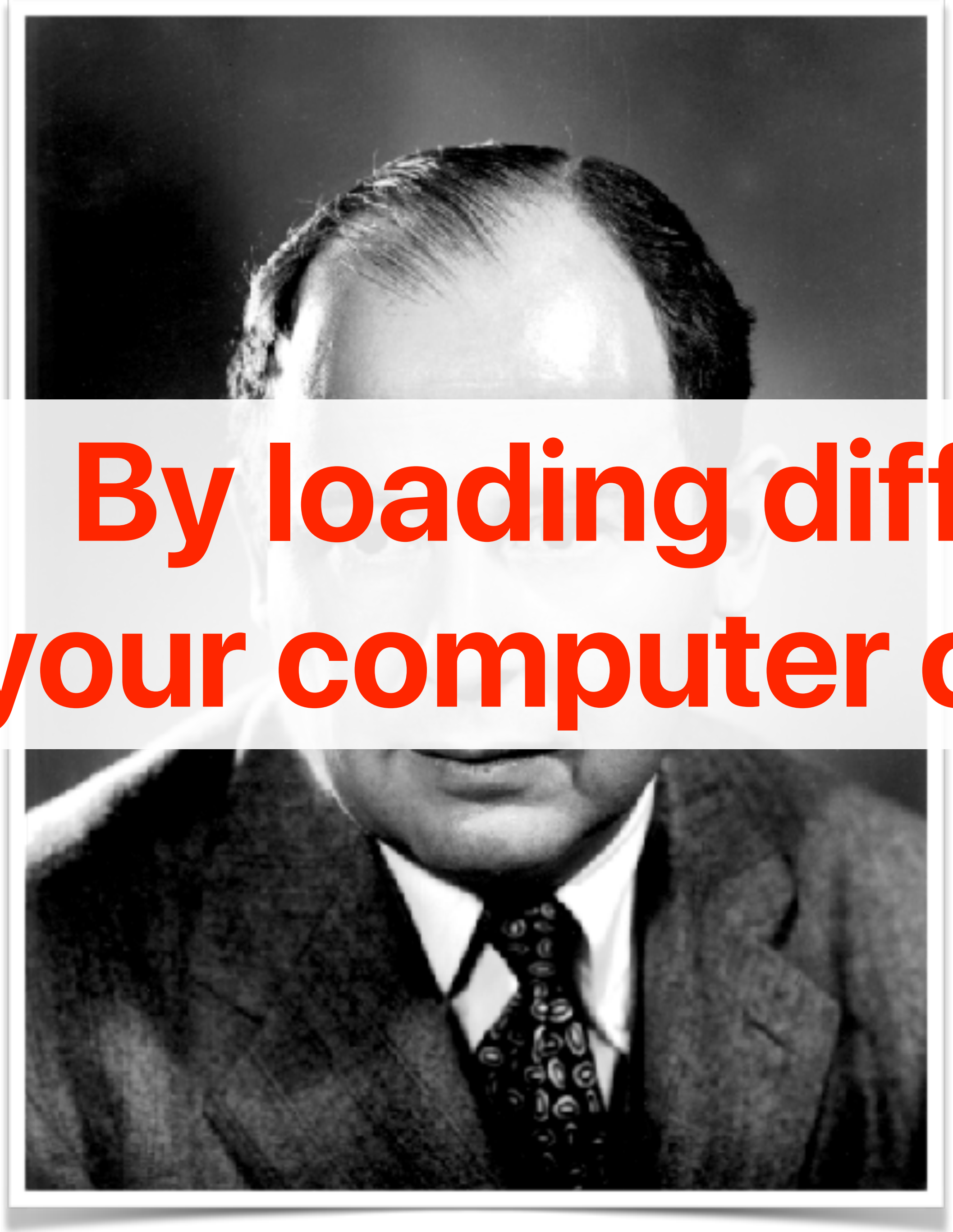
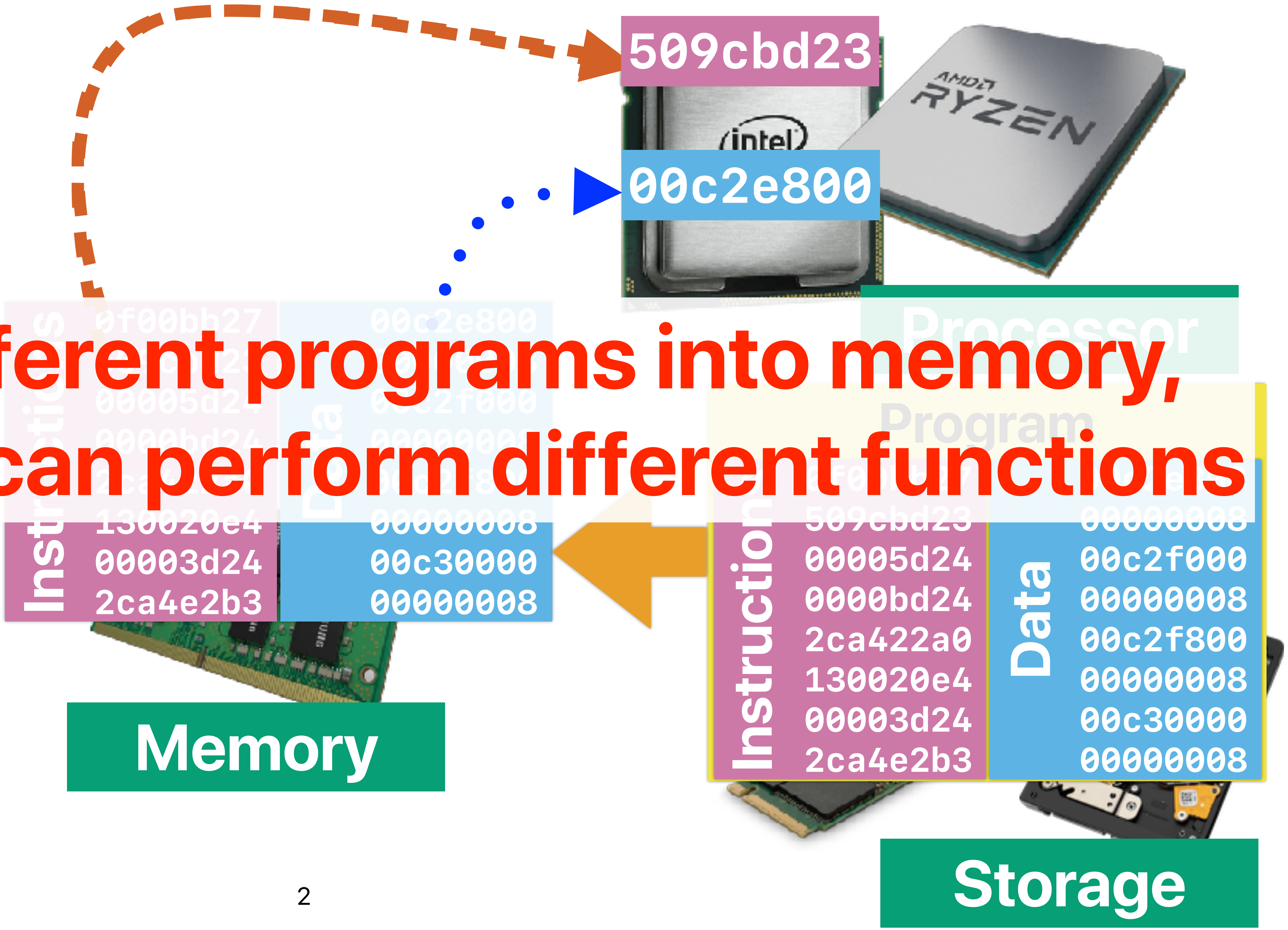
File system basics

Hung-Wei Tseng

Recap: von Neumman Architecture



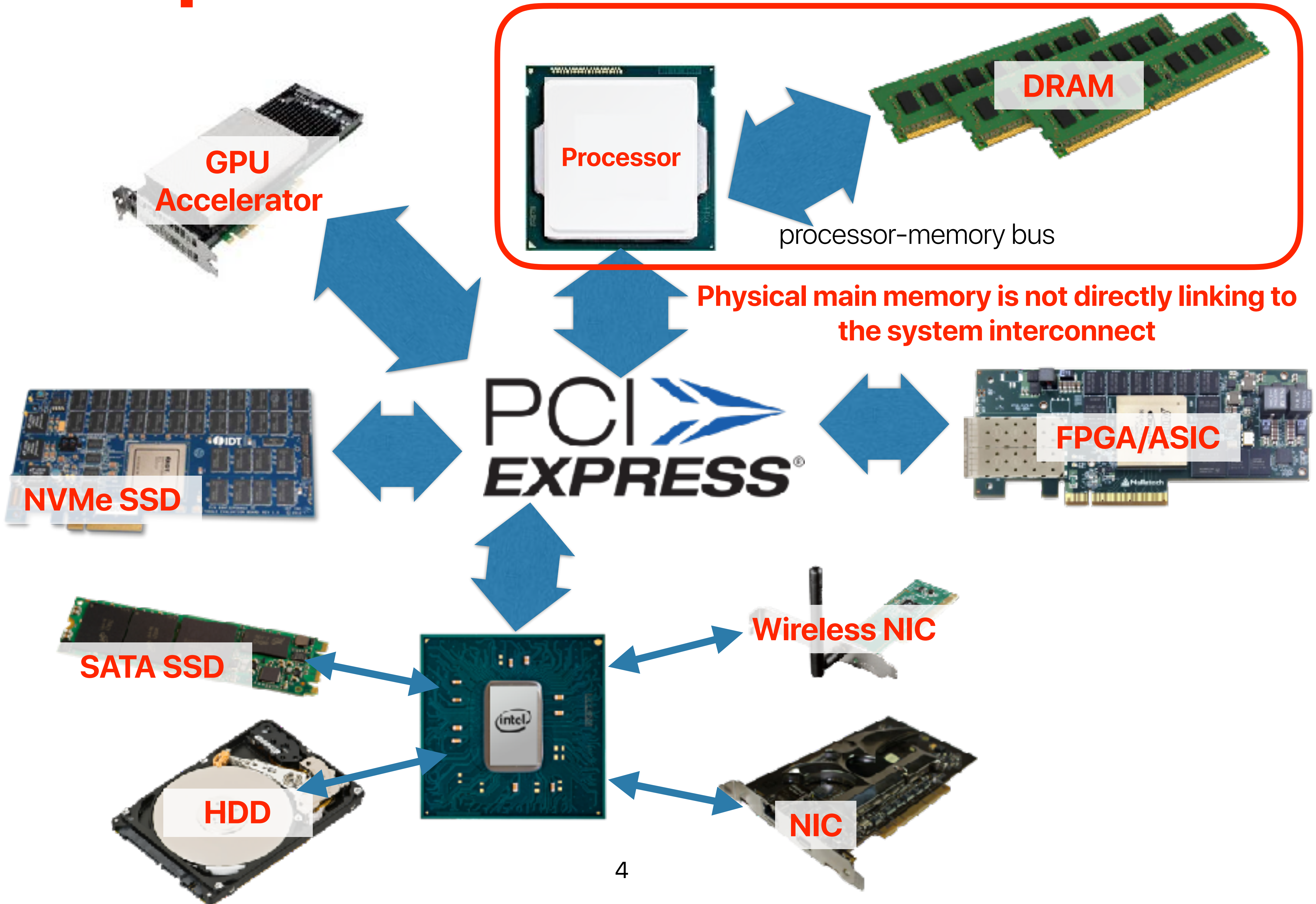


By loading different programs into memory, your computer can perform different functions

Outline

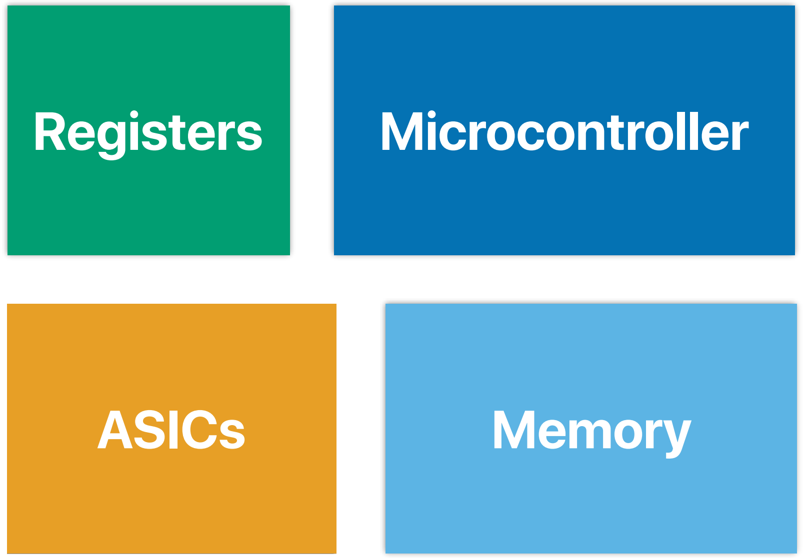
- How our systems interact with I/O
- The basics of storage devices
- File

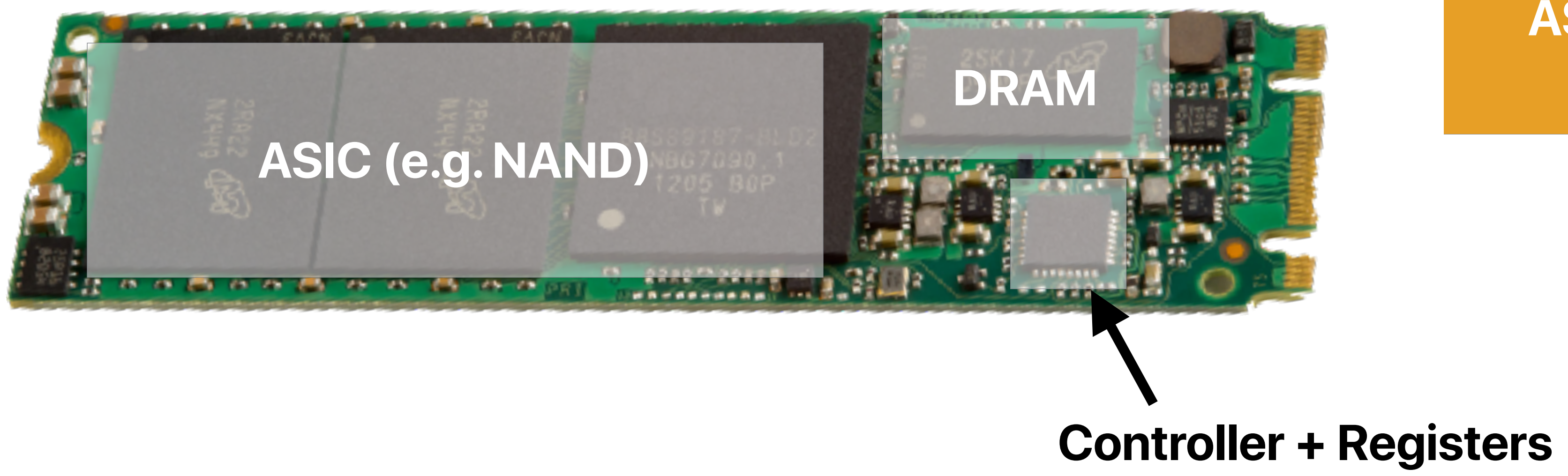
The computer is now like a small network



What's in each device?

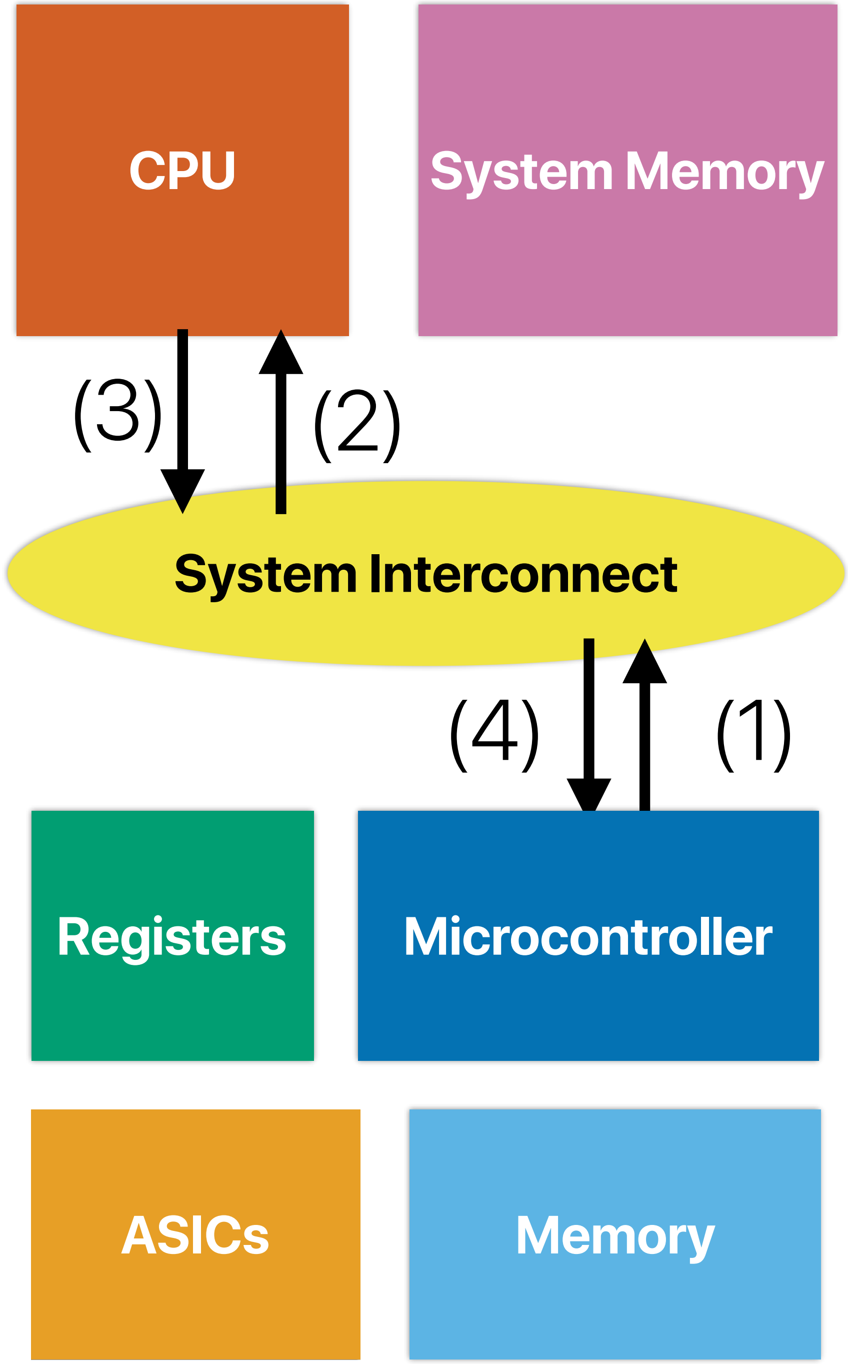
- Registers
  - Command: receiving commands from host
  - Status: tell the host the status of the device
  - Data: the location of exchanging data
- Microcontroller
- Memory
- ASICs





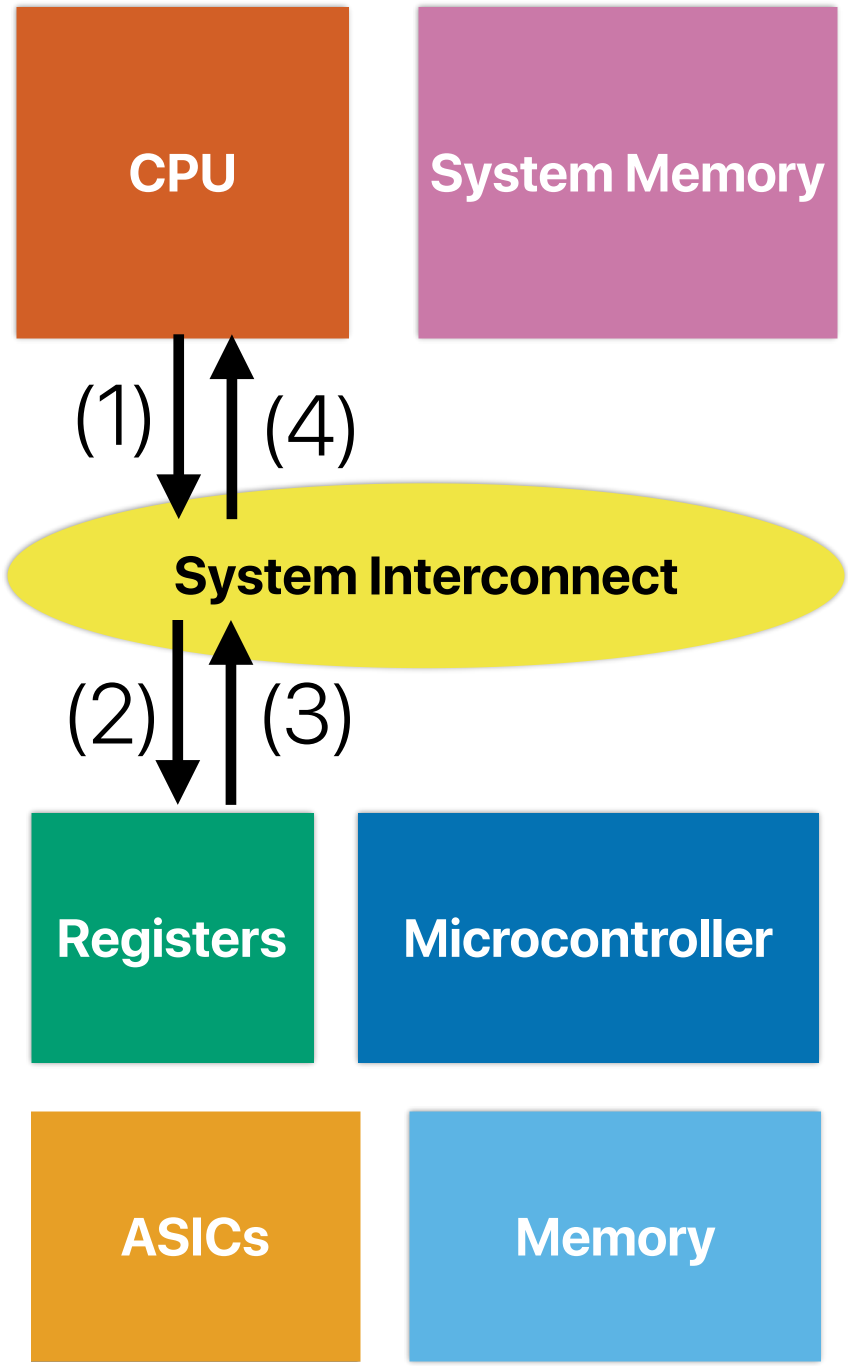
Interrupt

- The device signals the processor only when the device requires the processor/OS handle some tasks/data
- The processor only signals the device when necessary

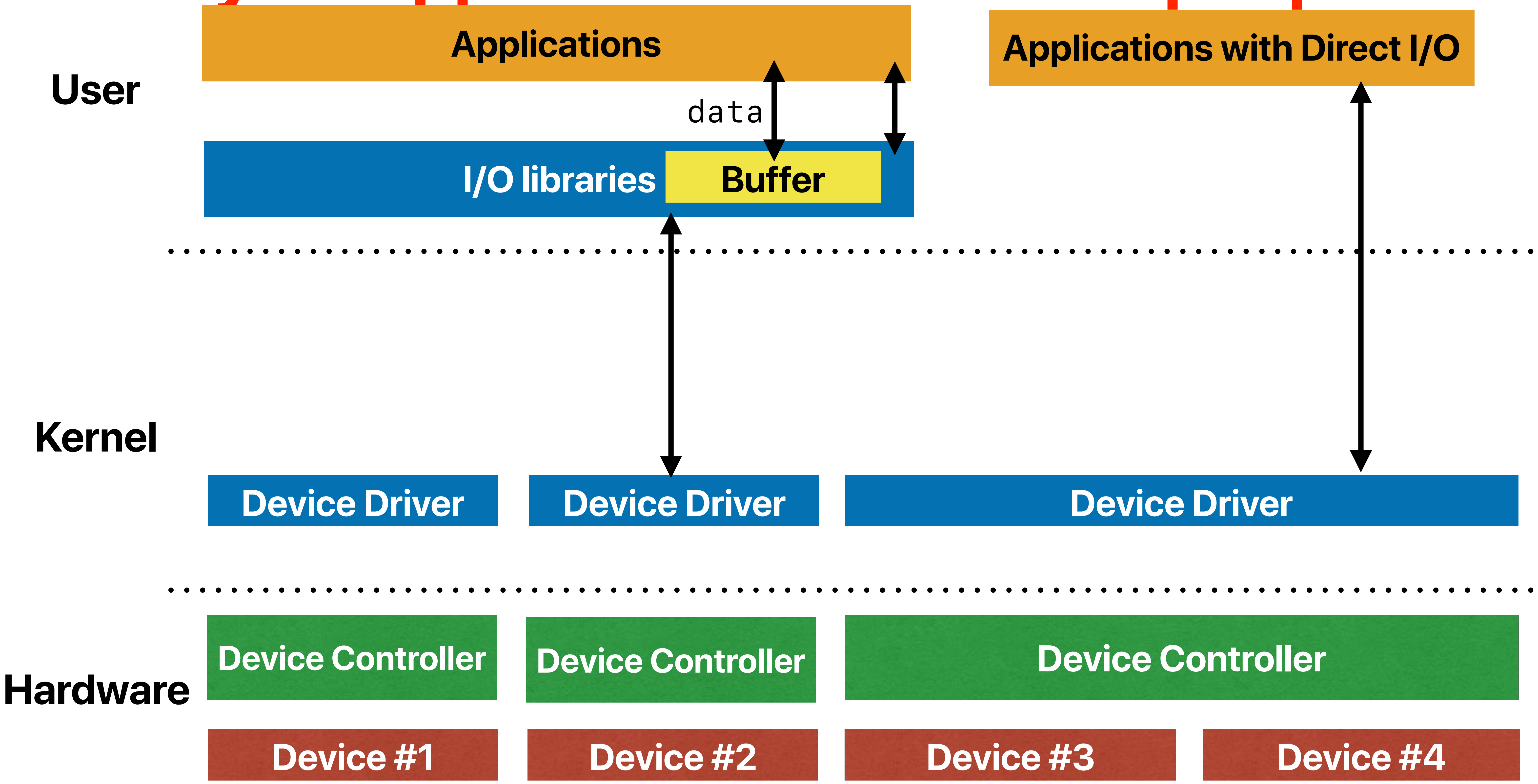


Polling

- The processor/OS constantly asks if the device (e.g. examine the status register of the device) is ready to or requires the processor/OS handle some tasks/data
- The OS/processor executes corresponding handler if the device can handle demand tasks/data or has tasks/data ready



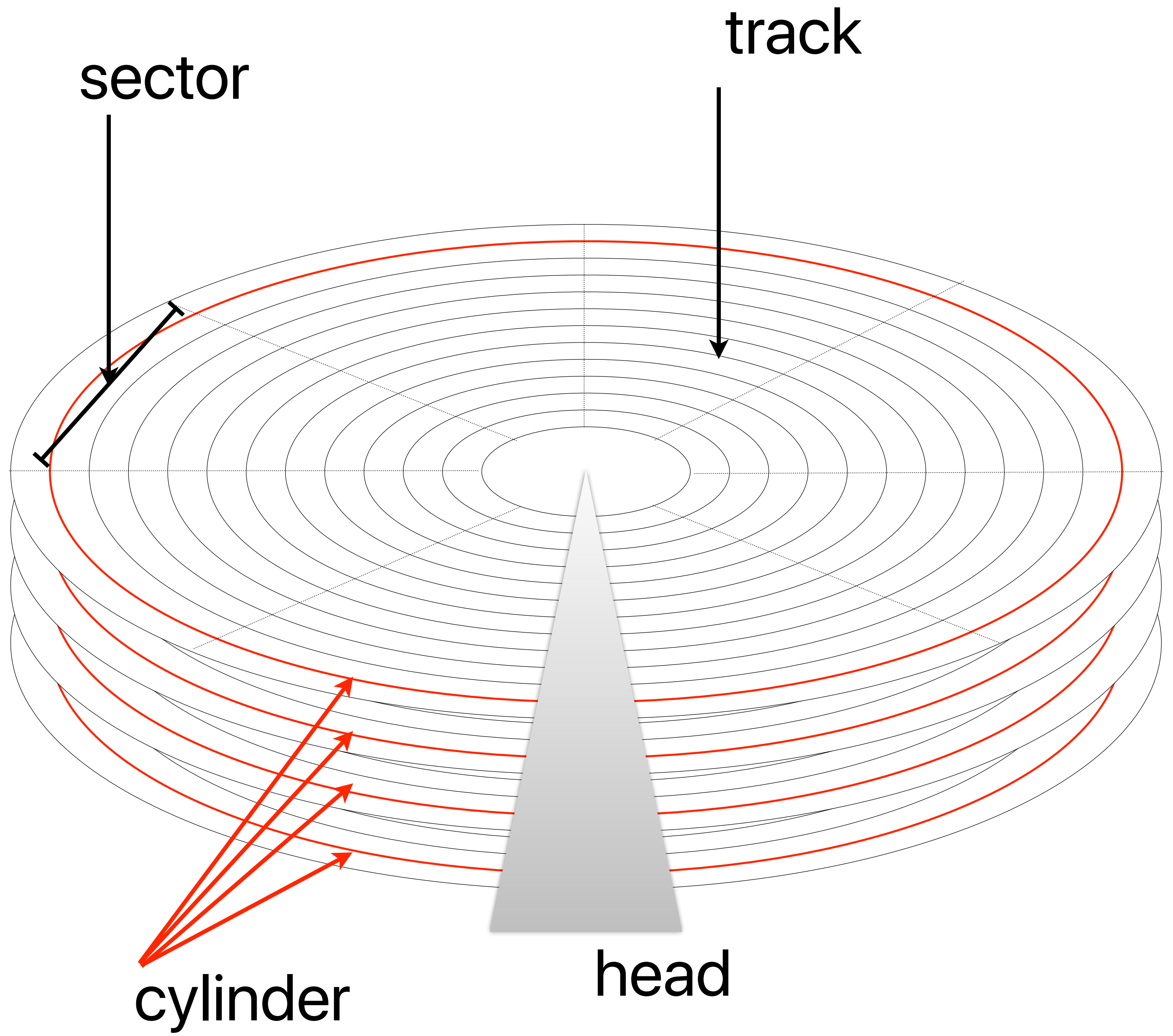
How your application interact with peripherals



Case study: interacting with hard disk drives

Hard Disk Drive

Each sector is identified, locate by an "block address"



- Position the head to proper track (seek time)
- Rotate to desired sector. (rotational delay)
- Read or write data from/to disk to in the unit of sectors (e.g. 512B)
- Takes at least 5ms for each access

Seagate Barracuda 12

- SATA II (300MB/s in theory), 7200 R.P.M., seek time around 8 ms. Assume the controller overhead is 0.2ms. What's the **latency** and **bandwidth** of accessing a 512B sector?

Latency = seek time + rotational delay + transfer time + controller overhead

$$8\ ms \quad + \frac{1}{2} \times \frac{1}{\frac{7200}{60}} \quad + \frac{\frac{0.5}{1024}}{300} \quad + 0.2\ ms$$

$$= 8\ ms + 4.17\ ms + 0.00167\ us + 0.2\ ms = 12.36\ ms$$

Bandwidth = volume_of_data over period_of_time

$$= \frac{0.5KB}{12.36ms} = 40.45KB/sec$$

Seagate Barracuda 12

- SATA II (300MB/s in theory), 7200 R.P.M., seek time around 8 ms. Assume the controller overhead is 0.2ms. What's the latency of accessing a consecutive 4MB data?

Latency = seek time + rotational delay + transfer time + controller overhead

$$8\ ms \quad + \frac{1}{2} \times \frac{1}{\frac{7200}{60}} \quad + \frac{4}{300} \quad + 0.2\ ms$$

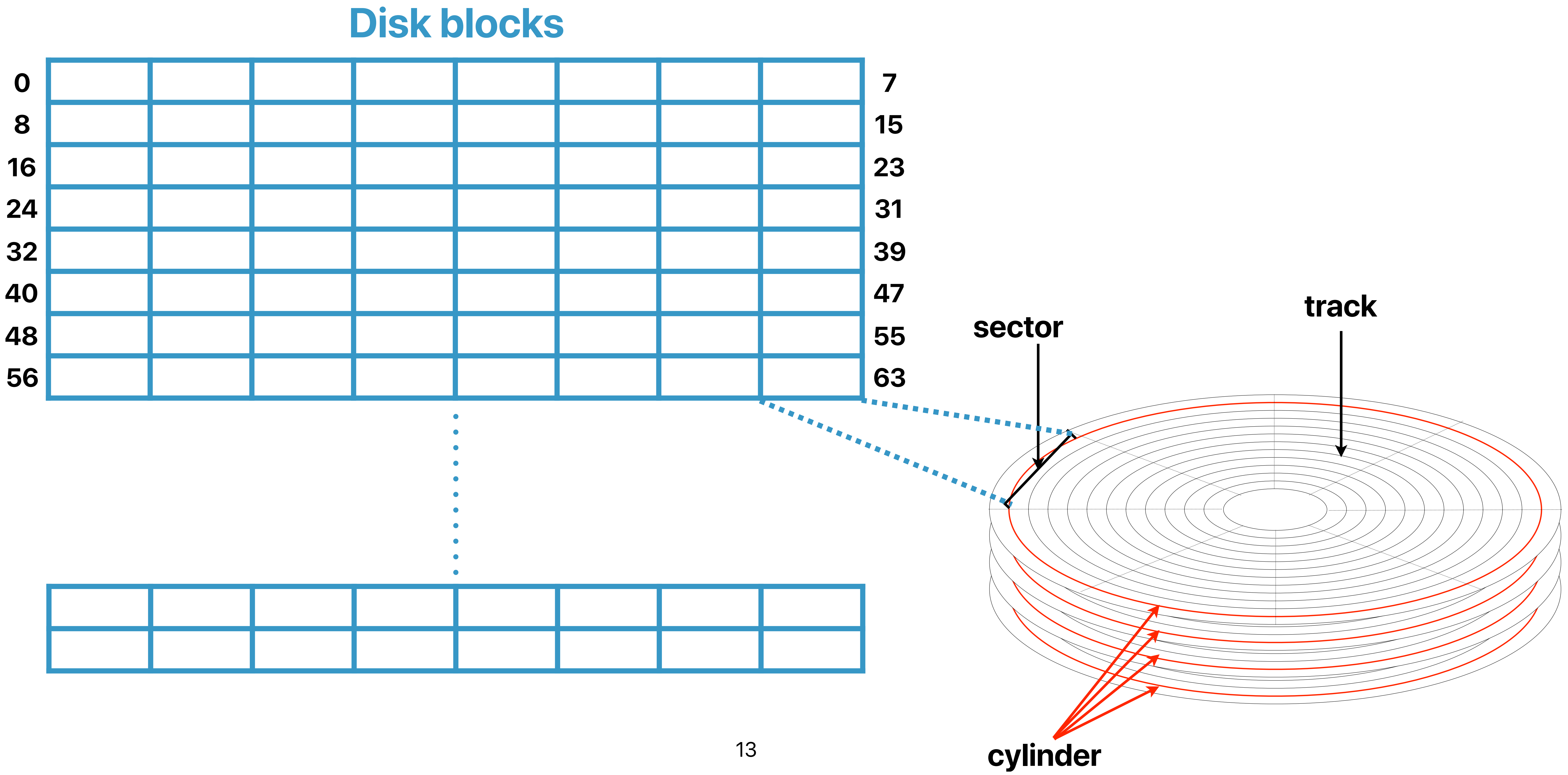
$$= 8\ ms + 4.17\ ms + 13.33\ ms + 0.2\ ms = 25.69\ ms$$

Bandwidth = volume_of_data over period_of_time

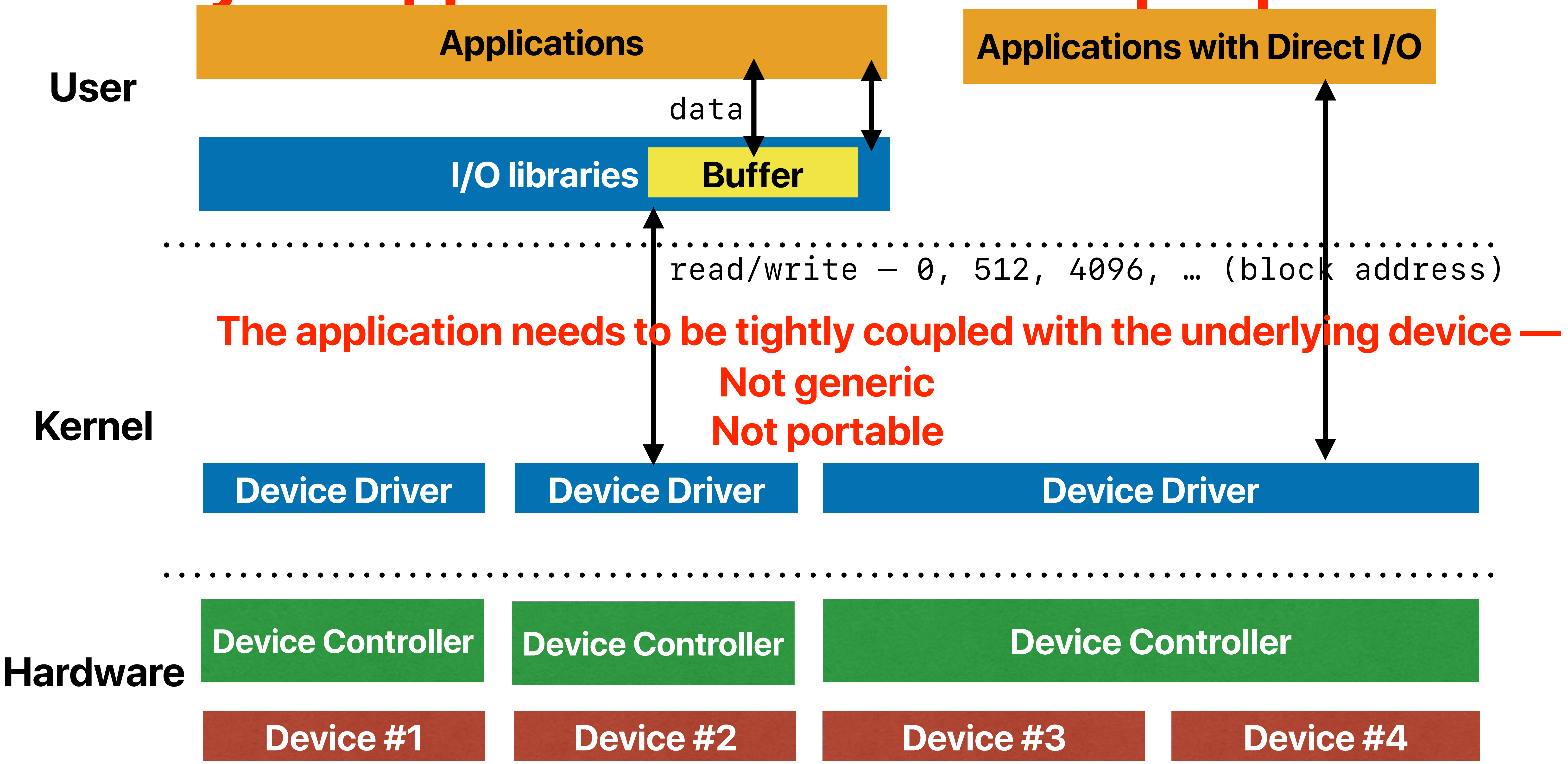
$$= \frac{4MB}{25.69ms} = 155.7\ MB/sec$$

Trading latencies with bandwidth

Numbering the disk space with block addresses



How your application interact with peripherals

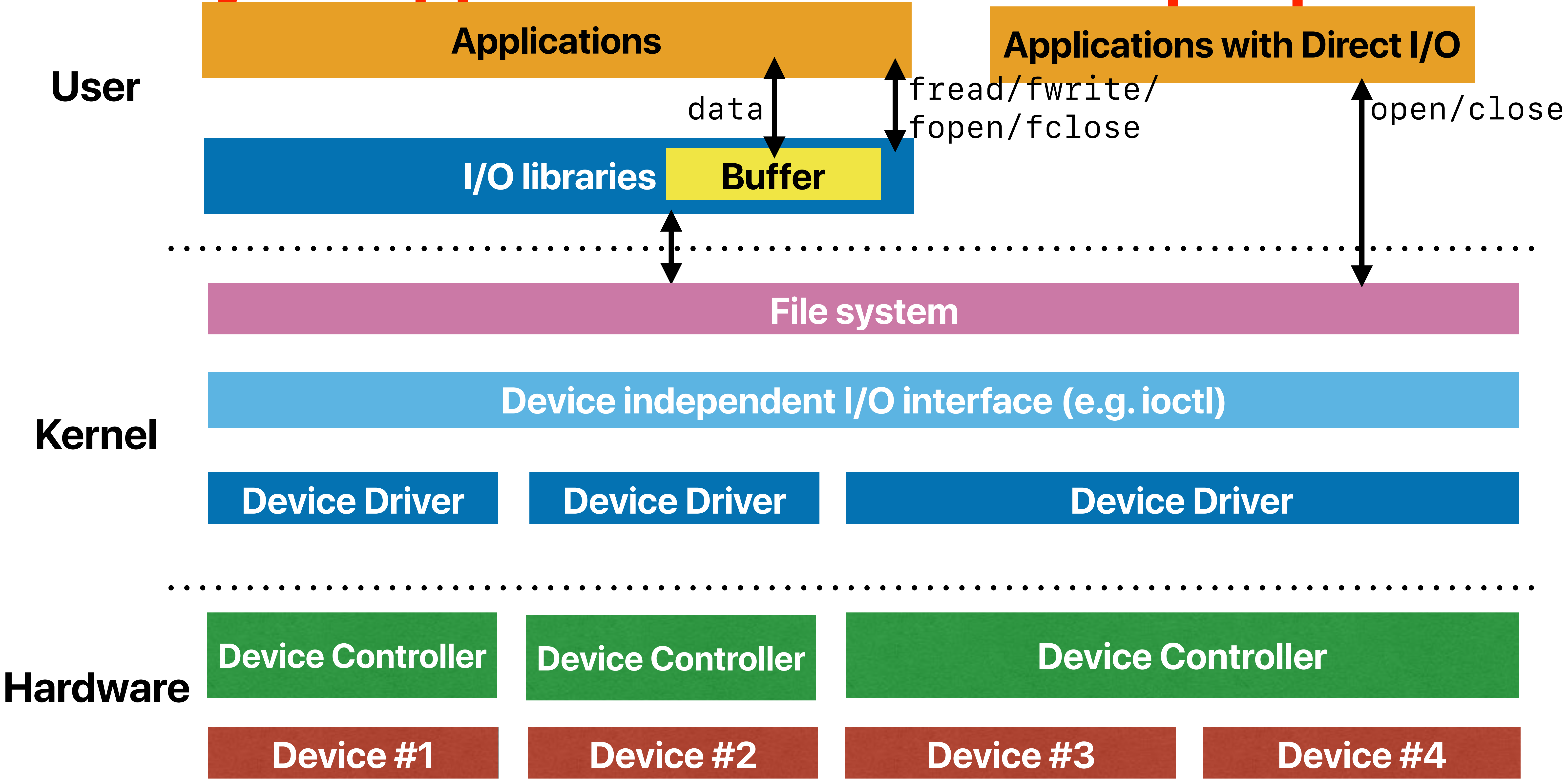


All problems in computer science can be solved by another level of indirection

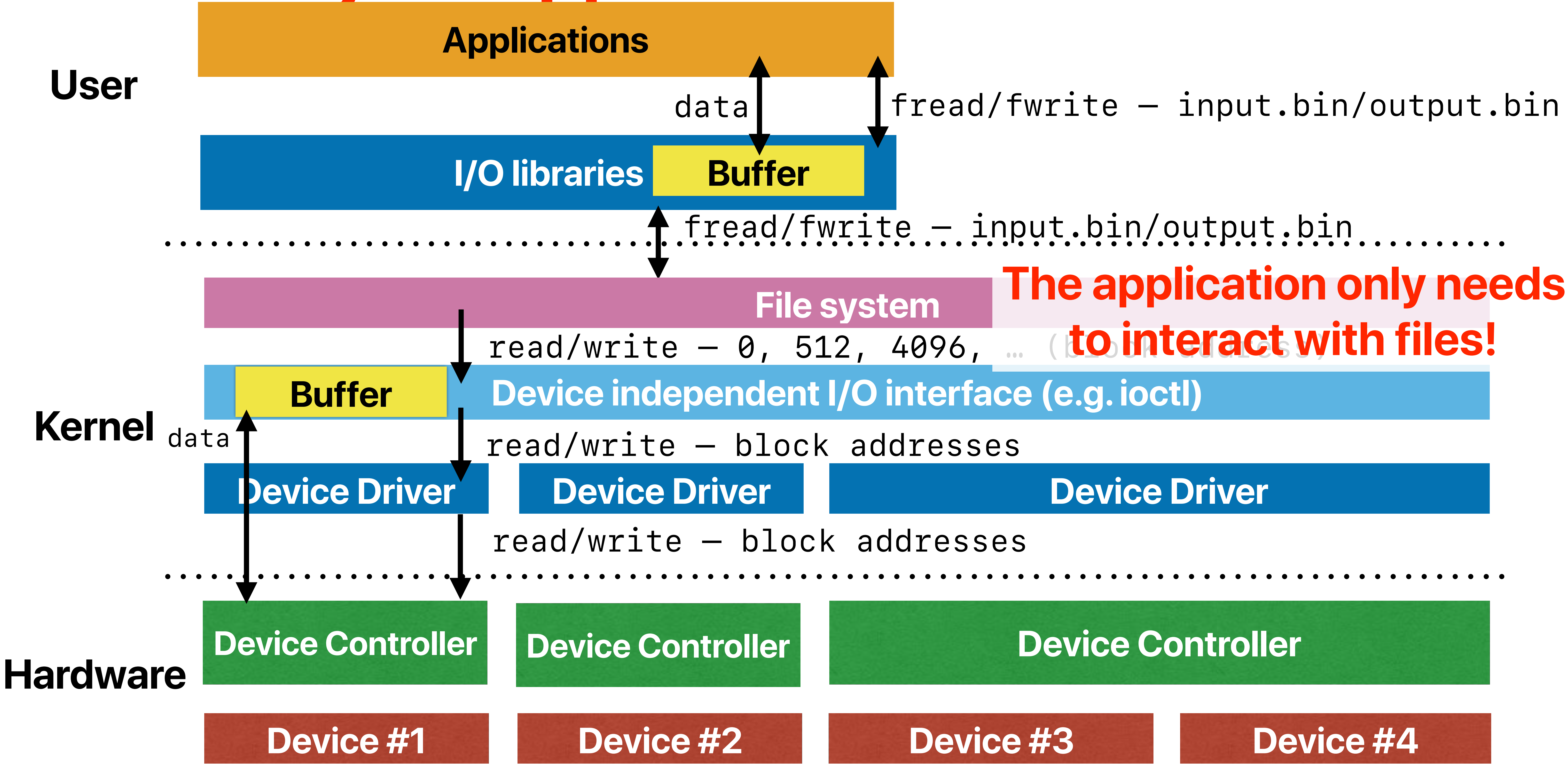
–David Wheeler

The file & file system abstraction

How your application interact with peripherals



How your application reaches H.D.D.



What we've learned in the past...

The most important role of UNIX is to provide a file system. From the point of view of the user, there are three kinds of files: ordinary disk files, directories, and special files.

3.1 Ordinary Files

A file contains whatever information the user places on it, for example symbolic or binary (object) programs. No particular structuring is expected by the system. Files of text consist simply of a string of characters, with lines demarcated by the new-line character. Binary programs are sequences of words as they will appear in core memory when the program starts executing. A few user programs manipulate files with more structure: the assembler generates and the loader expects an object file in a particular format. However, the structure of files is controlled by the programs which use them, not by the system.

3.2 Directories

Directories provide the mapping between the names of files and the files themselves, and thus induce a structure on the file system as a whole. Each user has a directory of his own files; he may also create subdirectories to contain groups of files conveniently treated together. A directory behaves exactly like an ordinary file except that it cannot be written on by unprivileged programs, so that the system controls the contents of directories. However, anyone with appropriate permission may read a directory just like any other file.

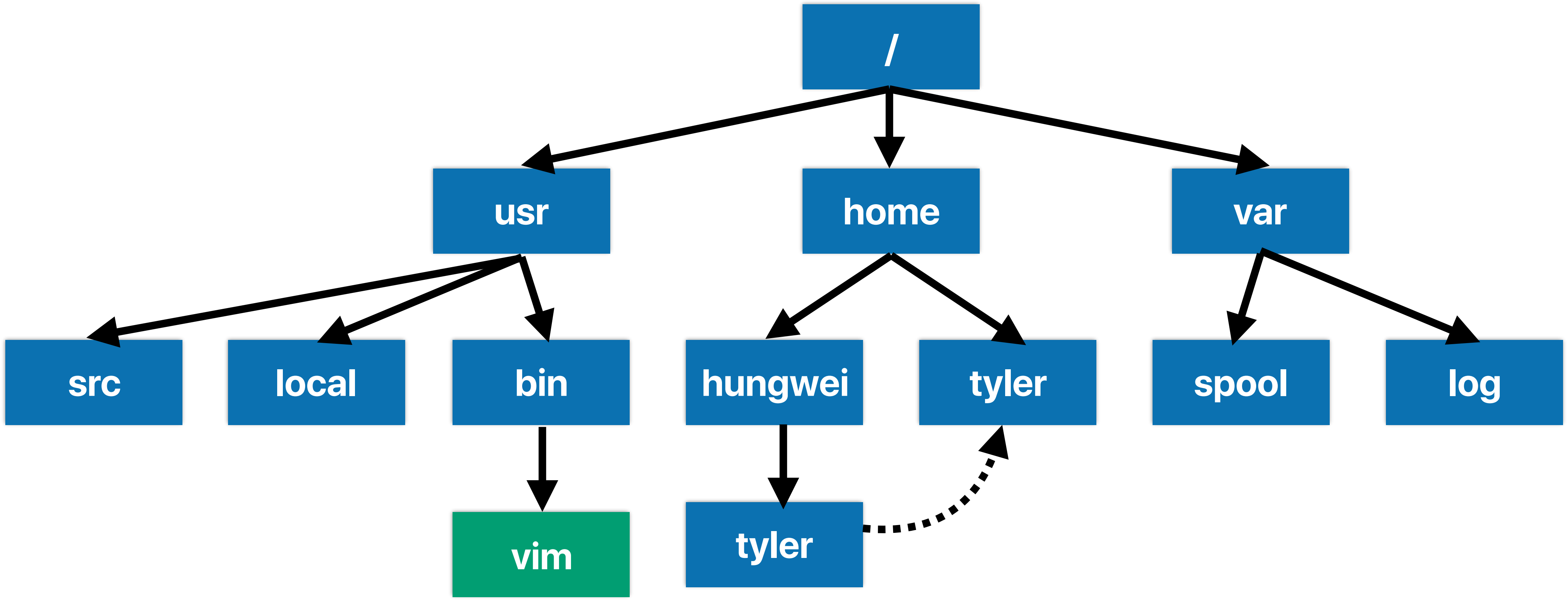
3.3 Special Files

Special files constitute the most unusual feature of the UNIX file system. Each I/O device supported by UNIX is associated with at least one such file. Special files are read and written just like ordinary disk files, but requests to read or write result in activation of the associated device. An entry for each special file resides in directory */dev*, although a link may be made to one of these files just like an ordinary file. Thus, for example, to punch paper tape, one may write on the file */dev/ppt*. Special files exist for each communication line, each disk, each tape drive, and for physical core memory. Of course, the active disks and the core special file are protected from indiscriminate access.

There is a threefold advantage in treating I/O devices this way: file and device I/O are as similar as possible; file and device names have the same syntax and meaning, so that a program expecting a file name as a parameter can be passed a device name; finally, special files are subject to the same protection mechanism as regular files.

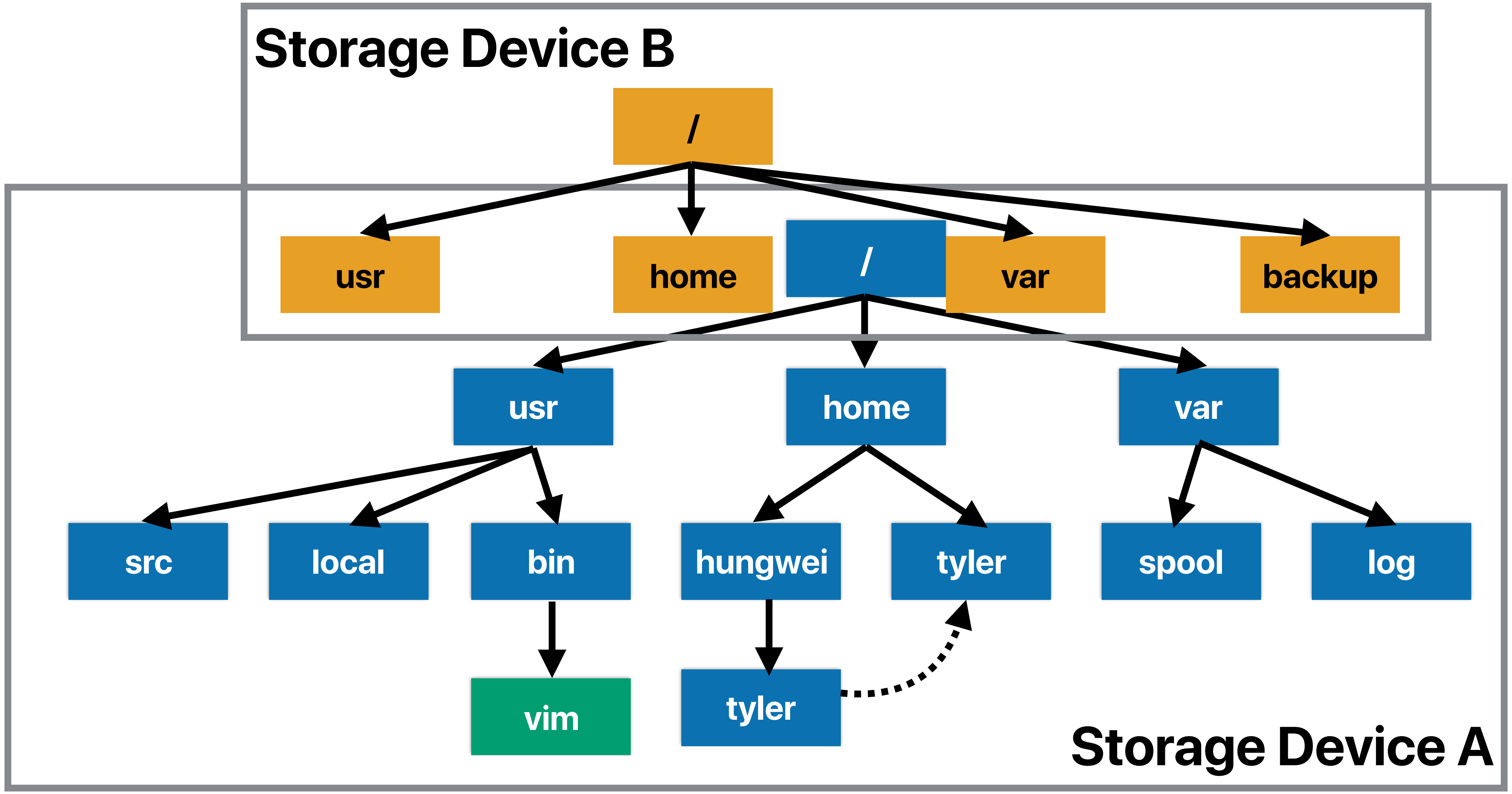
Hierarchical File System Structure

- Namespace has tree-like structure
- Root directory (/) with subdirectories, each containing its own subdirectories
- Links break the tree analogy



Mount

- The “/” on storage device A will become /backup now!

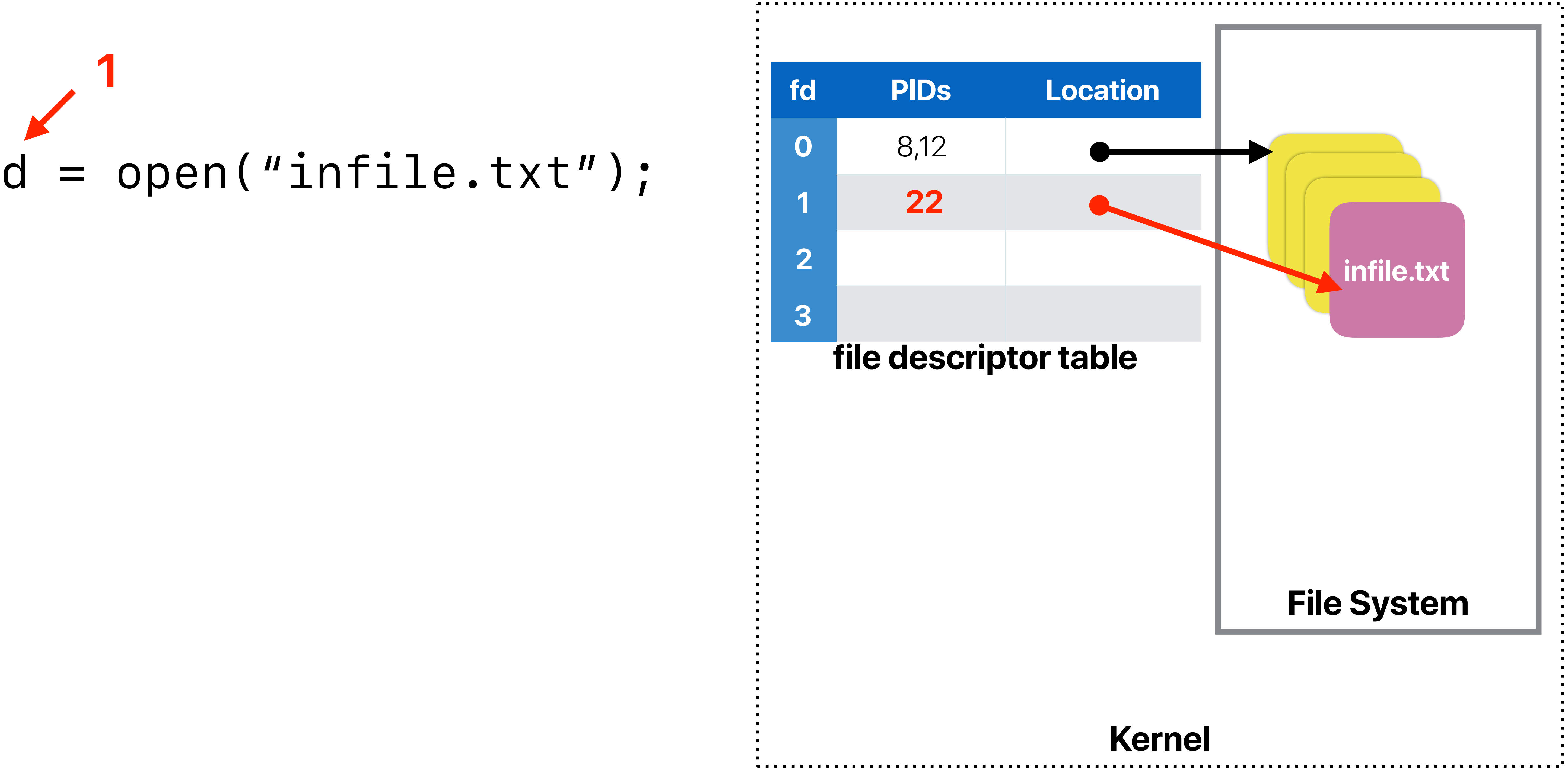


How you access files in C

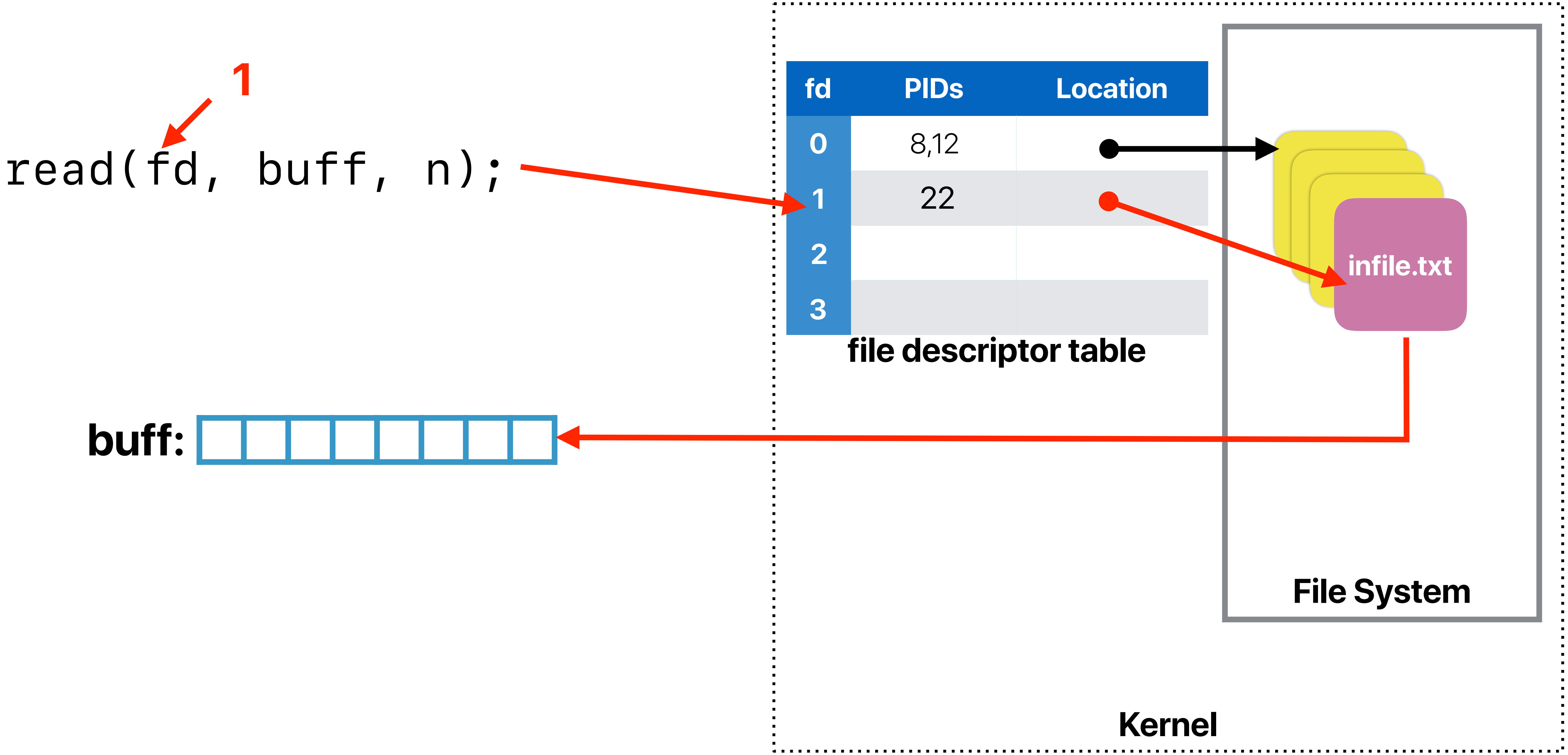
```
int fd, nr, nw;
void *in_buff;
in_buff = malloc(BUFF_SIZE);

fd1 = open("infile.txt", O_RDONLY);
fd2 = open("outfile.txt", O_RDWR | O_CREAT);
nr = read(fd1, in_buff, BUFF_SIZE);
nw = write(fd2, in_buff, BUFF_SIZE);
lseek(fd1, -8, SEEK_END);
nr = read(fd1, in_buff, 8); // read last 8 bytes
// more fancy stuff here…
close(fd1);
close(fd2);
```

open

```
fd = open("infile.txt");
```



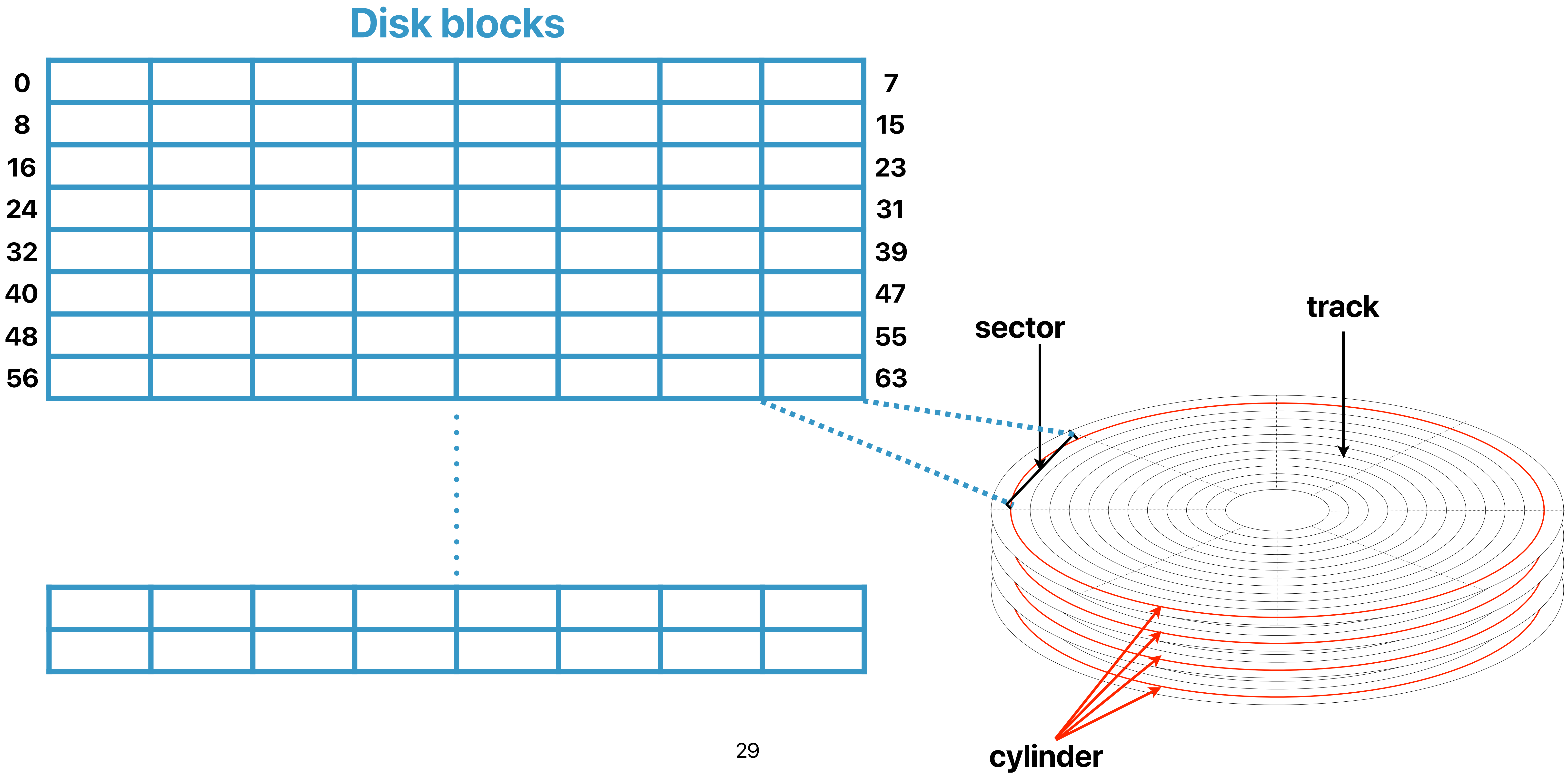
read



1
read(fd, buff, n);
fd
PIDs
Location
0
8,12
1
22
2
3
file descriptor table
infile.txt
File System
buff:
Kernel

The design of a file system

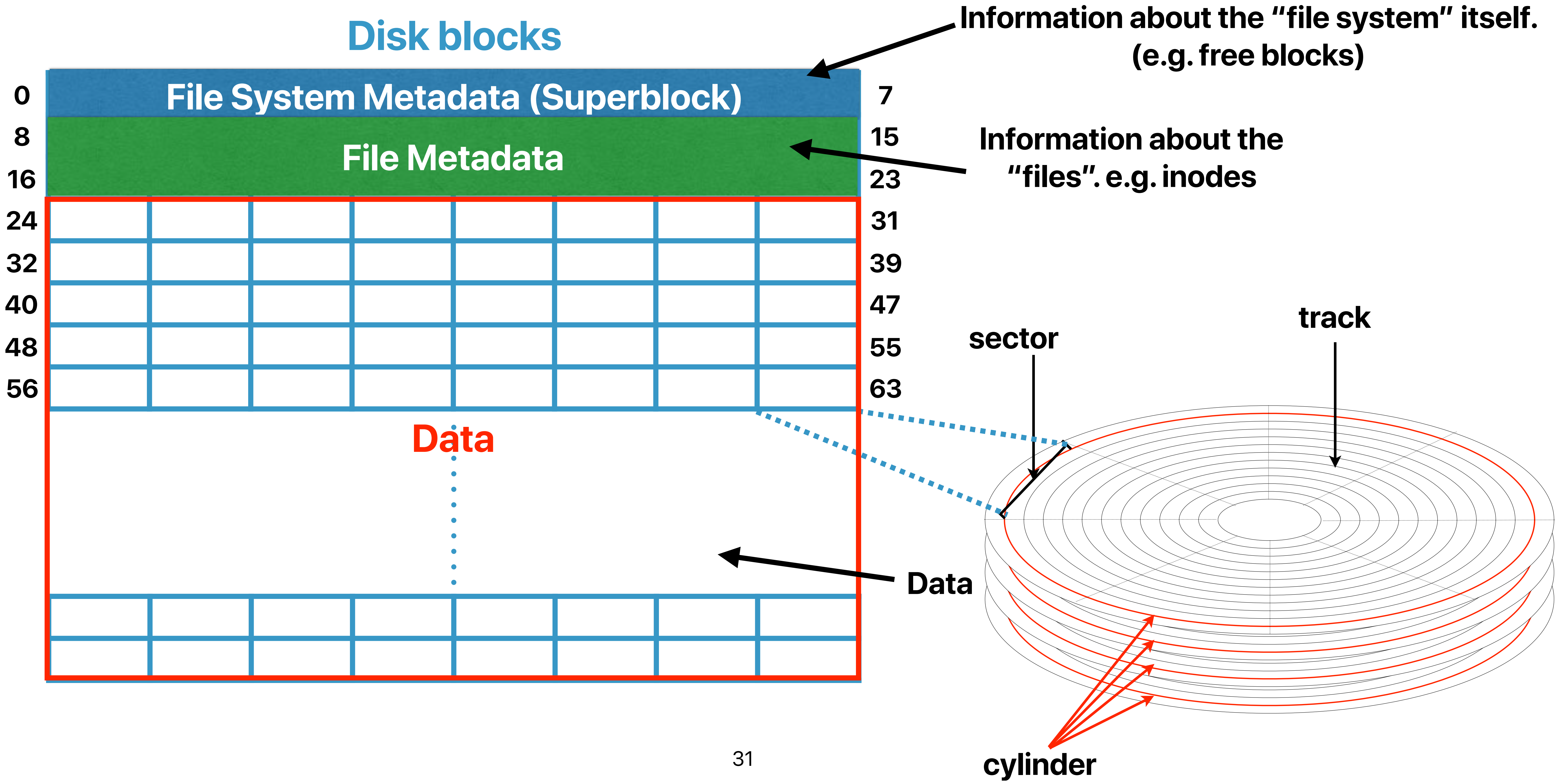
Recap: Numbering the disk space with block addresses



Questions for file systems

- How do we locate files?
  - How do we manage hierarchical namespace?
  - How do we manage file and file system metadata?
- How do we allocate storage space?
- How do we make the file system fast?
- How do we ensure file integrity?

How the original UNIX file system use these blocks



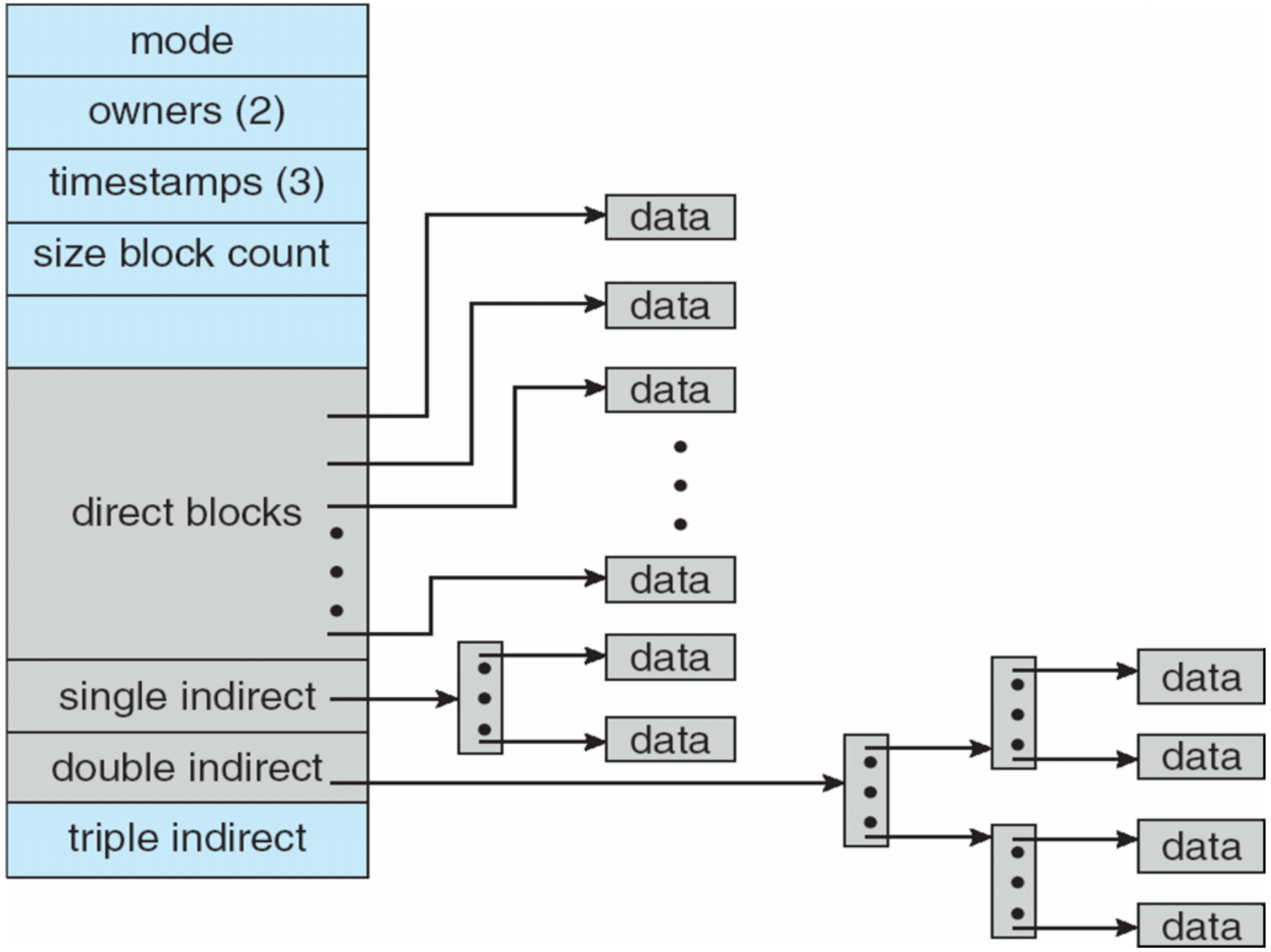
Superblock — metadata of the file system

- Contains critical file system information
  - The volume size
  - The number of nodes
  - Pointer to the head of the free list
- Located at the very beginning of the file system

inode — metadata of each file

- File types: directory, file
- File size
- Permission
- Attributes

Unix inode



- File types: directory, file
- File size
- Permission
- Attributes
- Types of pointers:
  - Direct: Access single data block
  - Single Indirect: Access n data blocks
  - Double indirect: Access n2 data blocks
  - Triple indirect: Access n3 data blocks
- inode has 15 pointers: 12 direct, 1 each single-, double-, and triple-indirect
- If data block size is 512B and n = 256: max file size = (12+256+2562+2563)*512 = 8GB

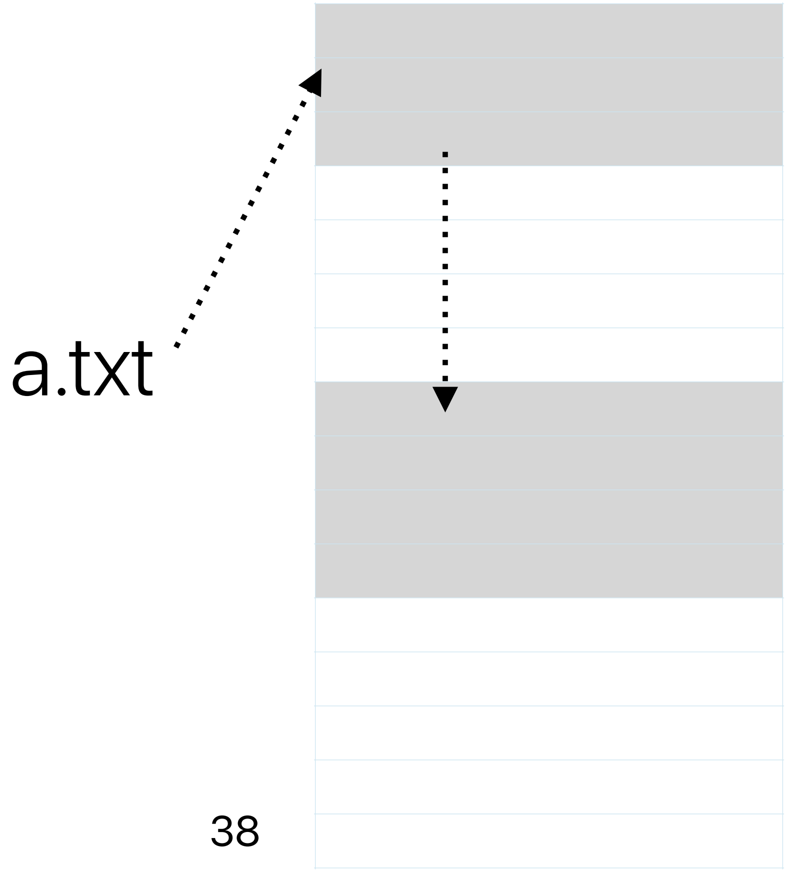
What must be done to reach your files

- Scenario: User wants to access /home/hungwei/CS202/foo.c
- Procedure: File system will…
  - Open "/" file (This is in known from superblock.)
  - Locate entry for "home," open that file
  - Locate entry for "hungwei", open that file
  - …
  - Locate entry for "foo.c" and open that file
- Let's use "strace" to see what happens

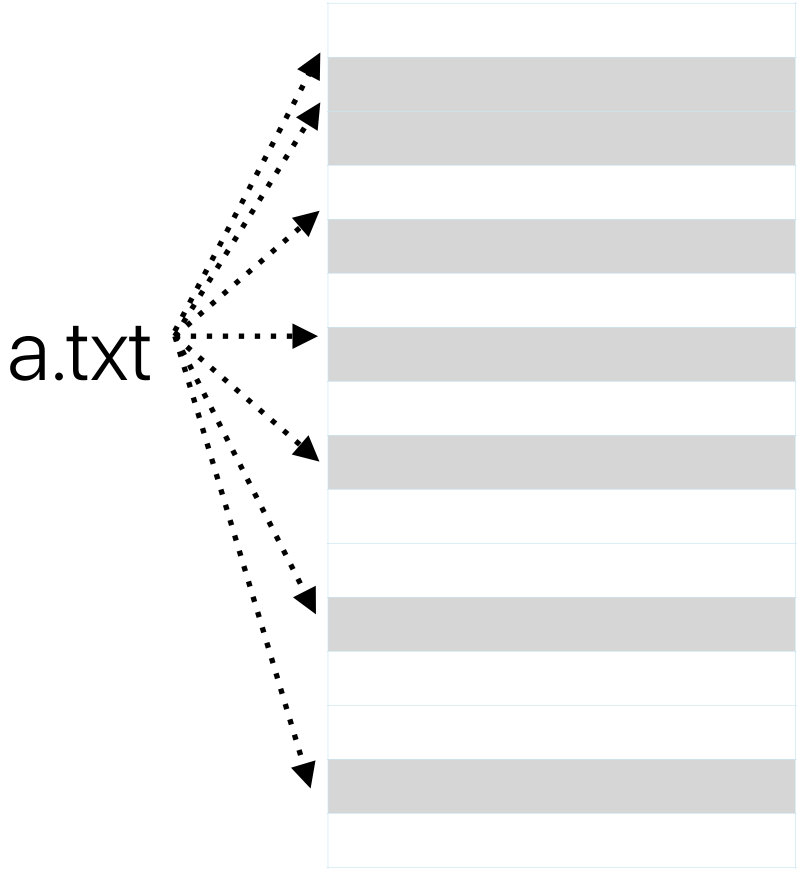
How do we allocate space?

- Contiguous: the file resides in continuous addresses

- Extents: the file resides in several group of smaller continuous address



- Non-contiguous: the file can be anywhere



Space overhead for storage allocation strategies

- Need to track location of blocks on per file basis
- Contiguous only needs a pair <start, size>
- Extents requires a table of pairs
- Non-contiguous requires either a linked list of blocks OR a table of block pointers (i.e. a map)

Now, what about performance?

- Disk accesses are slow!
  - Memory access: 100ns
  - Disk access: 5-12ms
  - Flash SSD: 30-120us
- Can reduce average access time by clustering data together… but still slow!
- Ideas: Reduce the number of disk accesses using:
- Read-ahead: Bring in multiple blocks when reading a single block (locality!)

Buffer Cache

- Buffer cache is a cache of recently used disk blocks resides in DRAM-based main memory
- Modern OSs aggressively use free DRAM space for buffer caches
- When accessing disk (read/write), we follow these steps:
  - Check if block is in cache; stop if in cache
  - If not in cache, access disk and place block in the cache
  - Replacement Policy: LRU implemented with a linked list
  - Head of list is next to replace
  - Tail of list is last to replace