File System Performance (and Abstractions)

Kevin Webb
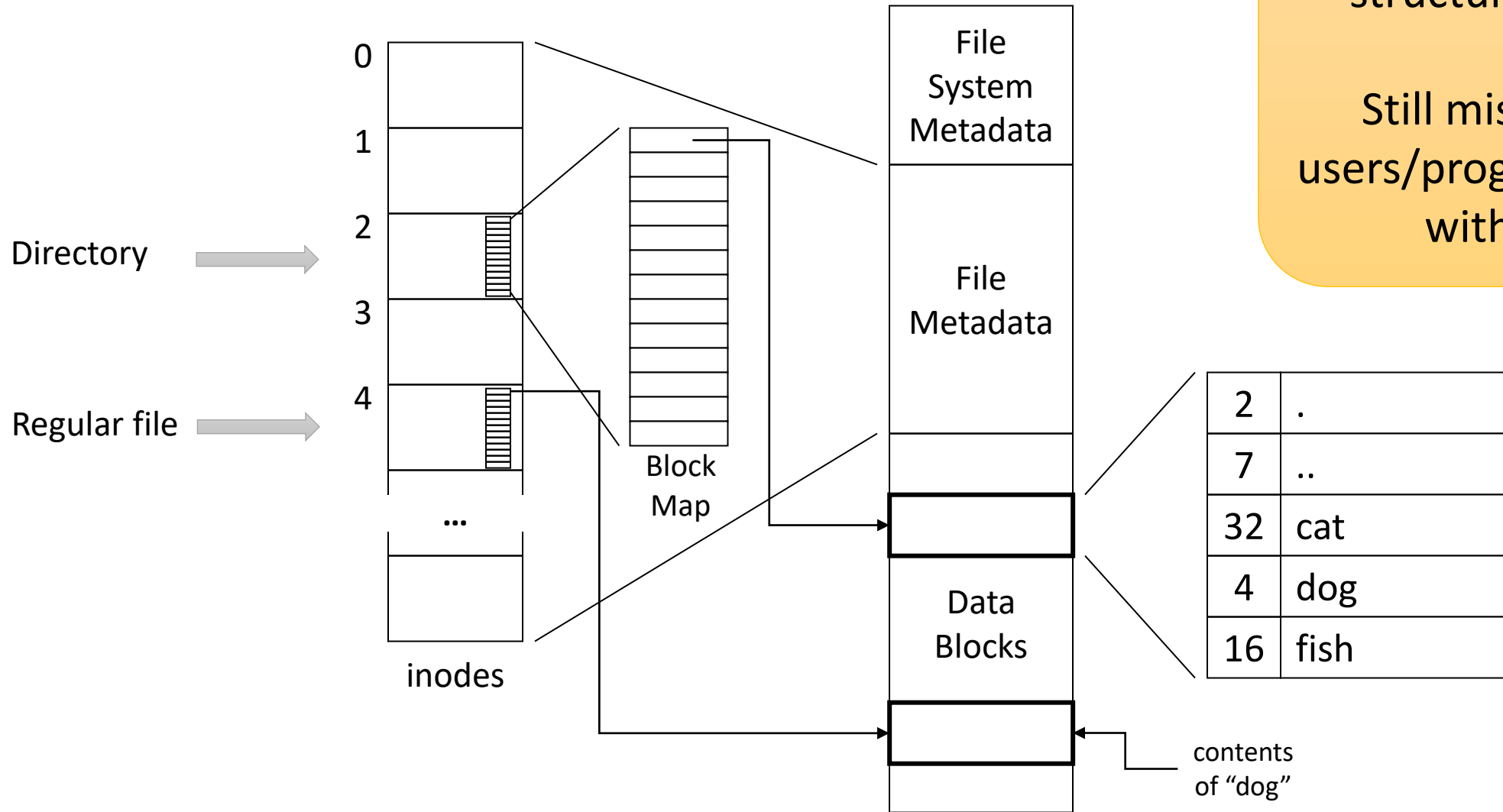
Swarthmore College

April 2, 2024

Today's Goals

- Supporting multiple file systems in one name space.

- Schedulers…not just for CPUs, but disks too!

- Caching and prefetching disk blocks.

- Organizing a file system for performance.

File Systems so far…

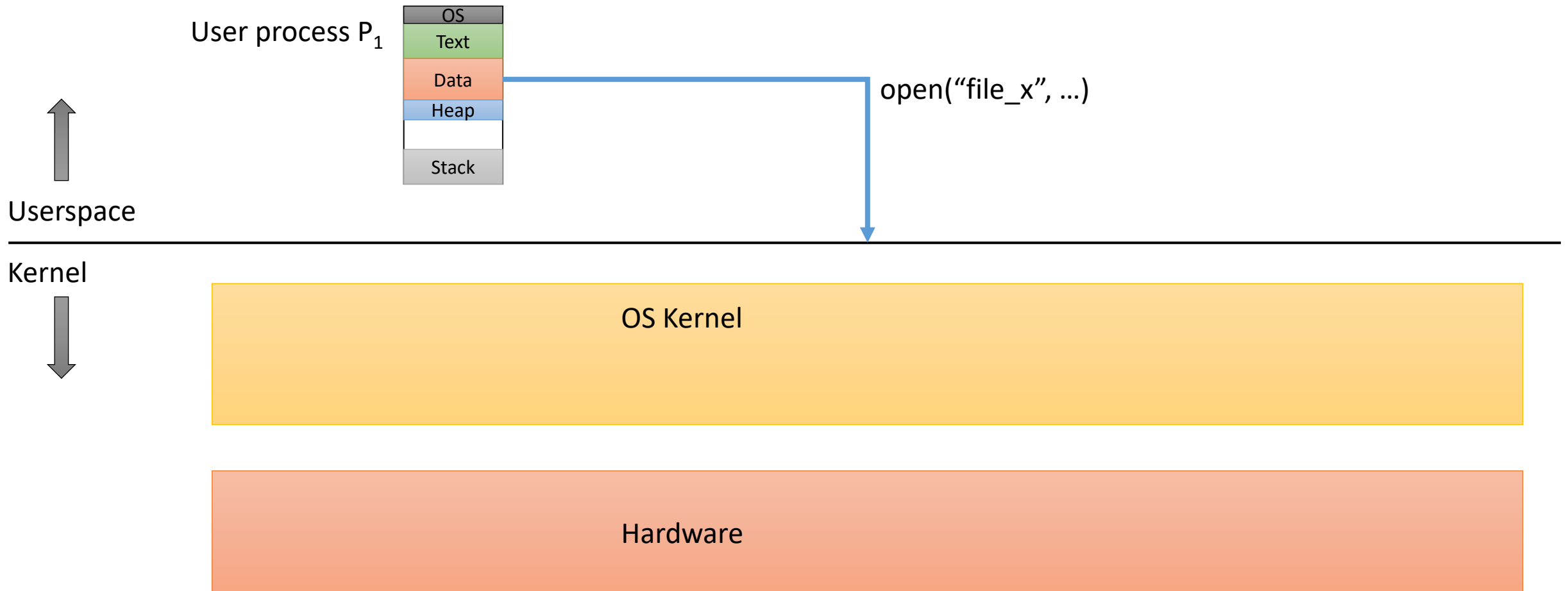So far, how FS is structured on disk.

Still missing: How users/programs interact with them!

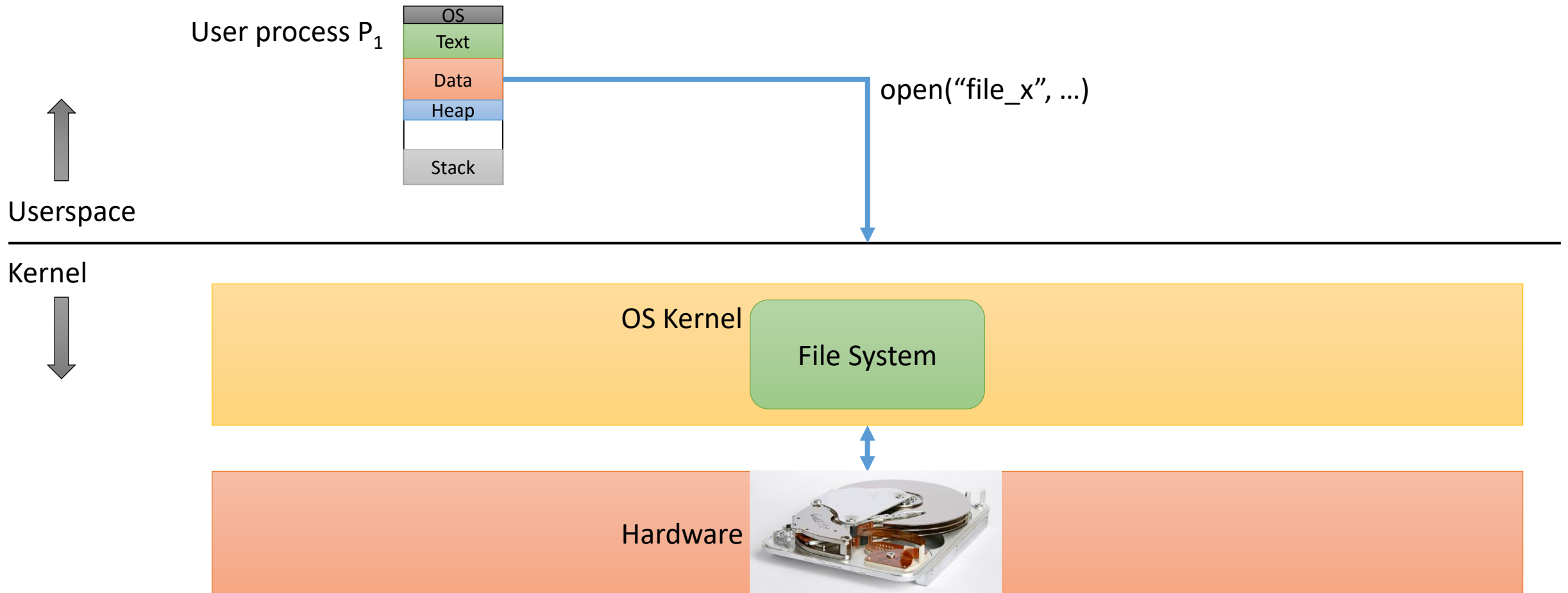|	
2	.
7	..
32	cat
4	dog
16	fish

Directory

Regular file

0
1
2
3
4
…

inodes

Block Map

File System Metadata

File Metadata

Data Blocks

contents of "dog"

Userspace Perspective

- Userspace processes make system calls to interact with files:

User process P$_1$

OS
Text
Data
Heap
Stack

open("file_x", …)

Userspace

Kernel

OS Kernel

Hardware

Userspace Perspective

- Userspace processes make system calls to interact with files:

User process P_1

| OS |
| Text |
| Data |
| Heap |
| |
| Stack |

open("file_x", ...)

Userspace

Kernel

OS Kernel

File System

Hardware

Userspace Perspective

- Userspace processes make system calls to interact with files:

User process P_1

OS
Text
Data
Heap
Stack

File descriptor (3)

Userspace

Kernel

P_1's Descriptor Table

...	...
3	file_x ...
...	...

OS Kernel

File System

Hardware

Userspace Perspective

- Userspace processes make system calls to interact with files:

User process P_1

OS
Text
Data
Heap
Stack

read(3, …)
write(3, …)
close(3)

This picture:
one process
one file system

Userspace

Kernel

P_1's Descriptor Table

OS Kernel

File System

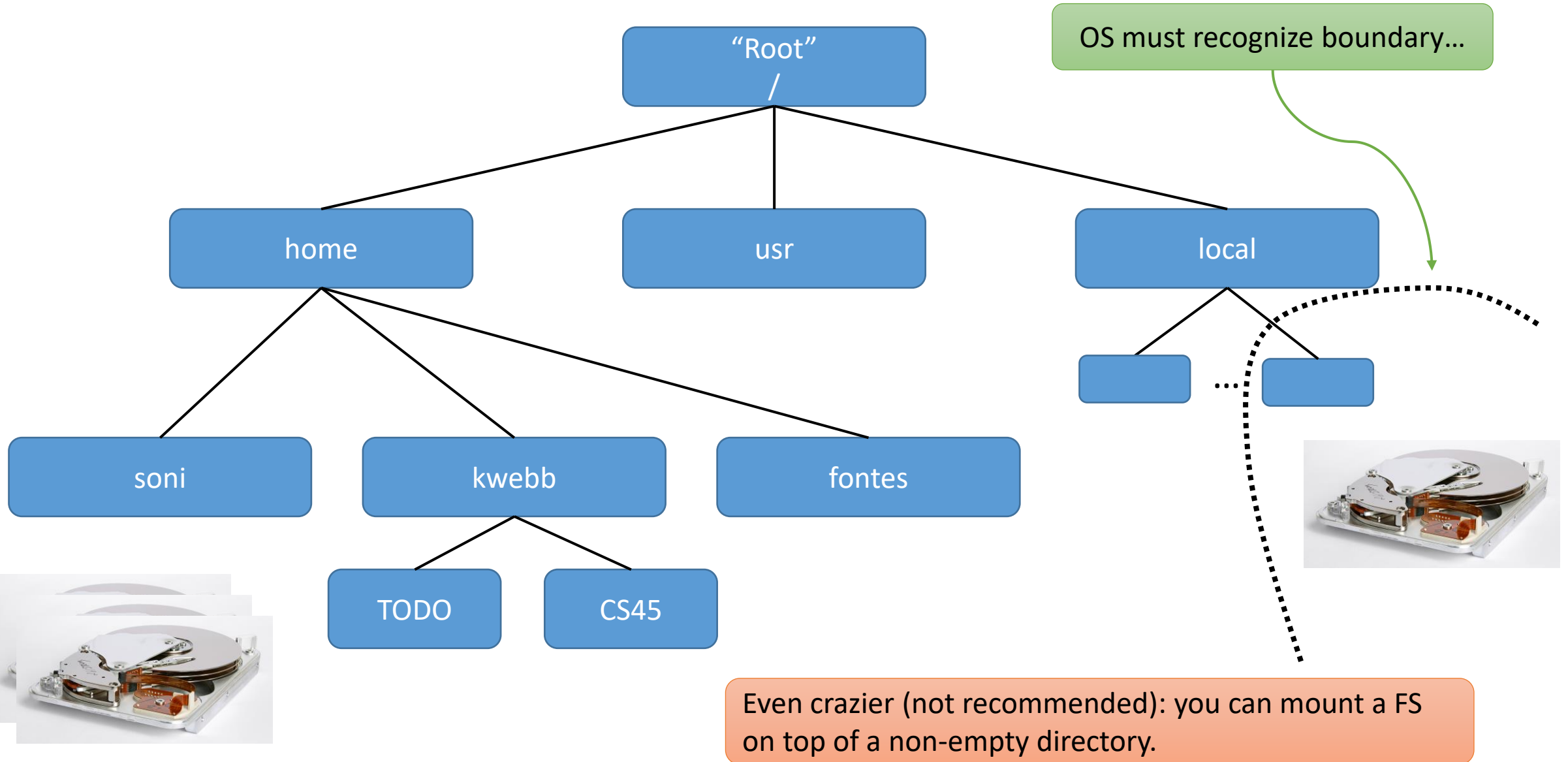| …	…
3	file_x …
…	…

Hardware
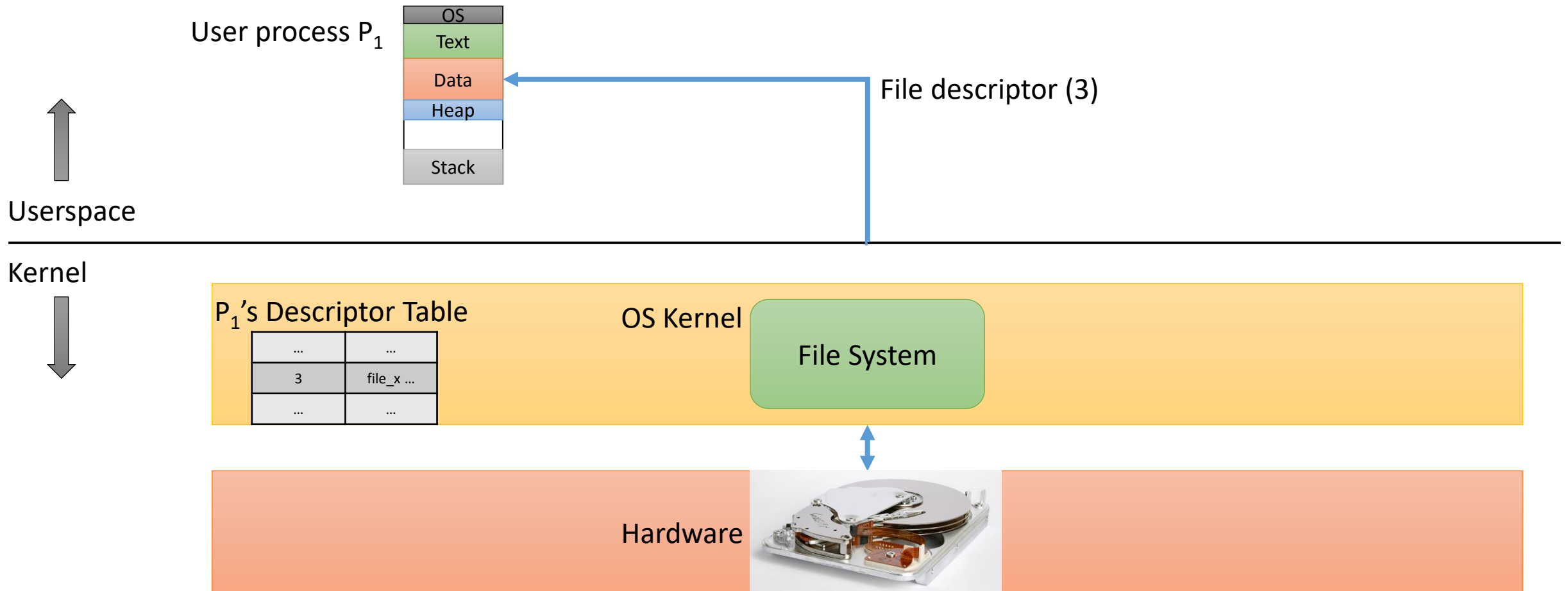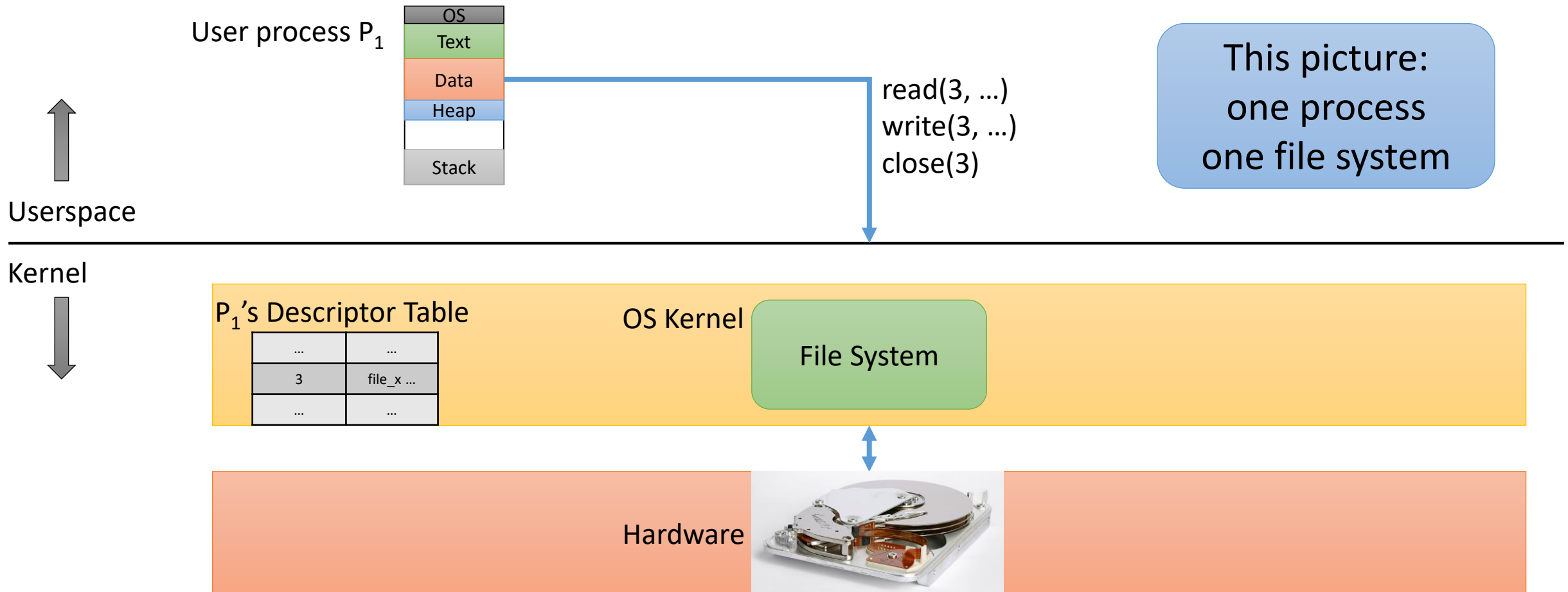
Challenges

- What if we have multiple disks?

- What if those disks use different file systems?
  - Example: CS department uses NFS for common files, /local for local disks
  - CS also: separate NFS mount for /scratch, Ameet has a research NFS mount

- The path in the file "tree" does NOT say anything about which FS files live on.

Single "name space", multiple backing stores

Single "name space", multiple backing stores

"Root"
/

home

usr

local

OS must recognize boundary…

soni

kwebb

fontes

TODO

CS45

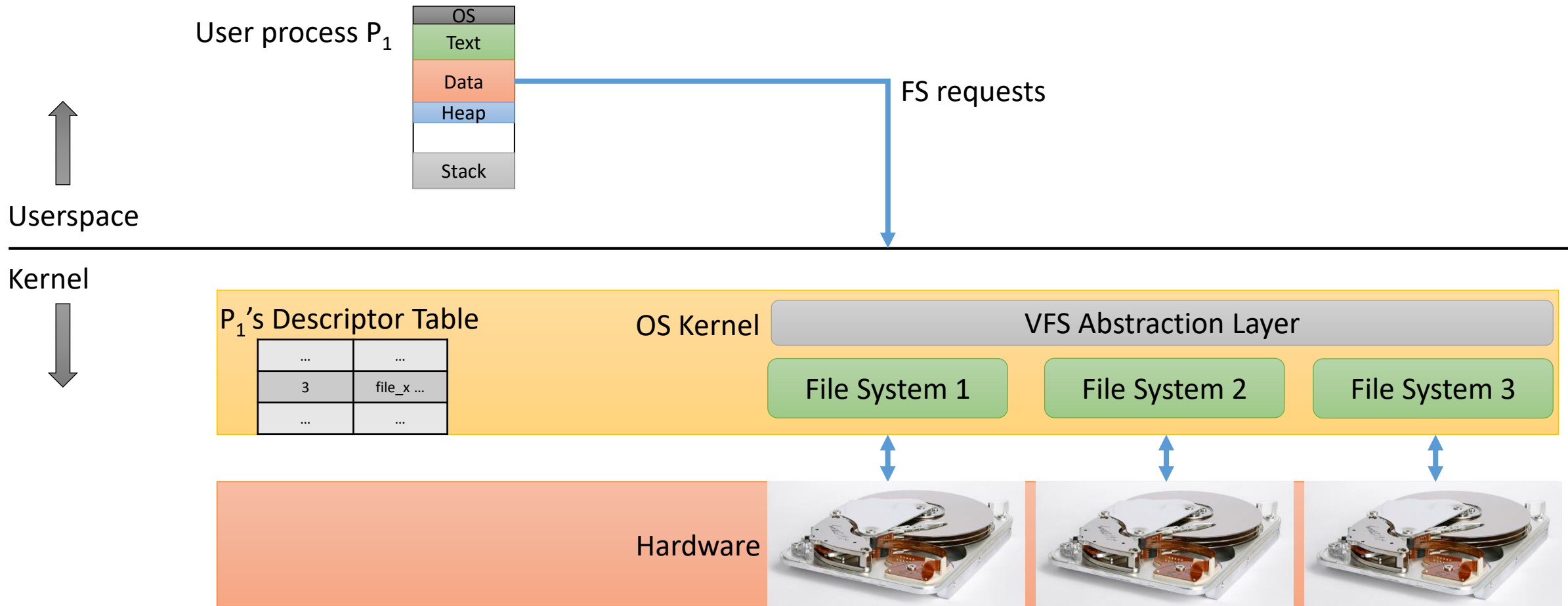Even crazier (not recommended): you can mount a FS on top of a non-empty directory.
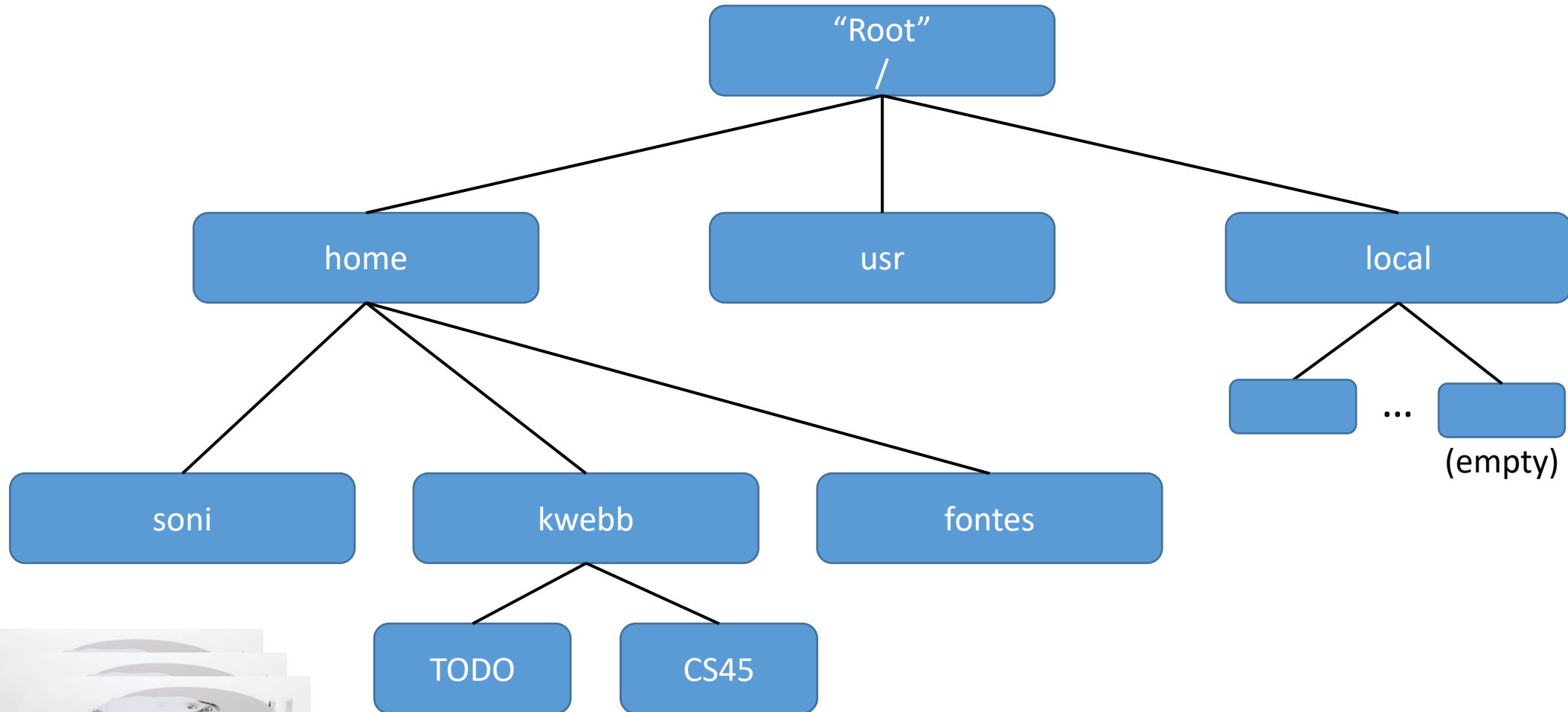
If you were tasked with solving this problem, what would you do?

- That is, how would you design the OS such that multiple file systems, possibly on different physical disks, can all share one file name space?
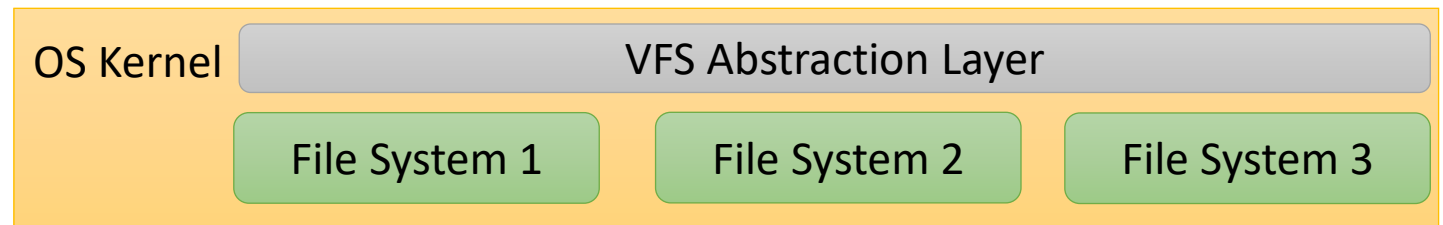
Virtual File System (VFS) Layer

- Userspace processes make system calls to interact with files:

User process P₁

OS
Text
Data
Heap
Stack

FS requests

Userspace

Kernel

P₁'s Descriptor Table | OS Kernel | VFS Abstraction Layer

...	...
3	file_x ...
...	...

File System 1	File System 2	File System 3

Hardware

VFS Layer

- Unifies the file name space and paths.
  - Paths all start from common root (/) and are passed to VFS layer.
  - VFS layer records which paths correspond to which FS.

- VFS translates application requests to appropriate low-level FS calls.

- Other benefits?

- Drawbacks?

OS Kernel	VFS Abstraction Layer		
	File System 1	File System 2	File System 3

Analogy: Instruction Set Architecture (ISA) interface
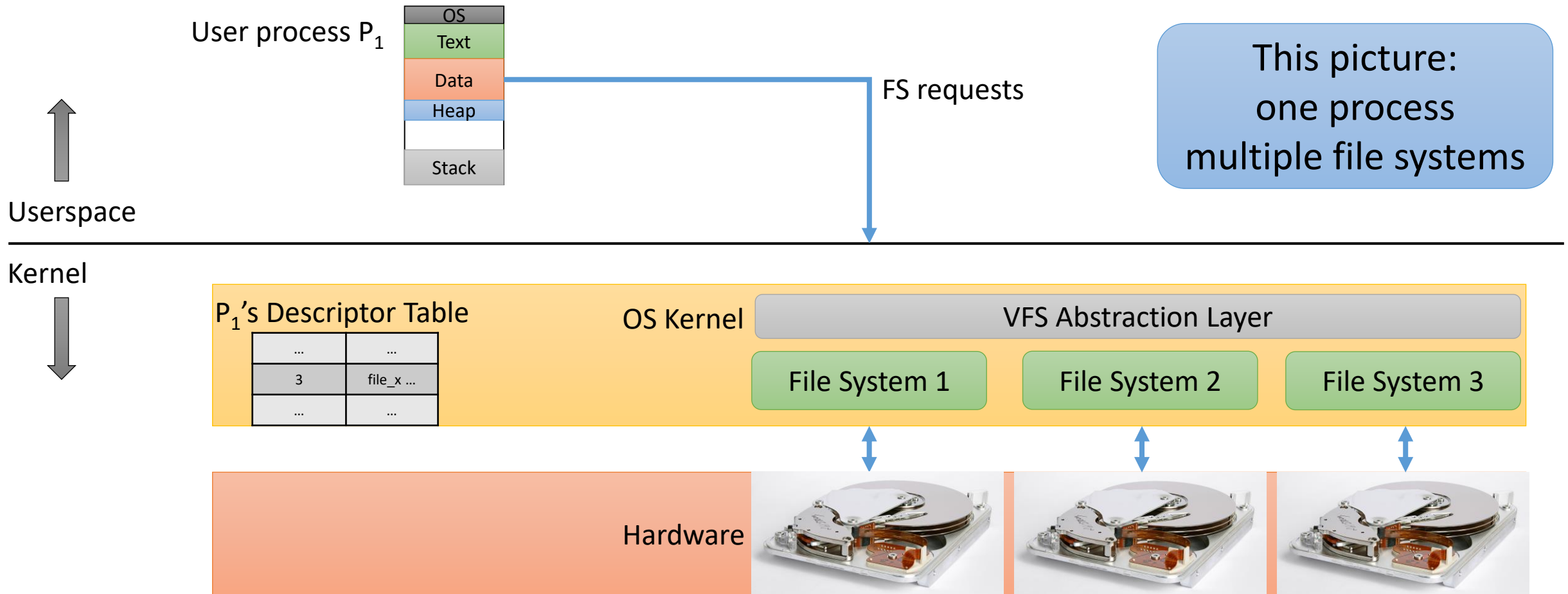
Having this VFS layer…

A. Is good for performance (why?)

B. Is bad for performance (why?)

C. Doesn't really mean much for performance (why not?)
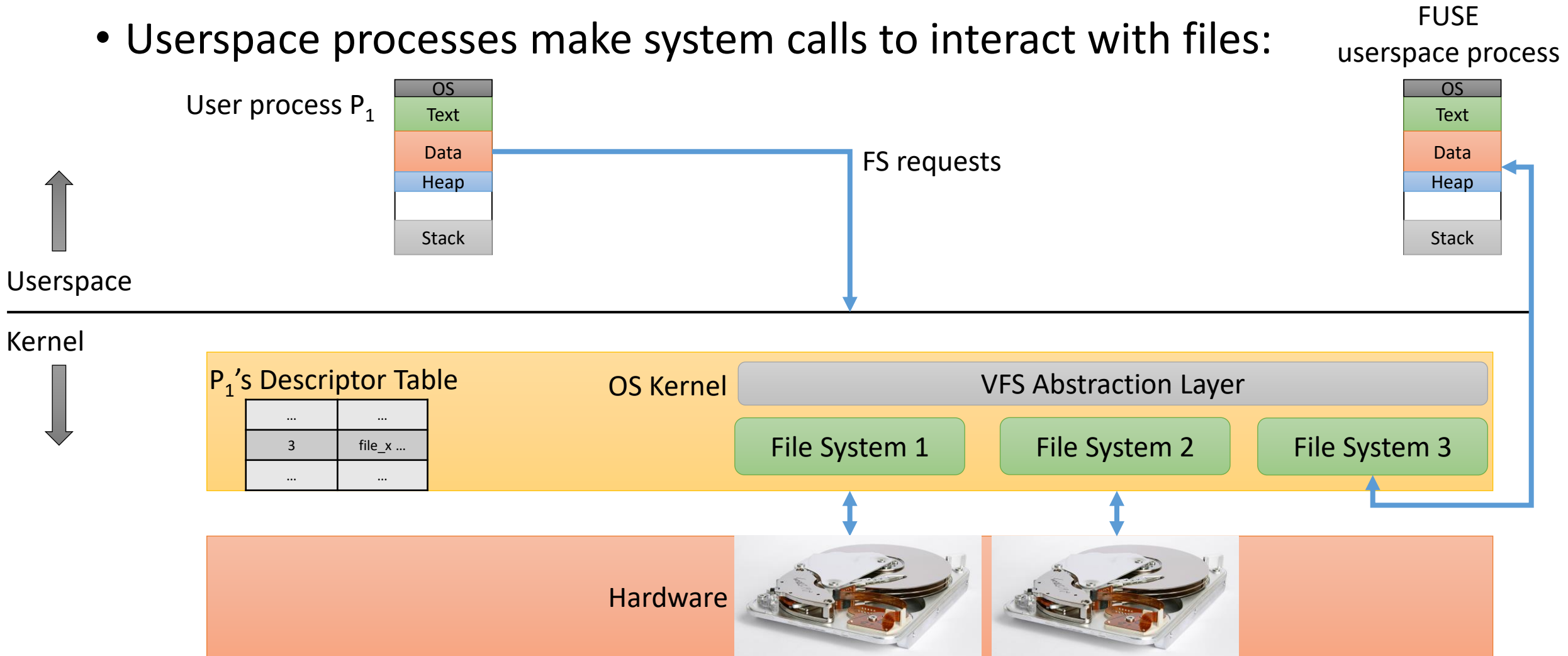
Virtual File System (VFS) Layer

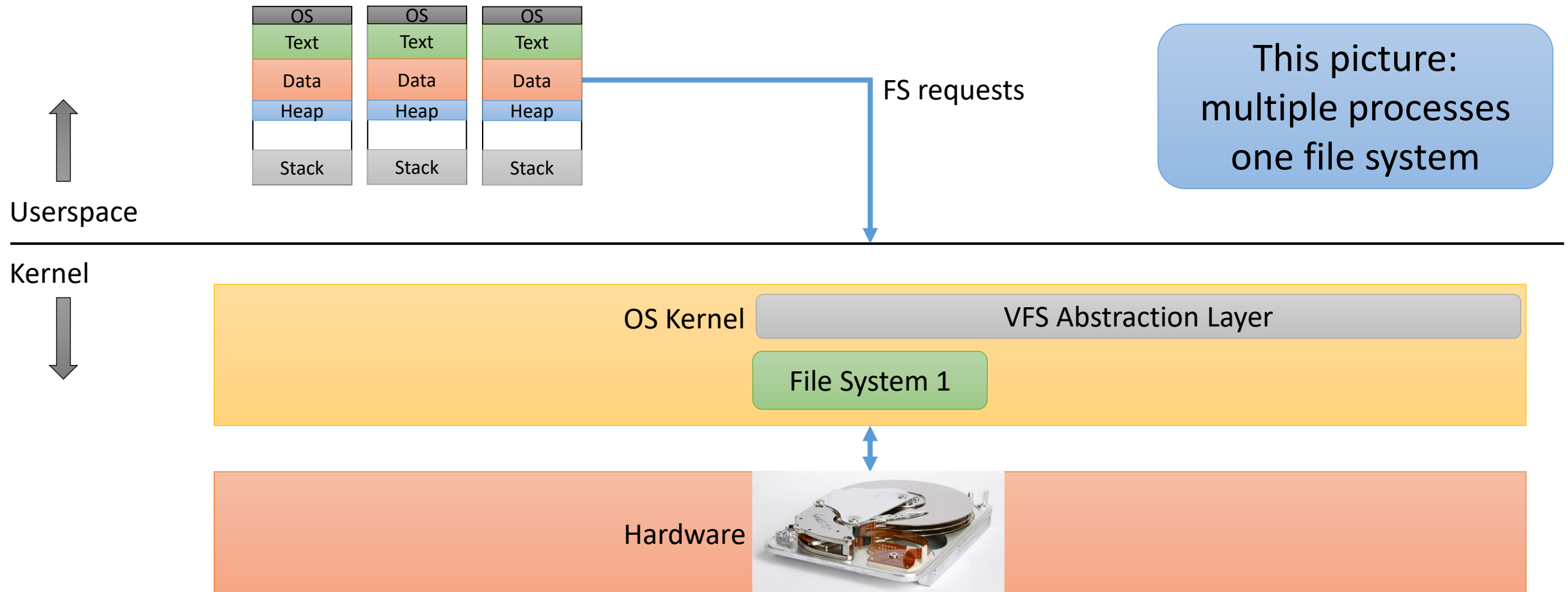- Userspace processes make system calls to interact with files:

User process P₁

| OS |
| Text |
| Data |
| Heap |
| |
| Stack |

FS requests

This picture:
one process
multiple file systems

Userspace

Kernel

P₁'s Descriptor Table

...	...
3	file_x ...
...	...

OS Kernel — VFS Abstraction Layer

File System 1   File System 2   File System 3

Hardware

FUSE

- Userspace processes make system calls to interact with files:

FUSE userspace process

User process P_1

OS
Text
Data
Heap
Stack

OS
Text
Data
Heap
Stack

FS requests

Userspace

Kernel

P_1's Descriptor Table

...	...
3	file_x ...
...	...

OS Kernel

VFS Abstraction Layer

| File System 1 | File System 2 | File System 3 |

Hardware

Multiple Concurrent Disk Requests

- Userspace processes make system calls to interact with files:

Userspace

Kernel

FS requests

This picture: multiple processes one file system

OS Kernel — VFS Abstraction Layer

File System 1

Hardware

Disk Request Queueing

- Recall I/O bound process behavior and the CPU scheduler:
  - If a process is making lots of I/O calls, let it run first, it will probably block

- Implication: If there are processes doing disk I/O on the system, they will often have an outstanding disk request.
  - If there are multiple such processes, there will often be a queue of disk requests that are waiting for service from the disk.

Wait a second…

- In this scenario, we have:
  - multiple processes, all wanting to use the same resource (disk).
  - only one process can use the resource at a time.
  - the OS needs to decide which one.

- This is a scheduling problem!
  - Might similar scheduling policies be relevant?

How about a simple policy like FIFO?
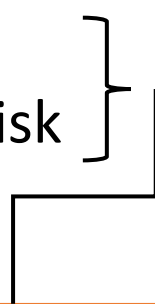
A. Good idea for disk scheduling. Why?

B. Bad idea for disk scheduling. Why?

C. It depends. On what?

Disk Scheduling

- Like CPU scheduling, disk scheduling is a policy decision
  - What *should* happen if multiple processes all want to access disk.

- Like CPU scheduling, the choice of metric influences the policy decision:
  - Priority: if a process is important, give execute its requests first
  - Throughput: maximize the data transfer from the disk
  - Fairness: give each process the same opportunity to access the disk

In some cases, trade-off between these two.

Disk Scheduling

- Unlike CPU scheduling, disk characteristics vary significantly. CPUs have different ISAs, but they *mostly* behave the same.

- For certain types of disks (solid state), FIFO might make a lot of sense (when targeting throughput):
  - The disk has no moving parts, so the fastest thing to do is just issue requests immediately as they come in.
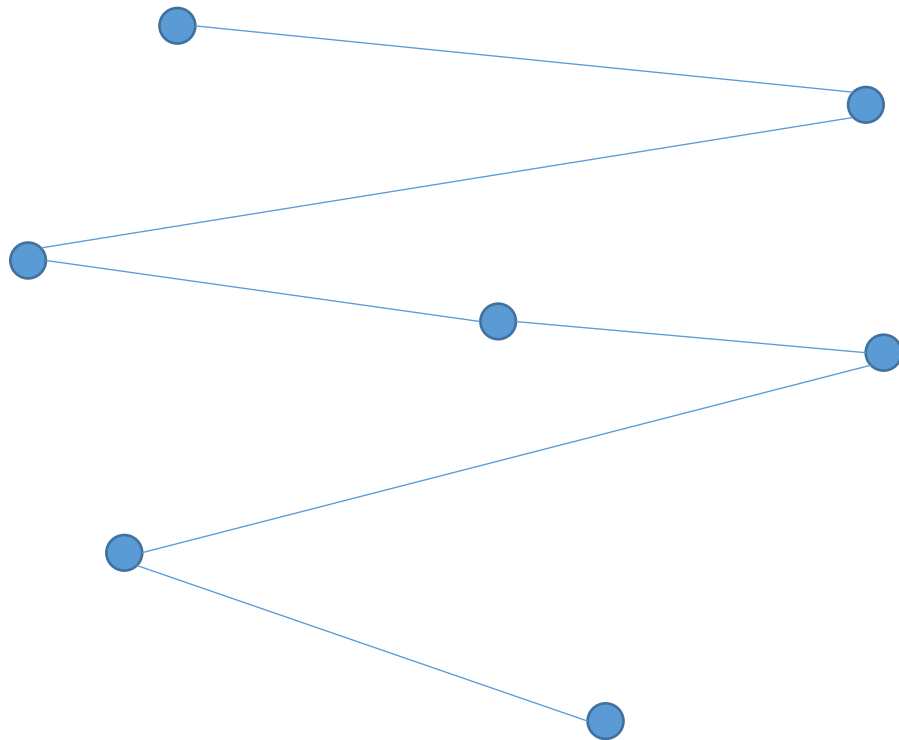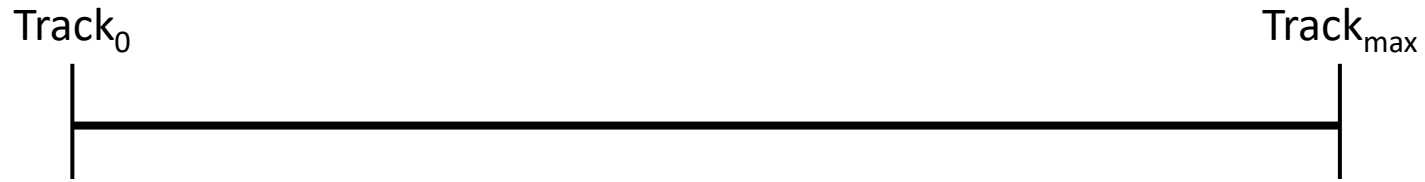
- For traditional spinning disks?

Disk Arm

- To simplify the discussion, let's assume the disk arm can move back and forth from left to right and right to left.

$Track_0$ $Track_{max}$

Request @ Track 20

time

$Track_0$ $Track_{max}$

wider horizontal distance -> longer seek time

FIFO Scheduling

Track$_0$ Track$_{max}$
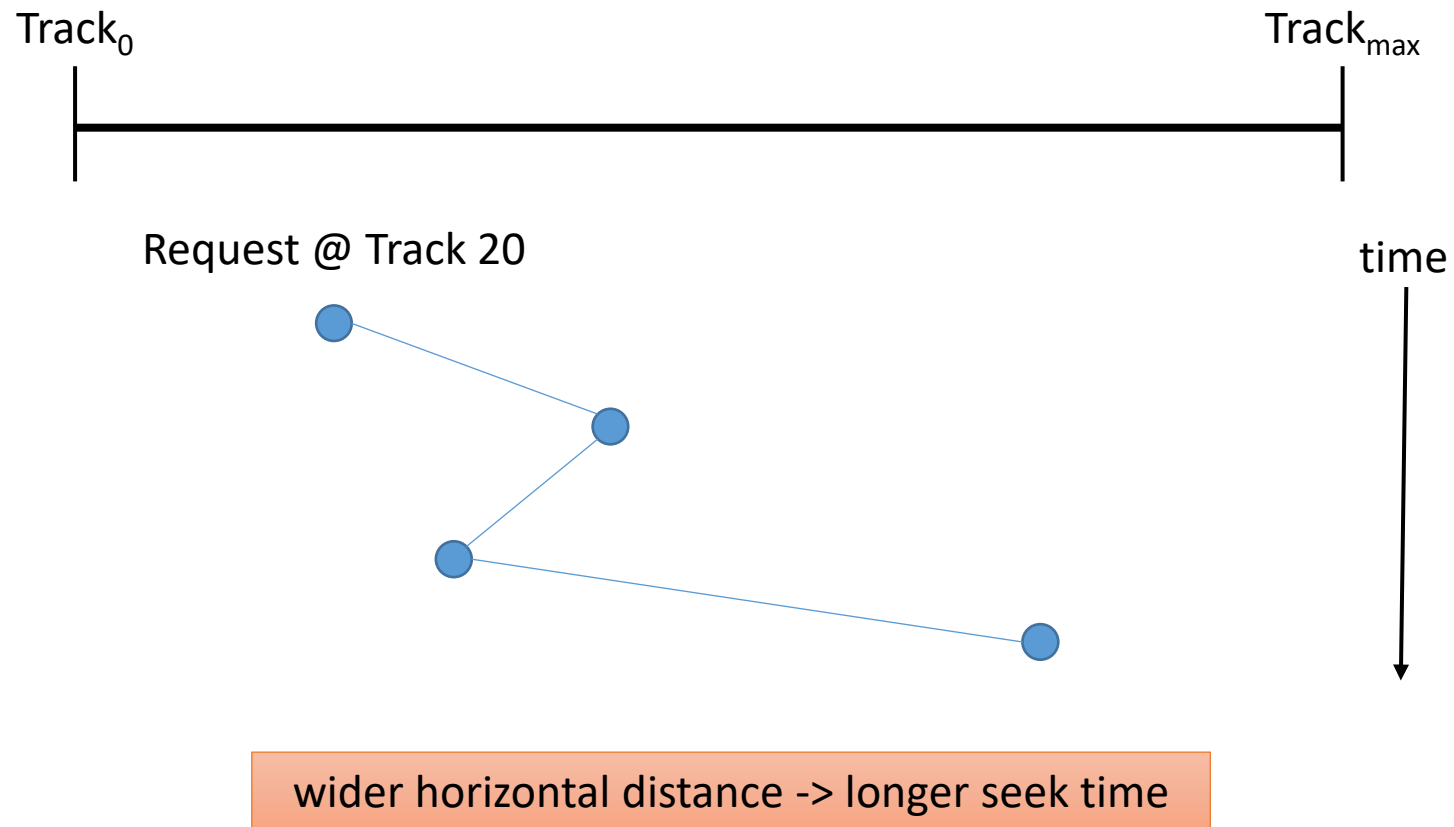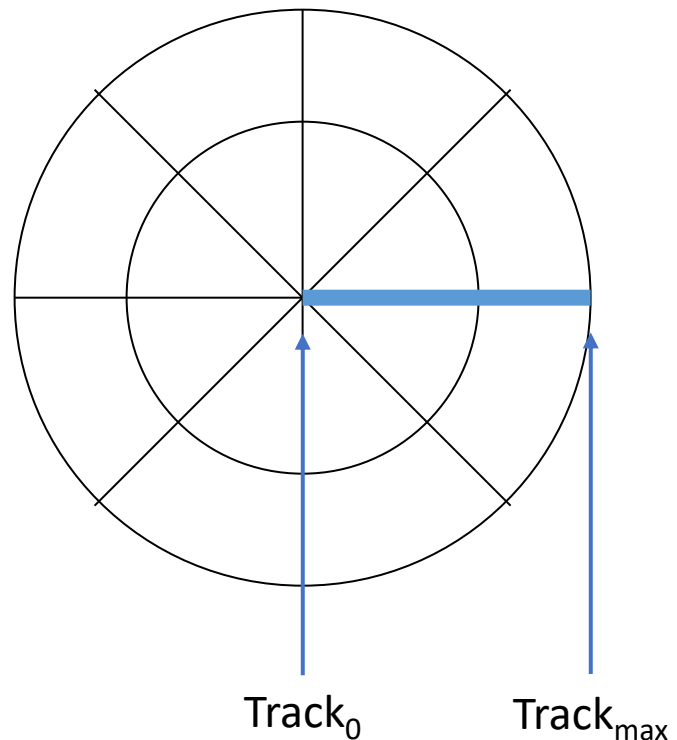
Performance is highly variable: depends on the order and track location of requests.

What might a better disk scheduling policy do? (for spinning disks)

- Policy to maximize throughput?

- Policy to maximize fairness?
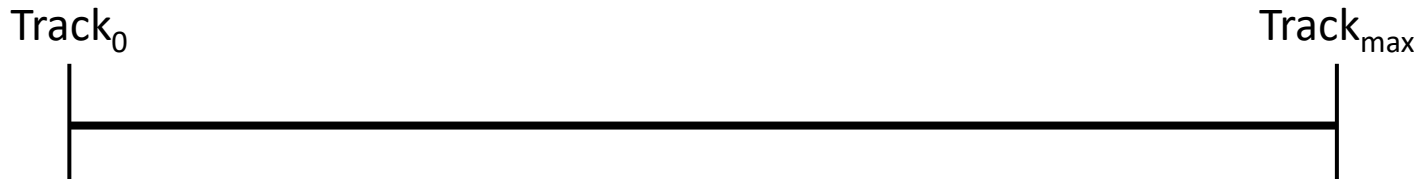
- Balanced policy?

$Track_0$ $Track_{max}$

What might the diagram look like for your policies?

Shortest Seek Time First (SSTF)

Track$_0$ Track$_{max}$
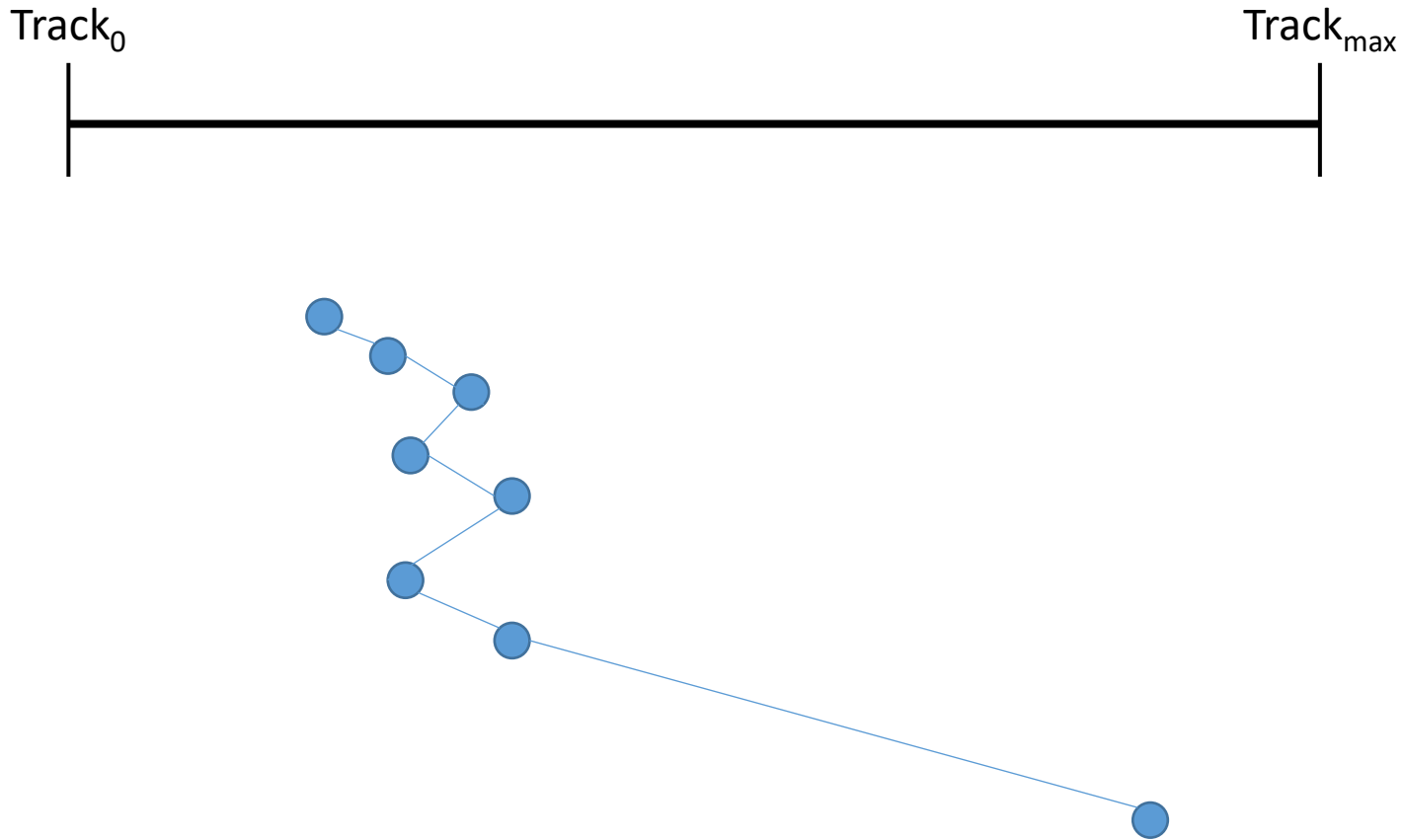
- Always choose the request that is closest to current arm position.

- Goal: minimize arm movement

- Good: less seeking, more throughput!

- Bad: potentially long delays for far-away locations (starvation)

Elevator (SCAN)

Track$_0$ .... Track$_{max}$
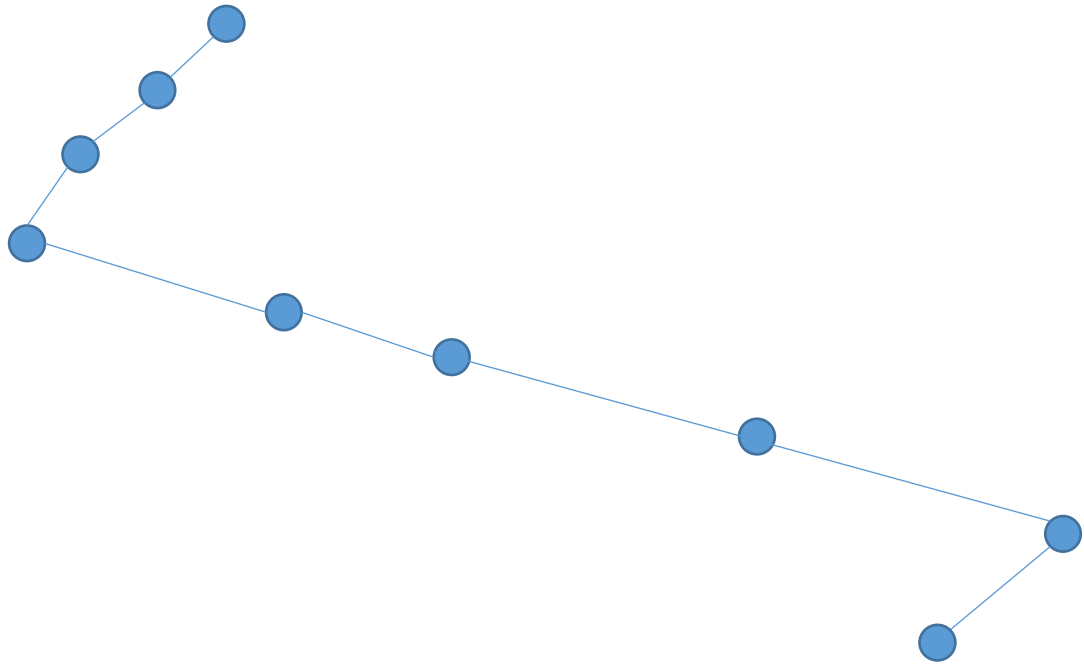
- Move end-to-end in one direction, they go back the other way.

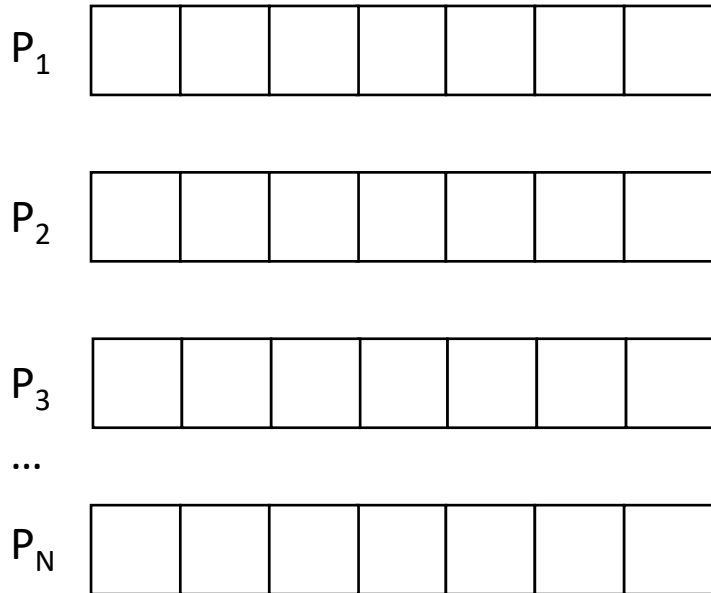- Goal: balance

- Intuition: like an elevator in a tall building.

Many other variants…

- Circular SCAN (C-SCAN)
- LOOK
- Circular LOOK (C-LOOK)
- …

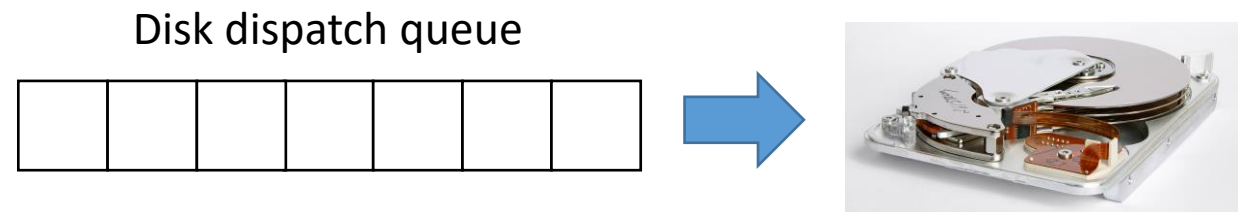- Some care about variance, fairness, performance, whatever…

Linux's Default Scheduler: CFQ

- Completely Fair Queueing (CFQ)
  - Not to be confused with the "completely fair scheduler (CFS)" for the CPU...

P_1 [| | | | | |]

P_2 [| | | | | |]

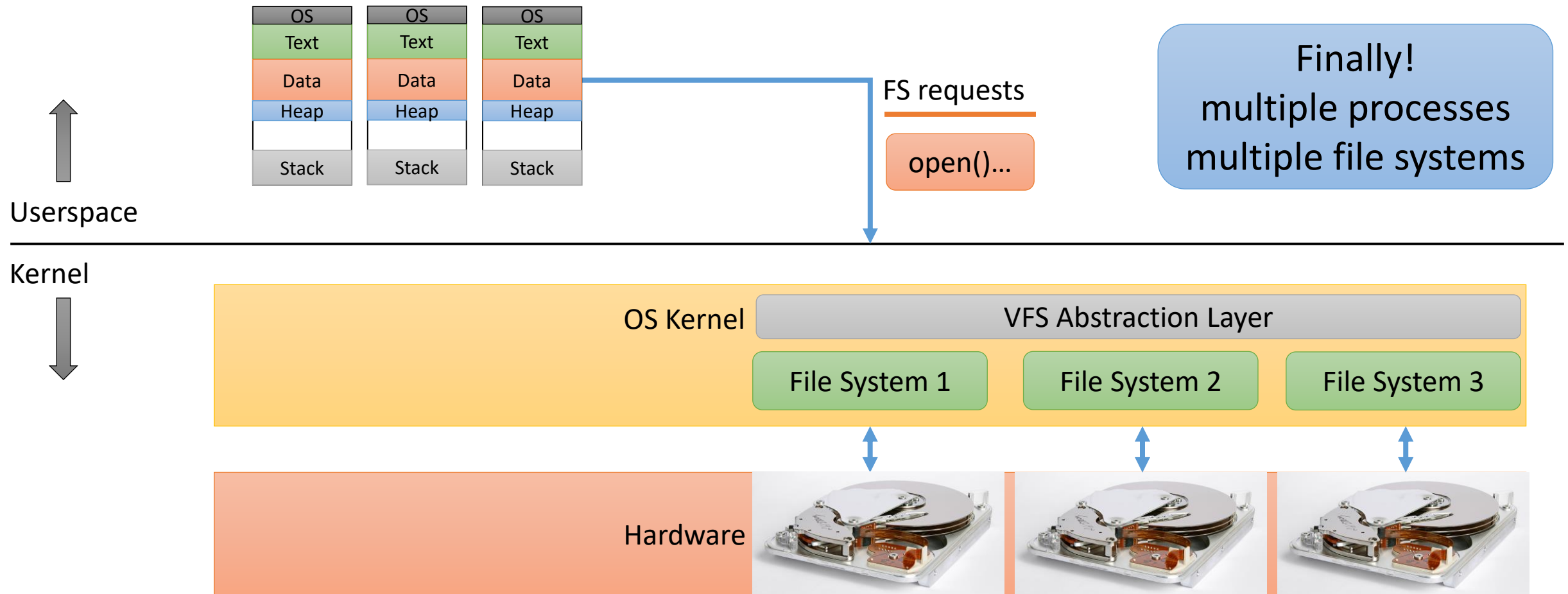P_3 [| | | | | |]

…

P_N [| | | | | |]

- Keep a disk request queue for each process.

- Move requests from process queues to dispatch queue in round-robin fashion (if equal priority).

- Dispatch queue can re-order for throughput.

Disk dispatch queue

[| | | | | |] →

The story so far…

- Userspace processes make system calls to interact with files:

OS | Text | Data | Heap | | Stack

FS requests

open()...

Finally!
multiple processes
multiple file systems

Userspace

Kernel

OS Kernel | VFS Abstraction Layer

File System 1 | File System 2 | File System 3

Hardware

Why do we have an `open()` call, as opposed to just `read()`ing or `write()`ing a file path?

A. To check file permissions

B. To improve performance

C. To lookup the file's location on disk

D. Two of the above

E. All of the above

Improving Performance

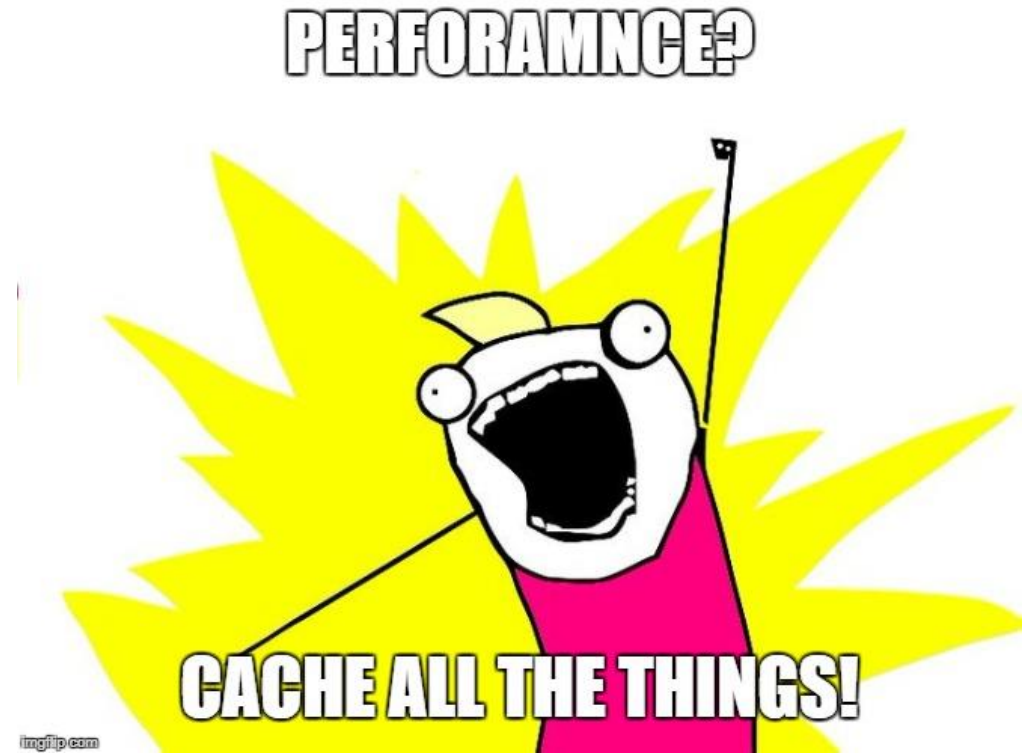1. Take advantage of locality!

2. Changes to FS structure.

Improving Performance

1. Take advantage of locality!

2. Changes to FS structure.

Where should we do FS / disk caching?

A. On the disk device

B. In the OS file system implementation

C. In the OS VFS layer

D. In applications
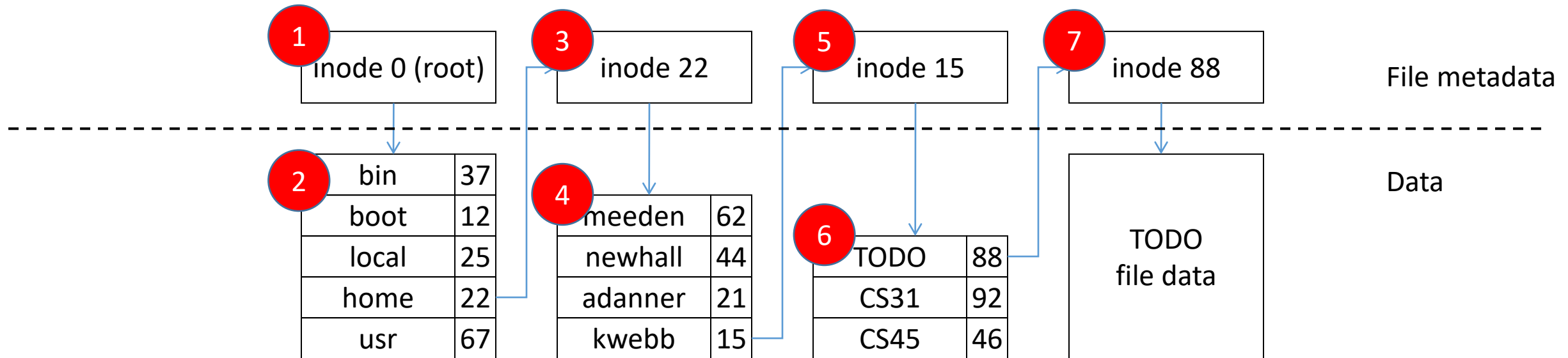
E. Somewhere else (where?)

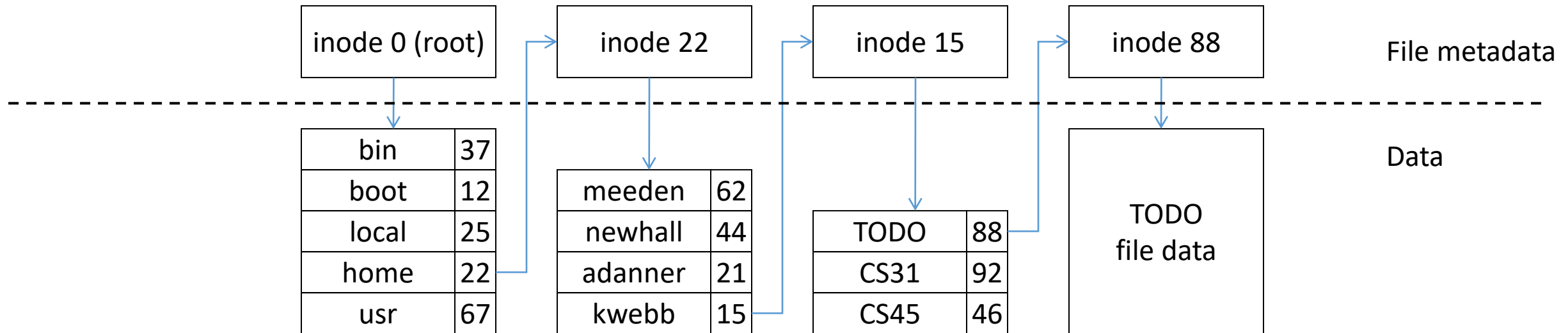# What sort of information should we cache?

Caches

- Directory cache – caching the results of directory lookups

- inode cache – inode data for frequently/recently used files

- Block cache – data blocks for frequently/recently used files

Directory Cache

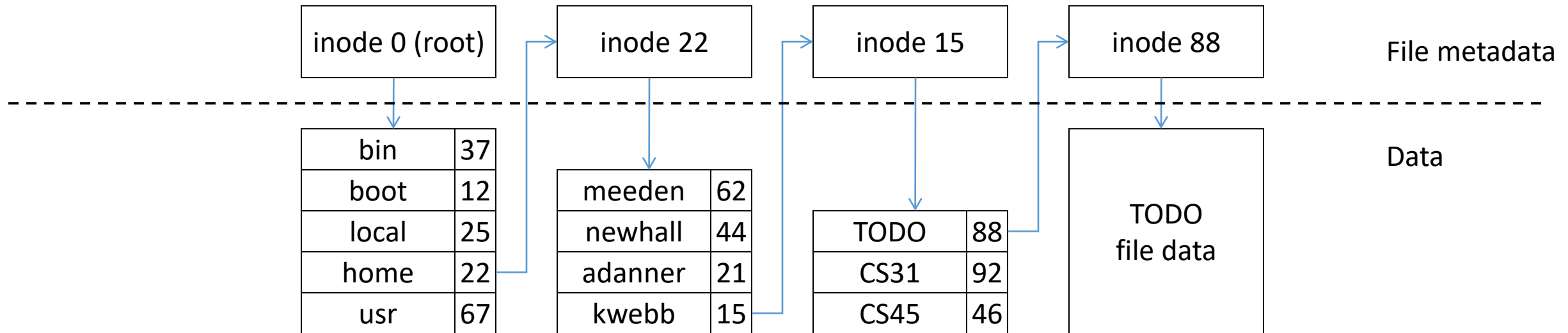- Recall path lookup: /home/kwebb/TODO

- This is an *extremely* expensive operation: several disk accesses.
  - In this example, at least seven disk reads just to get the file's inode (metadata)

Directory Cache

inode 0 (root)

inode 22

inode 15

inode 88

File metadata

- -

Data

bin	37
boot	12
local	25
home	22
usr	67

meeden	62
newhall	44
adanner	21
kwebb	15

TODO	88
CS31	92
CS45	46

TODO file data

- Solution: in the VFS layer, keep a small table in software.
    - Combination of hash table (fast lookup) and linked lists (for LRU replacement)
    - Maps commonly used paths to the final inode number.

- (The structure of this cache is similar to that of the block cache – coming up soon)

Directory Cache

inode 0 (root)	inode 22	inode 15	inode 88	File metadata

bin	37
boot	12
local	25
home	22
usr	67

Data

meeden	62
newhall	44
adanner	21
kwebb	15

TODO	88
CS31	92
CS45	46

TODO
file data

- In this example, every system in CS cluster probably knows where /home is all the time. Everybody uses /home!

- On my desktop (sesame) it should always know where /home/kwebb is too…

inode cache

- If a file is accessed frequently, it will likely have its inode read/written frequently too.
  - Especially true when writing – update size, block count, modification time

- It would be crazy to have to read the inode from the disk every time!

- Solution: keep another table…
  - Similar structure to directory cache and block cache.
  - Maps inode number to inode data contents.

Block Cache

- In-memory cache of recently used disk data (blocks).

- On read or write operation:
  - Check if block is in cache; if it is, great! No disk access necessary.
  - If not in cache, access disk, place data in cache.

- Eventually, like any cache, it will fill up: need a replacement policy

- LRU: keep a list ordered by access
  - Remove old block at head of list
  - Put new block at tail of list

Comparing disk caching and virtual memory...

- In both cases: the user requests some data.
    - For VM: it's a page of memory
    - For file: it's a block on the disk

- In both cases: the data is either in memory or on disk.

- In both cases: the data in memory is a limited subset and we need a replacement scheme that exploits locality.

# Why does it make sense to use full LRU for the disk block cache, but for page replacement, we wanted an approximation?

A. The block cache is more important than memory.

B. Disk accesses are slower than memory accesses.

C. A system call has to be made for disk data accesses.

D. The disk block cache will be smaller, so less state will be necessary.

E. Some other reason(s) (what?).

Block Cache

- Two primary concerns:
    1. Fast to access (is the block in the cache?)
    2. Block replacement (which block should we evict?)

- Data structures:
    - Hashed access (for 1)
    - Linked LRU list (for 2)
    - Independent from one another!

Searching the Block Cache

- Green
  - Fast accesses
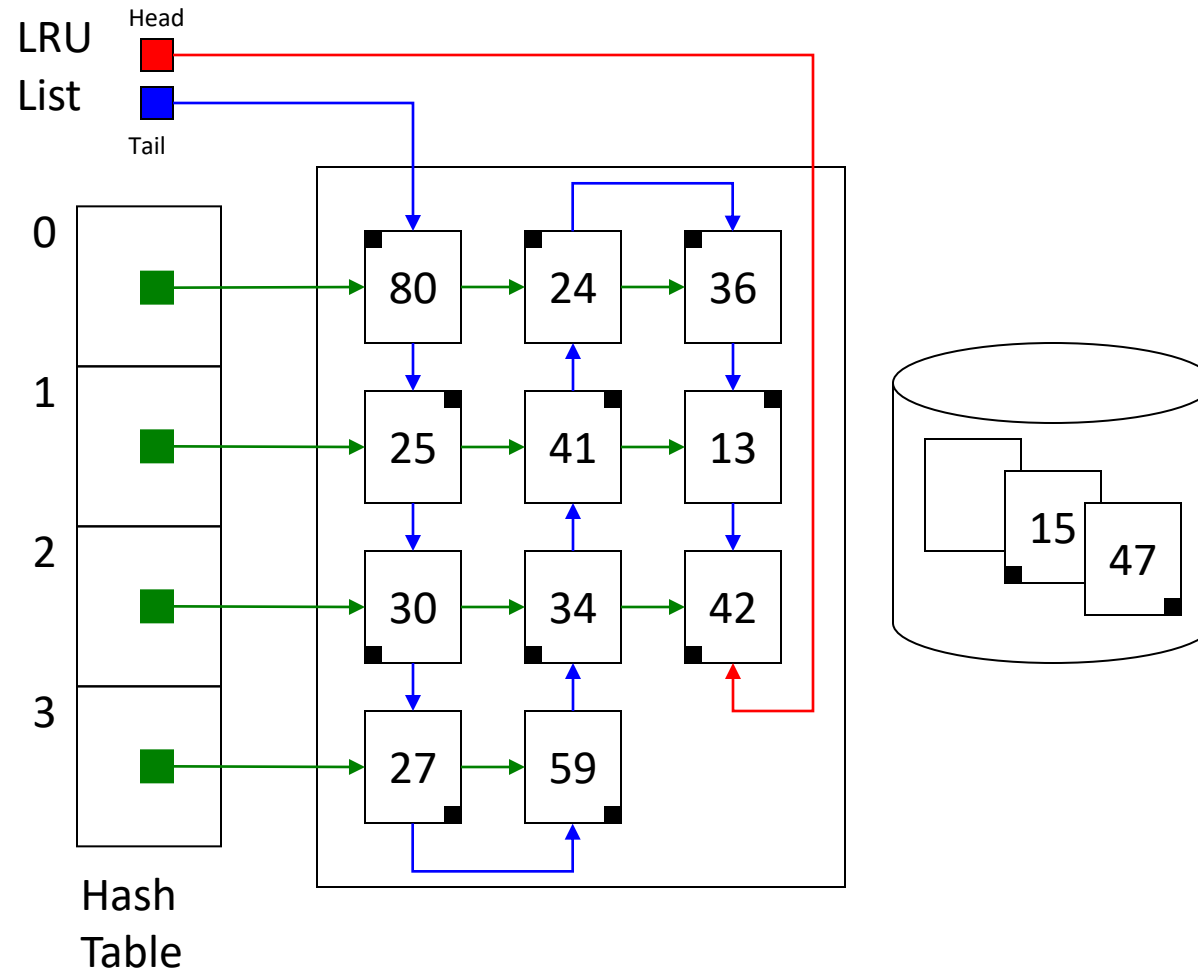 - Hash table

- Red/Blue
  - LRU Pointers
  - Linked list
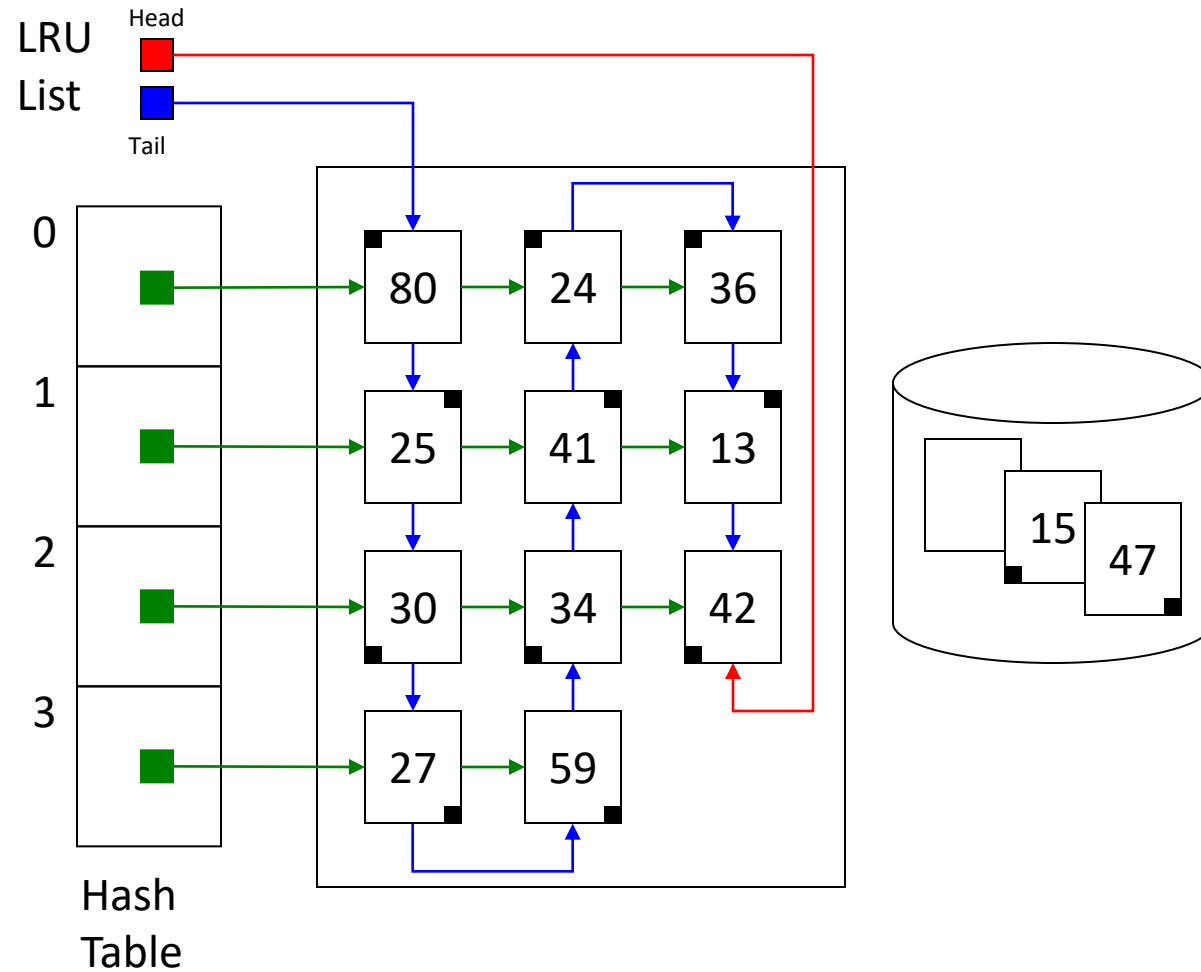
Searching the Block Cache

- For this example:

  - "Hash" is block num % 4

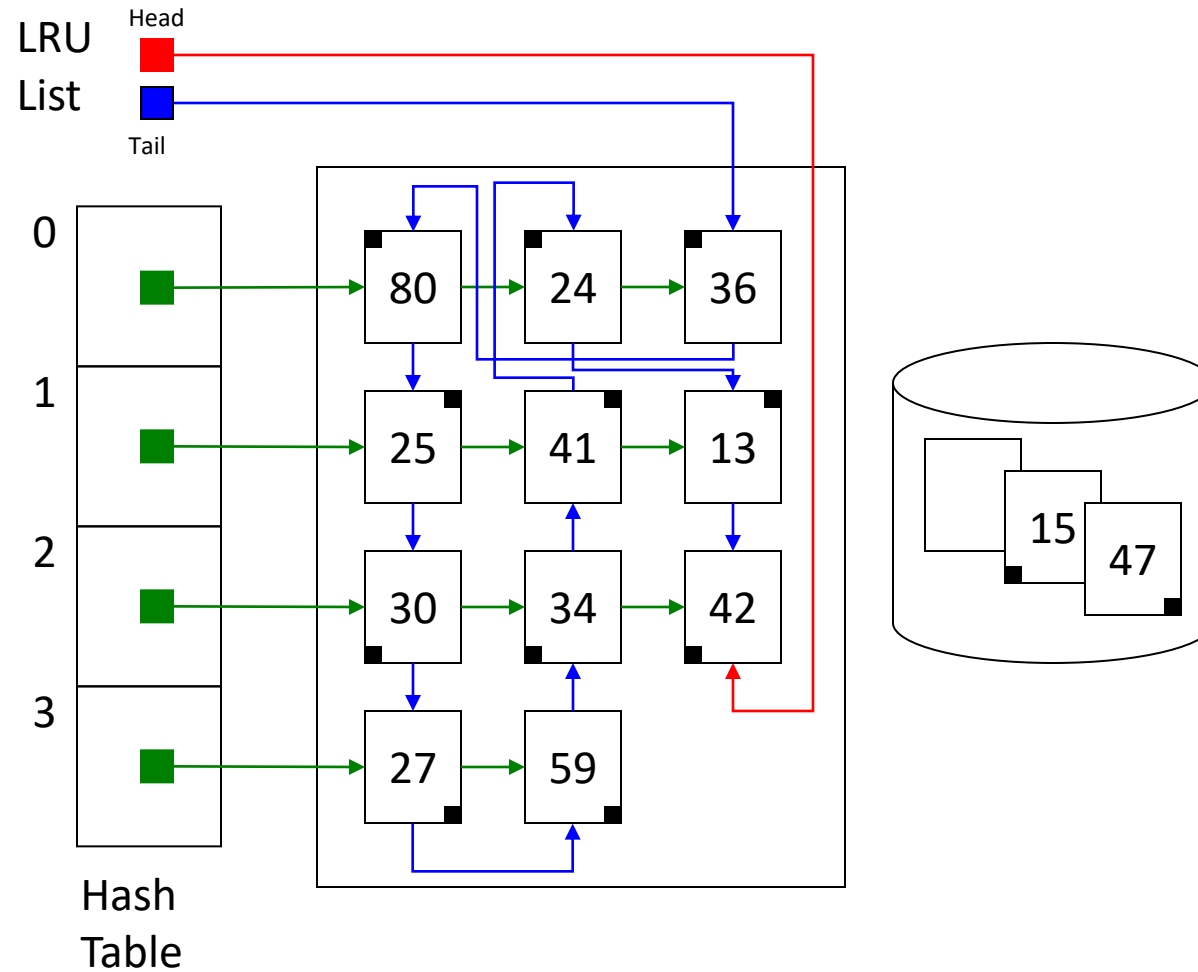 - Total number of blocks allowed in cache is 12, but they can be in any row.
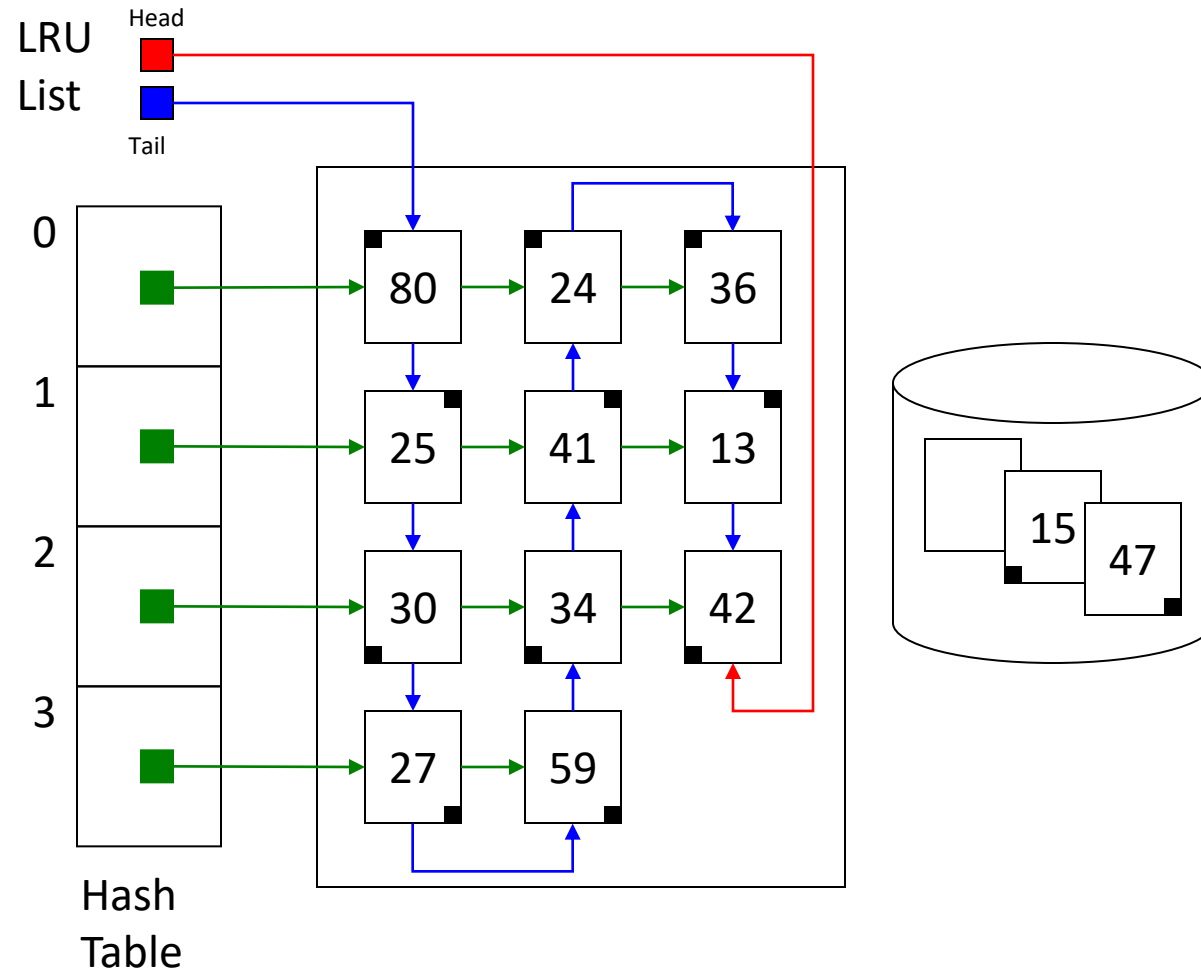
Searching the Block Cache
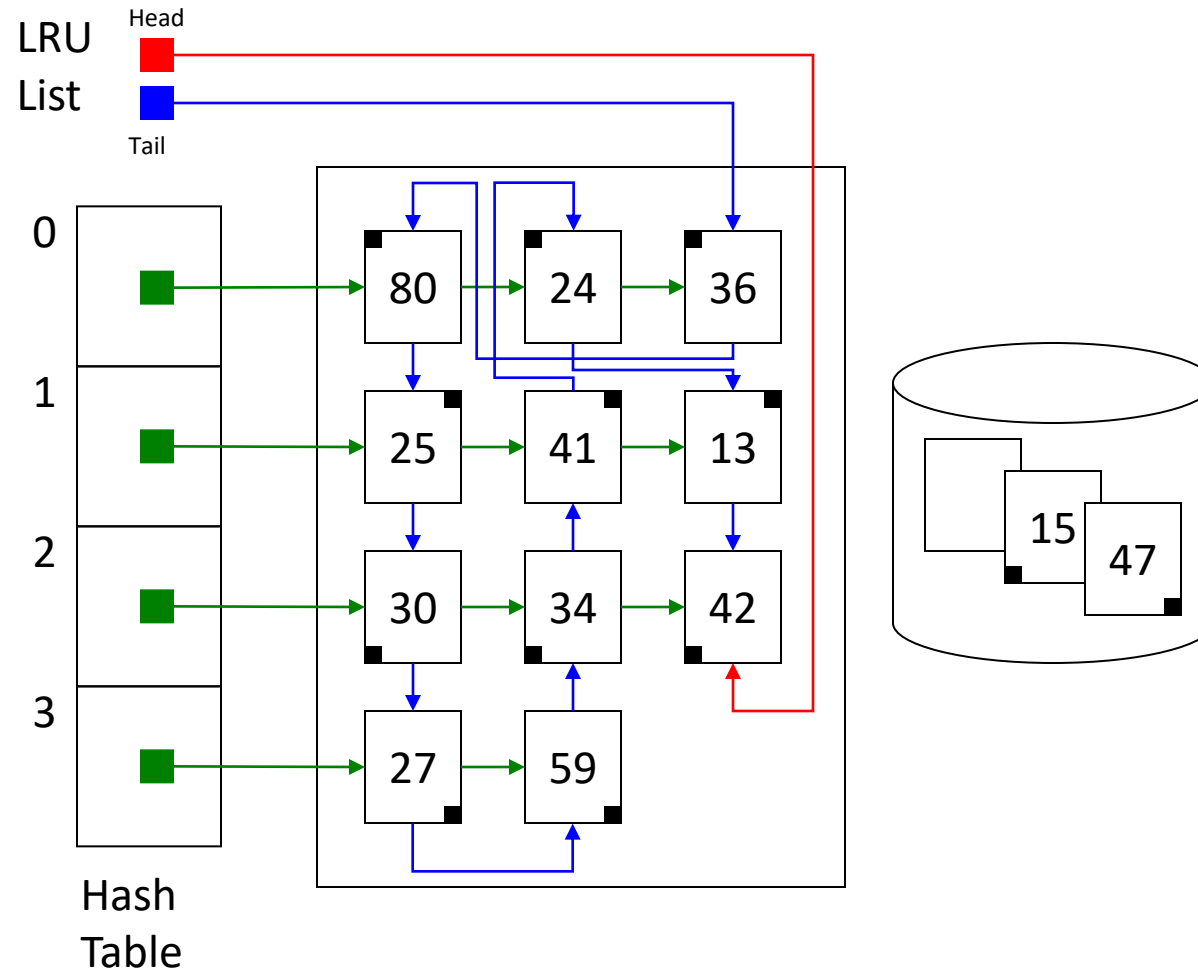
- Read (36)

- 36%4 = 0

- Search list 0 for 36

Searching the Block Cache

- Read (36)
- 36%4 = 0
- Search list 0 for 36
- Cache hit!
- *Update LRU list*

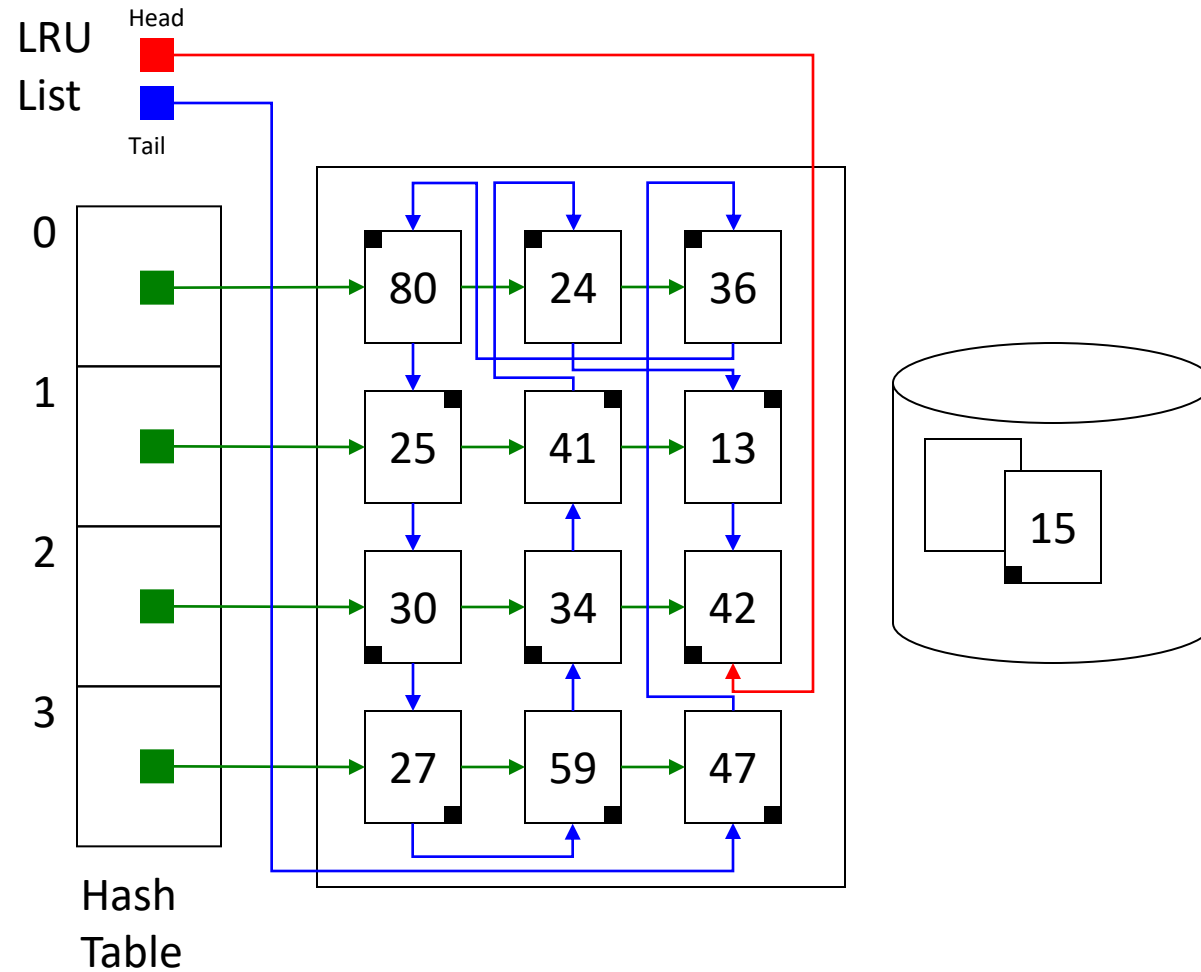# Searching the Block Cache
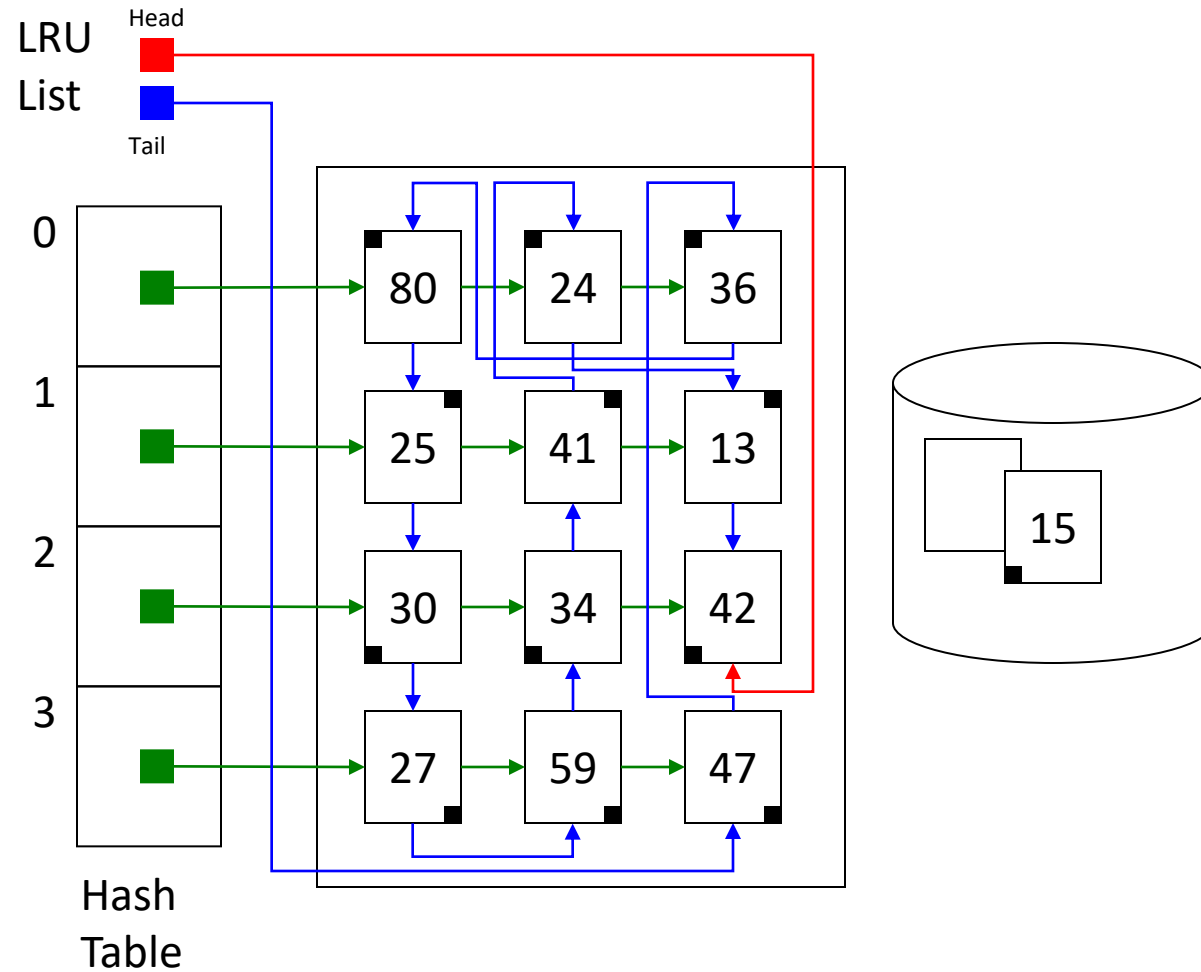
- Read (47)
- 47%4 = 3
- Search list 3 for 47

Searching the Block Cache

- Read (47)

- 47%4 = 3

- Search list 3 for 47
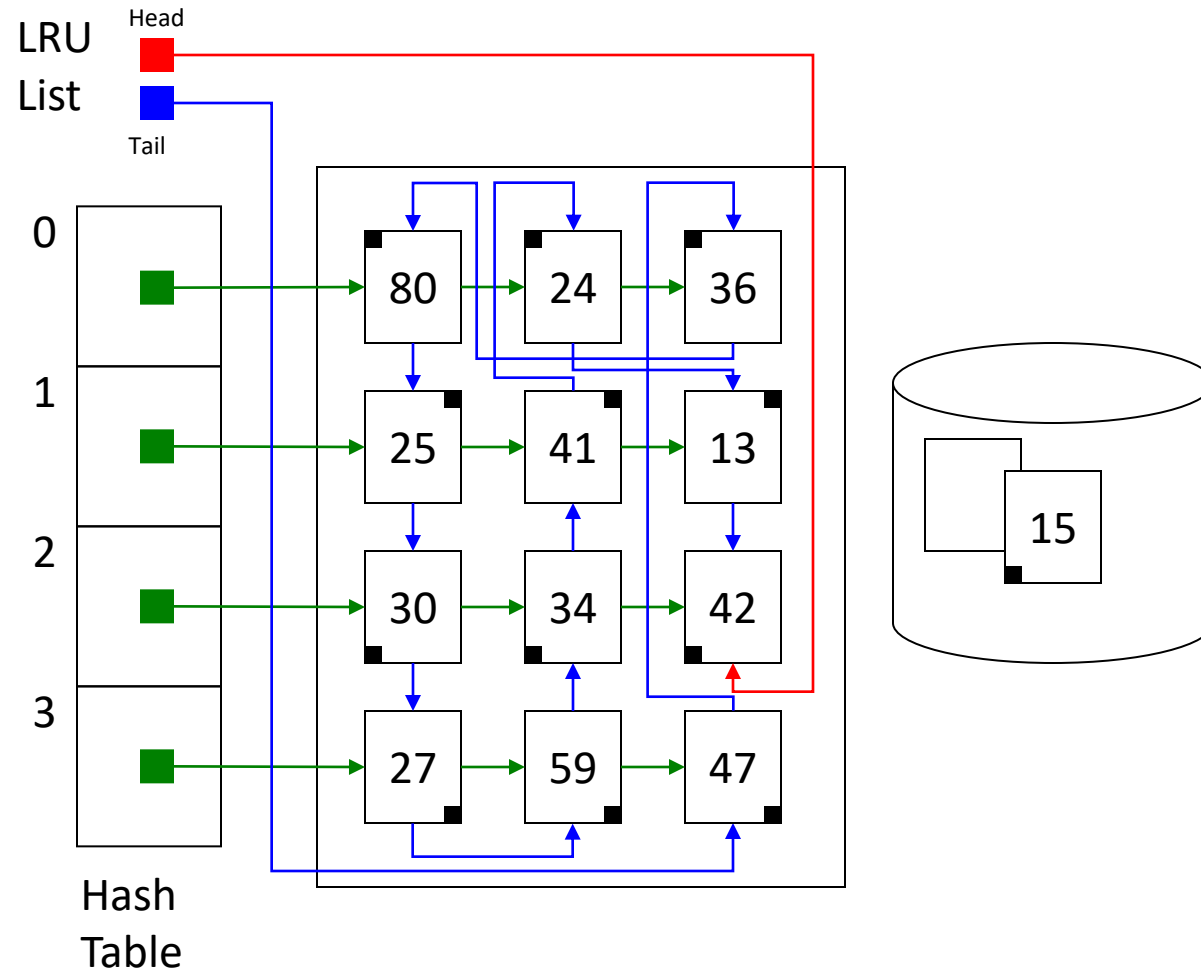
- Cache miss!

- *Retrieve 47*

- *Update LRU list*

Searching the Block Cache

- Read (15)

- 15%4 = 3

- Search list 3 for 15
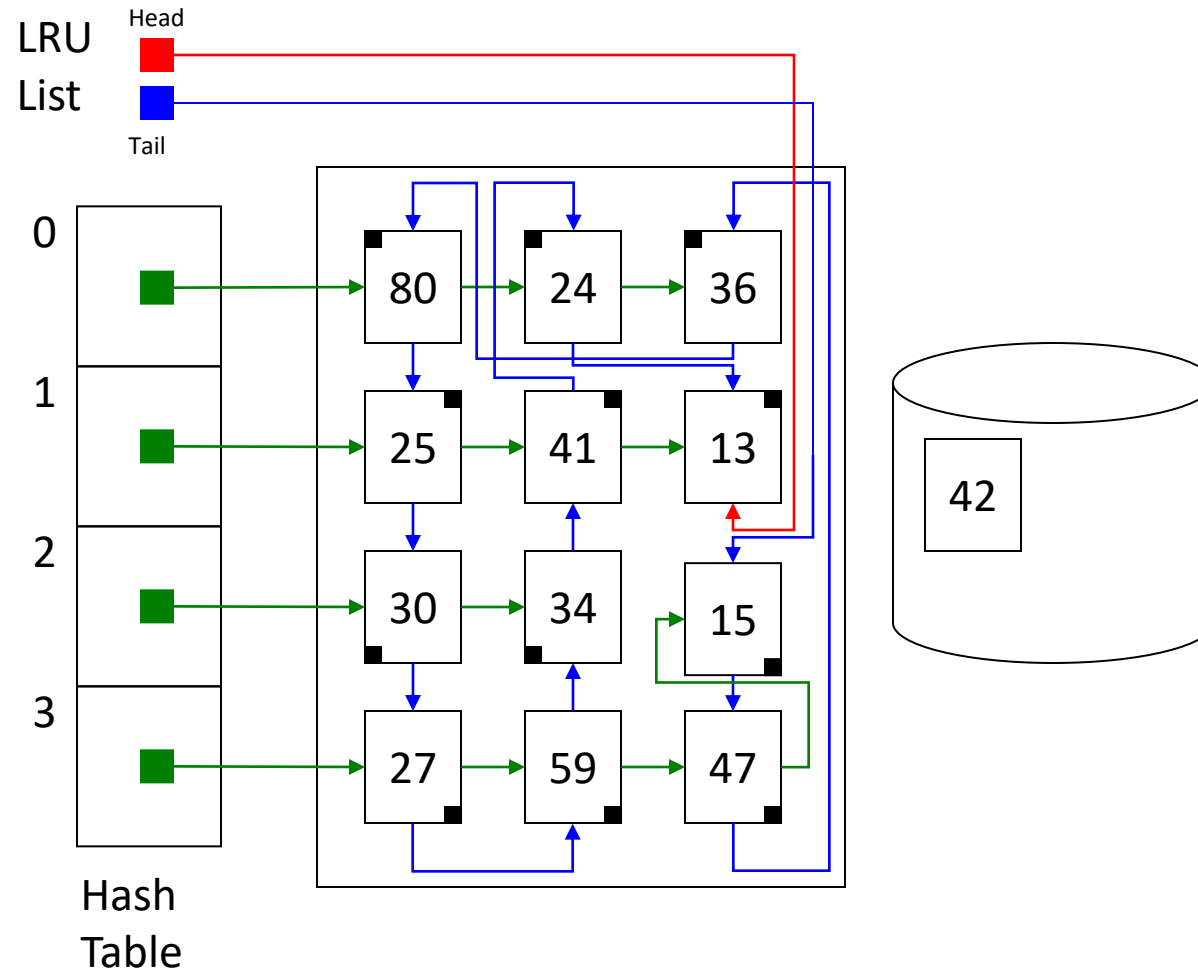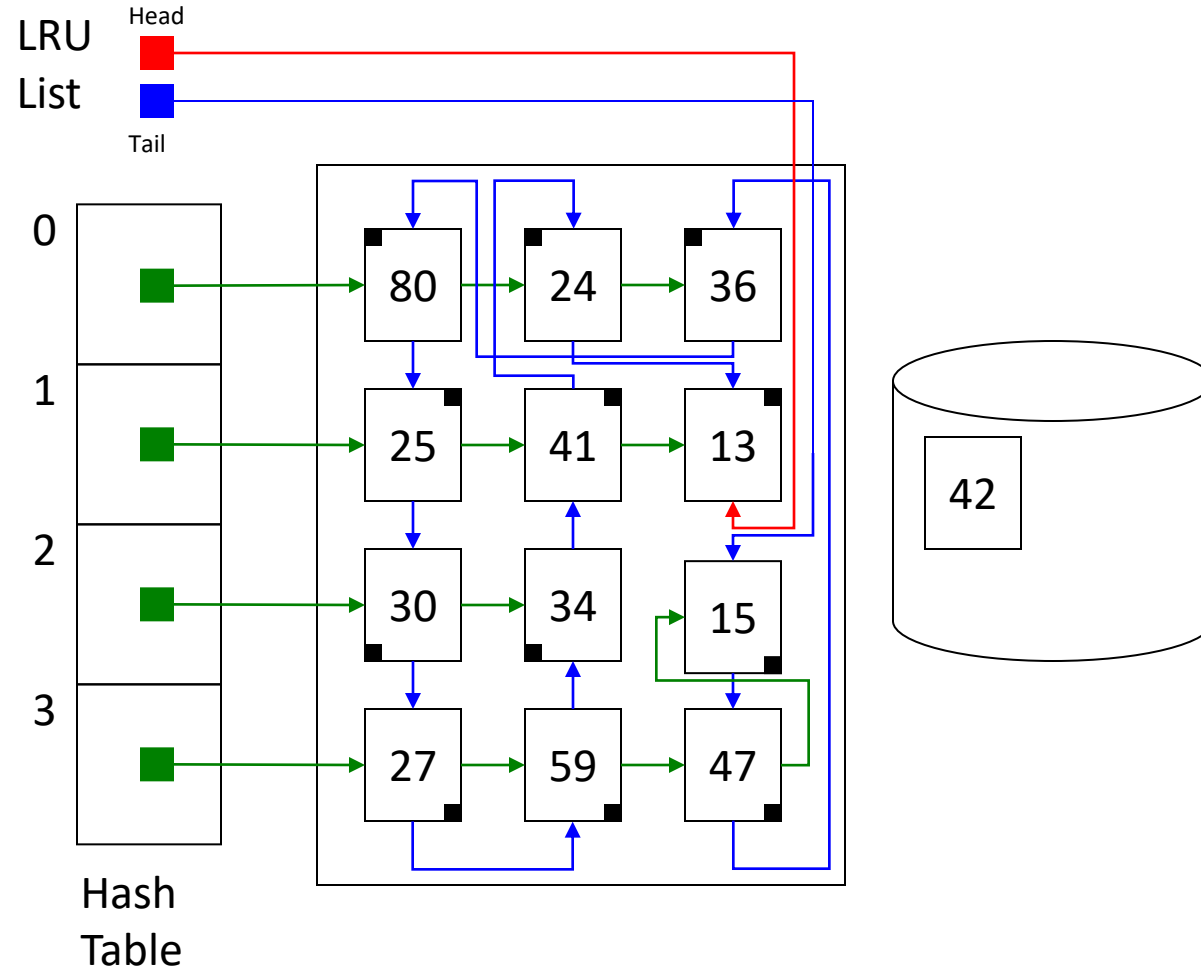
Searching the Block Cache

- Read (15)
- 15%4 = 3
- Search list 3 for 15
- Cache miss!
- *Remove 42*
- *Retrieve 15*

Cache Structure

- This same general structure is used for the other caches too.

- Most of you probably did this in your lab 4 LRU implementation.

Prefetching ("readahead")

- Observation: the disk is slow to react when we ask it for something (moving parts), but once it finds the data, it can deliver it quickly.

- Idea: Look for patterns in file accesses.

- Simple example: application calls read() asking for data from the file's first block. Then the second block, then the third block…
  - This pattern looks sequential: request 4th and 5th block from the disk now
  - Being "wrong" removes old, unused data (LRU), "right" minimal latency

Consistency

- After you perform a write, the next read should retrieve the most recently written changes.

- This can be a *very* complex problem in a distributed system.

- Not as difficult when we're talking about one disk.
  - Caching can introduce some important decisions.
  - A write is "safe" or "reliable" only if it's written *to the disk.*
  - For performance, we'd like to write the data to the in-memory block cache.

What about writes? When should we write data to the disk to ensure reliability?

A. Write blocks to disk immediately, to ensure safety.

B. Write blocks when we evict them from the cache, to maximize performance.

C. Write blocks when the user explicitly tells us to, since the user/application knows best.

D. Write blocks to disk at some other time.

USB Disk Drives

The disk was not ejected properly. If possible, always eject a disk before unplugging it or turning it off.

To eject a disk, select it in the Finder and choose File > Eject. The next time you connect the disk, Mac OS X will attempt to repair any damage to the information on the disk.

OK

USB2.0 Flash Disk USB Device Properties

General | Policies | Volumes | Driver | Details

Removal policy

○ Quick removal (default)
Disables write caching on the device and in Windows, but you can disconnect the device safely without using the Safely Remove Hardware notification icon.

○ Better performance
Enables write caching in Windows, but you must use the Safely Remove Hardware notification icon to disconnect the device safely.

Writing to Disk

- When writing metadata, write to disk immediately (write-through)
  - inode changes, directory entries, free block lists
  - metadata is usually small (update one block), critical for reliability

- When writing data, write it to memory cache. Flush to disk when…
  1. User says to do so.
  2. The block is getting evicted from the cache.
  3. The FS is being unmounted.
  4. Enough time has passed (approx. 5-30 seconds)

Improving Performance

1. Take advantage of locality!


2. Changes to FS structure.

Block Size

- What should the block size be?
  - Larger block size: better throughput (disk reads/writes larger chunks)
  - Smaller block size: less wasted space

- Technology trends
  - Disk density is increasing faster than disk speed
  - Make disk blocks larger: 512 B -> 4096 B

- Underlying assumption (typically true for spinning disks): data that is contiguous can be accessed more quickly (reduced seeking)

Block map: extents

- Current block map: one pointer, one block. Blocks potentially scattered around disk.
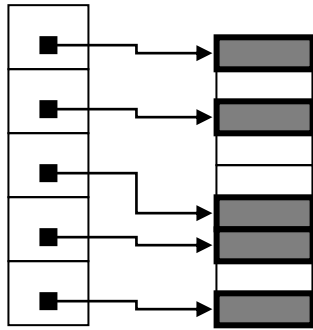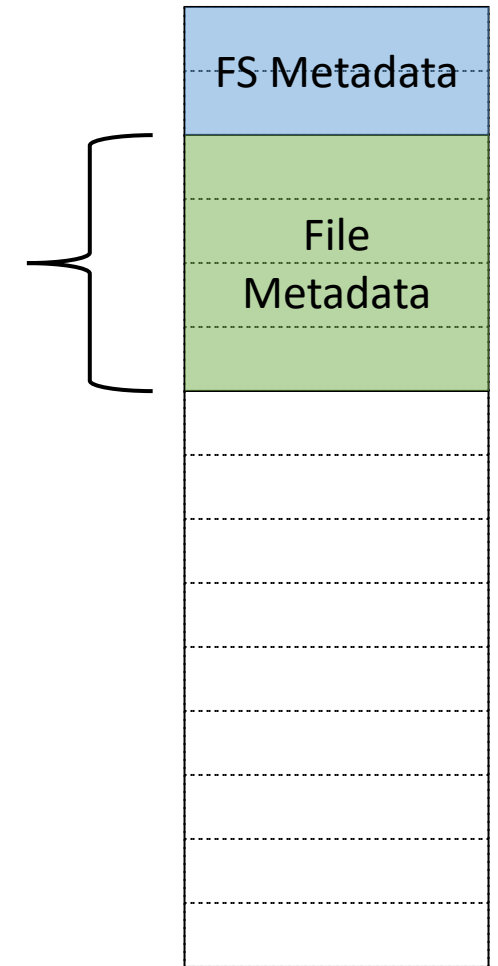
- Extents: each pointer contains more info: start and length
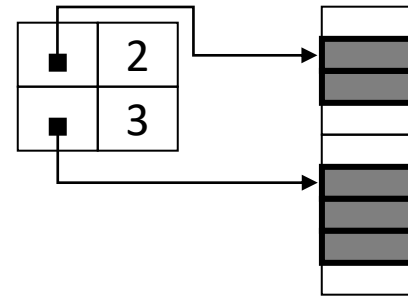
- Fewer pointers needed if disk has large free regions.

- Data contiguous on disk.

Organizing Metadata

- Observations:
  - Why allocate all the inodes in advance?
  - Why store them in a fixed-size table?

B+Tree Structure

- Generalized BST structure: if tree is balanced, find items quickly
  - Commonly used in databases to create an index for records

- FS insight: use this structure for file metadata (inodes), directory entries, and file block maps
  - Requires fewer disk accesses for name lookups

- Widely used today, popularized by Hans Reiser's ReiserFS in 2001

Other FS Structures

- Log-structured FS:
  - Treat all disk blocks as a circular array, and always append any written data or metadata to the end, like a log file.
  - Writes are always sequential, giving good throughput, reads more difficult.
  - Useful for database transaction logs, crash recovery.

- Cluster FS:
  - Treat the disks of a large compute cluster as one giant logical disk.
  - Files replicated for redundancy and performance.
  - Popularized by Google's GFS: designed to store ALL the web crawling data.

Summary

- VFS layer separates user-facing file interface from FS implementation.

- OS's scheduling of disk accesses impacts throughput, fairness.

- Disks are slow, so we cache directories, inodes, and disk data.
    - Combined data structures for fast access and LRU.

- Many other FS structures exist, with wildly different characteristics.