

# [544] File Systems

Meenakshi Syamkumar

# Learning Objectives

- compare the performance characteristics of different kinds of block devices (HDDs and SSDs)
- describe different kinds of file systems
- interpret the output of tools like "mount" and "df" to understand the structure of a mount namespace

# Outline

Block Devices (overview, HDD, SSD)

File Systems

Demos

# Block Devices

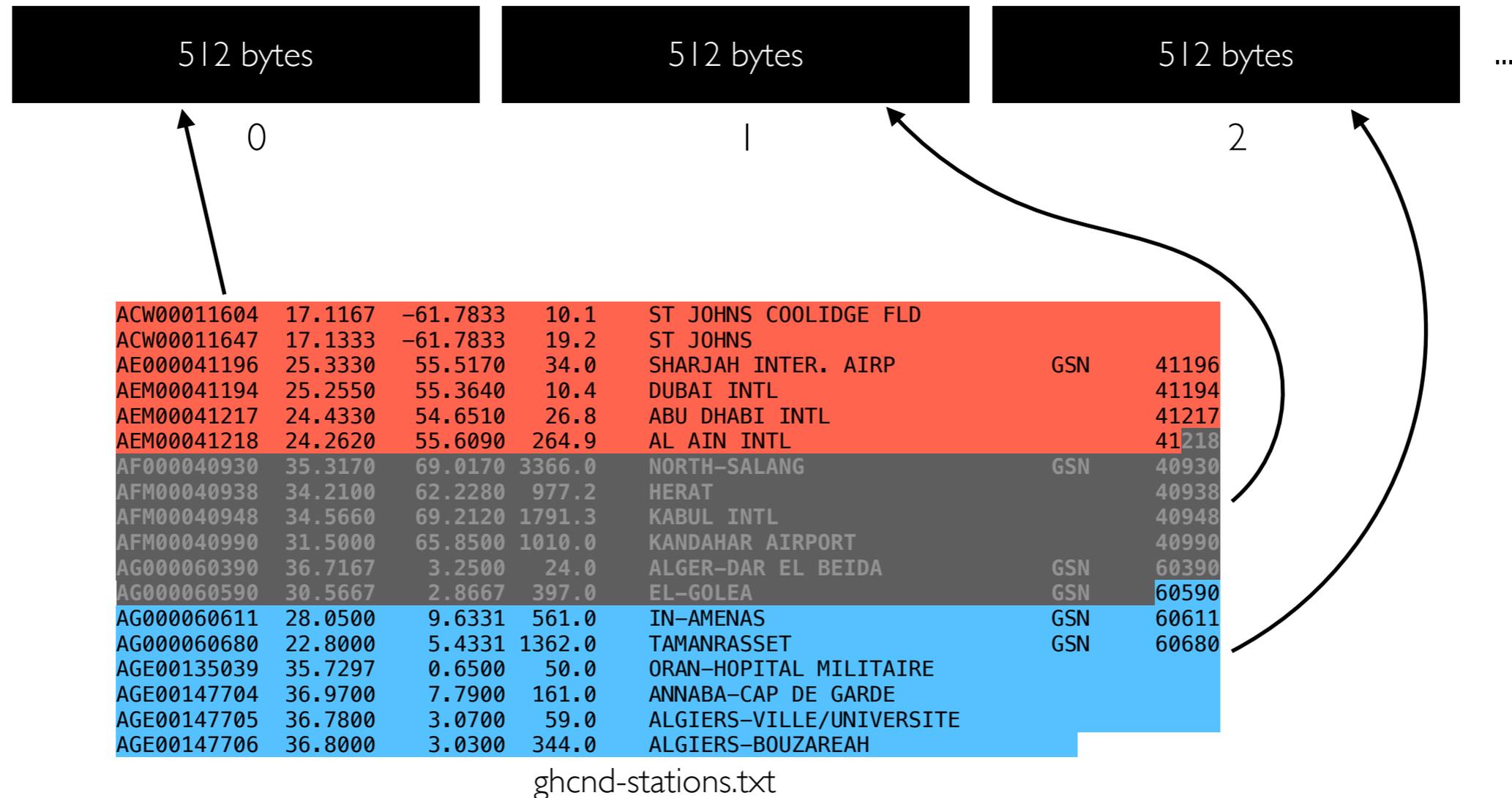
Memory is **byte addressable**



Block storage devices are accessed in units of **blocks** (512 bytes, few KBs, etc)



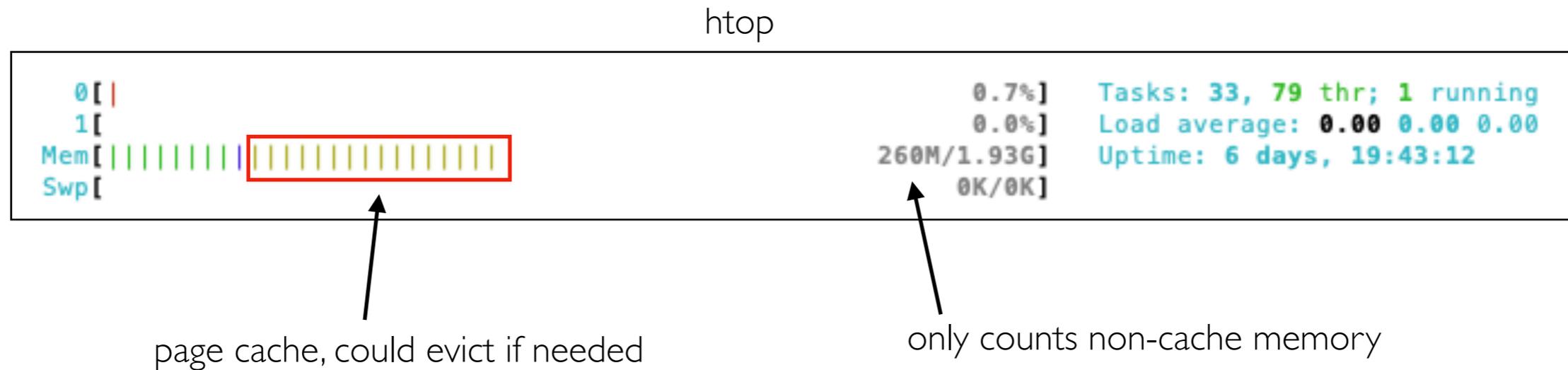
# Optimizing Disk I/O with Memory: Caching and Buffering



We might want to process one line a time, but it would be wasteful to repeatedly read the same block from the device

- the Linux **page cache** stores pages from files in RAM (usually 4KB pages, often larger than device blocks)
- Python (and other) programs might **buffer** chunks of data to avoid asking Linux too many times for small pieces of data

# Optimizing Disk I/O with Memory: Caching and Buffering



We might want to process one line a time, but it would be wasteful to repeatedly read the same block from the device

- the Linux **page cache** stores pages from files in RAM (usually 4KB pages, often larger than device blocks)
- Python (and other) programs might **buffer** chunks of data to avoid asking Linux too many times for small pieces of data

# Small Reads (<512 bytes): Performance

goal: collect all station IDs

ACW00011604	17.1167	-61.7833	10.1	ST JOHNS COOLIDGE FLD		
ACW00011647	17.1333	-61.7833	19.2	ST JOHNS		
AE000041196	25.3330	55.5170	34.0	SHARJAH INTER. AIRP	GSN	41196
AEM00041194	25.2550	55.3640	10.4	DUBAI INTL		41194
AEM00041217	24.4330	54.6510	26.8	ABU DHABI INTL		41217
AEM00041218	24.2620	55.6090	264.9	AL AIN INTL		41218
AF000040930	35.3170	69.0170	3366.0	NORTH-SALANG	GSN	40930
AFM00040938	34.2100	62.2280	977.2	HERAT		40938
AFM00040948	34.5660	69.2120	1791.3	KABUL INTL		40948
AFM00040990	31.5000	65.8500	1010.0	KANDAHAR AIRPORT		40990
AG000060390	36.7167	3.2500	24.0	ALGER-DAR EL BEIDA	GSN	60390
AG000060590	30.5667	2.8667	397.0	EL-GOLEA	GSN	60590
AG000060611	28.0500	9.6331	561.0	IN-AMENAS	GSN	60611
AG000060680	22.8000	5.4331	1362.0	TAMANRASSET	GSN	60680
AGE00135039	35.7297	0.6500	50.0	ORAN-HOPITAL MILITAIRE		
AGE00147704	36.9700	7.7900	161.0	ANNABA-CAP DE GARDE		
AGE00147705	36.7800	3.0700	59.0	ALGIERS-VILLE/UNIVERSITE		
AGE00147706	36.8000	3.0300	344.0	ALGIERS-BOUZAREAH		

ghcnd-stations.txt

```
start = time.time()
with open("ghcnd-stations.txt") as f:
    for line in f:
        stations.append(line[:11])
print(time.time() - start)
```

simple version that reads everything: **66 ms**

**format issue:** no good way to read one column without everything else

(similar to issues with bad cache line usage)

```
stations = []
line_len = 86

start = time.time()
with open("ghcnd-stations.txt",
         "rb", buffering=0) as f:
    offset = 0
    while True:
        f.seek(offset)
        station = str(f.read(11), "utf-8")
        offset += line_len

        if station:
            stations.append(station)
        else:
            break
print(time.time() - start)
```

"optimized" version that only reads stations: **171 ms**

# Hard Disk Drives (HDDs)

Steps to read/write

1. move head to correct track
2. wait for spinning disk to rotate until data is under head
3. transfer the data

these steps dominate unless transferring lots of data (few MBs)



Layout

- assign block numbers to platter locations so **sequential** (like 5,6,7,8, ...) reads/writes will be fast
- programmers should assume **random** accesses (like 2, 9, 5, 1, ...) will be slow

# Capacity vs. I/O and Short Stroking

Storage resources

1. capacity
2. I/O (input/output often more limited when using HDDs)



Short Stroking

- head moves over platter faster near outside track
- smaller block addrs correspond to outside tracks
- strategy: only use outside tracks
- pros: faster I/O
- cons: less space

# Solid State Drives (SSDs) - Flash

Reading and writing

- no moving parts
- inherently parallel



SSD internals:

- "block" and "page" have different meanings in this context
- "page" => unit that we can read or write (couple KBs)
- pages cannot be individually re-written
- "block" => unit that is erased together (maybe 100s of KBs)

block



# Solid State Drives (SSDs) - Flash

want to write X. Options:

- erase whole block and re-write A, C, and D too
- write X somewhere else

X



# Solid State Drives (SSDs) - Flash

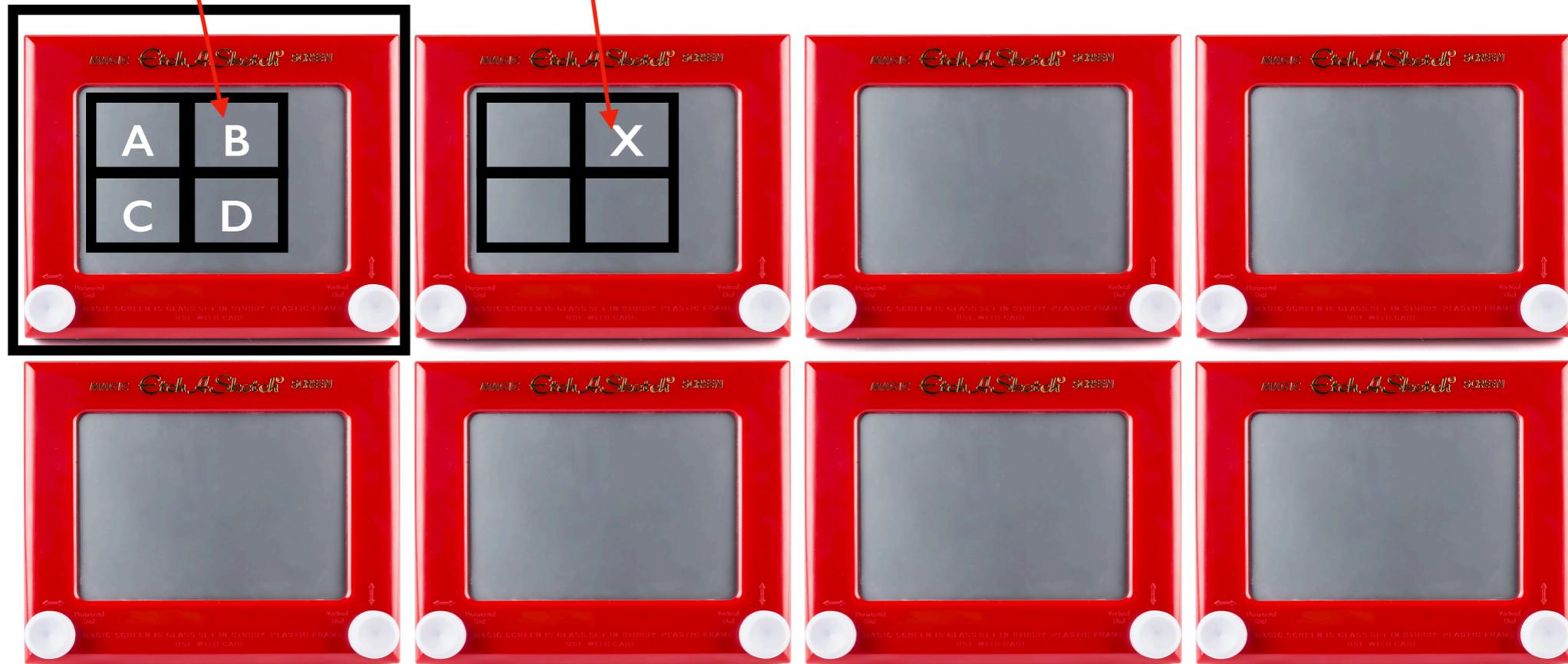
want to write X. Options:

- erase whole block and re-write A, C, and D too
- write X somewhere else

disadvantages

- need extra bookkeeping (in SSD) to know where data is
- need to eventually move things around to reclaim the space wasted by B
- **strategy:** sequentially write whole blocks (when possible)

garbage



# HDDs vs. SSDs

## Metrics

- **capacity**: how many bytes can we store?
- **latency**: how long does it take to start transferring data
- **IOPS** (I/O operations, of some max size, per second): how many small/random transfers can we do per second
- **throughput**: how many bytes can we transfer per second

### Metric:

capacity

latency

random IOPS

throughput (sequential)

throughput (random writes)

throughput (random reads)

### Relative to HDDS, **SSDs** are:

*worse*

*much better (no moving parts)*

*even better -- low latency AND in parallel*

*little better*

*better (but block erase is a concern)*

*much better*

# Partitions and RAID

Block devices can be divided into **partitions**:



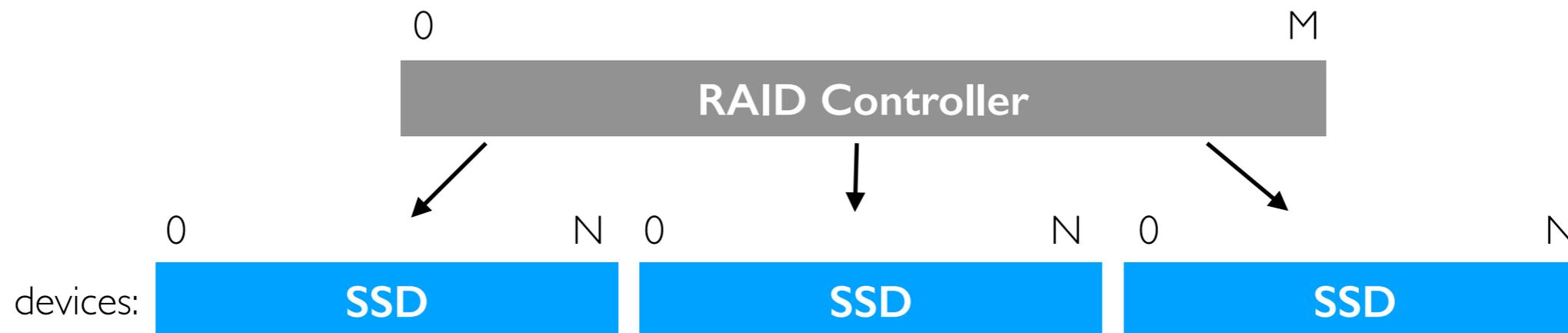
```
root@instance-1:/home/trh# ls /dev/sd*
```

```
/dev/sda    /dev/sda1    /dev/sda14    /dev/sda15  
/dev/sdb    /dev/sdb1
```

2 devices

4 partitions

**RAID** controllers (Redundant Array of Inexpensive Disks) can make multiple devices appear as one:



Many configs use **redundancy** (e.g., same data on >1 disk) to avoid data loss when one device dies.

# Outline

Block Devices (overview, HDD, SSD)

File Systems

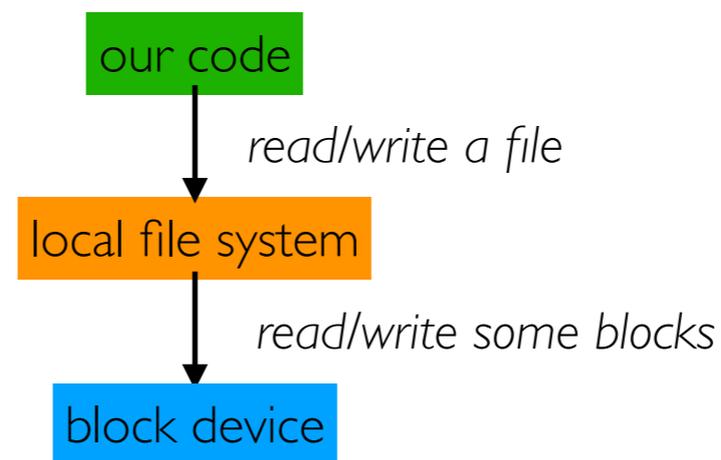
Demos

# File Systems

**Difficult:** writing code to store data in **blocks**

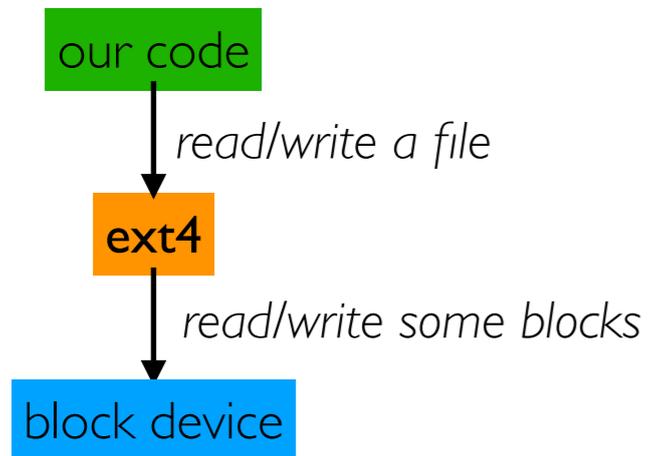
**Easier:** writing code to store data in **files**

File systems *abstract* storage for us. We write to data **blocks** without thinking about it by writing data to **files** in a **local file system**.

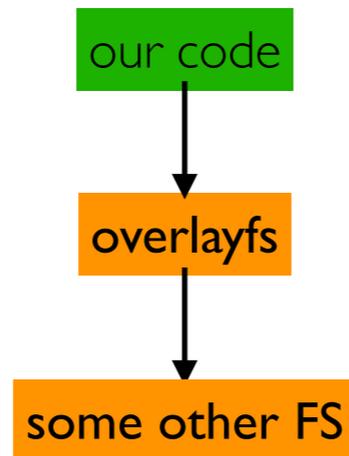


# Types of File System (FS)

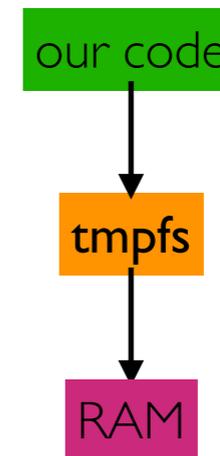
local FS



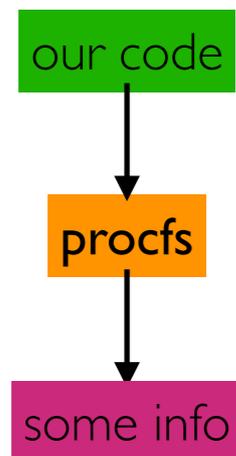
layered FS (for Docker)



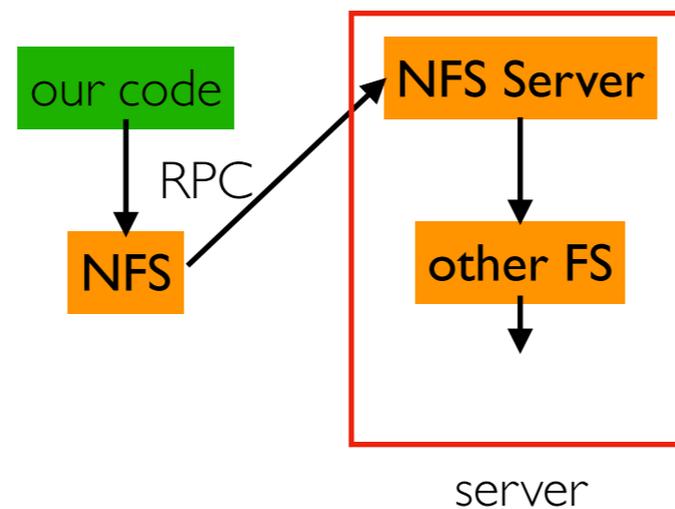
in-memory FS (Temp Files)



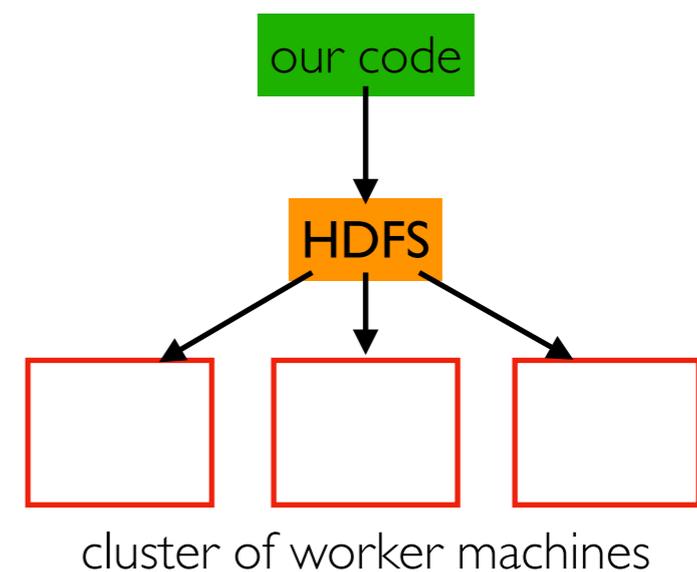
pseudo FS (Stats)



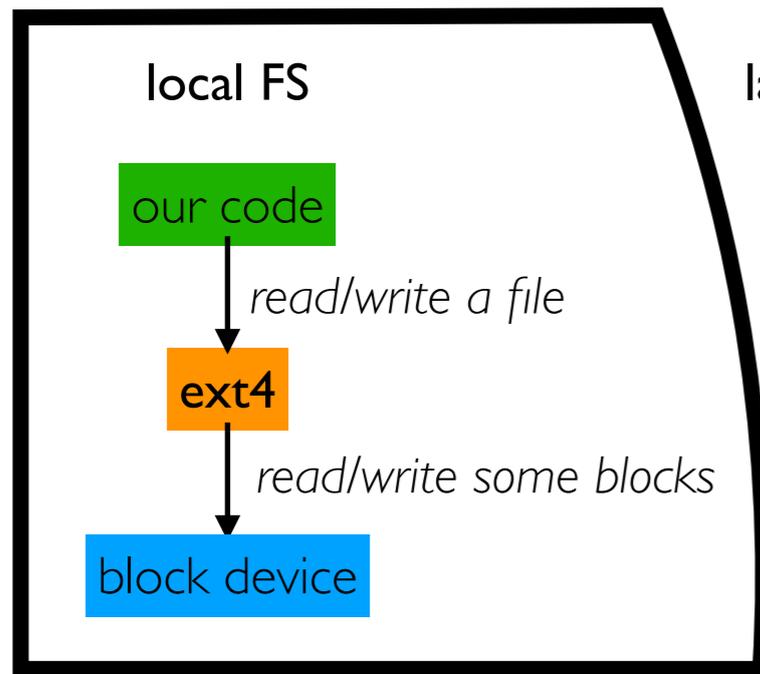
network FS



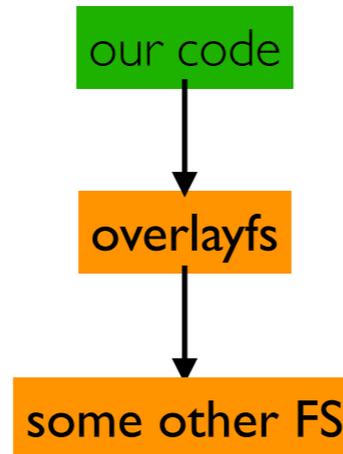
distributed FS



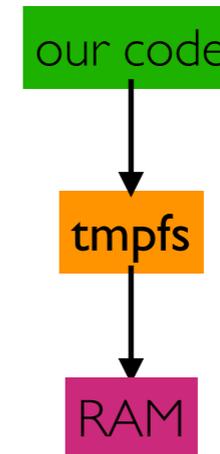
# Types of File System (FS)



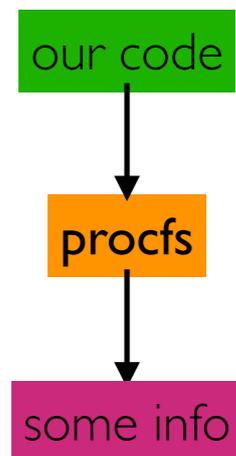
layered FS (for Docker)



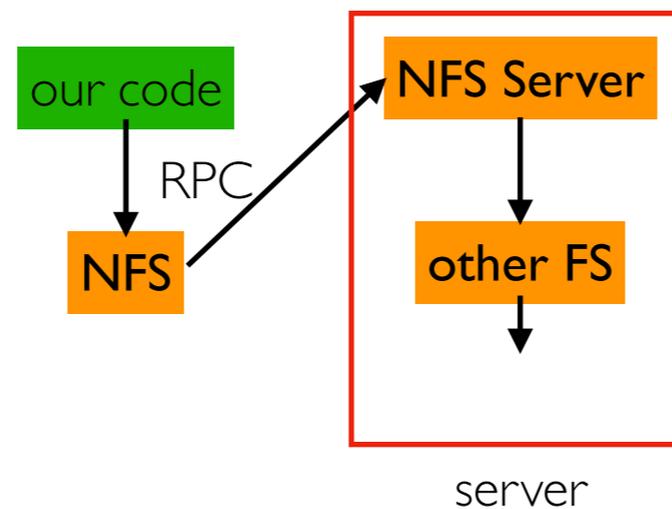
in-memory FS (Temp Files)



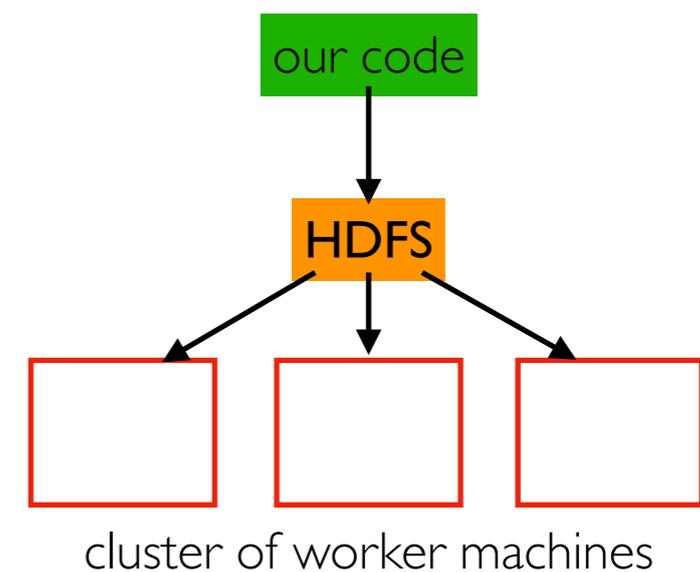
pseudo FS (Stats)



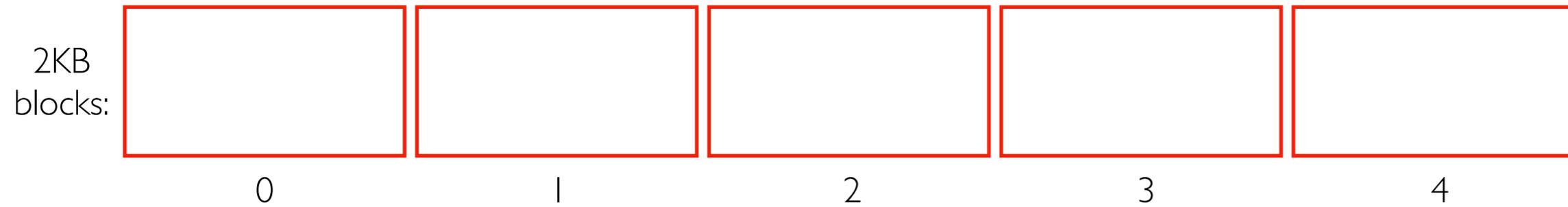
network FS



distributed FS

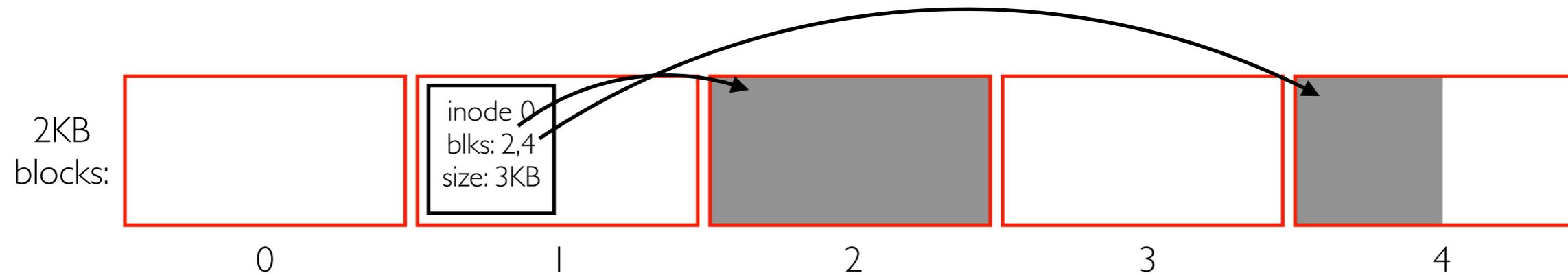


# Local File Systems



How does a local FS use blocks?

# Local File Systems

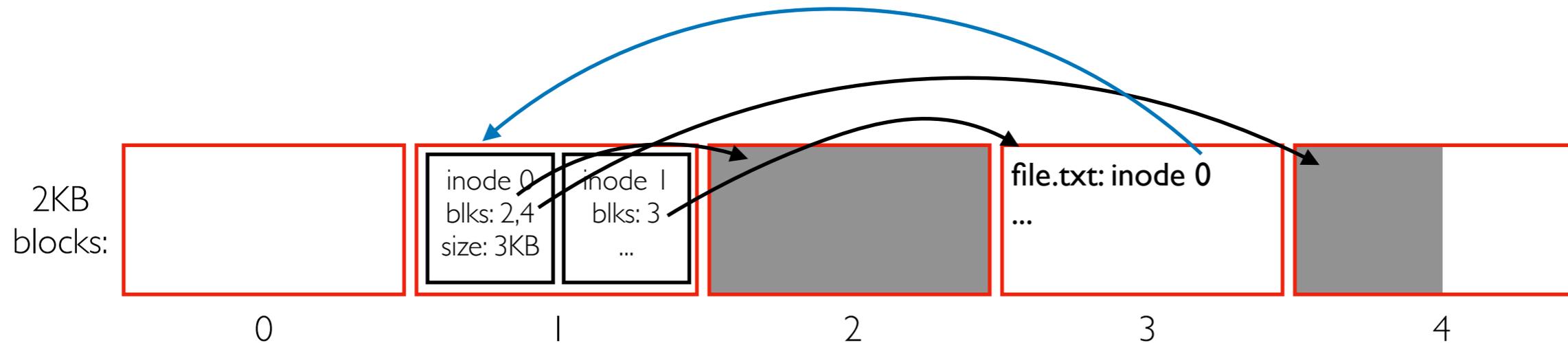


How does a local FS use blocks? Many possibilities. One example...

## Files

- some metadata, like size, block locations
- each is represented by an "inode" structure (above file is fragmented)

# Local File Systems



How does a local FS use blocks? Many possibilities. One example...

## Files

- some metadata, like size, block locations
- each is represented by an "inode" structure (above file is fragmented)

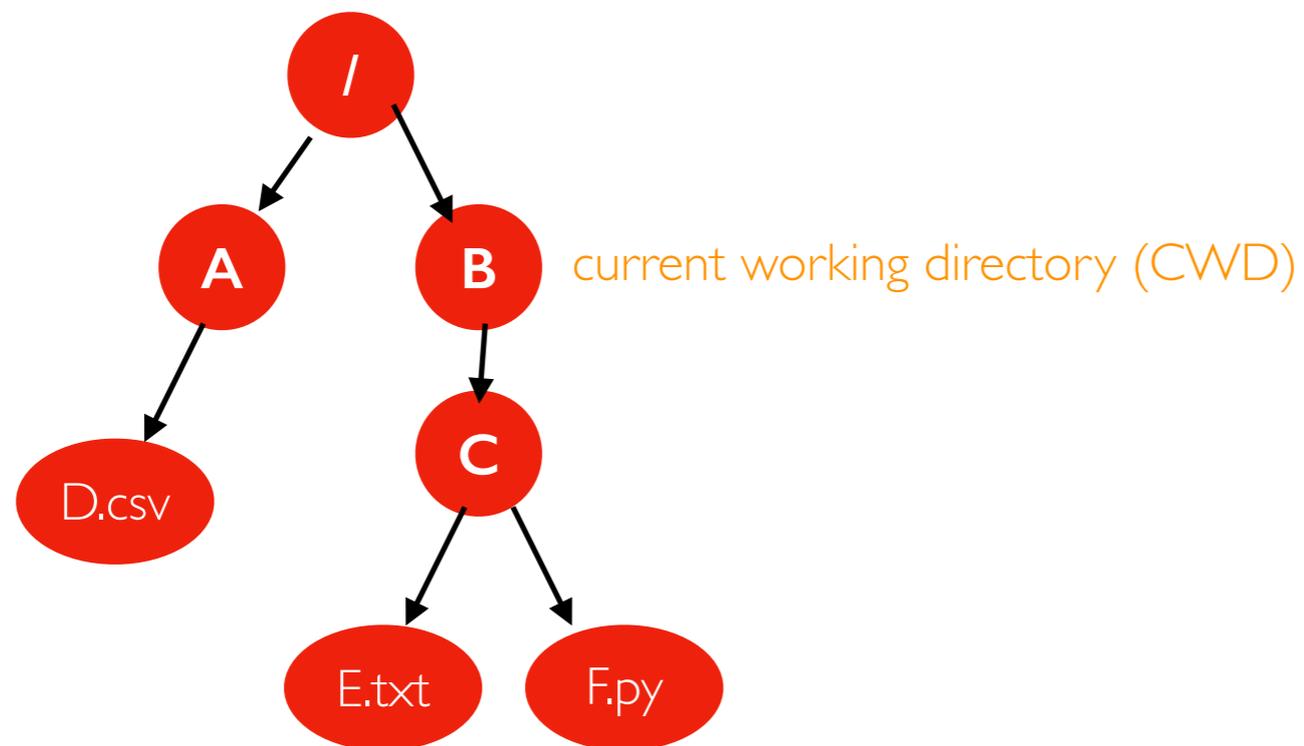
## Directories

- special files containing name => inode mappings
- file extensions (like .txt) don't mean anything to the file system (just for documentation)
- the same inode could be in multiple directories
- each file system has a "root" directory from which you can reach everything else recursively
- **formatting** a disk creates initial structures (like the root directory)

# File System Trees

Nesting of directories and files logically create "trees"

- technically DAGs (directed acyclic graphs) because the same inode number can have multiple names in different directories
- leaves: files and empty directories



relative path to E.txt: C/E.txt

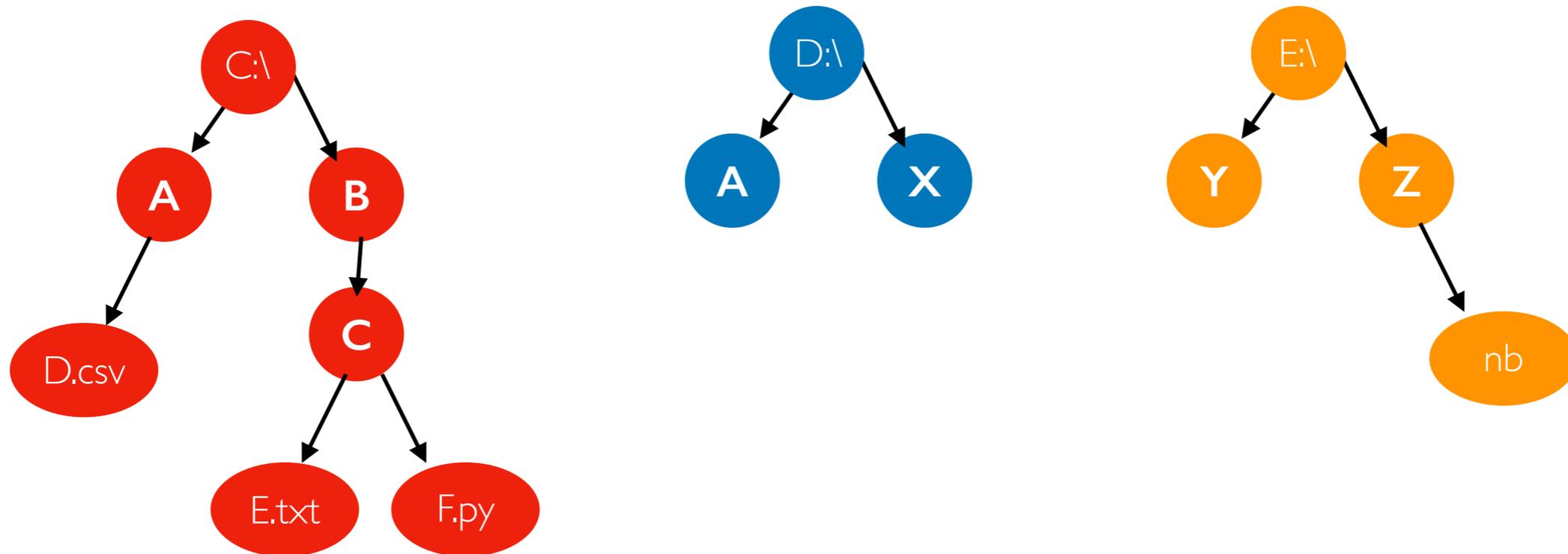
absolute path to E.txt: /B/C/E.txt

relative path to D.csv: ../A/D.csv

absolute path to F.py: TopHat

# Multiple File Systems: Windows Approach

have multiple trees (each is a "drive")



# Multiple File Systems: Unix Approach

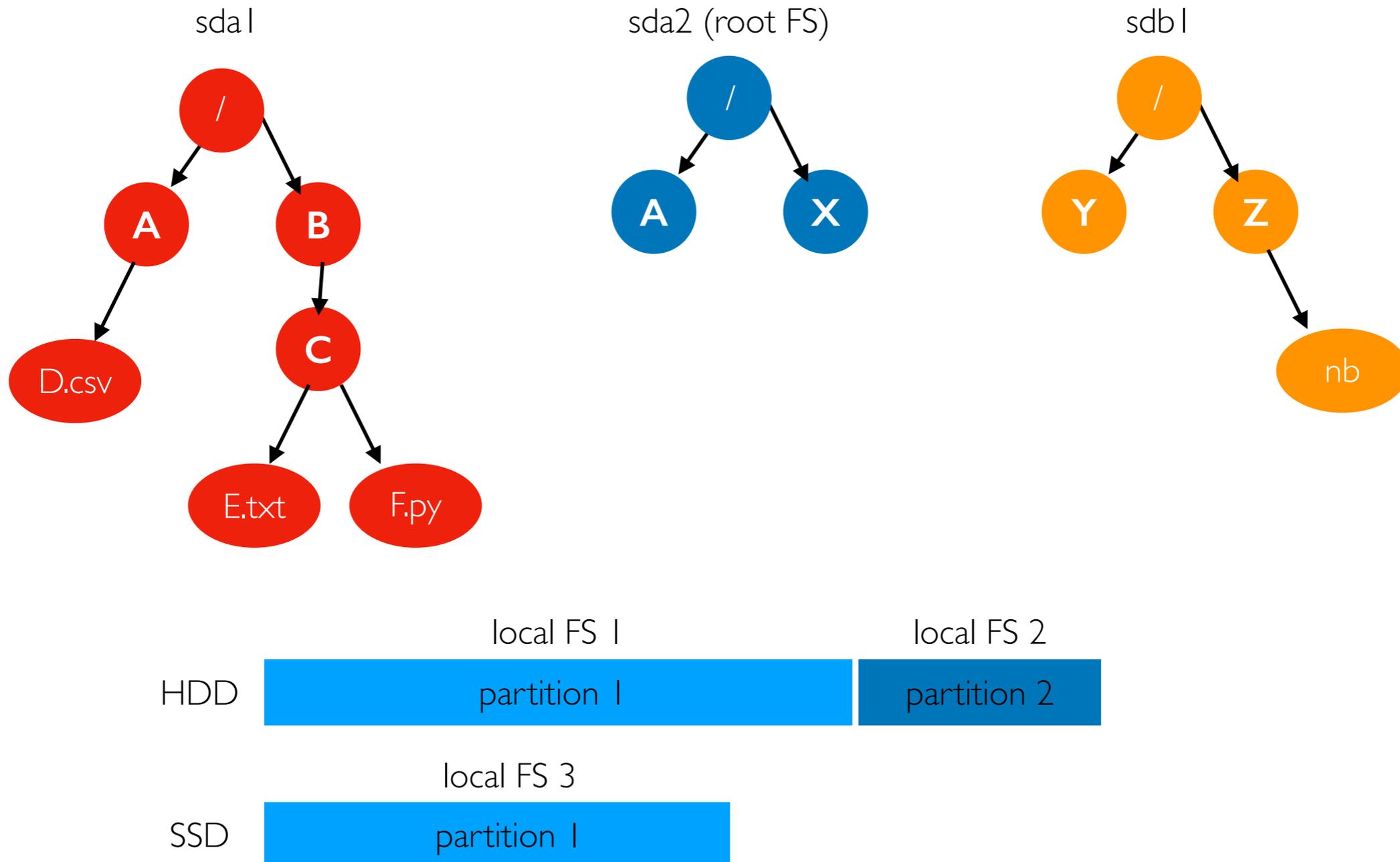
mount file systems over directories of other file systems to make one big tree



<https://www.brit.co/fruit-salad-tree/>

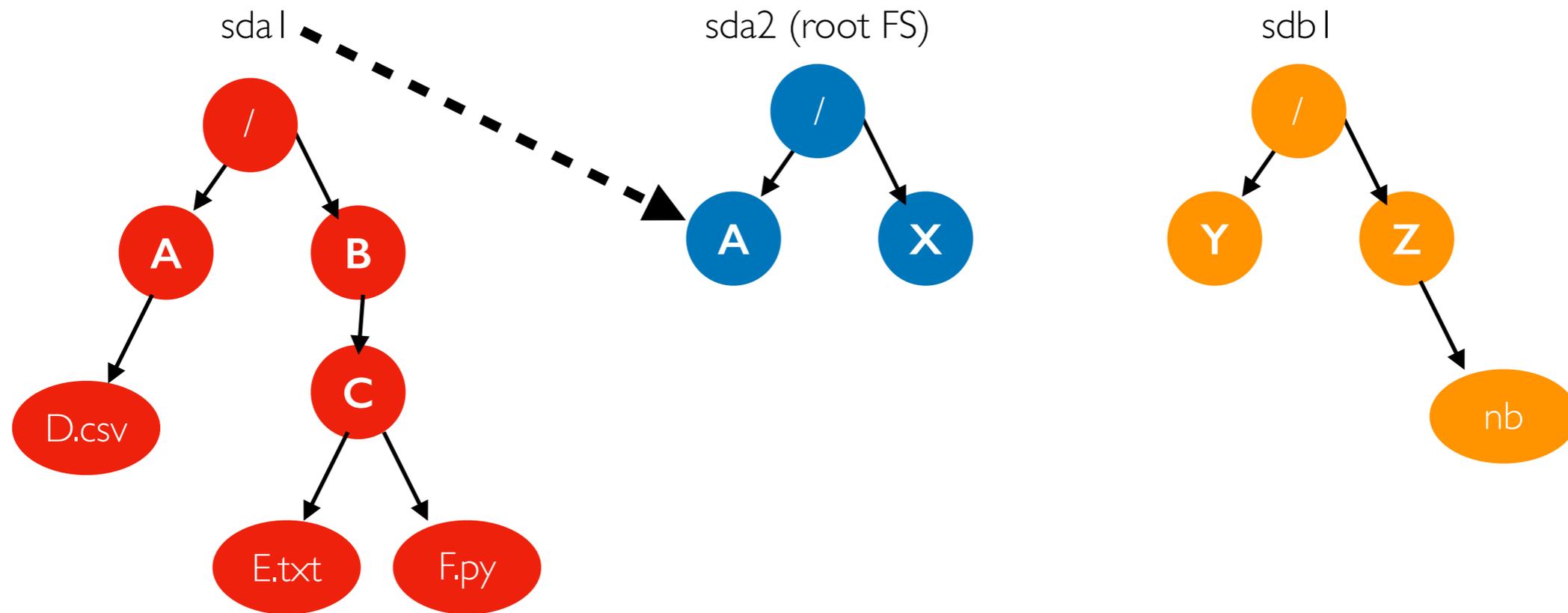
# Multiple File Systems: Unix Approach

mount file systems over directories of other file systems to make one big tree



# Multiple File Systems: Unix Approach

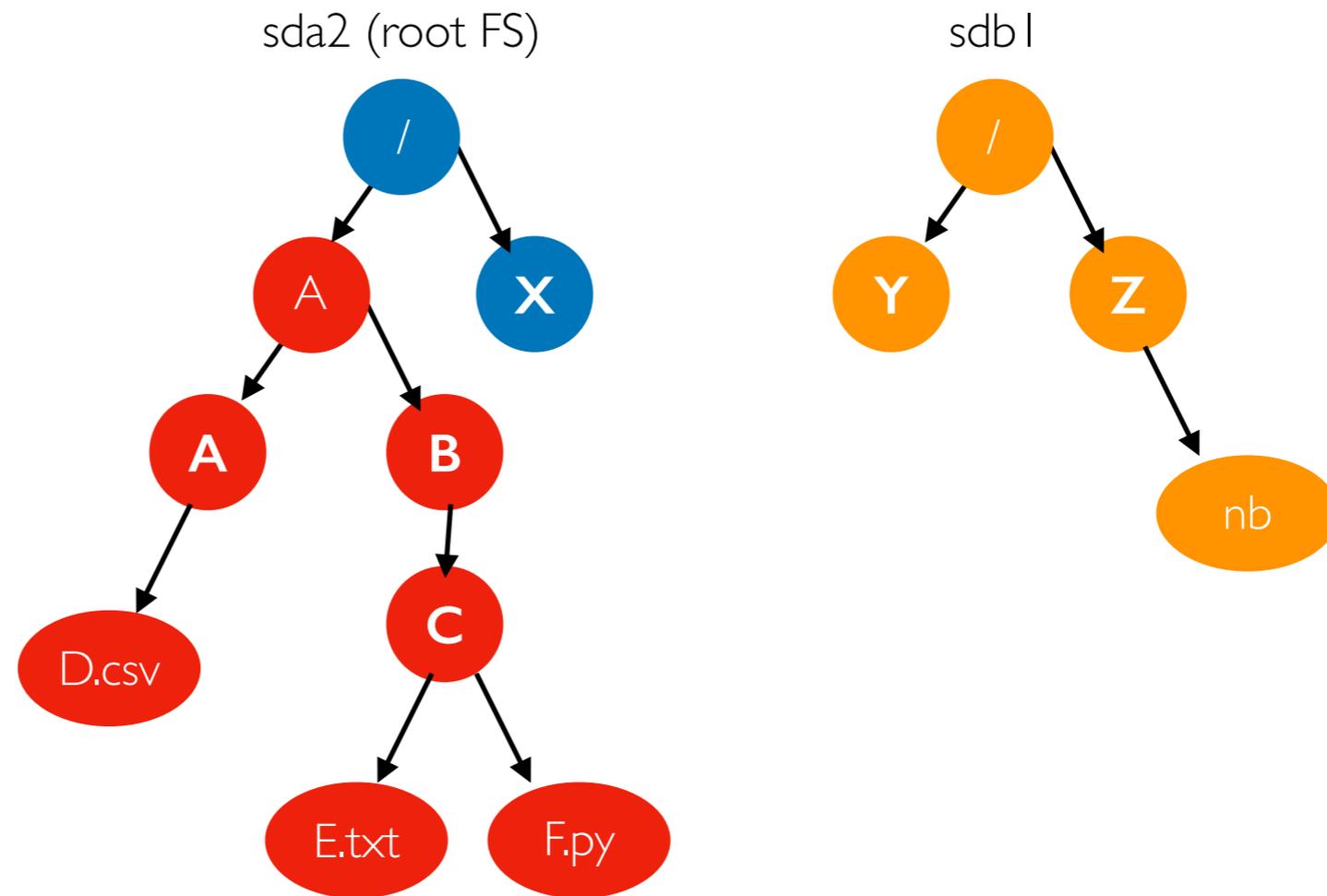
mount file systems over directories of other file systems to make one big tree



```
mount /dev/sda1 /A
```

# Multiple File Systems: Unix Approach

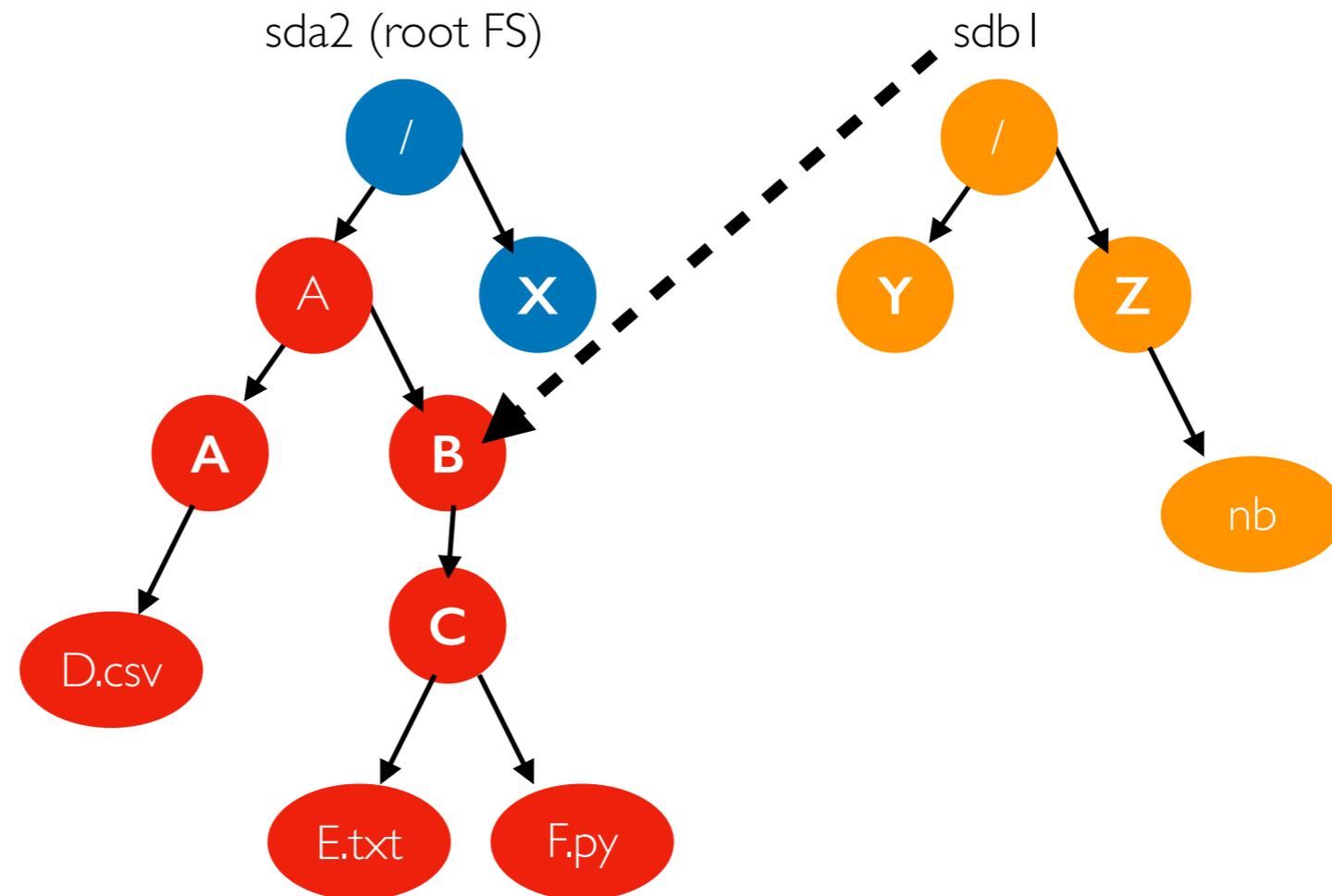
mount file systems over directories of other file systems to make one big tree



```
mount /dev/sda1 /A
```

# Multiple File Systems: Unix Approach

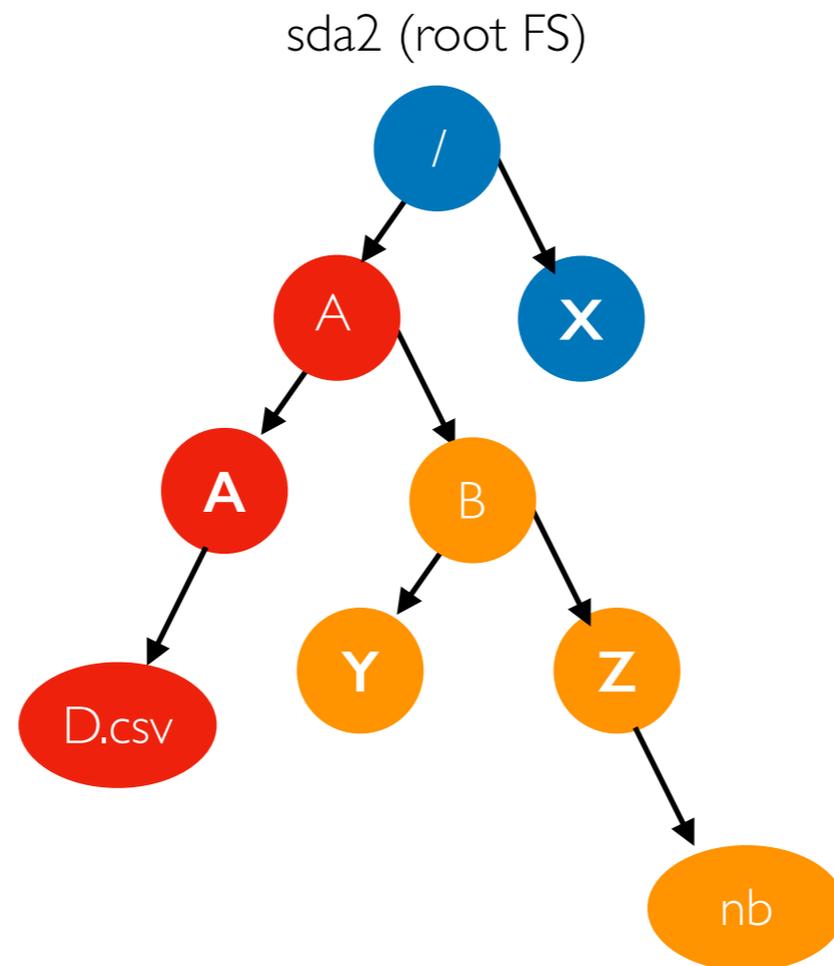
mount file systems over directories of other file systems to make one big tree



```
mount /dev/sda1 /A  
mount /dev/sdb1 /A/B
```

# Multiple File Systems: Unix Approach

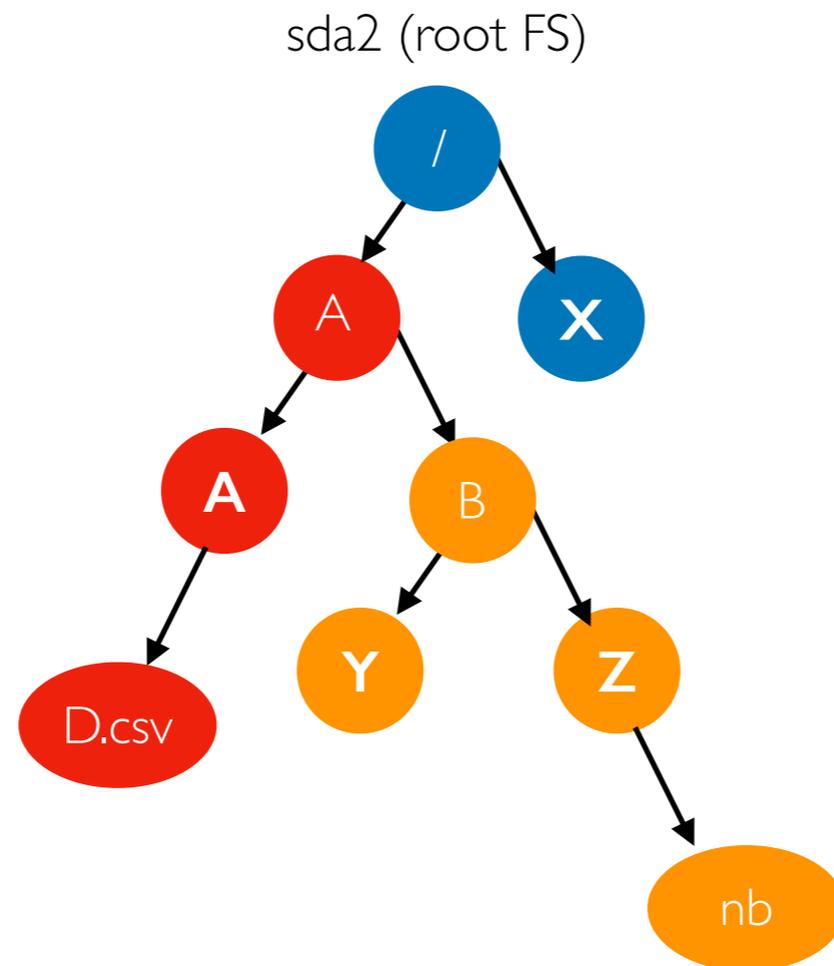
mount file systems over directories of other file systems to make one big tree



```
mount /dev/sda1 /A  
mount /dev/sdb1 /A/B
```

# Multiple File Systems: Unix Approach

mount file systems over directories of other file systems to make one big tree

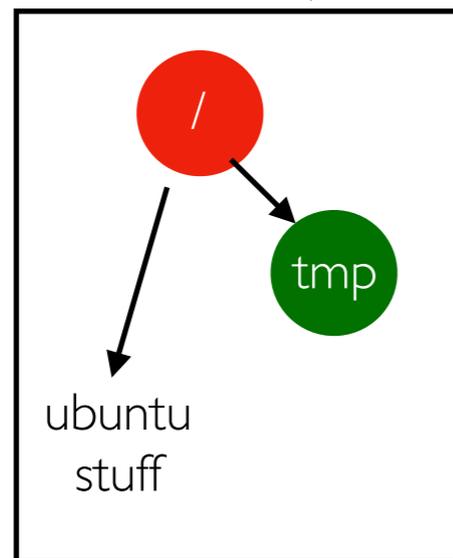
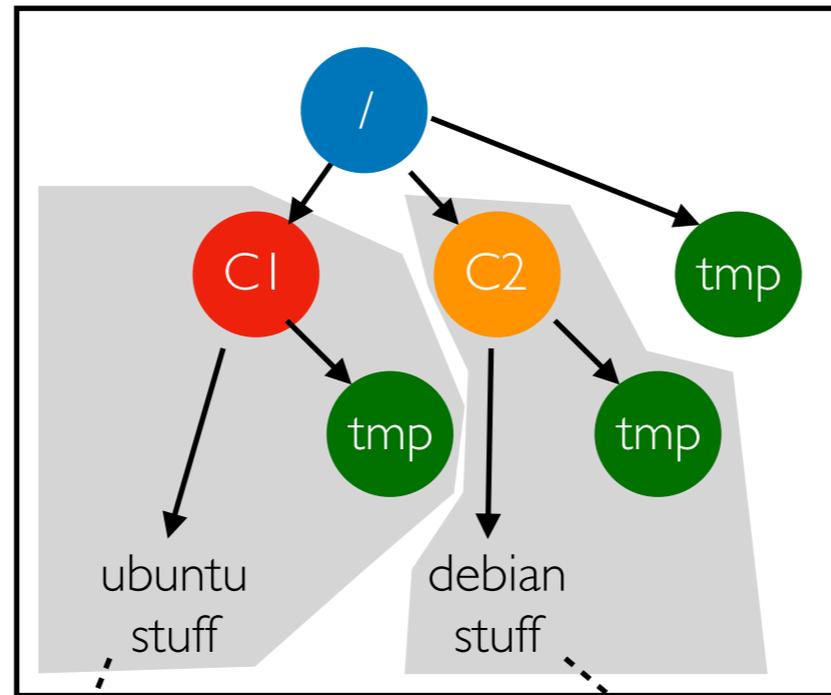


Note: each container has its own root file system and mount namespace

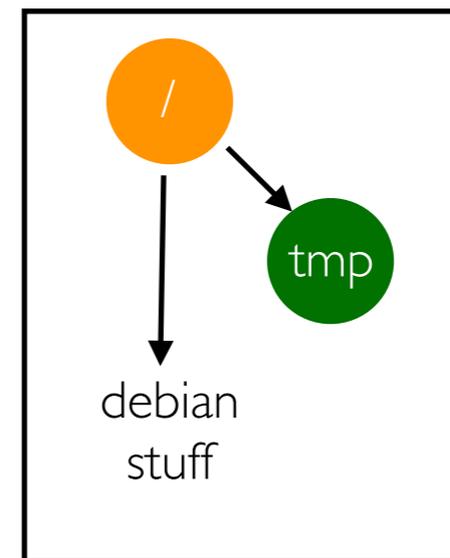
```
mount /dev/sda1 /A  
mount /dev/sdb1 /A/B
```

# Container File Systems (Simplified)

mount namespace (VM)



mount namespace (container 1)



mount namespace (container 2)

# Outline

Block Devices (overview, HDD, SSD)

File Systems

Demos