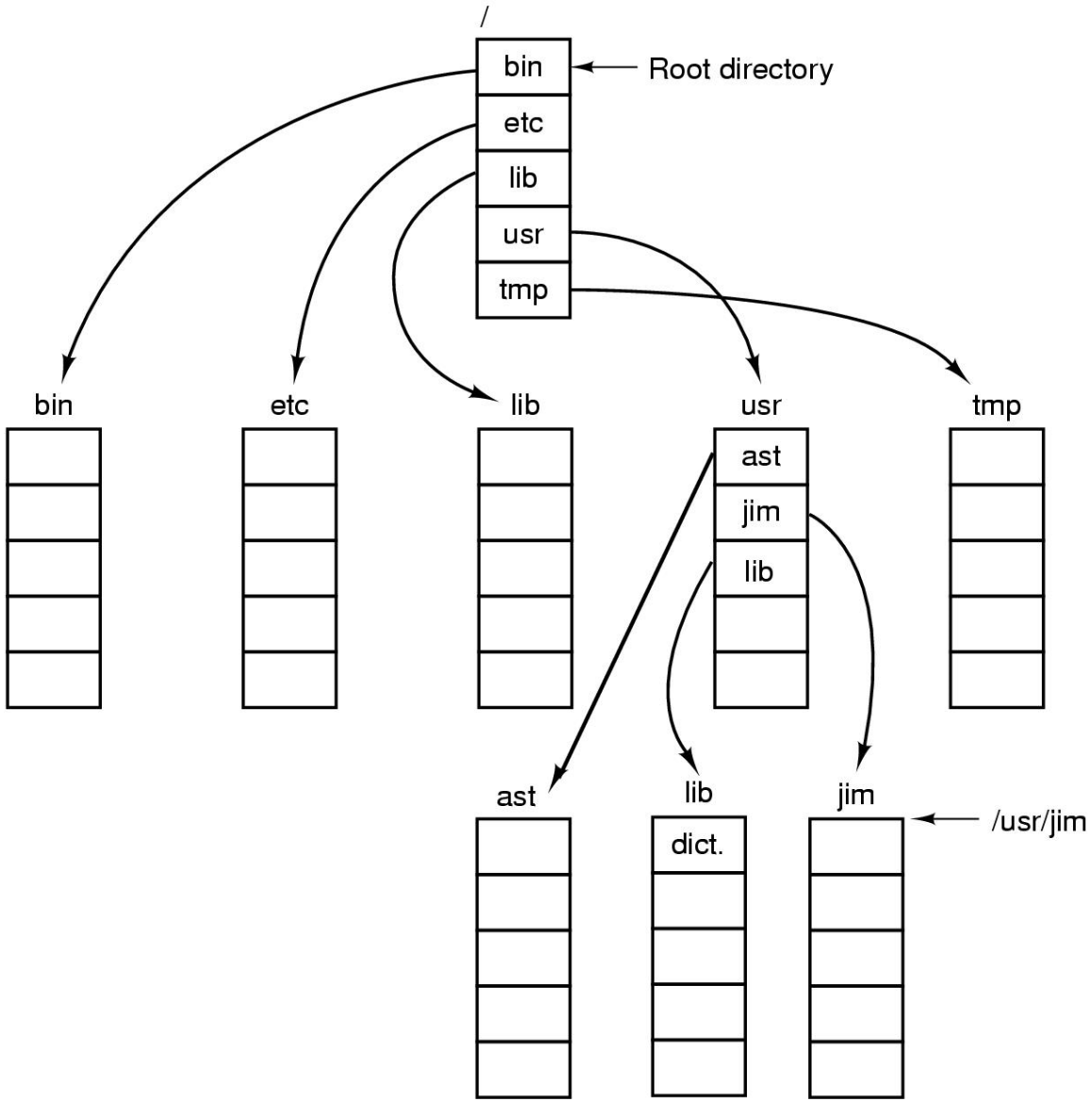
Miscellaneous Issues

- File Sharing
- Journalling File System
- I/O Buffering
- More Kernel-level data structures

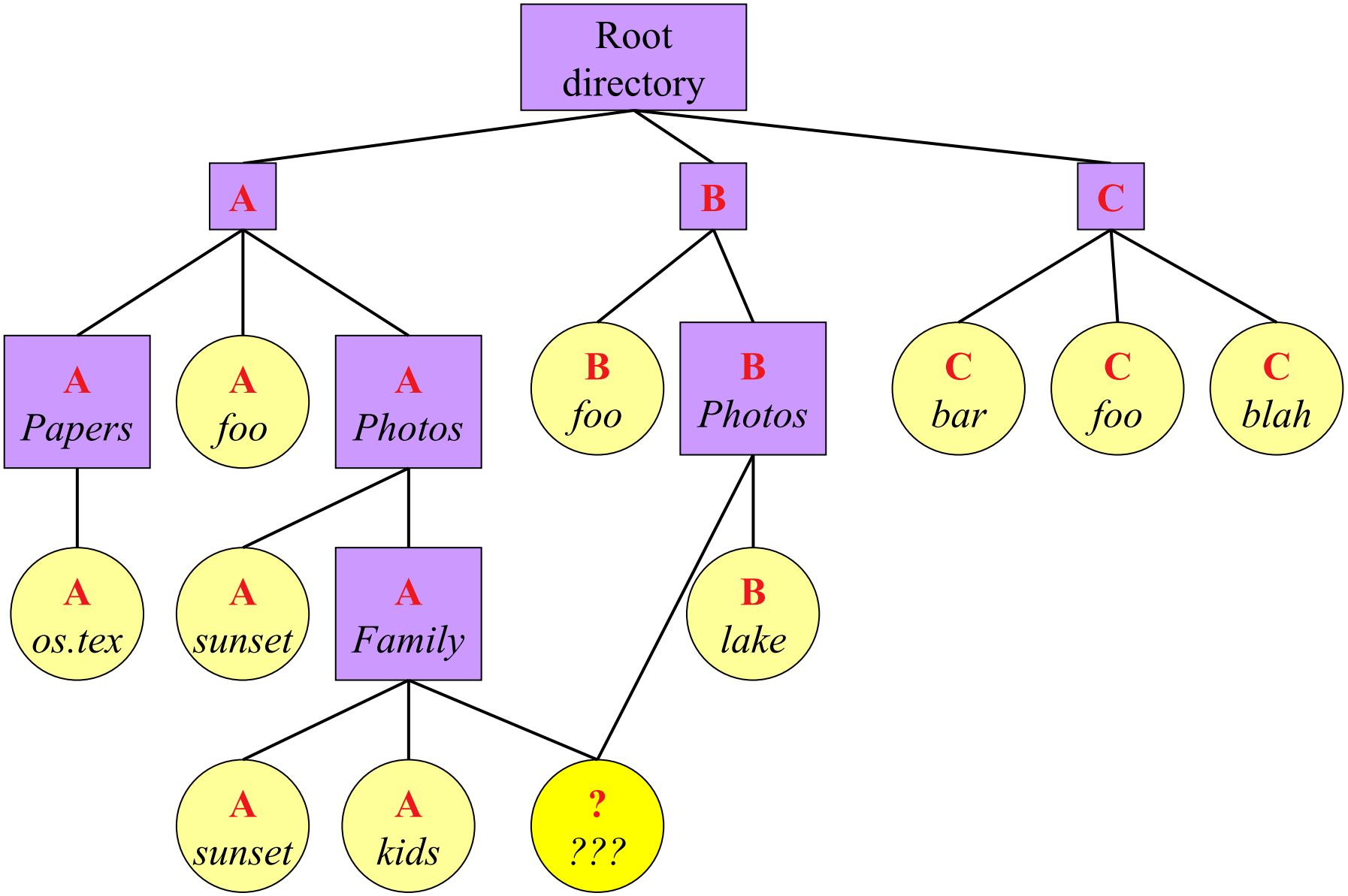
Hierarchical directory system



Unix directory tree

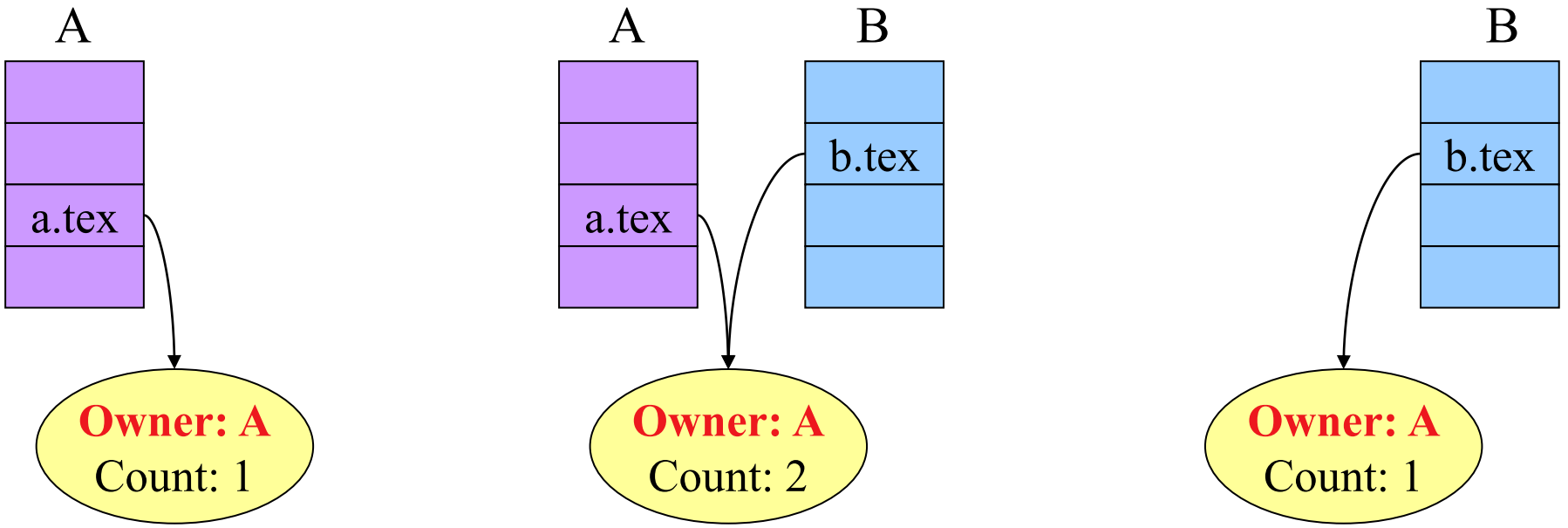


Sharing files

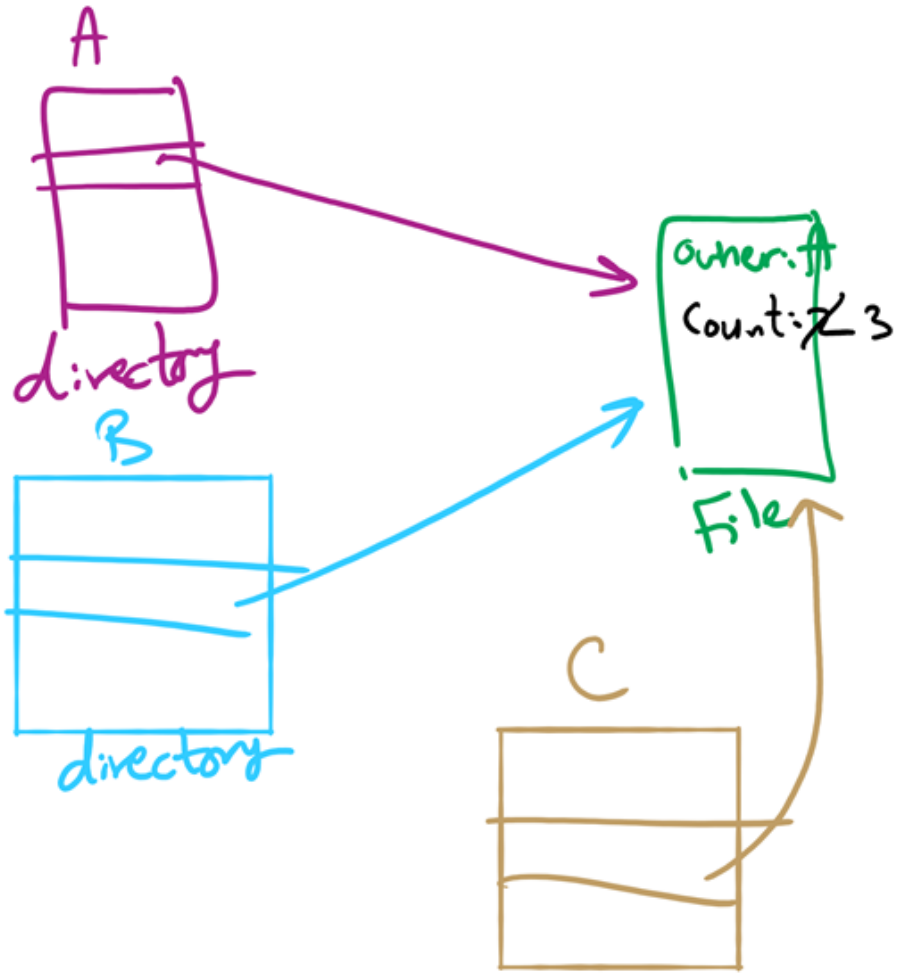


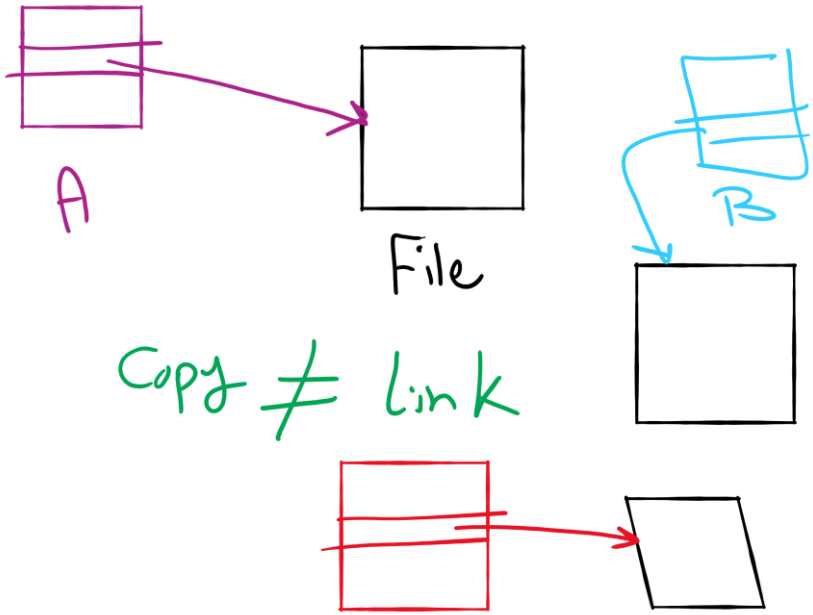
Solution: use links

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

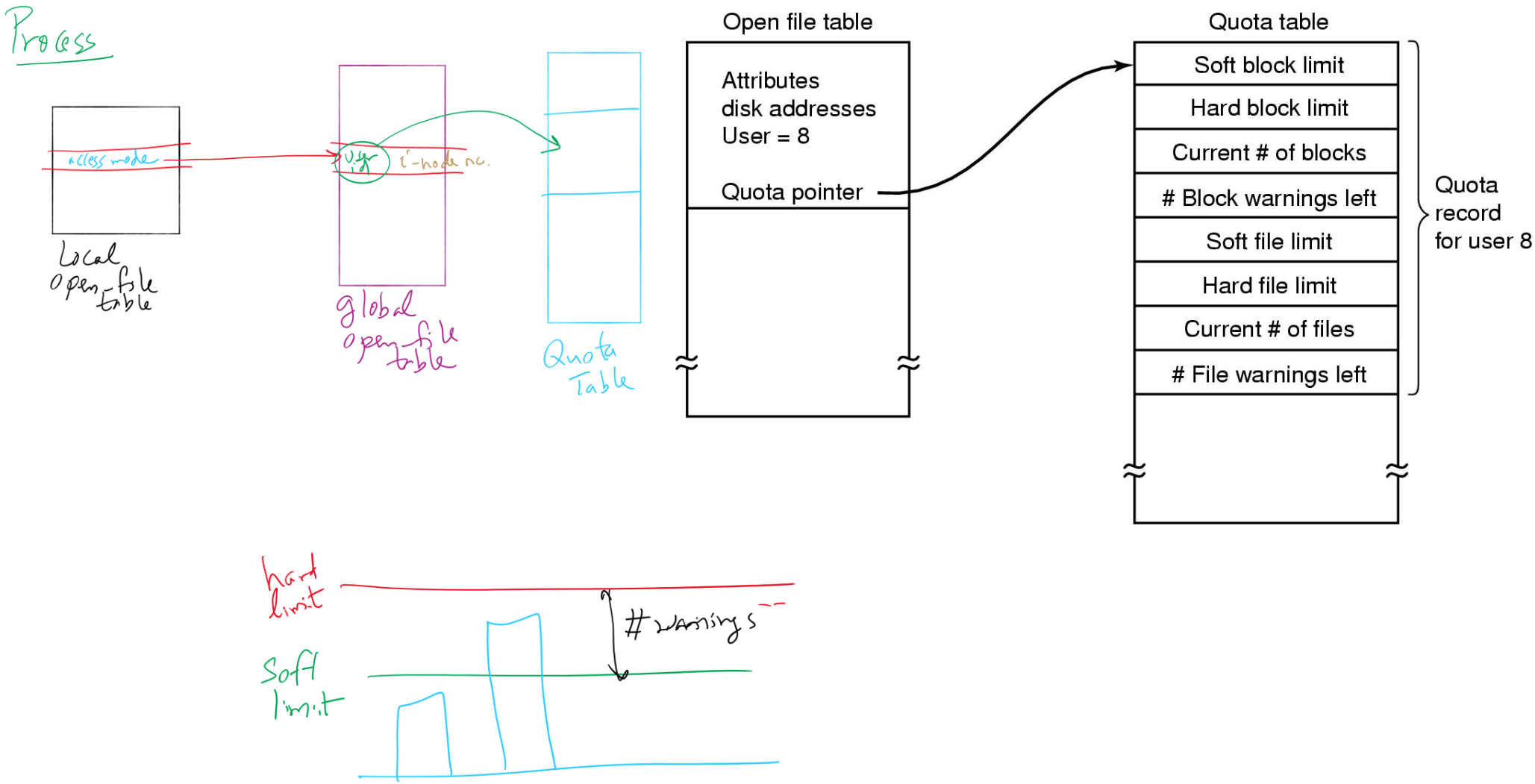


File Linking (left) vs. File Copying (right)





File-related kernel structures: open file tables and disk quotas



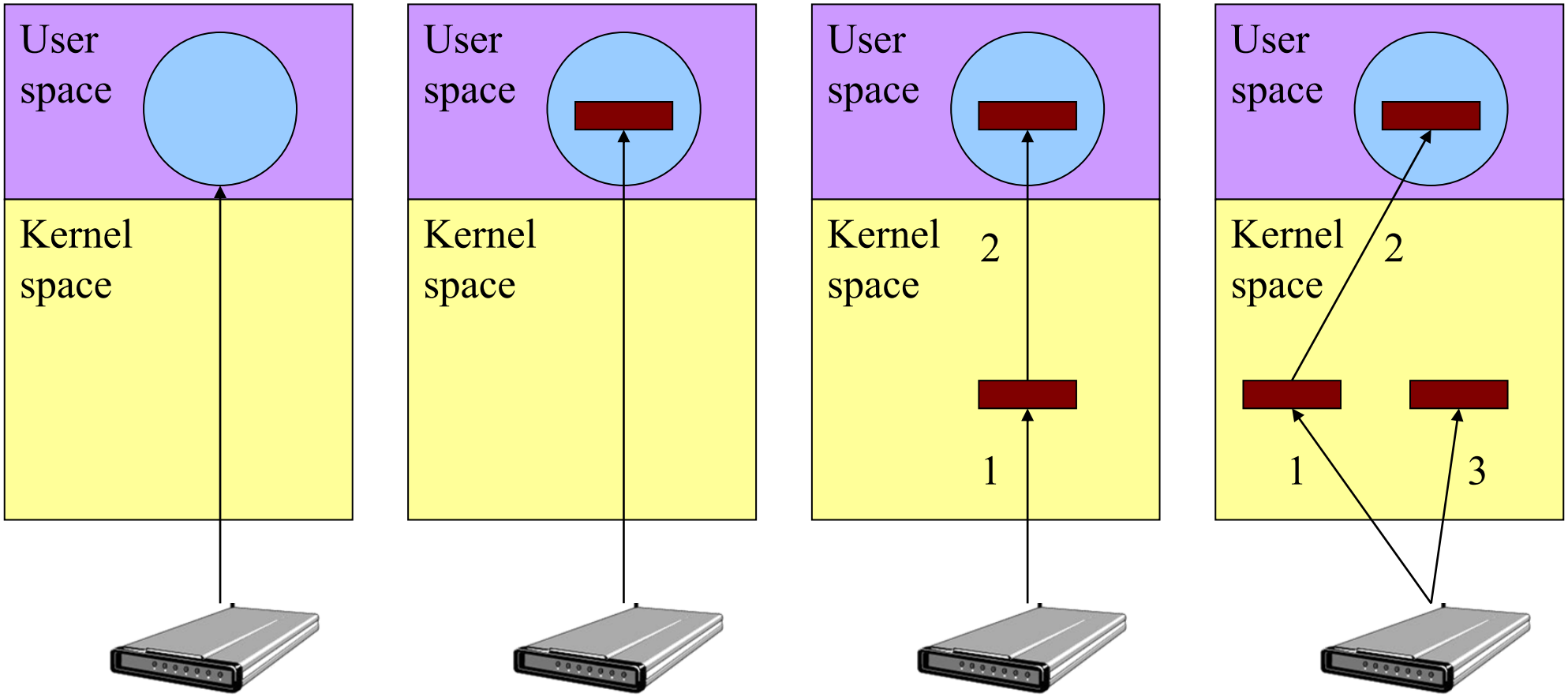
Journaling File System

- In a regular file system, changes to files and directories result in multiple separate writes to disk
  - prone to power failures
- Write the changes twice
  - first to an on-disk *journal*
    - for efficiency, journal can be put on SSD or NVRAM
    - modified data itself may or may not be written to the journal
  - second to main file system
- Examples
  - Windows NTFS
  - Linux ext3, ext4

Journaling File System

- Interaction of disk arm scheduling?
  - out-of-order writes
  - ext3 and ext4 force disk to flush its cache at certain points (barriers)
- Journaling vs. Log-structured file system
  - Journaling is not needed in LFS

Buffering device input



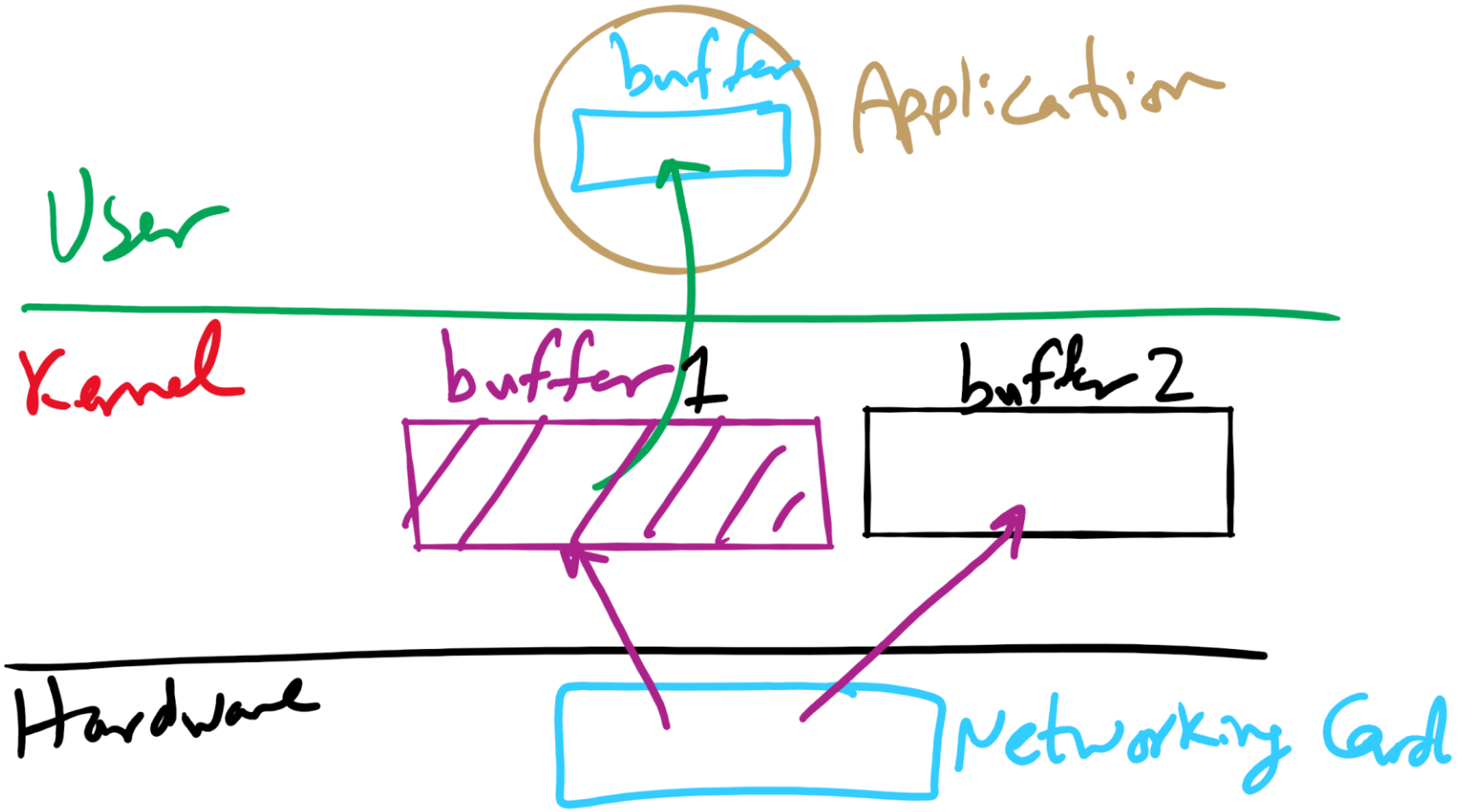
Unbuffered input

Buffering in user space

Buffer in kernel
Copy to user space

Double buffer in kernel

I/O Buffering



Max Partition Size

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

Max Partition Size

32-bit block no. (block no. size)

Max. Partition Size = 2^{32} * block size

FAT-12 : $2^{12} * \frac{1}{2}$ KB

$= 2^{12} * 2^{9} = 2^{21} = 2$ MB

FAT-16 : $2^{16} * 2$ KB

$= 2^{16} * 2^{11} = 2^{27} = 128$ MB

FAT-32 : $2^{28} * 4$ KB

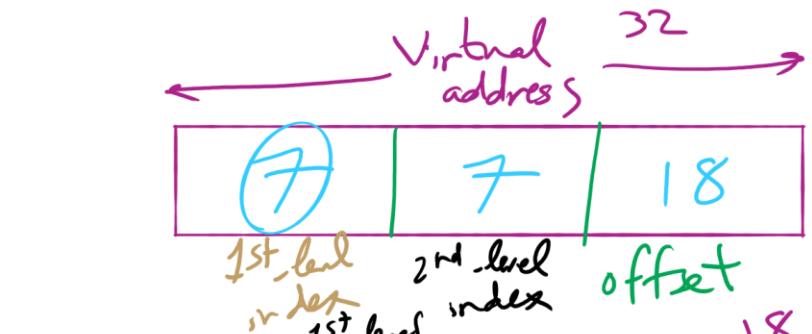
$= 2^{28} * 2^{12} = 2^{40} = 1$ TB

sector no. size

$2^{32} * 512 = 2^{32} * 512$

$= 2^{41} = 2$ TB

HW 10: Q 2-4

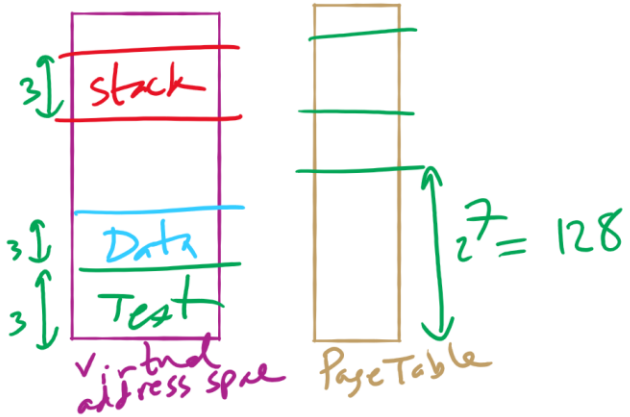


PDEs = $2^{\text{1st\_level index size}}$ 2^8 KB = 2^{18} bytes

$= 2^7$

PTEs in 2nd_level Page Table = $2^{\text{2nd\_level index size}}$

$= 2^7$



HW 10: Q 10-13

single-level Page Table

$EAT = h(a+m) + (1-h)(a+m+m)$

2-level Page Table

$EAT = h(a+m) + (1-h)(a+m+m+m)$

(PID, VPN)

(30, 1)

PageNo

0x1200

offset 12

4 KB = 2

⇒ offset : 12 bits

Frame no. : 3

3x200

3200

PID

1

(10, 7)

0x4500

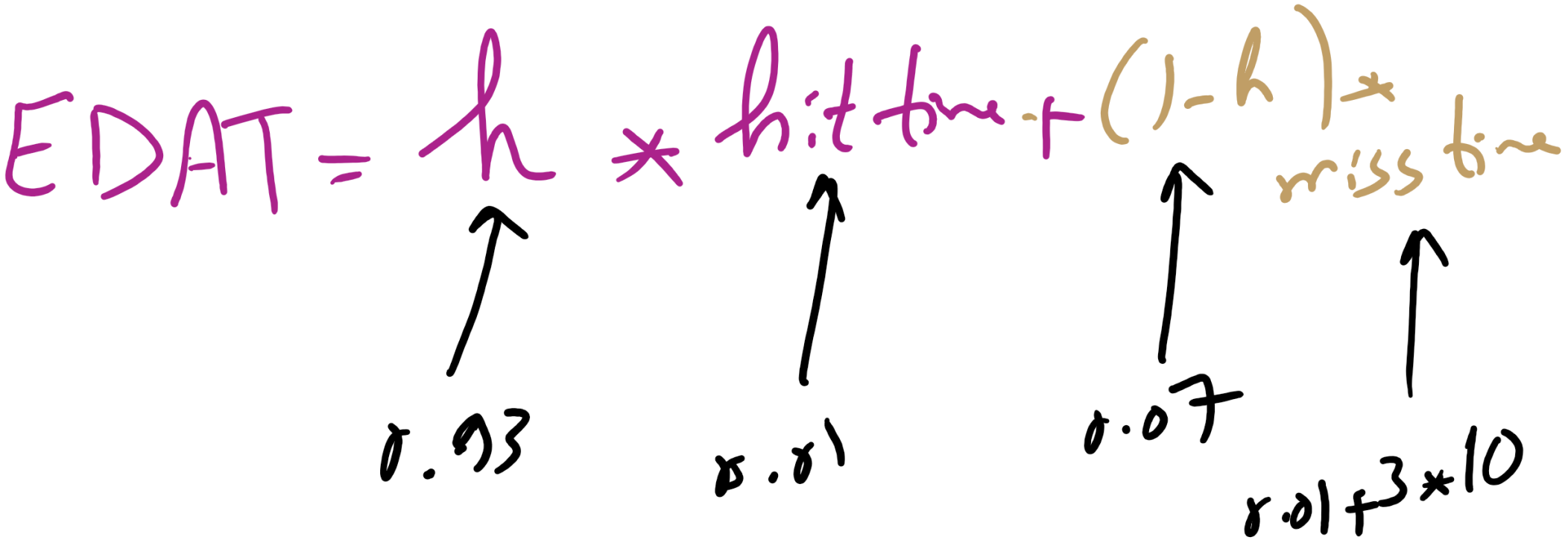
offset

Frame no.

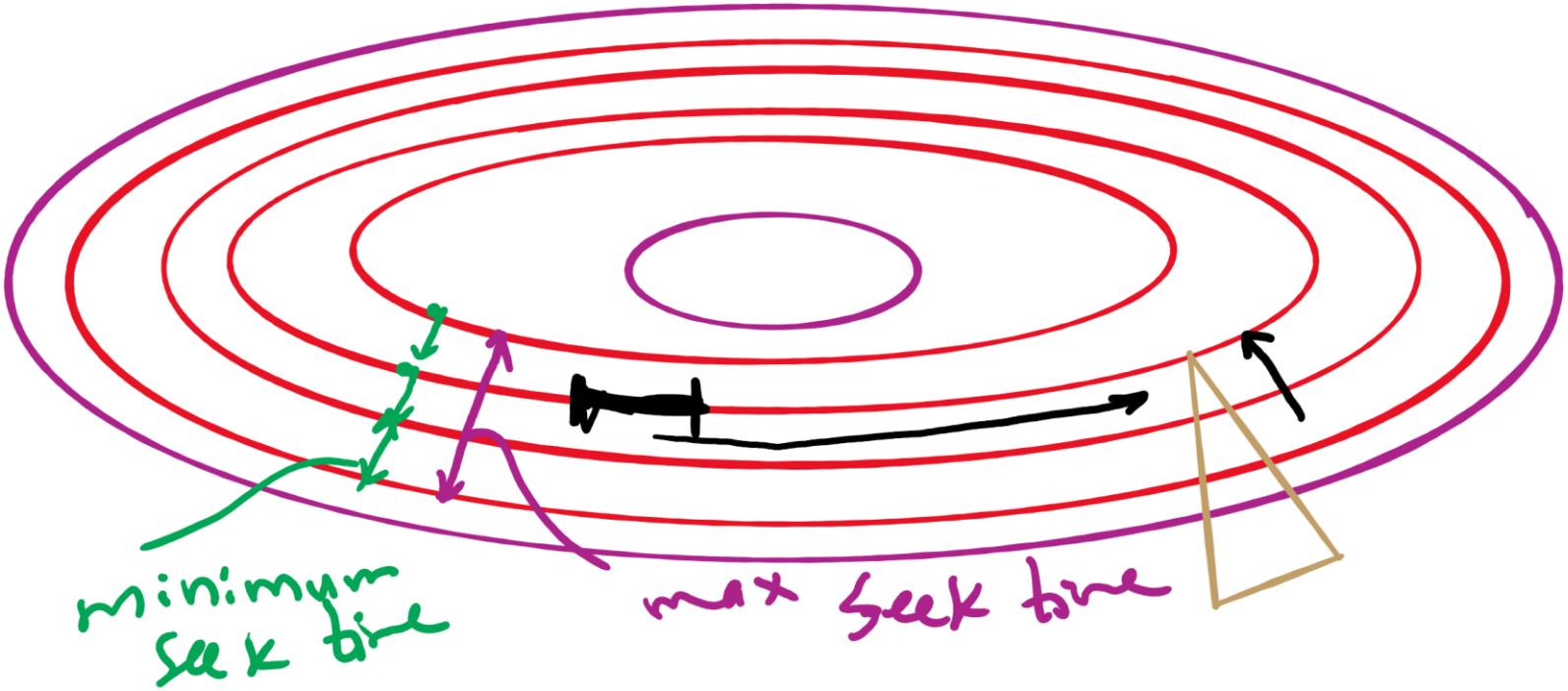
Frame no. : 4

0x7500

Effective Disk Access Time



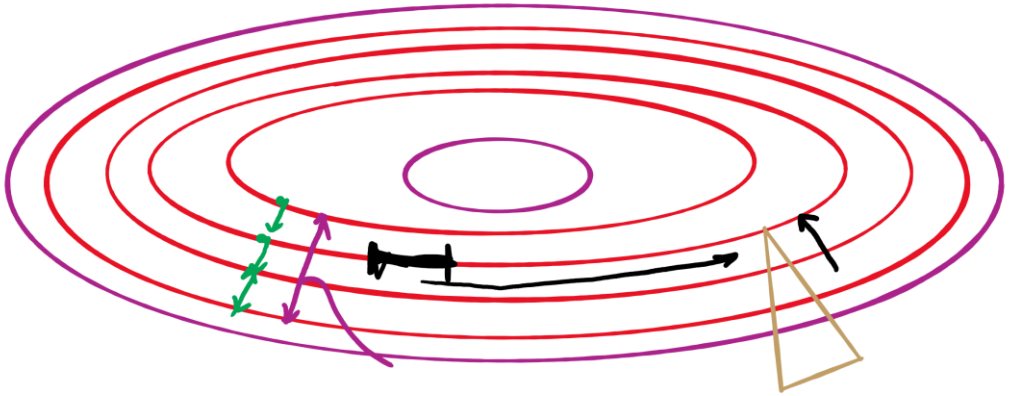
Minimum and Maximum Seek Time



max seek time = min seek time * (# tracks - 1)

cylinders

Average Rotational Delay



$$\text{Average rotational delay} = \frac{\text{one full rotation time}}{2} = \frac{1/\text{rotation speed}}{2} = \frac{(1/4800)*60*1000}{2}\ \text{ms}$$