Operating Systems

5. Persistence

Pablo Prieto Torralbo

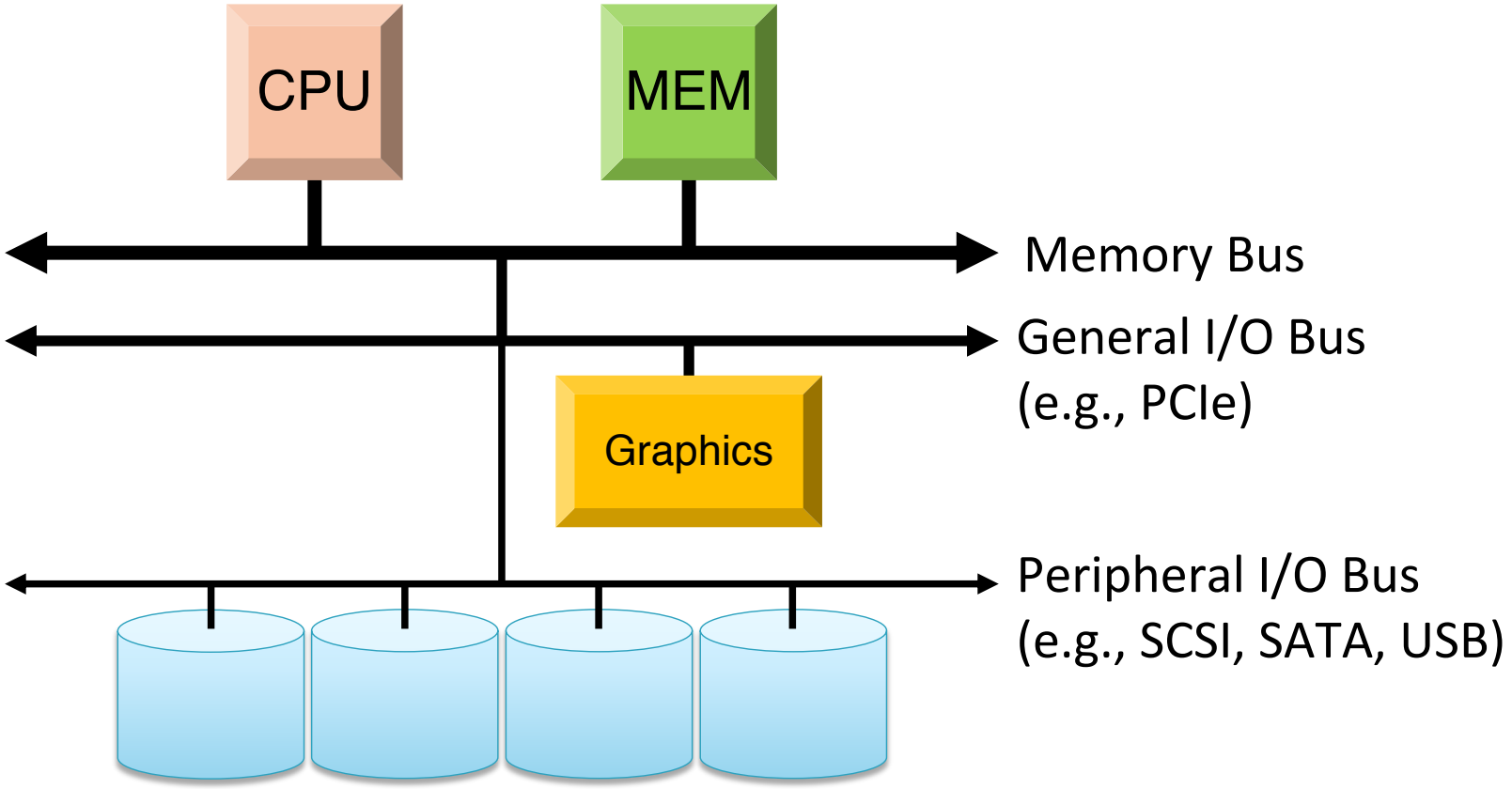
DEPARTMENT OF COMPUTER ENGINEERING AND ELECTRONICS

5.1 Persistence: I/O Devices

Motivation

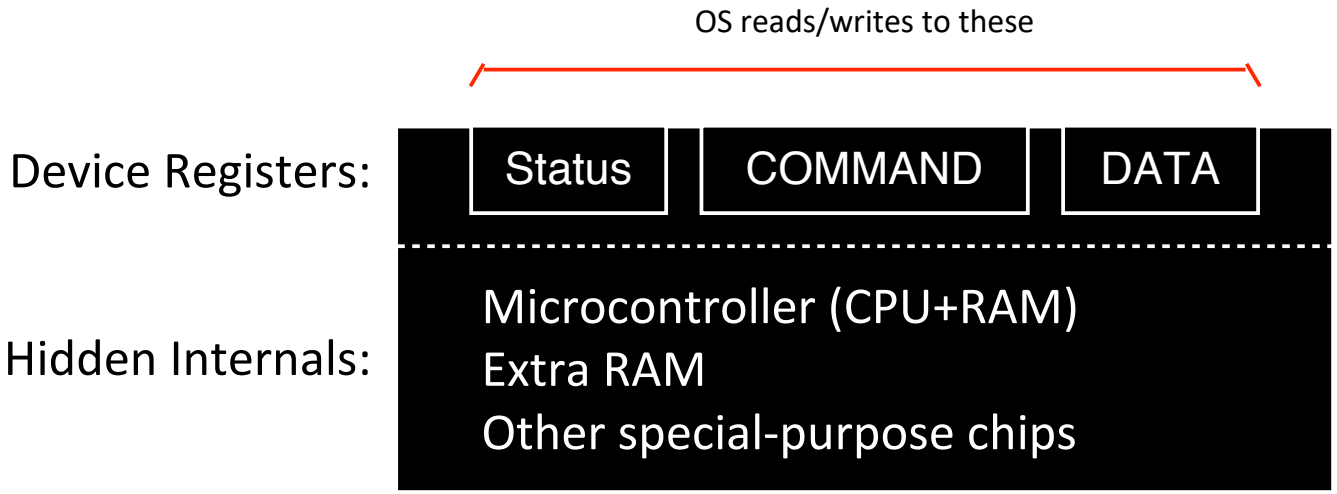
- How good is a computer without any I/O devices?
  - keyboard, display, disks
- We want:
  - H/W that will let us plug in different devices
  - OS that can interact with different combinations

Hardware support for I/O



Hierarchical buses

Canonical Device




```
while (STATUS == BUSY)
  ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
  ; // spin
```

Let's write a Protocol

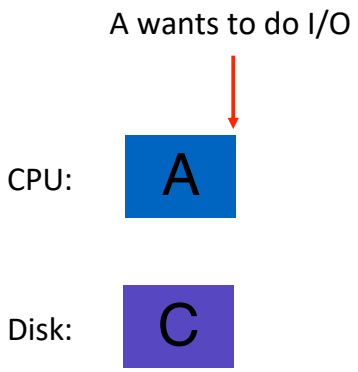
Example: Write Protocol

CPU:

Disk:

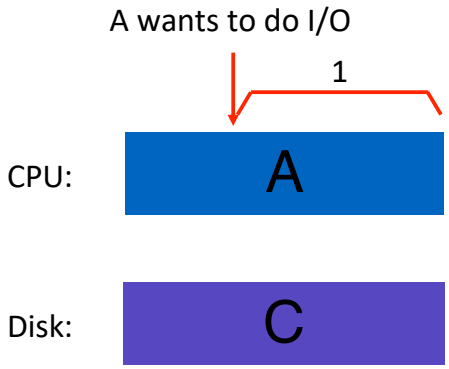
```
while (STATUS == BUSY)                  //1
  ; // spin
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
  ; // spin
```

Example: Write Protocol



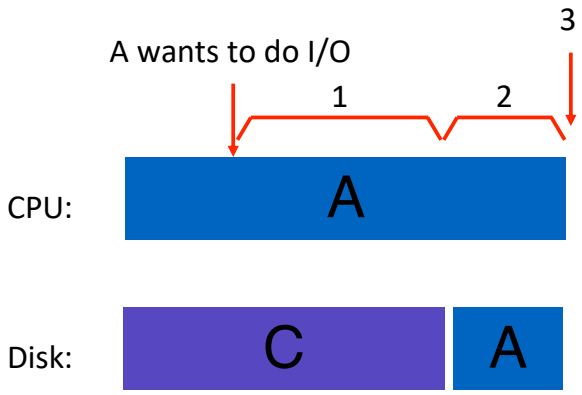

```
while (STATUS == BUSY)                  //1
  ; // spin
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
  ; // spin
```

Example: Write Protocol



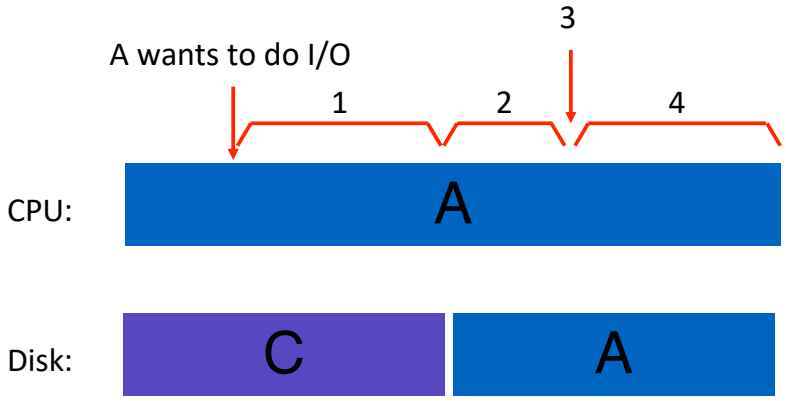

```
while (STATUS == BUSY)                  //1
  ; // spin
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
  ; // spin
```

Example: Write Protocol



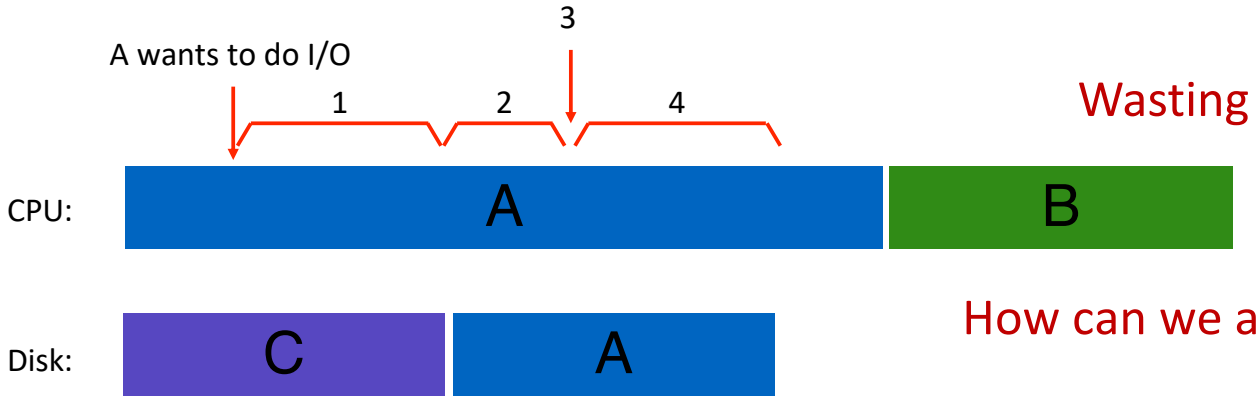

```
while (STATUS == BUSY)                  //1
  ; // spin
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
  ; // spin
```

Example: Write Protocol




```
while (STATUS == BUSY)                  //1
  ; // spin
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
  ; // spin
```

Example: Write Protocol

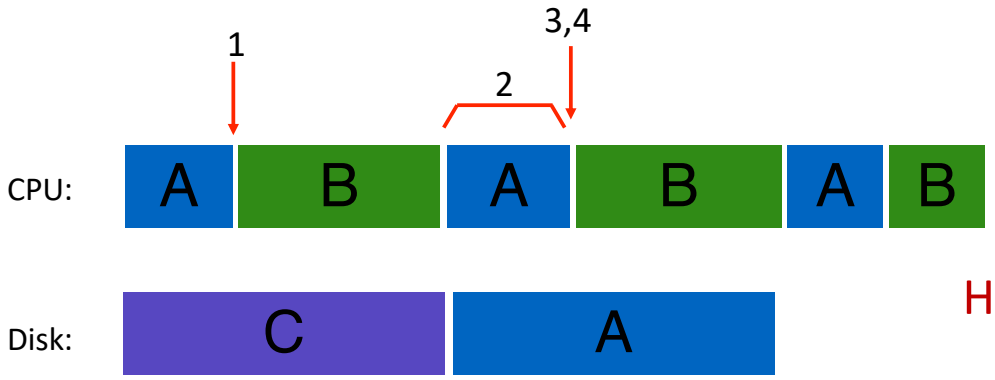


Wasting CPU

How can we avoid spinning?

```
while (STATUS == BUSY)                  //1
  ; // spin
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
  ; // spin
```

Example: Write Protocol



How can we avoid spinning?

INTERRUPTS!

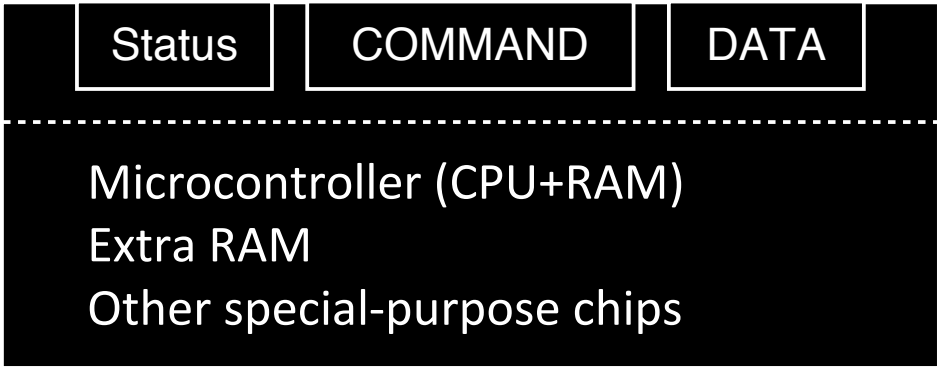
```
while (STATUS == BUSY)                  //1
    wait for interrupt;
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
    wait for interrupt;
```

Interrupts vs. Polling

Are interrupts ever worse than polling?

- Fast device: Better to spin than take interrupt overhead
  - Device time unknown? Hybrid approach (spin then use interrupts)
- Flood of interrupts arrive
  - Can lead to livelock (always handling interrupts)
  - Better to ignore interrupts while make some progress handling them

Protocol Variants



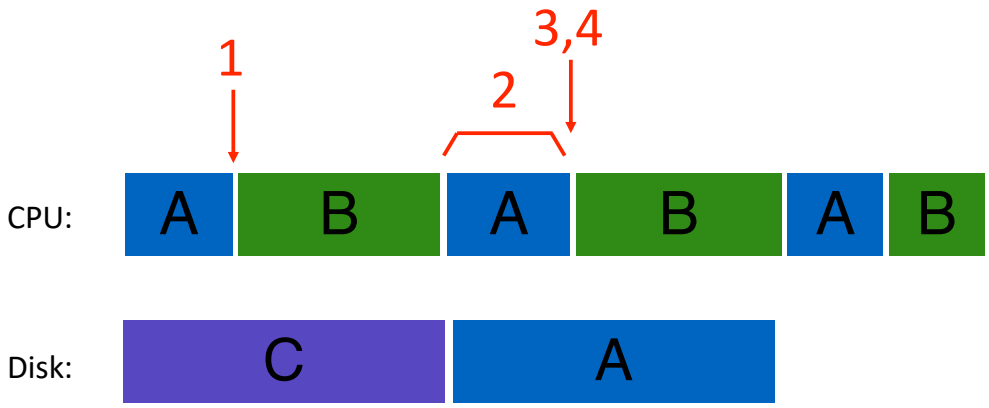
- Status checks: polling vs. interrupts
- Data: PIO vs. DMA ← How can we optimize data transfer?
- Control: special instructions vs. memory-mapped I/O

Programmed I/O vs. Direct Memory Access

- PIO (Programmed I/O):
  - CPU directly tells device what the data is.
  - CPU usage.

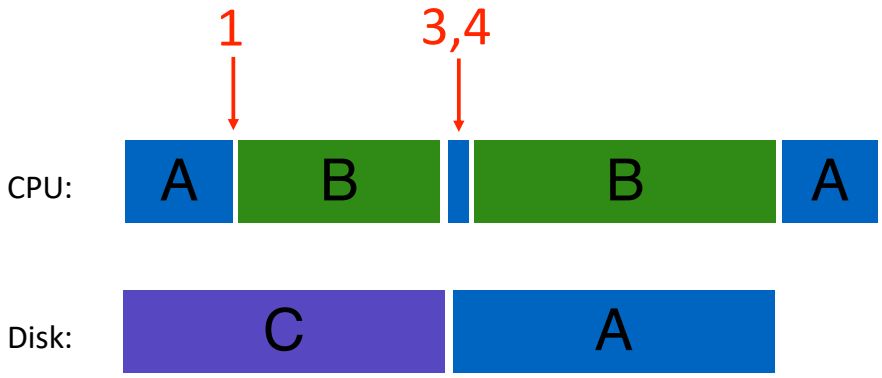
- DMA (Direct Memory Access):
  - CPU leaves data in memory.
  - Specific device transfers data directly between memory and the I/O device.

Example: Write Protocol



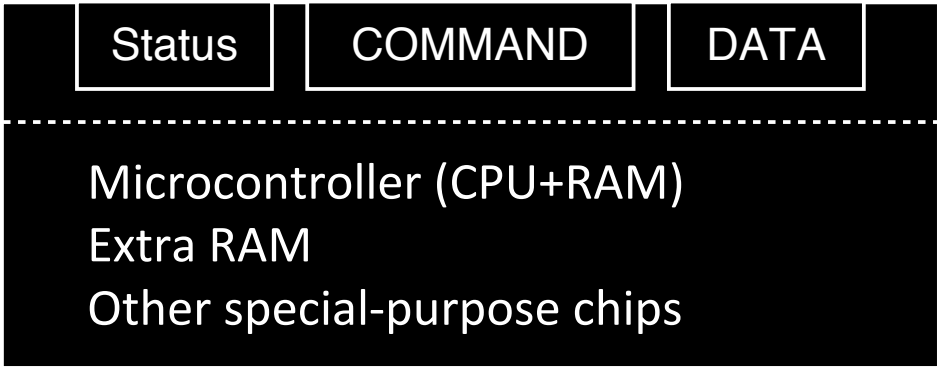

```
while (STATUS == BUSY)                  //1
    wait for interrupt;
Write data to DATA register             //2
Write command to COMMAND register       //3
while (STATUS == BUSY)                  //4
    wait for interrupt;
```

Example: Write Protocol




```
while (STATUS == BUSY)                    //1
    wait for interrupt;
Write data to DATA register               //2
Write command to COMMAND register         //3
while (STATUS == BUSY)                    //4
    wait for interrupt;
```

Protocol Variants



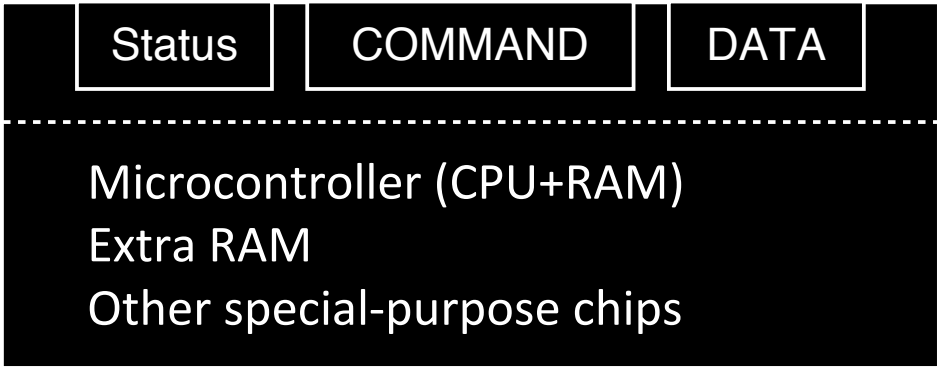
- Status checks: polling vs. interrupts
- Data: PIO vs. DMA
- Control: special instructions vs. memory-mapped I/O

how does OS read and write registers?

Special Instructions vs. Mem-Mapped I/O

- Special instructions
  - Explicit privileged instructions.
  - x86: in/out instructions to communicate with devices.
  - Each device has a specific port to name it.
- Memory-Mapped I/O
  - H/W maps device registers into address space.
  - loads/stores sent to these addresses → hardware translates them to particular device registers.
  - No new instructions needed.
- Doesn’t matter much (both are used)

Protocol Variants

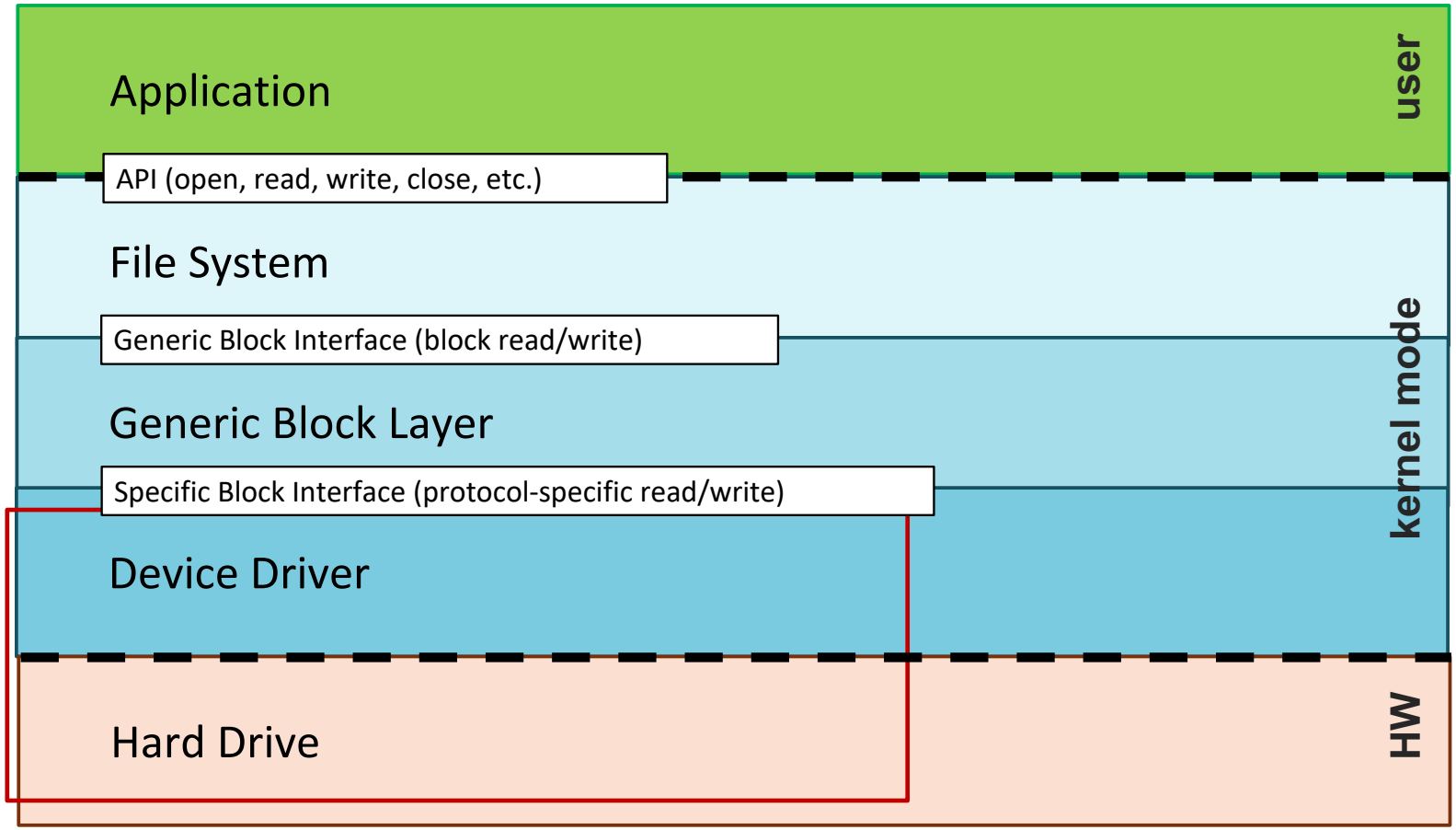


- Status checks: polling vs. interrupts
- Data: PIO vs. DMA
- Control: special instructions vs. memory-mapped I/O

Variety is a Challenge

- Problem:
  - many, many devices
  - each has its own protocol
- How can we avoid writing a slightly different OS for each H/W combination?
- Abstraction: Write device driver for each device.
- A large percentage of OS source code are drivers (~70% on Linux).

Storage Stack



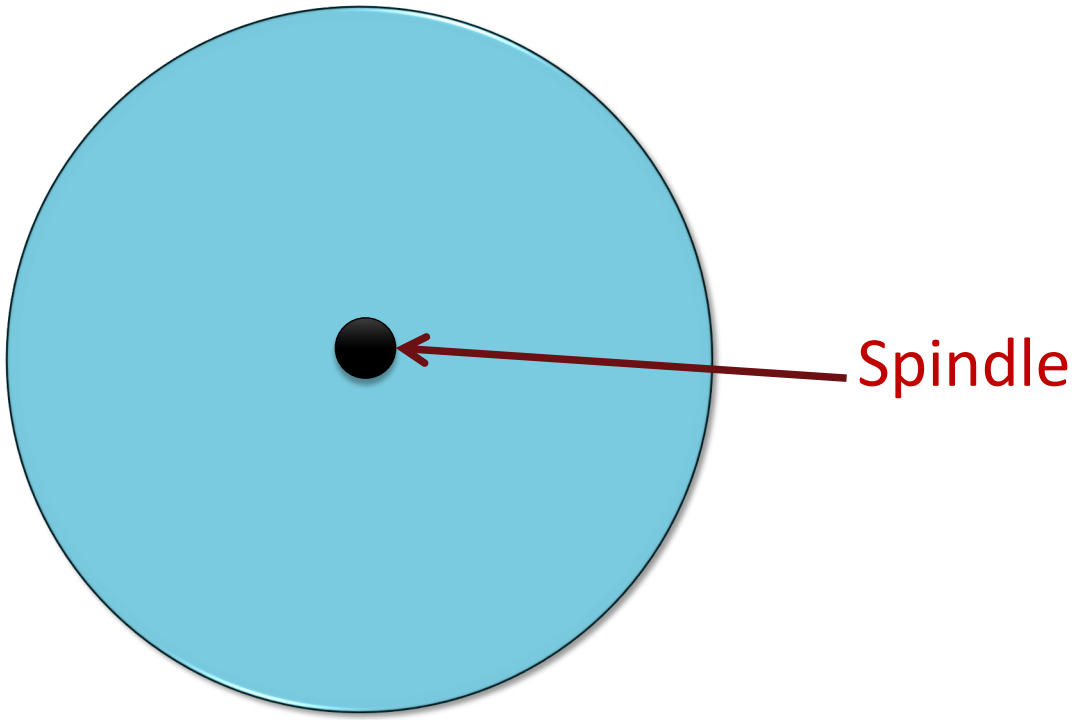
5.2 Persistence: Hard Disk Drives

HDD Basic Interface

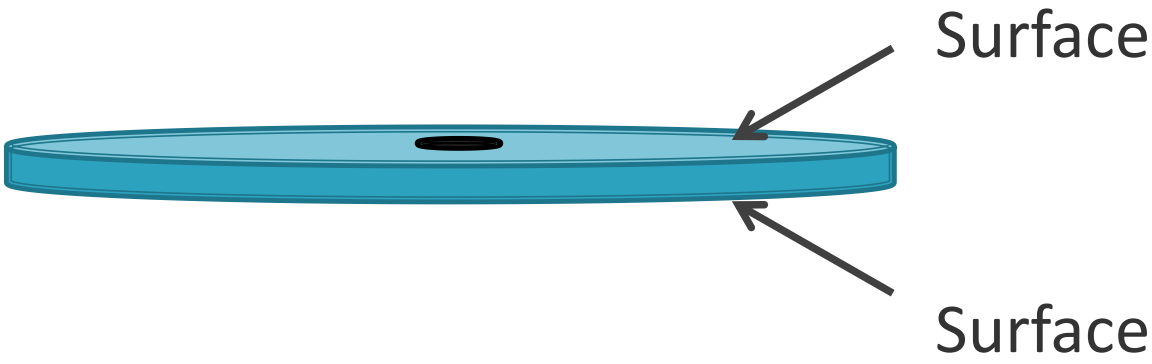
- Disk has a sector-addressable address space
  - Appears as an array of sectors
- Sectors are typically 512 bytes or 4096 bytes
- Main operations: reads + writes to sectors
- Mechanical (slow) nature makes management “interesting”

Disk Internals

- Platter (covered with a magnetic film)

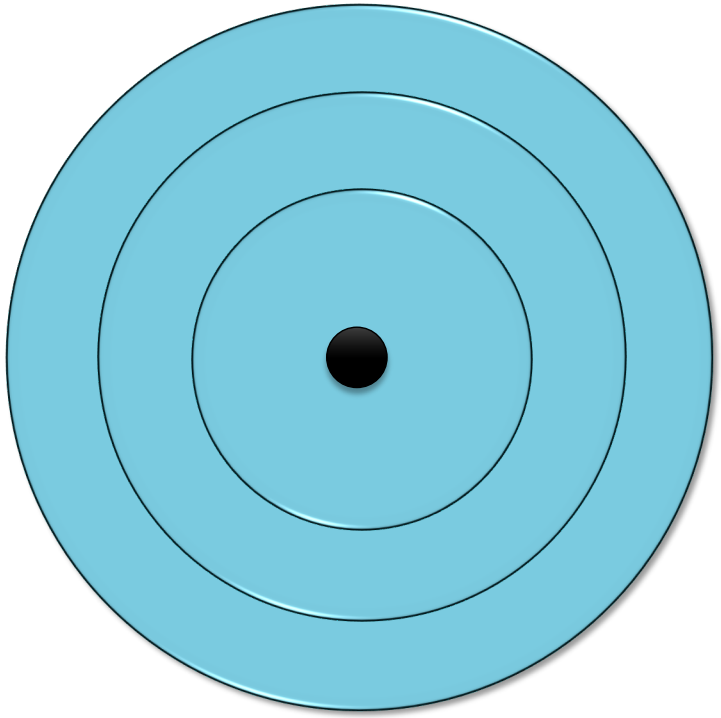


Disk Internals



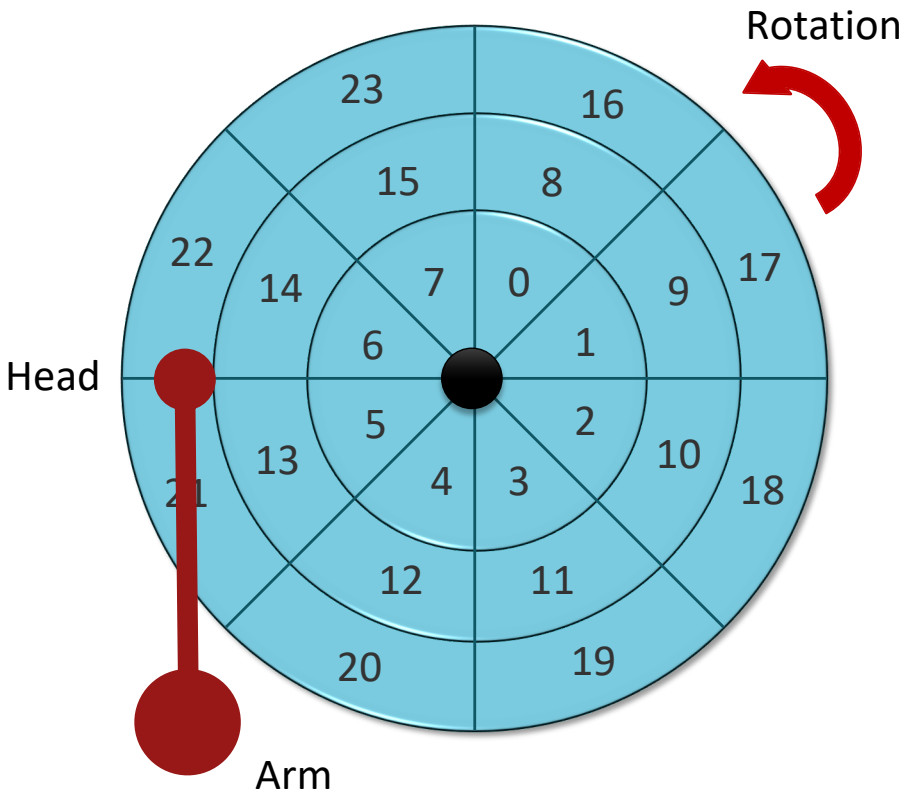
Disk Internals

- Each surface is divided into rings called tracks.
- A stack of tracks (across platters) is called a cylinder.

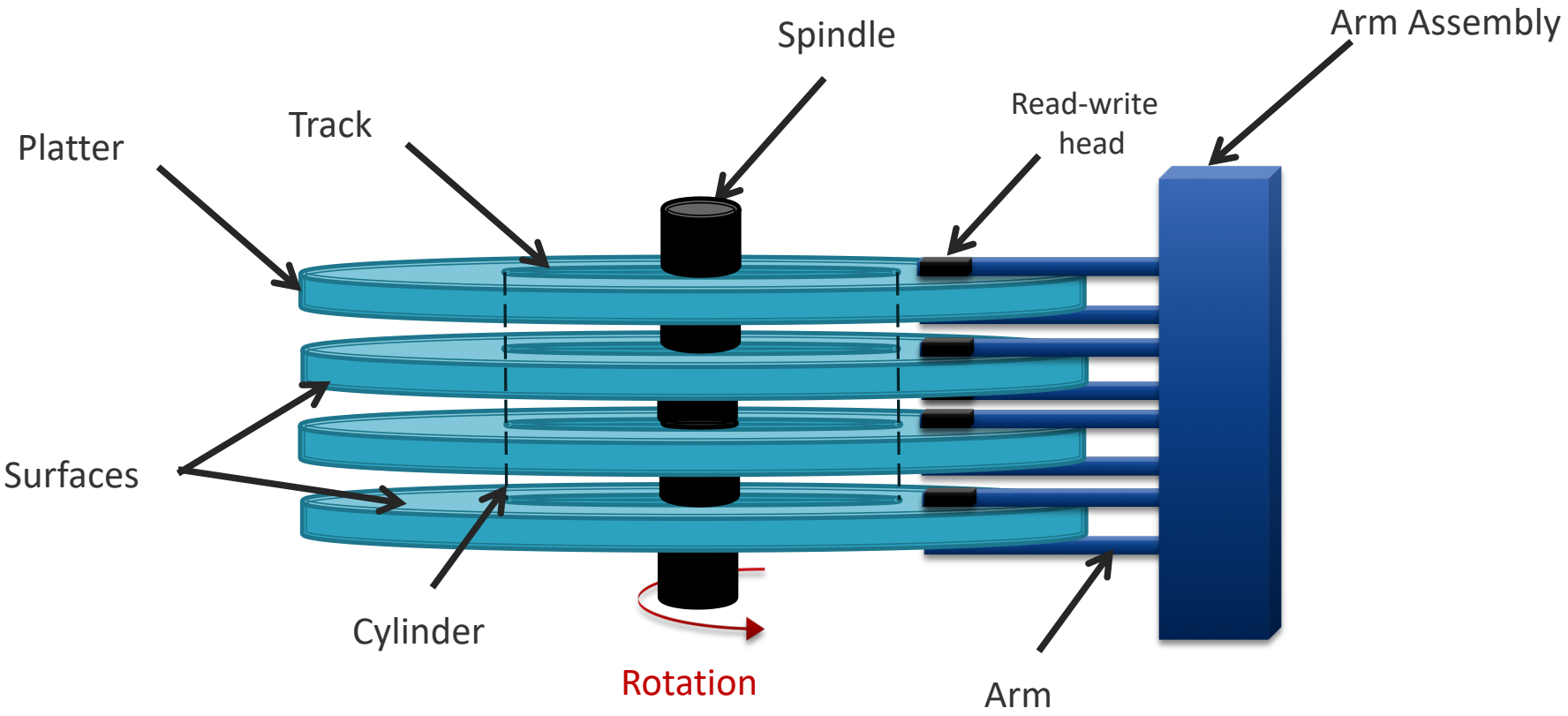


Disk Internals

- The tracks are divided into numbered sectors.
- Heads on a moving arm can read from each surface.



Disk Internals



Many platters may be bound to the spindle.

I/O Time

- Seek Time
  - There are multiple tracks per surface (many million).
  - Move the disk arm to the correct track.
    - Acceleration + Coasting + Deceleration.
    - Settling (quite significant 0.5-2ms) → Be sure it is in the right track.
  - During the seek, the platter keeps rotating.
  - Entire seeks often take several milliseconds (4 - 10 ms).
  - Approximate average seek distance = 1/3 max seek distance.

Slow!

- Rotational Delay
  - Wait for the desired sector to rotate under the disk head.
  - Depends on rotations per minute (RPM).
  - Average Rotational Delay = 1/2 rotation.
  - At 7200RPM → 8.3ms/rotation. Average = 4.15ms.

Slow!

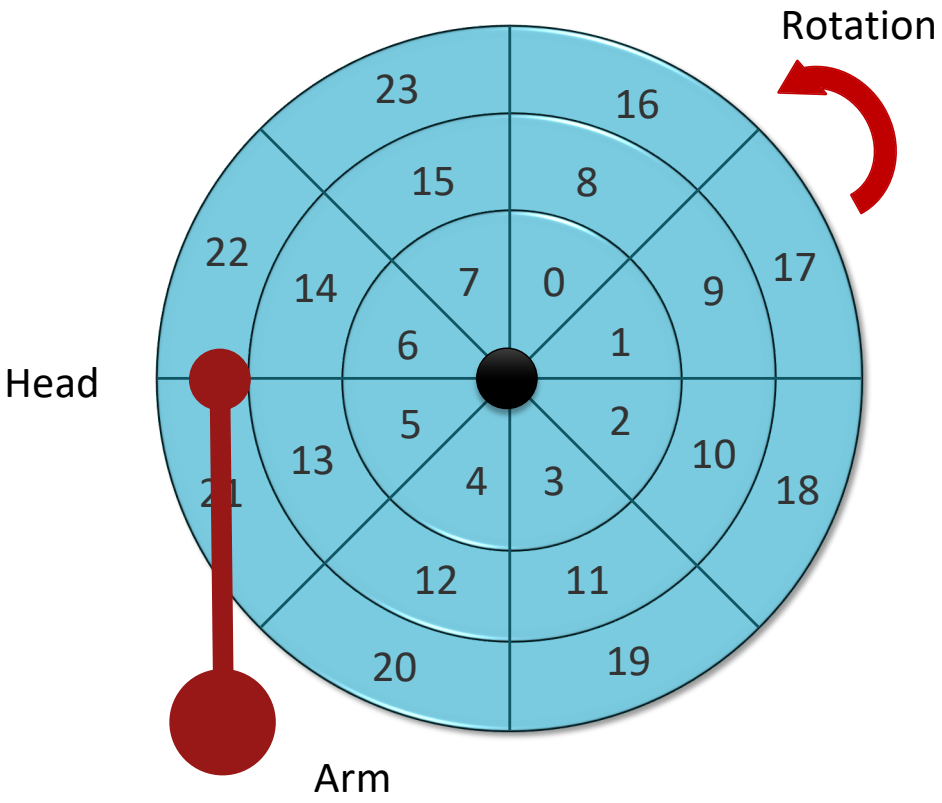
- Transfer Time
  - When the sector passes under the disk head, the data is either read from or written to the surface (transfer).
  - Pretty fast (>100 MB/s).
  - 512Bytes → ~5us.

Fast!

- $T_{I/O}=T_{seek} + T_{rotation} + T_{transfer}$

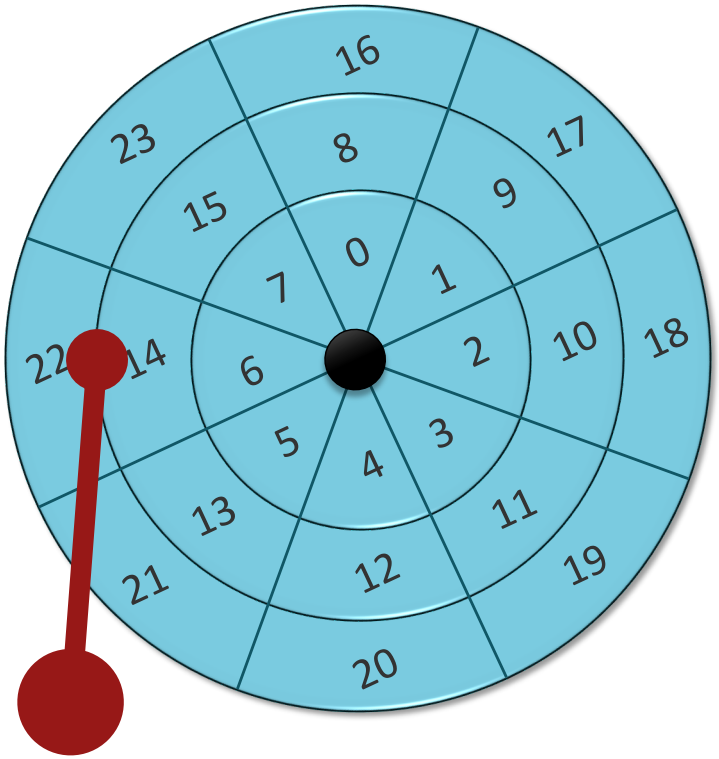
I/O Time

- Let's Read 12!



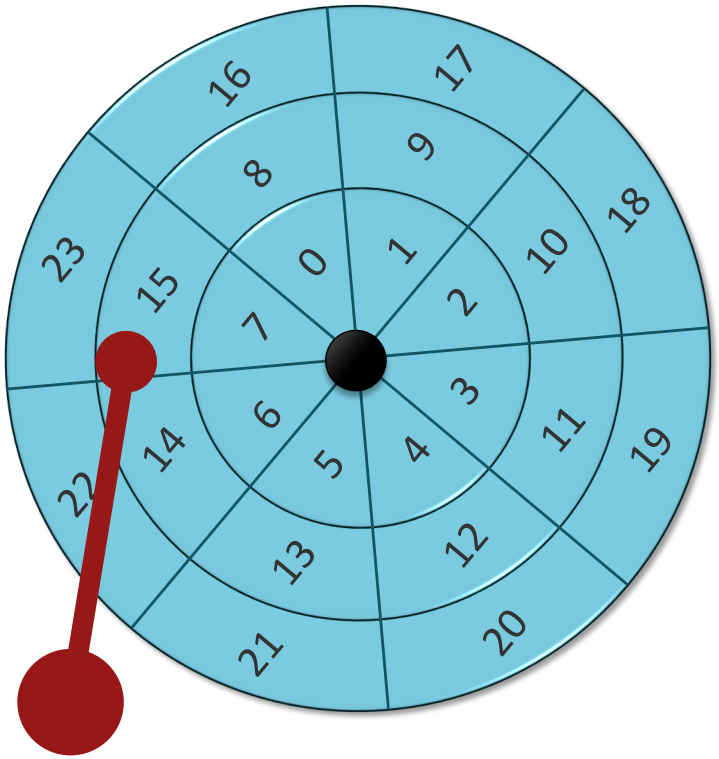
I/O Time

- Let's Read 12!
- Seek to right Track.



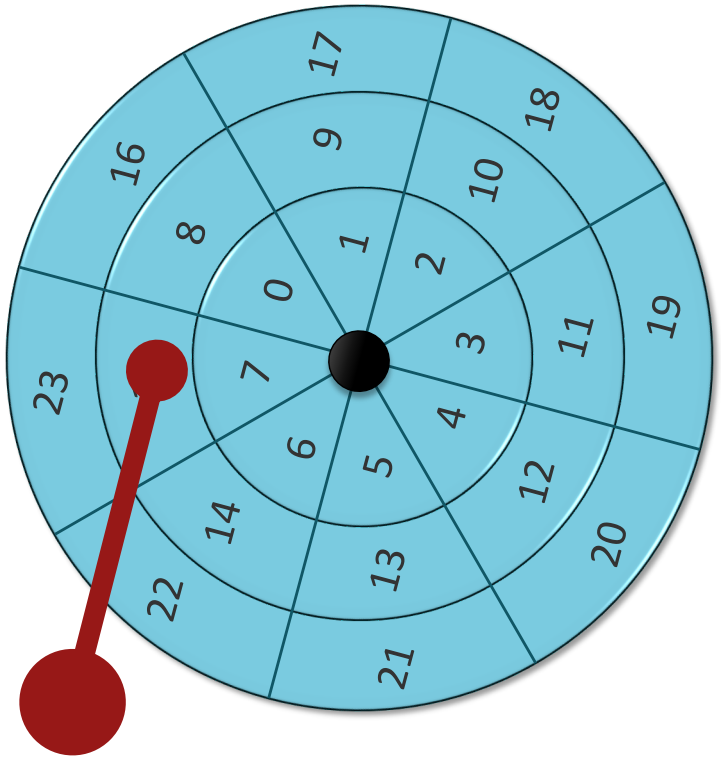
I/O Time

- Let's Read 12!
- Seek to right Track.



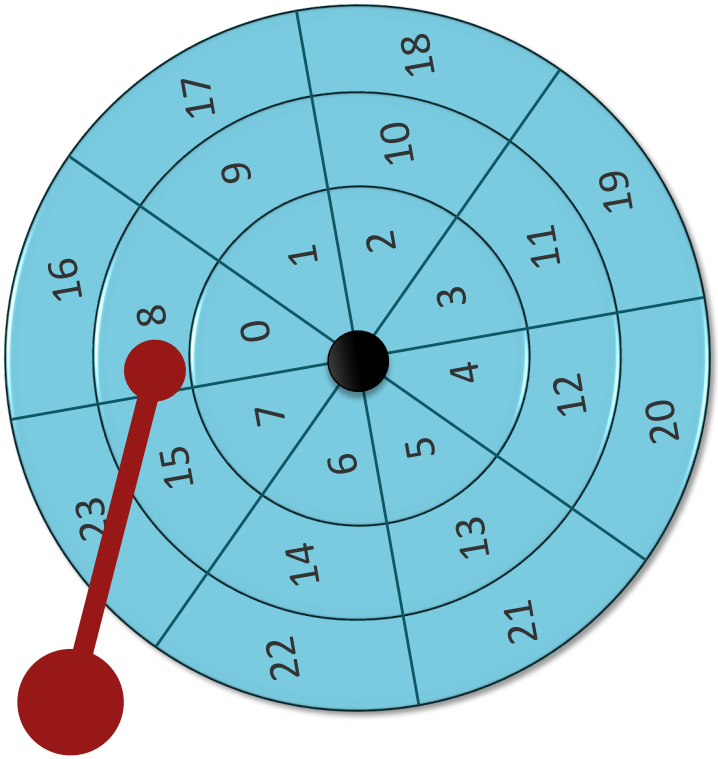
I/O Time

- Let's Read 12!
- Seek to right Track.



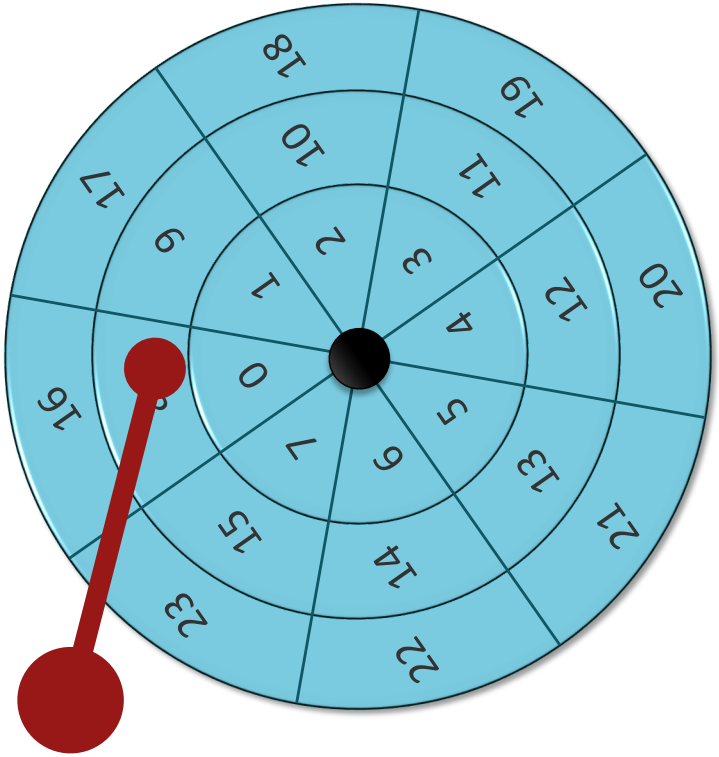
I/O Time

- Let's Read 12!
- Wait for Rotation.



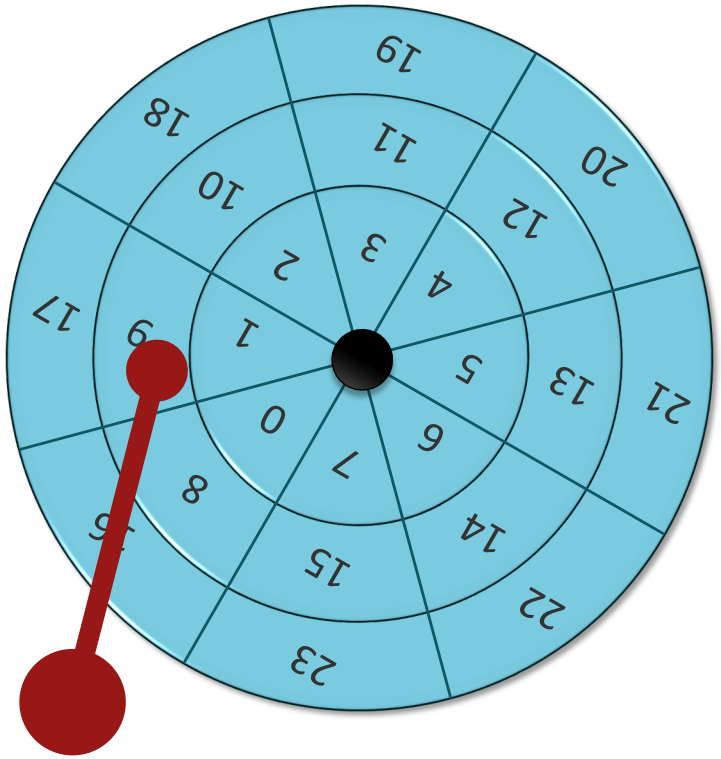
I/O Time

- Let's Read 12!
- Wait for Rotation.



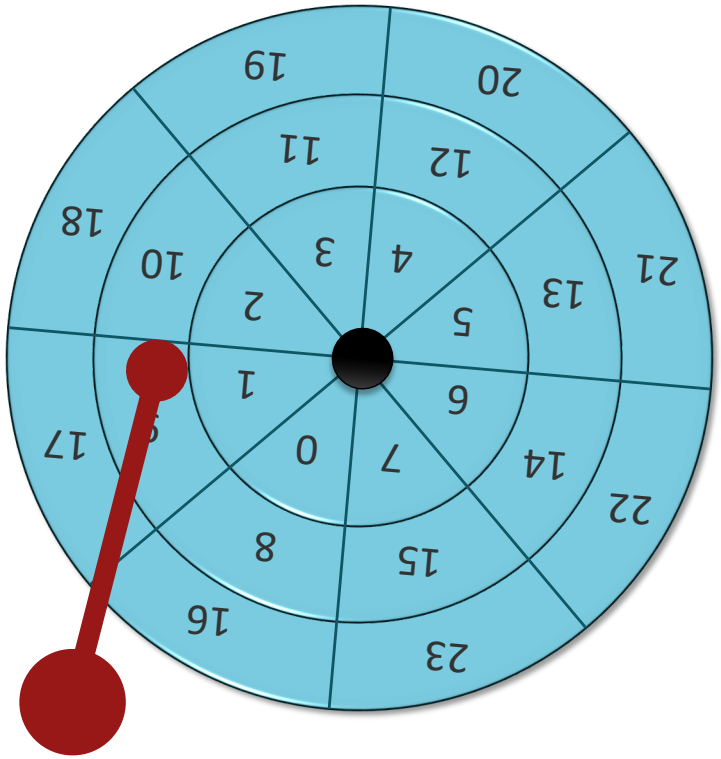
I/O Time

- Let's Read 12!
- Wait for Rotation.



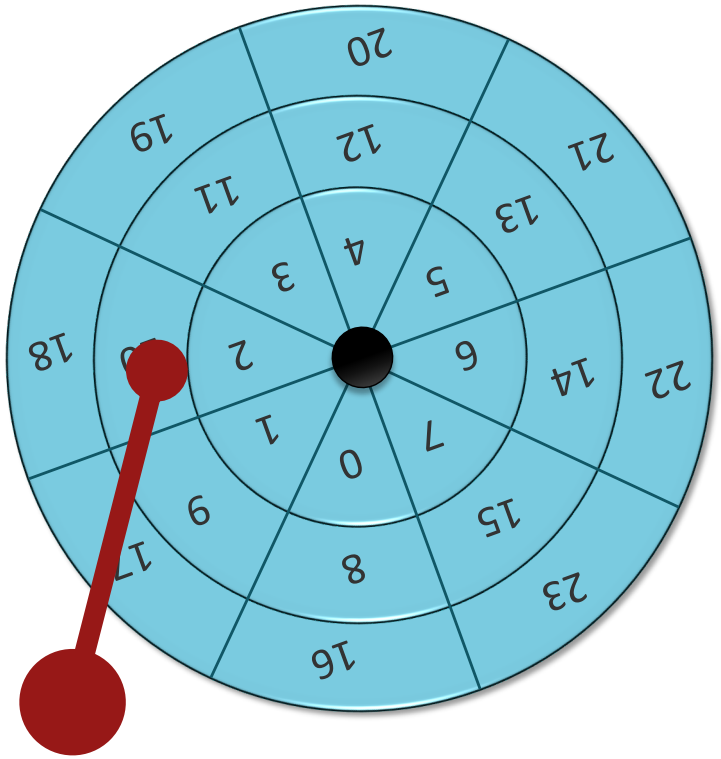
I/O Time

- Let's Read 12!
- Wait for Rotation.



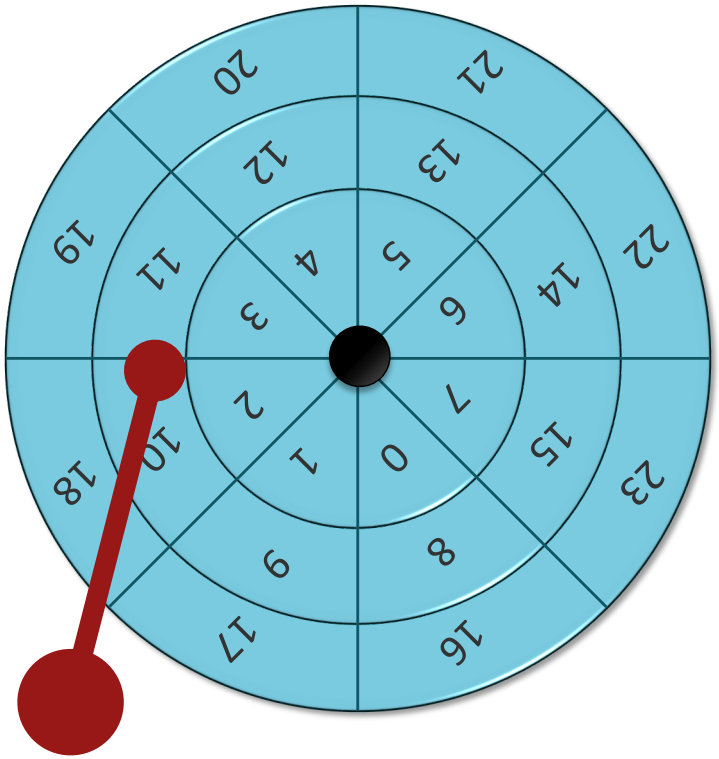
I/O Time

- Let's Read 12!
- Wait for Rotation.



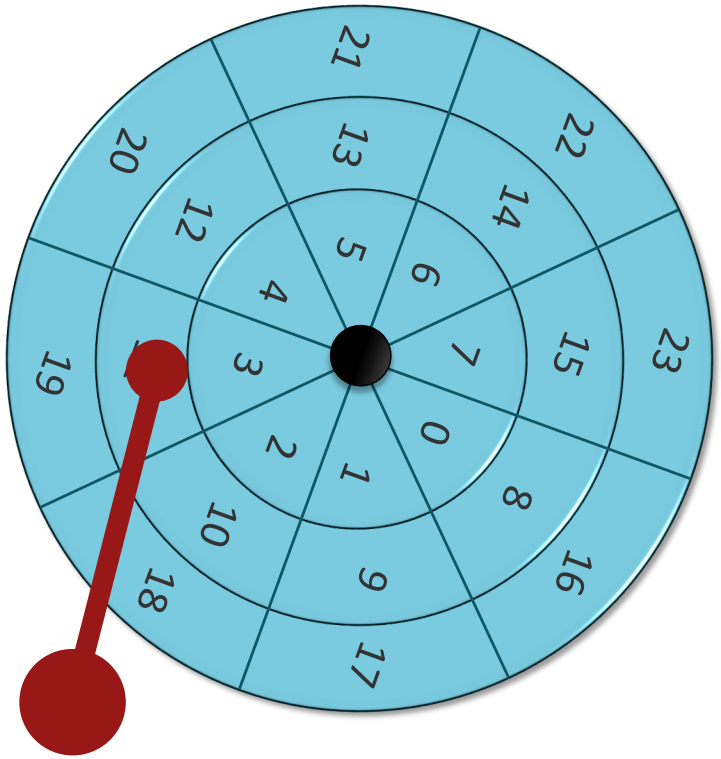
I/O Time

- Let's Read 12!
- Wait for Rotation.



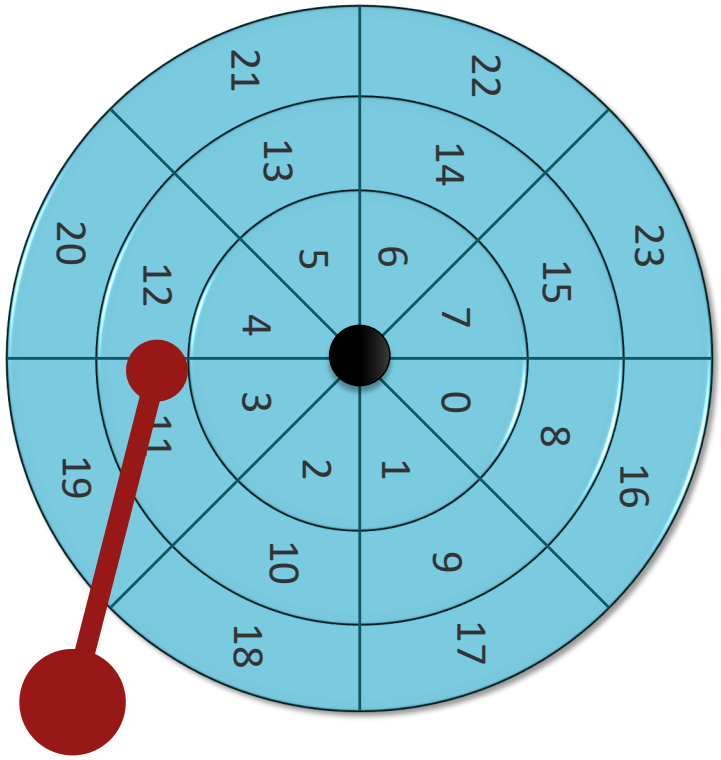
I/O Time

- Let's Read 12!
- Wait for Rotation.



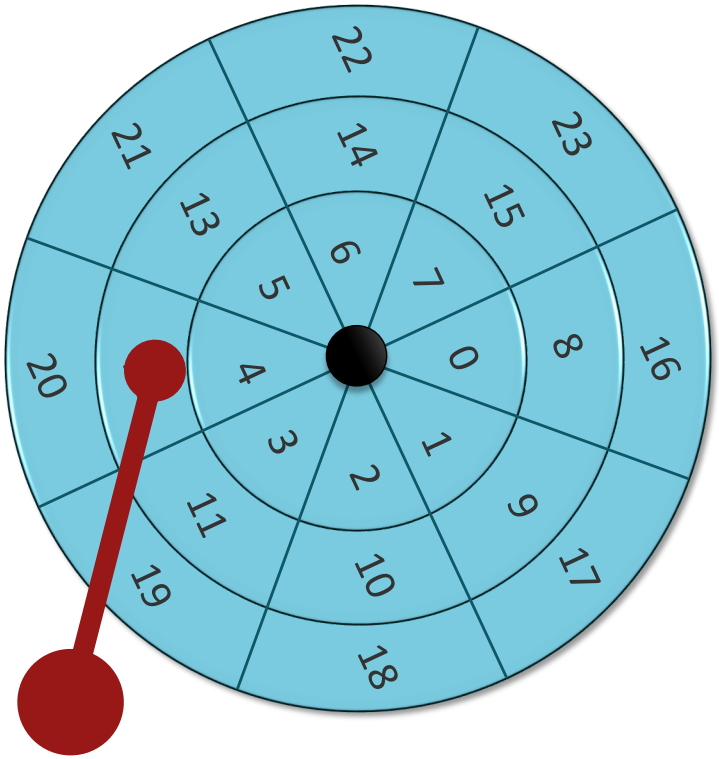
I/O Time

- Let's Read 12!
- Transfer Data!



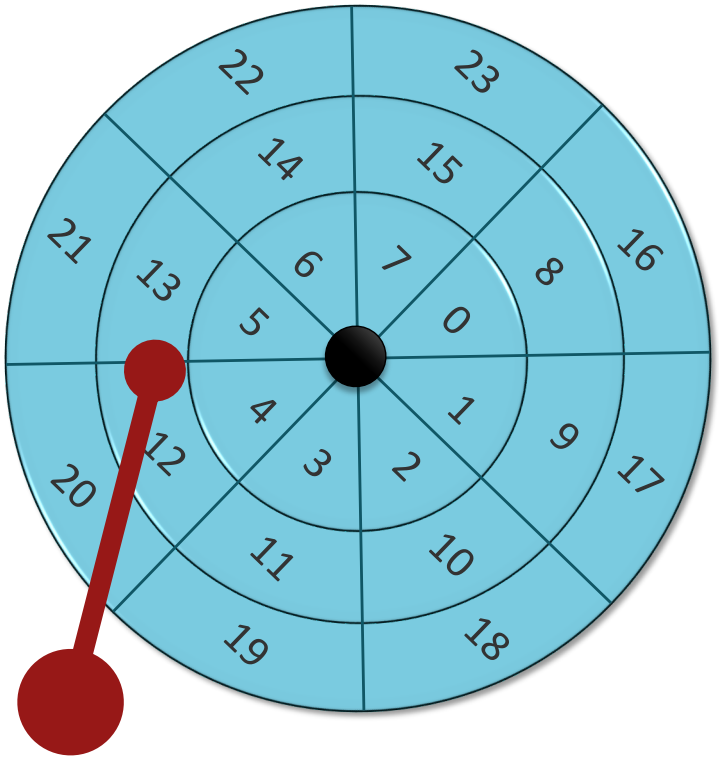
I/O Time

- Let's Read 12!
- Transfer Data!



I/O Time

- Let's Read 12!
- Done!



Hard Drive Demo

https://www.youtube.com/watch?v=L0nbo1VOF4M

I/O Time - Seek

- Many phases:
  - Acceleration
  - Coasting
  - Deceleration
  - Settling
- Often takes several milliseconds.
  - Setting alone can take 0.5 – 2 ms.
  - Entire seek often takes 4 – 10 ms.
- Seeks are slow!

I/O Time - Rotate

- Depends on rotations per minute (RPM)
  - 7200 RPM is common, 15000 is high end.

- 1/7200 RPM =

  1 minute /7200 rotations =

  1 second / 120 rotations =

  ~8.3 ms / rotation

  Average ~4.2 ms.

- Rotations are slow!

I/O Time - Transfer

- Pretty fast
  - depends on RPM and sector density.
- >100 MB/s is common.
  - 1s/100MB =
    10 ms. /MB
    Typically 4.9µs/sector (512 bytes sector)
- Transfers are fast!

Workload Performance

- So…
  - Seeks are slow
  - Rotations are slow
  - Transfers are fast

- What kind of workload is fastest for disks?
  - Sequential: access sectors in order (transfer dominated)
  - Random: access sectors arbitrarily (seek+rotation dominated)

Workload Performance

	Cheetah	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Avg Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	32 MB

Random Workload: Assume 16KB reads

Seek + rotation + transfer. → Per access!

Cheetah: 4ms + 2ms + 125µs ≈ 6.1 ms.
Throughput: 2.5 MB/s.

Barracuda: 9ms + 4.2ms + 149µs ≈ 13.3ms.
Throughput: 1.2 MB/s.

Workload Performance

	Cheetah	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Avg Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	32 MB

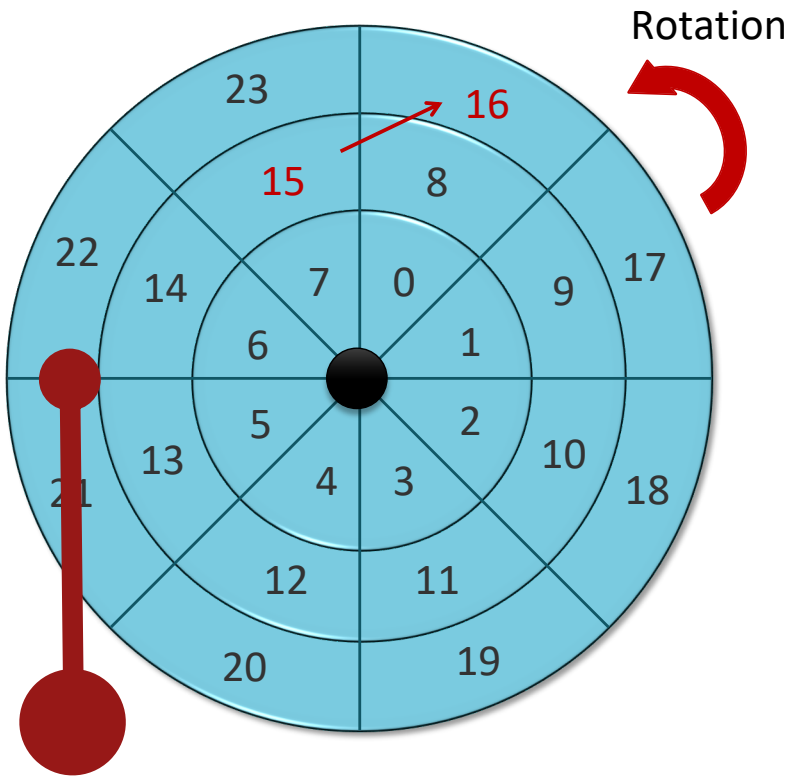
	Cheetah	Barracuda
Sequential	125 MB/s	105 MB/s
Random	2.5 MB/s	1.2 MB/s

Improvements

- Track Skew
- Zones
- Cache

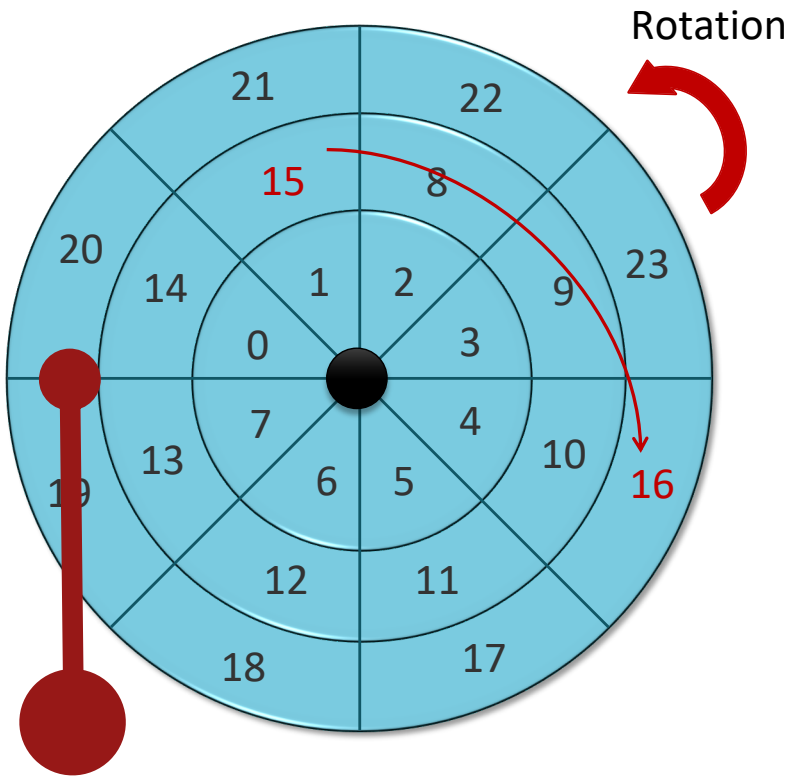
Track Skew

- When reading 16 after 15, the head won't settle quick enough, so we need to do a rotation.



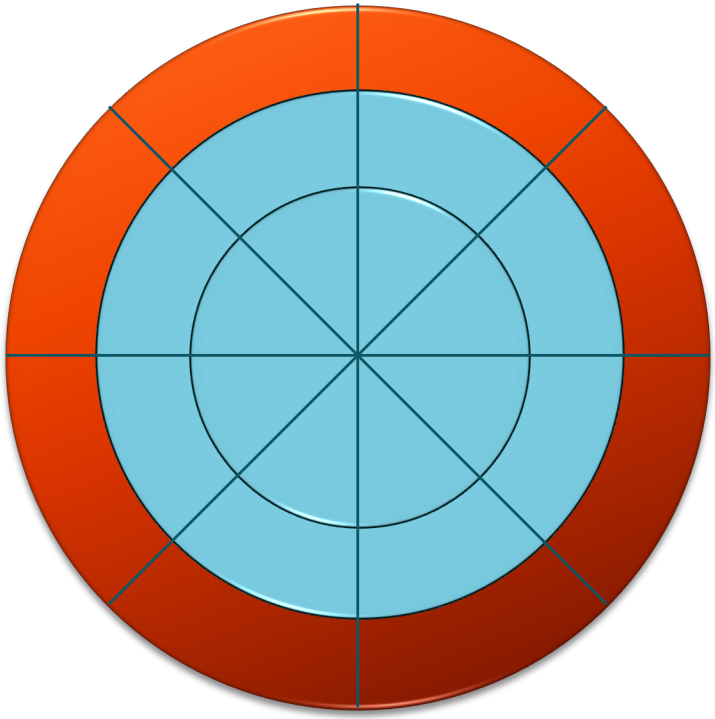
Track Skew

- Enough time to settle now



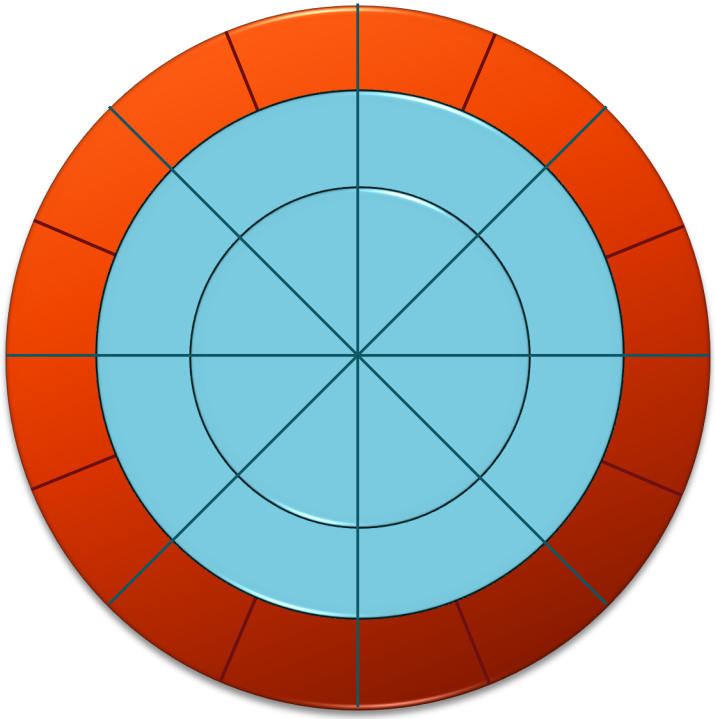
Zones

- Not every sector has the same “size”.



Zones

- Not every sector has the same “size”.
- Outer zones have more sectors than inner zones.



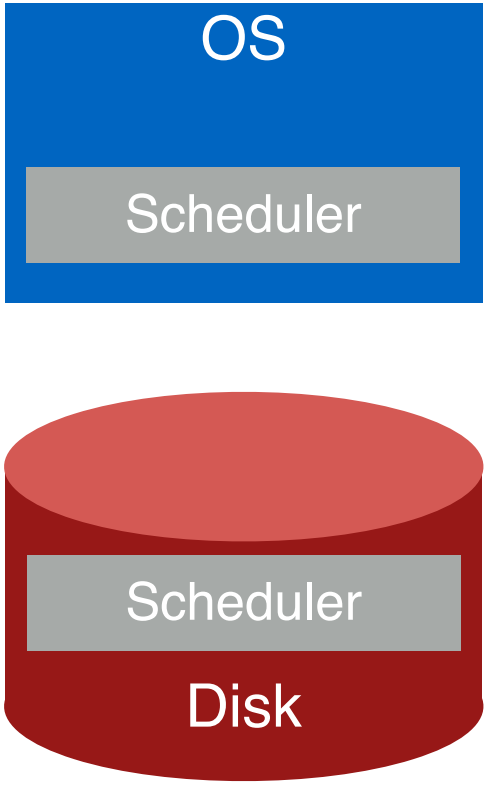
Drive Cache

- Drives may cache both reads and writes.
  - a.k.a. Track buffer.
  - Usually around 8-16MB.

- While reading
  - The drive might decide to read all the sectors on that track and cache it.
    - Allowing quick response to any subsequent request to the same track.
- While Writing
  - Acknowledge the write as the data is in cache.
    - Faster, but can lead to ordering problems.
    - Write can be delayed.
      - Avoid writing a file that soon is destroyed
      - Could imply data loss → fsync().
  - Acknowledge the write when it is actually written to disk.
    - Slower.

- Tagged command queuing
  - Have multiple outstanding requests to the disk
  - Disk can reorder (schedule) requests for better performance

Schedulers



Where should the scheduler go?

Schedulers

- FCFS (First Come First Served)
- SSF (Shortest Seek First)
  - Serves first requests closest to the current track.
  - Does not consider rotation time
  - How is this implemented in the OS?
    - Drive geometry not available at OS.
    - Use Nearest-Block-First (NBF)
  - Requests can starve.

How can starvation be avoided?

- SPTF (Shortest Positioning Time First)
  - Chooses request that requires least positioning time (time for seeking and rotating)
  - Greedy algorithm (just looks for best NEXT decision)
  - Usually implemented inside the drive
  - Easy for far away requests to starve.

Elevator Algorithms

- SCAN:
  - Sweeps back and forth, from one end of disk to the other, serving requests as it passes that cylinder
  - Sorts by cylinder number; ignores rotation delays
  - Pros/Cons?
    - Optimal → requests with uniform distribution
    - Don't take rotation into consideration

- C-SCAN (circular scan)
  - Only sweeps in one direction (outer-to-inner).
  - More fair (SCAN favors middle tracks).

Work Conservation

- Work conserving schedulers always try to do work if there's work to be done
- Sometimes, it's better to wait instead if system anticipates another ("better") request will arrive
- Such non-work-conserving schedulers are called anticipatory schedulers
  - CFQ (Completely Fair Queuing) → linux default

CFQ

- Completely Fair Queuing.
- Queue for each process.
- Do weighted round-robin between queues, with slice time proportional to priority.
- Optimize order within queue.
- Yield slice only if idle for a given time (kind of anticipation).

Solid State Disk

- Nonvolatile memory used like a hard drive.
- Many technology variations.
- Holds charge in cells. No moving parts!
- Inherently parallel.
  - HDD usually has only one head.
- No seeks!
  - HDD requires mechanical seek and rotate.
  - Better random I/O.

Solid State Disk

- Can be more reliable than HDDs.
- Much faster.
  - Throughput: HDD ~125MB/s vs. SDD ~200MB/s.
  - Latency: HDD ~10ms. vs. SDD 10µs – 2ms. (read-erase).
- More expensive per MB.
- May have shorter life span.
  - flash cells wear out after being overwritten too many times. (10k – 100k times).
  - Usage Strategy: Wear leveling.
- No direct HDD replacement.
  - Needs a complex layer for emulating hard disk API.

5.3 Persistence: File System

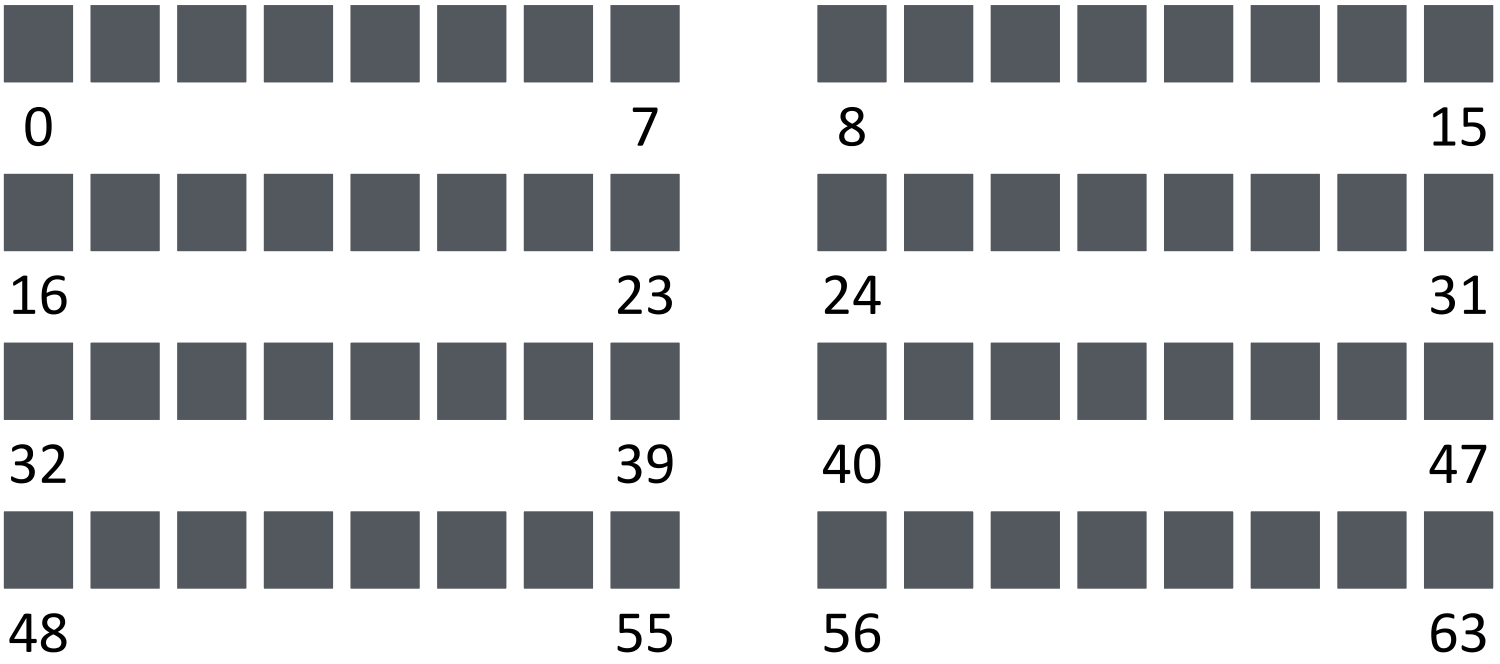
File System Abstraction

- The file system is pure software.
  - Refers to a collection of files.
  - Also refers to part of the OS that manages those files.
- What is a file?
  - An array of persistent bytes that can be read or written.
- File system consists of many files.
- Files need names so programs can choose the right one.

File Names

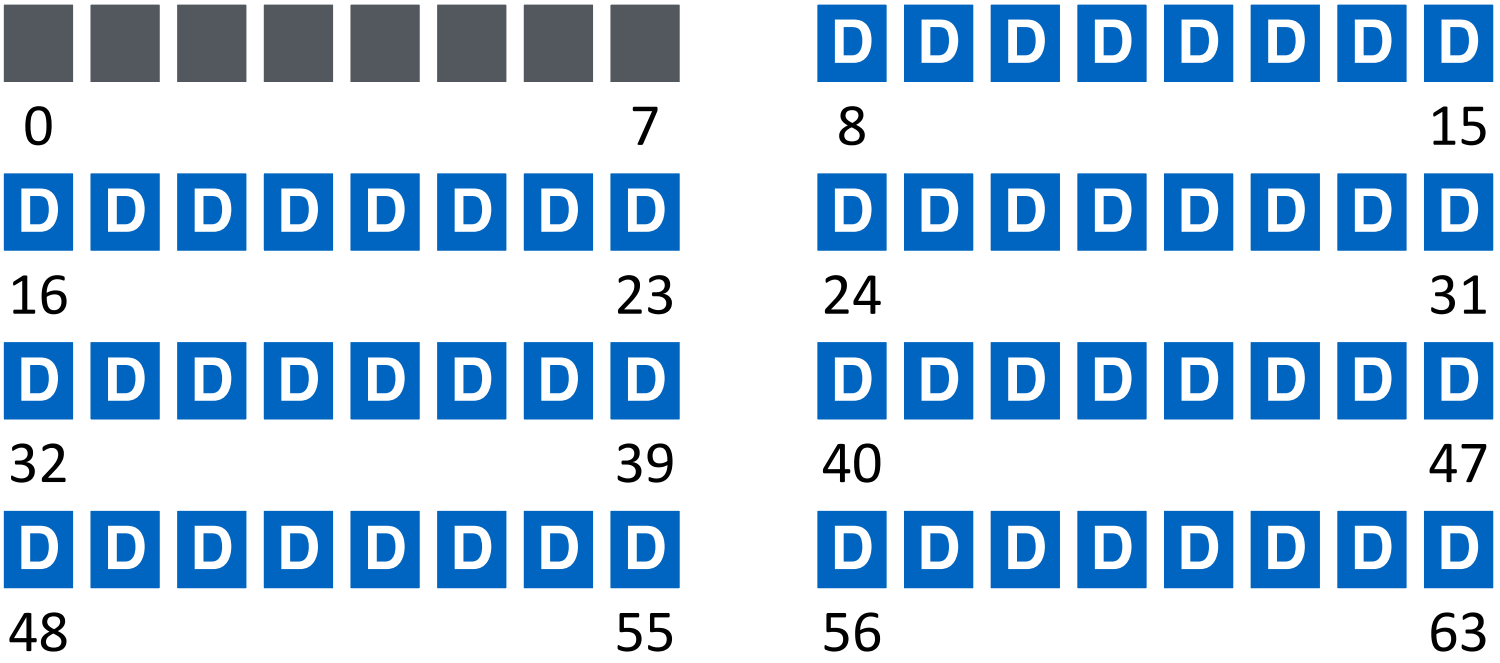
- Three types of file names:
  - File Descriptor
    - An integer, private per process, used to access files.
  - inode
    - Low level name.
    - Each file has exactly one inode number.
    - Unique within file system.
    - Records meta-data about file: file size, permissions, etc.
  - Path
    - User friendly name. Inside a directory tree.
    - Stores path-to-inode mappings for each directory.
    - “root” file typically inode 2.

FS Structs: Empty Disk

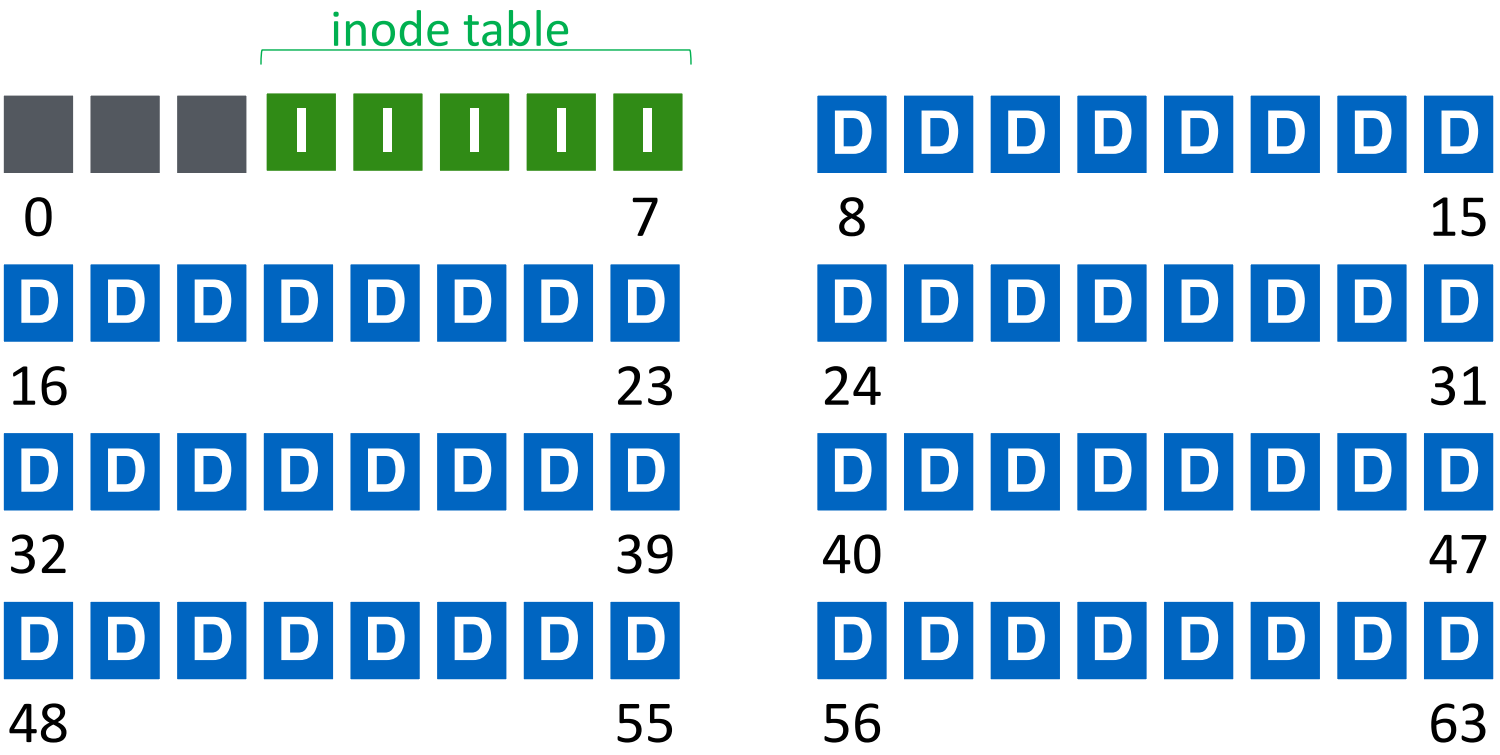


Assume each block is 4KB

Data Blocks



Inodes



One Inode Block

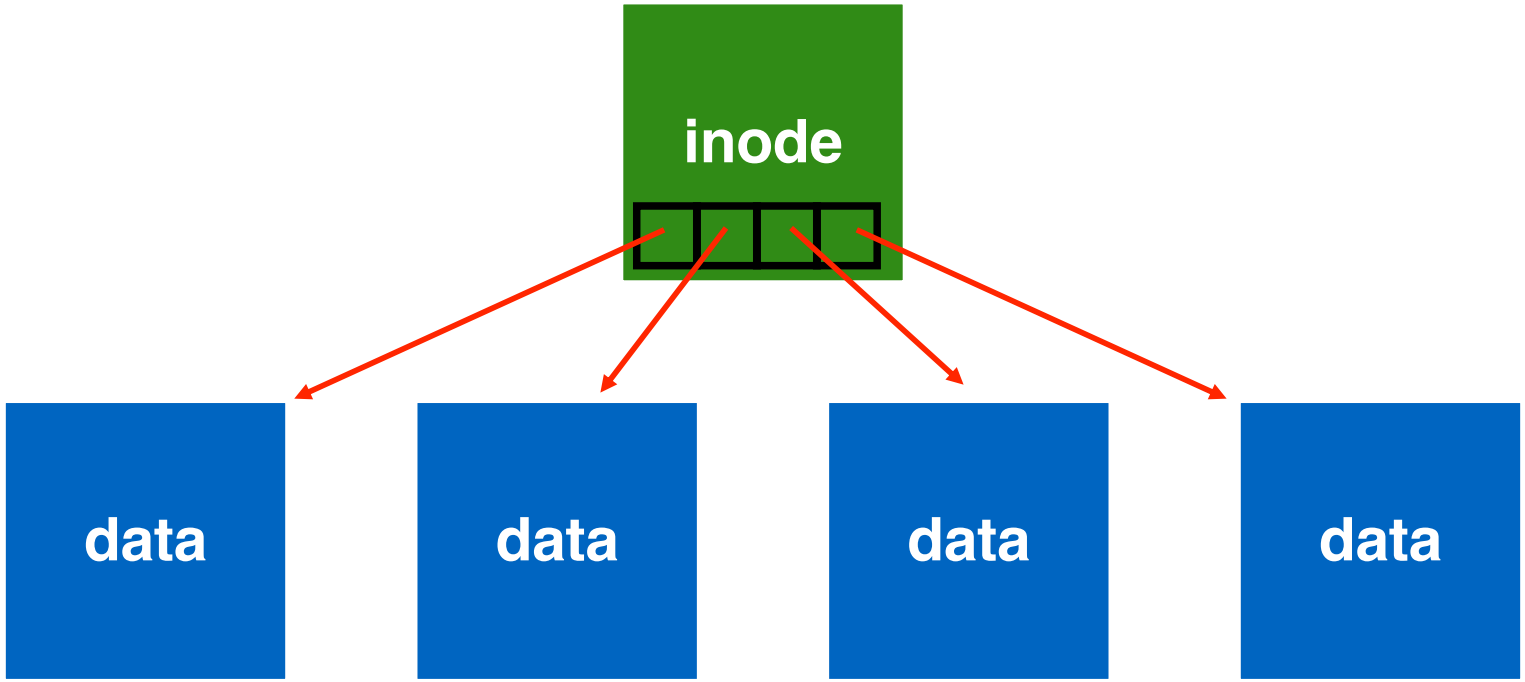
- Each inode is typically 256 bytes (depends on the FS, maybe 128 bytes)
- 4KB disk block
  - 16 inodes per inode block.
- What is max file size?
  - Assume 256-byte inodes

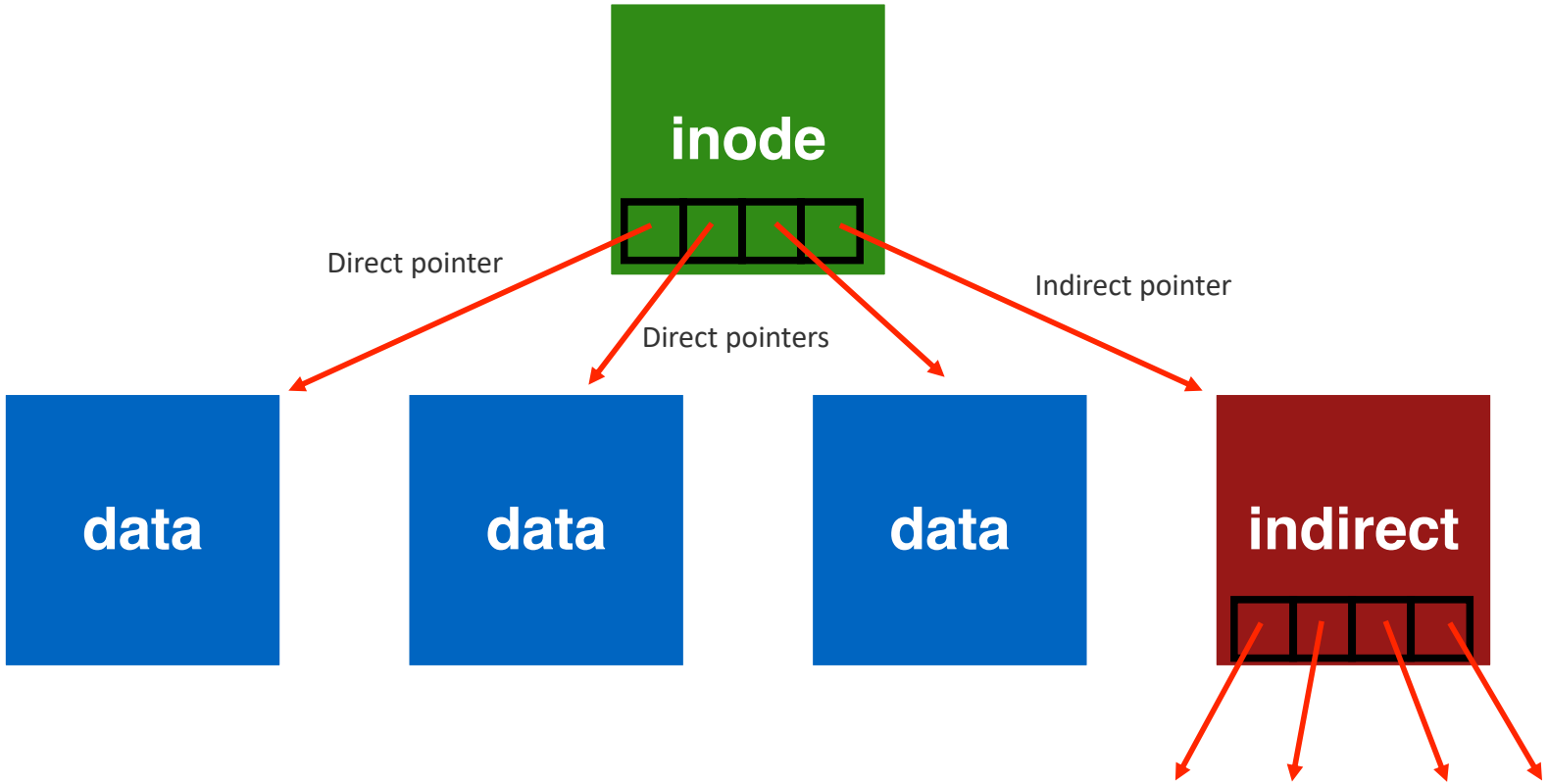
  (all can be used for pointers)
  - Assume 4-byte block addrs

(256/4)*4K=256KB

type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
Blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

How can larger files be obtained?


inode
data
data
data
data



- Multilevel Index
  - Inode can have one pointer devoted to indirect block.
    - A data block full of direct pointers → with 4byte addresses, 1024 pointers in the data block.
  - For bigger files, the inode can have a double indirect pointer.
    - A pointer to a data block that contains pointers to indirect blocks.
  - Triple indirect pointer... Multi-level index.
  - e.g. 12 direct pointers, 1 single indirect, 1 double → over 4GB in size.

Why direct pointers?

Most files are small!

Inode approaches

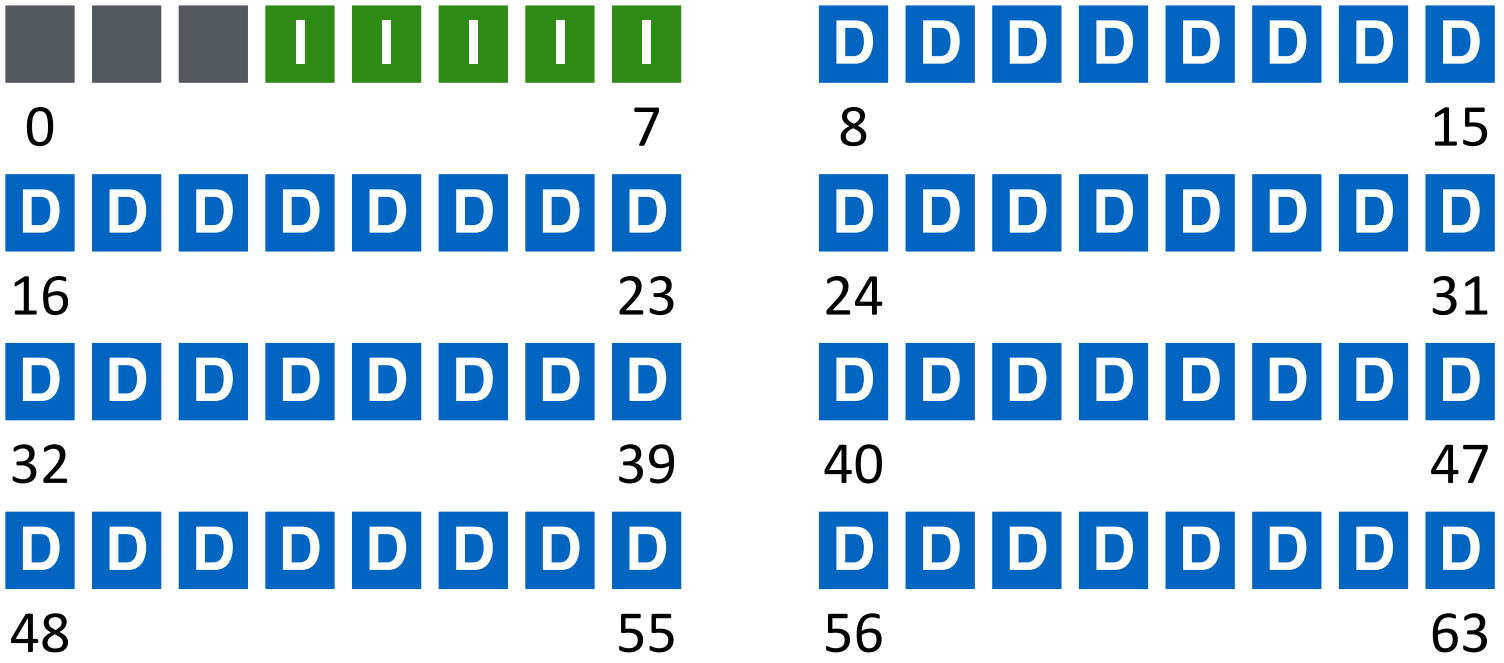
- Indexed
  - Allocate fixed-sized blocks for each file.
  - No external fragmentation.
  - Files can be easily grown up to max file size.
  - Supports random access but wastes space for unneeded pointers.
- Multi-level Indexed
  - Dynamically allocate hierarchy of pointers to blocks as needed.
  - Meta-data: Small number of pointers allocated statically.
  - Linux ext2, ext3, original UNIX file system.

- Extent-based
  - Similar to segments in memory virtualization.
  - Extent instead of block pointers: disk pointer + length in blocks.
  - Helps fragmentation.
  - Can grow (until run out of extents).
  - Still good performance and little overhead.
- Flexible # of Extents
  - Dynamic multiple contiguous regions (extents) per file.
  - Organize extents into multi-level tree structure.
  - Linux ext4.

- Linked
  - Each block contains a pointer to next block.
  - No external fragmentation.
  - Can easily grow.
  - Relatively poor performance and some overhead (pointer per block).
- File-allocation Tables (FAT)
  - Keep linked-list information for all files in on-disk FAT table.
  - Allow random access with better performance than linked.
  - Read from two disk locations for every data read.
  - Not exactly inodes.

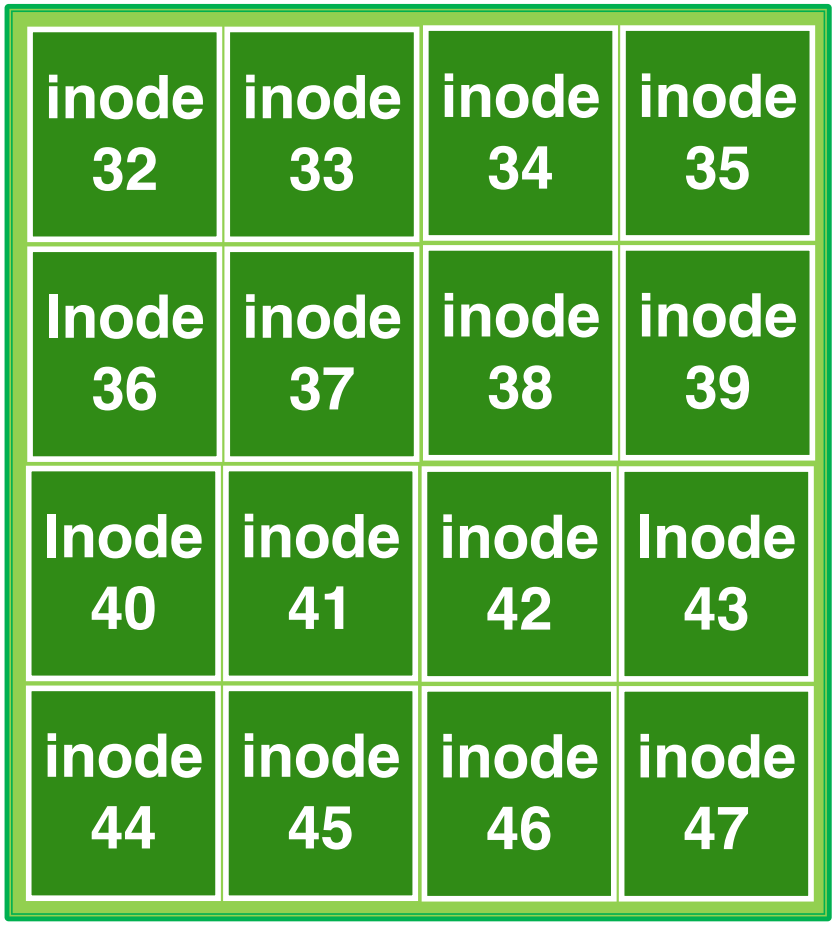
Example

Assume 256 byte inodes (16 inodes/block).
What is offset for inode with number 40?

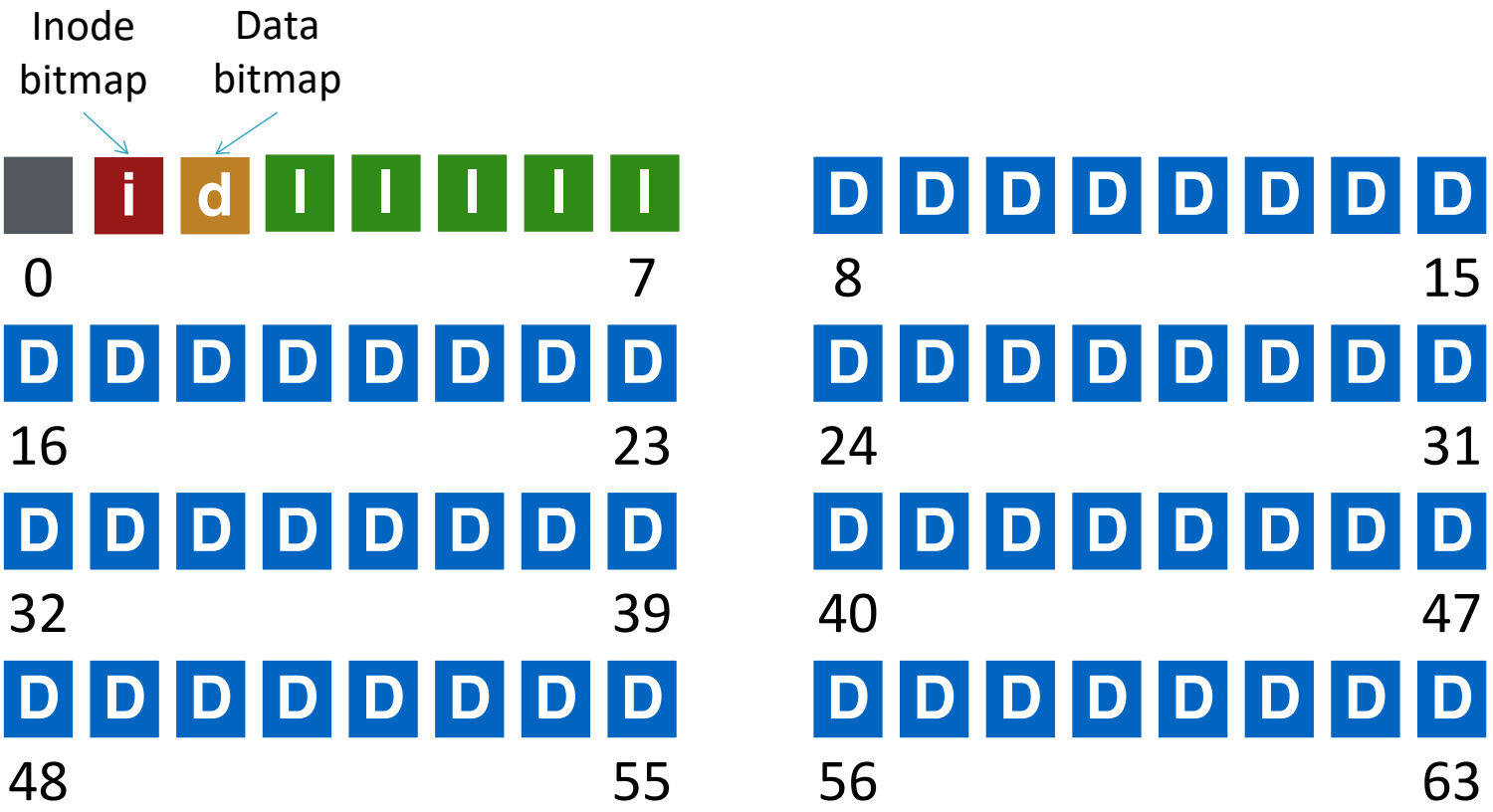


Example

- Each inode is typically 256 bytes (depends on the FS, maybe 128 bytes)
- 4KB disk block
  - 16 inodes per inode block.
- Inodes start at 12KB (superblock at 0KB, i-bitmap at 4KB, d-bitmap at 8KB and inode at 12KB).
- Inode 40 → 12KB+40*sizeof(inode) = 12KB+40*256 = 22KB (third inode block)
- Disk is sector addressable (i.e. 512bytes)
  - Inode 40 (22KB)
  - Sector = 22*1024/512 = 44

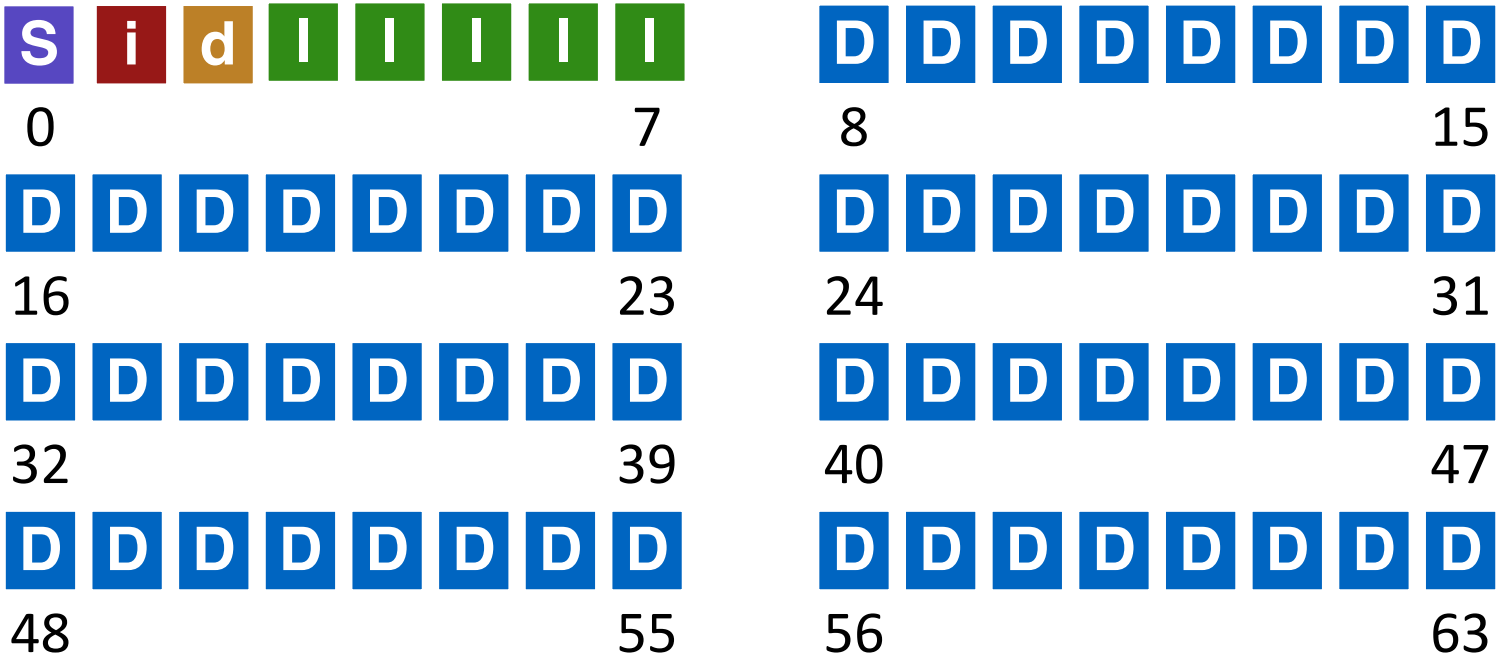


Bitmaps



32k inodes, 32k datablocks. More than enough in this FS

Superblock



- Contains information about the file system:
  - How many inodes and data blocks.
  - Where the inode table begins.
  - Magic number to identify the file system type.

Directory Organization

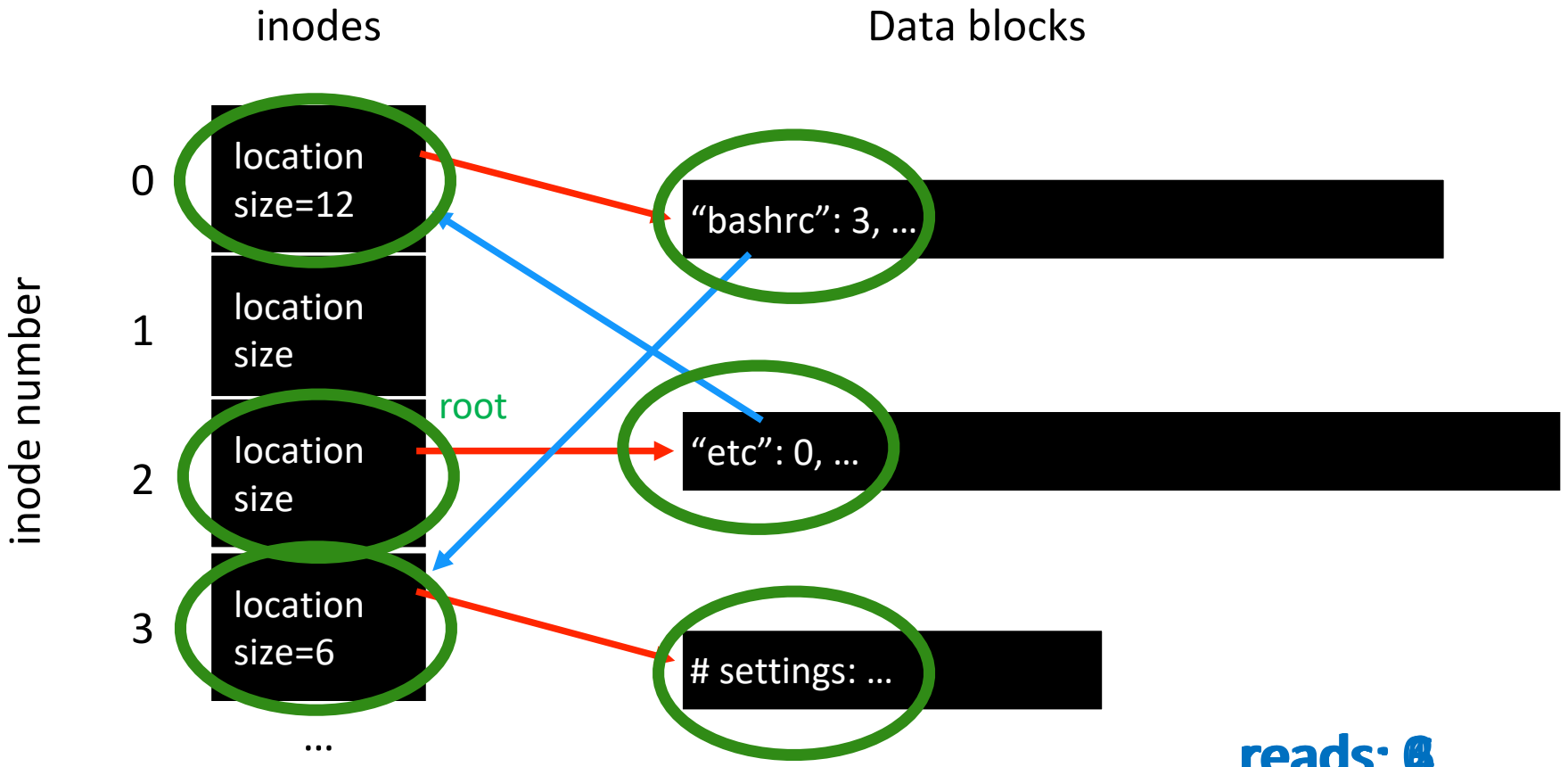
- Common design:
  Store directory entries in data blocks
  - Large directories just use multiple data block
  - Use bits in the inode to distinguish directories from files
  - For each file or directory in the directory, there is a directory entry with the name and the inode of the element.
- Example:

inum	rclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

- In the example: each entry has an inode number, record length, string length, and the name of the entry.

Example

read **/etc/bashrc**



Reads for getting final inode called "traversal"

create /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
		read					
					read		
			read				
						read	
	read						
	write						
						write	
				read			
				write			
			write				

- 1. read root inode and get its data location.
- 2. read data location and look for “foo” inode.
- 3. read “foo” inode and get its data location.
- 4. read “foo” data and look for “bar” inode (verify doesn’t exist).
- 5. read inode bitmap and locate a free inode.
- 6. mark the bit of the corresponding inode as used.
- 7. write “bar”-inode (recently acquired) entryin the “foo” directory.
- 8. read “bar” inode.
- 9. Initialize “bar” inode (write).
- 10. Update directory inode.

open /foo/bar (assume file exist)

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
		read					
					read		
			read				
						read	
				read			

- 1. read root inode and get its data location.
- 2. read root data and look for “foo” inode.
- 3. read “foo” inode and get its data location.
- 4. read “foo” data and look for “bar” inode.
- 5. load “bar” inode in memory (FS checks permission, assign file descriptor...).

write /foo/bar (assume file is open)

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
read							
write							
							write
				write			
				read			
read							
write							
							write
				write			

- 1. read “bar” inode (already open, but with no data assigned).
- 2. read data bitmap and locate a free data block.
- 3. mark the bit of the corresponding data block as used.
- 4. write data into “bar” data block (4K).
- 5. Update “bar” inode.

read /foo/bar (assume file is open)

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data
				read			
							read
				write			
				read			
							read
				write			

- 1. read “bar” inode and get its data location.
- 2. read “bar” data.
- 3. update “bar” inode.

close /foo/bar

data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data

nothing to do on disk!

Crash Consistency

- File system is appending to a file and must update:
  - - inode
  - - data bitmap
  - - data block
- What happens if crash happens after only updating some blocks?
  - a) bitmap: lost block (space leak)
  - b) data: nothing bad (as if write never occurred)
  - c) inode: point to garbage / another file may use it
  - d) bitmap and data: lost block
  - e) bitmap and inode: point to garbage
  - f) data and inode: another file may use it

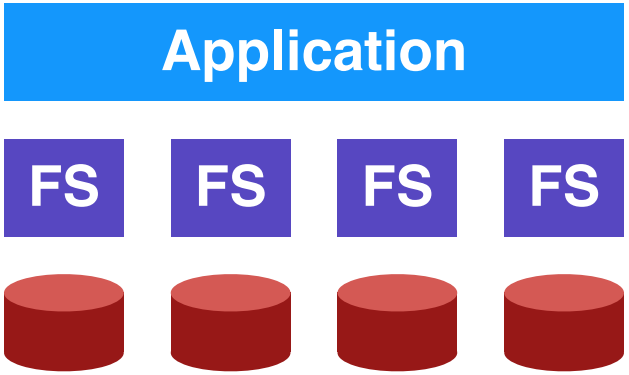
Solutions

- FSCK = file system checker
  - After crash, scan whole disk for contradictions and “fix” if needed.
  - Keep file system off-line until FSCK completes.
  - Some states look consistent but they are not
    - like inode pointing to garbage data.

- Journaling
  - Don’t move file system to just any consistent state, get correct state → Atomicity.
  - Write-ahead logging.
  - Never delete ANY old data, until, ALL new data is safely on disk
  - Add work during updates to reduce work during recovery.
  - Extra blocks are called a “journal”.
  - Most modern file systems use journals (ext3, ext4, NTFS...).

5.4 Persistence: Multiple Disks - RAID

Solution 1: JBOD



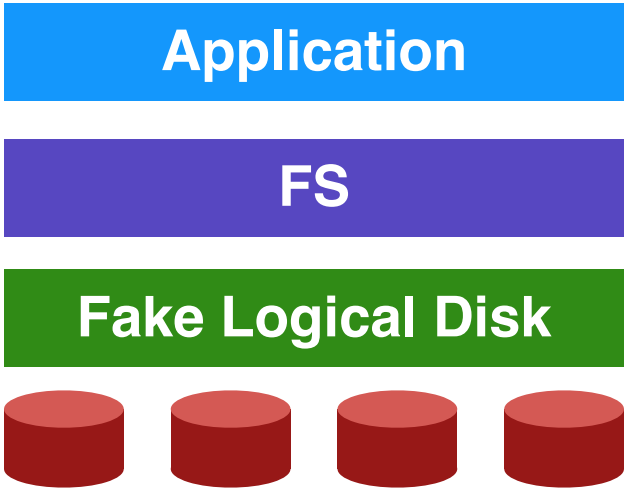
Application is smart, stores different files on different file systems.

JBOD: **J**ust a **B**unch **O**f **D**isks

Solution 2: RAID

RAID is:

- transparent
- deployable



Logical disk gives

- capacity
- performance
- reliability

Build logical disk from many physical disks.

RAID: **R**edundant **A**rray of **I**nexpensive **D**isks

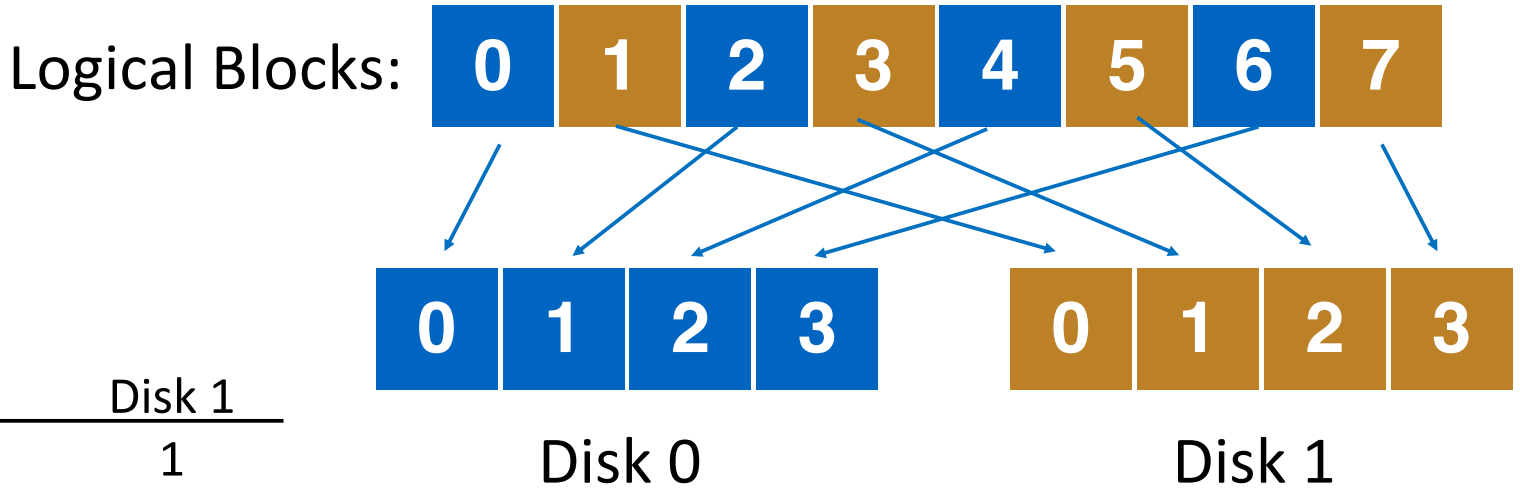
Metrics

- Capacity: how much space can apps use?
- Reliability: how many disks can we safely lose?
  (assume fail stop!)
- Performance: how long does each workload take?
- Normalize each to characteristics of one disk

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

RAID-0: Striping

- Optimize for capacity. No redundancy



Disk 0	Disk 1
0	1
2	3
4	5
6	7

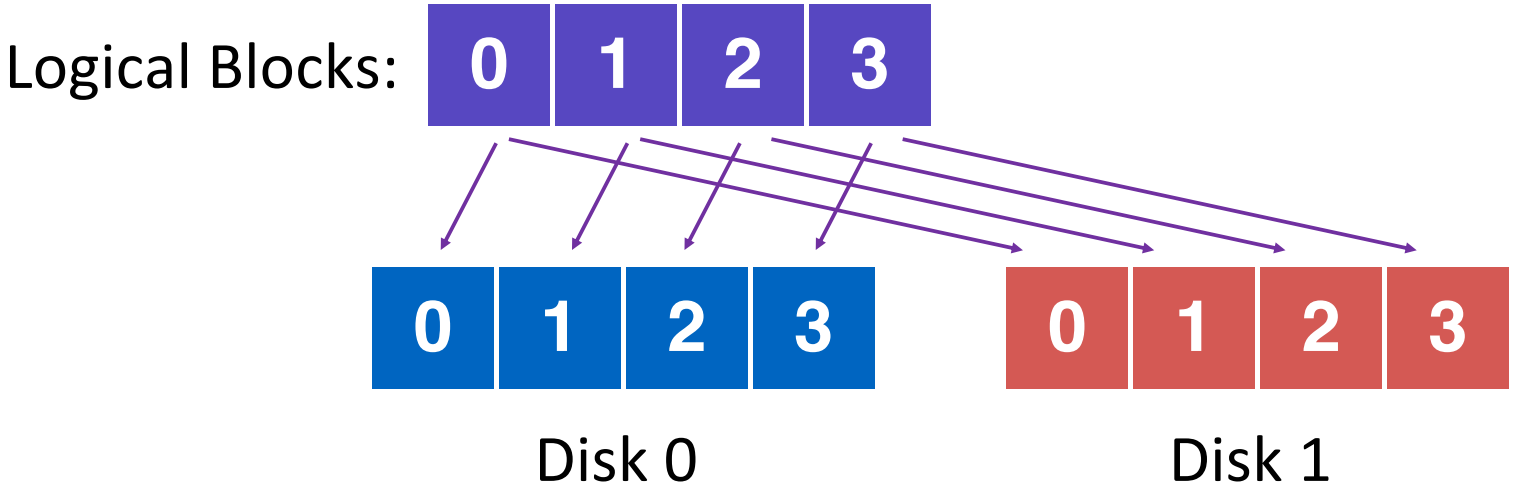
RAID-0: Analysis

- Capacity? **N * C**
- How many disks can fail? **0**
- Latency **D**
- Throughput (sequential, random)? **N*S , N*R**

Buying more disks improves throughput, but not latency!

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

RAID-1: Mirroring



Keep two copies of all data.

Redundancy improves reliability
but decreases space efficiency

Raid-1 Layout

2 disks

Disk 0	Disk 1
0	0
1	1
2	2
3	3

4 disks

Disk 0	Disk 1	Disk 2	Disk 4
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

RAID-1: Analysis

- Capacity? **N/2 * C**
- How many disks can fail? **1 (or maybe N / 2)**
- Latency (read, write)? **D (slightly higher on writes)**
- What is steady-state throughput for
  - random reads? **N * R**
  - random writes? **N/2 * R**
  - sequential writes? **N/2 * S**
  - sequential reads? **Book: N/2 * S (other models: N * S)**

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

Raid-4 Strategy

- Reduce redundancy → Use parity disk
- In algebra, if an equation has N variables, and N-1 are known, you can often solve for the unknown.
- Treat sectors across disks in a stripe as an equation.
- Data on bad disk is like an unknown in the equation.

	Disk0	Disk1	Disk2	Disk3	Disk4
Stripe:	5	X	0	1	9
					(parity)

	Disk0	Disk1	Disk2	Disk3	Disk4
Stripe:	5	3	0	1	9
					(parity)

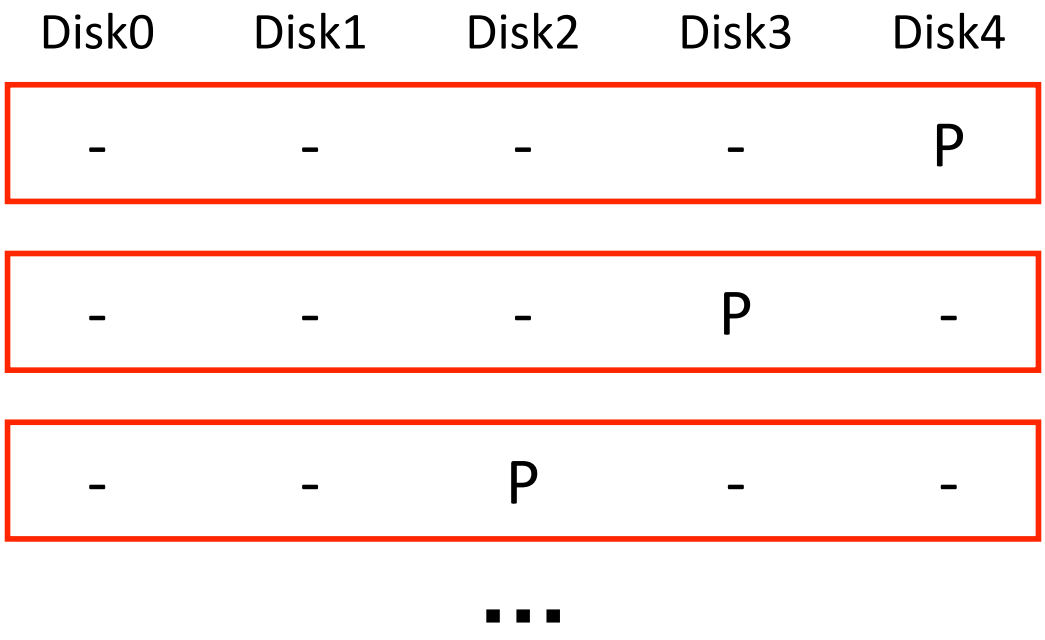
RAID-4: Analysis

- Capacity? **(N-1) * C**
- How many disks can fail? **1**
- Latency (read, write)? **D, 2*D (read and write parity disk)**
- What is steady-state throughput for
  - random reads? **(N-1) * R**
  - random writes? **R/2 (read and write parity disk)**
  - sequential writes? **(N-1) * S** All at once
  - sequential reads? **(N-1) * S**

$P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old}$

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

RAID-5



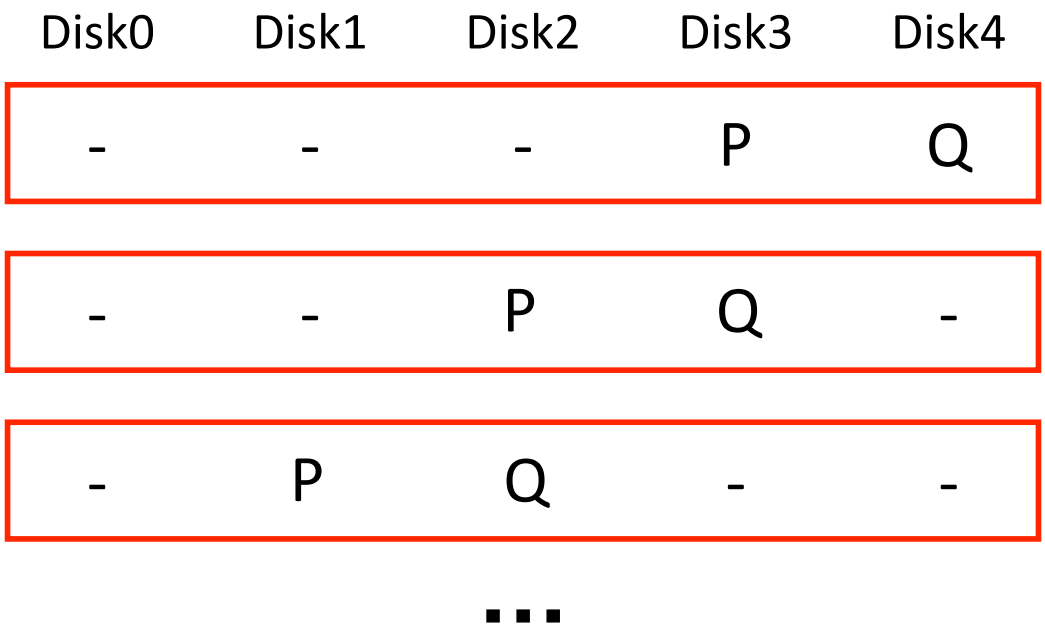
Rotate parity across different disks

RAID-5: Analysis

- Capacity? **(N-1) * C**
- How many disks can fail? **1**
- Latency (read, write)? **D, 2*D (read and write parity disk)**
- What is steady-state throughput for
  - random reads? **N * R**
  - random writes? **N* R/4 (each write needs 4 ops)**
  - sequential writes? **(N-1) * S**
  - sequential reads? **(N-1) * S**

N := number of disks
C := capacity of 1 disk
S := sequential throughput of 1 disk
R := random throughput of 1 disk
D := latency of one small I/O operation

RAID-6



Double parity check

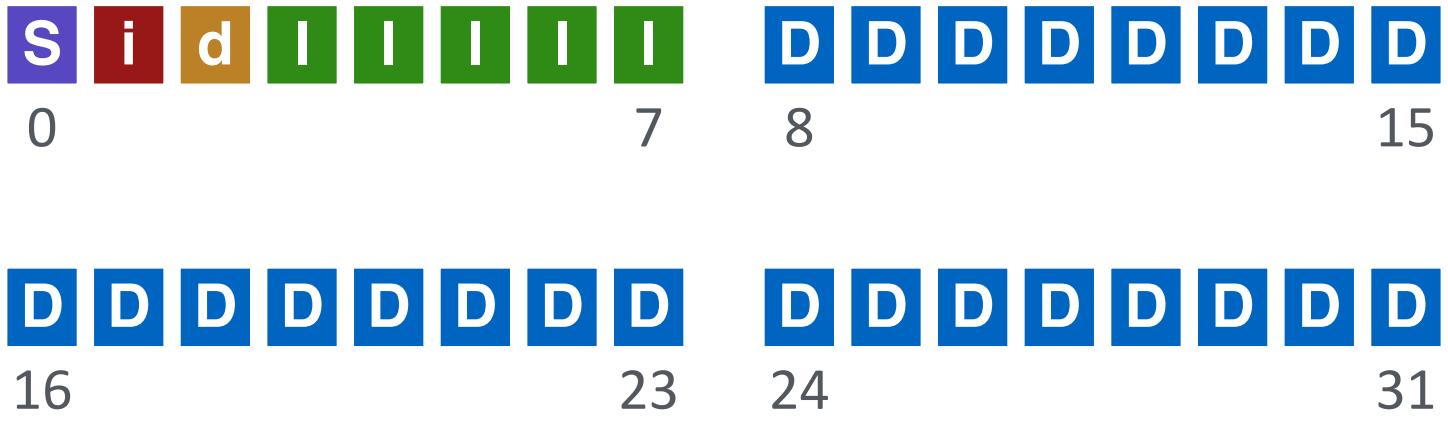
RAID Level Comparisons

	Reliability	Capacity	Read Latency	Write Latency	Seq Read	Seq Write	Rand Read	Rand Write
RAID-0	0	C*N	D	D	N * S	N * S	N * R	N * R
RAID-1	1	C*N/2	D	D	N/2 * S	N/2 * S	N * R	N/2 * R
RAID-4	1	(N-1) * C	D	2D	(N-1)*S	(N-1)*S	(N-1)*R	R/2
RAID-5	1	(N-1) * C	D	2D	(N-1)*S	(N-1)*S	N * R	N/4 * R

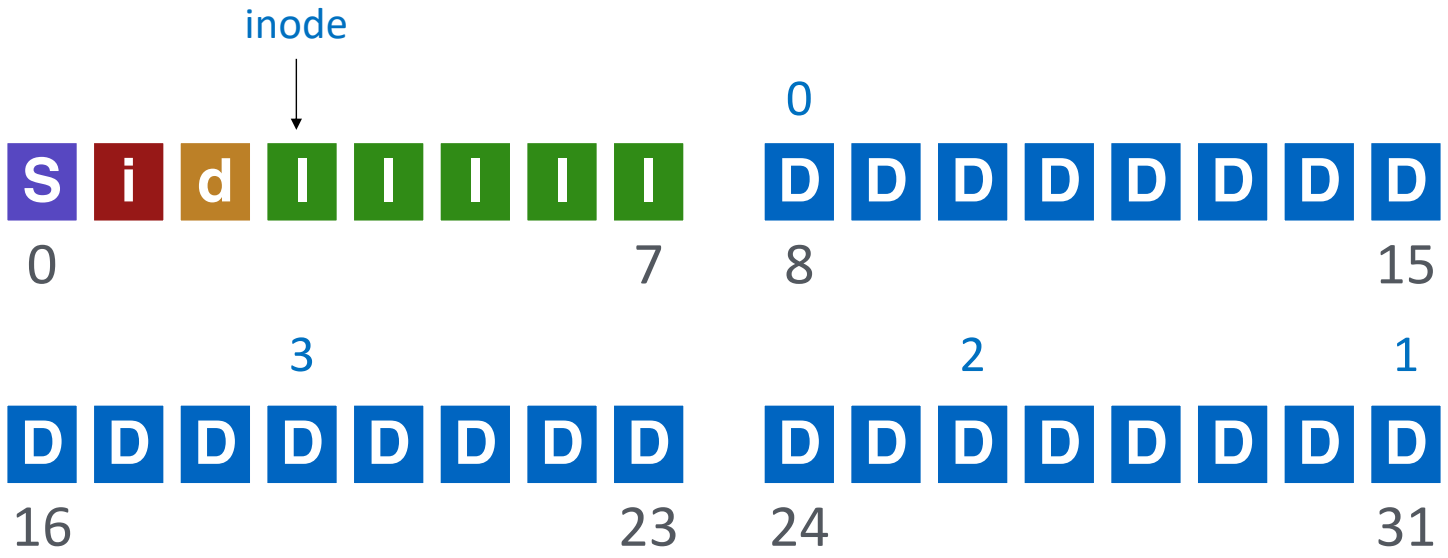
- RAID-5 is strictly better than RAID-4 other than a large write
- RAID-0 is always fastest and has best capacity (but at cost of reliability)
- RAID-5 is better than RAID-1 for sequential workloads
- RAID-1 is better than RAID-5 for random workloads (although with less capacity)
- RAID-6 can handle double-disk failure
- RAID 10: 0+1
- RAID 50: 0+5

5.5 Persistence: Fast File System

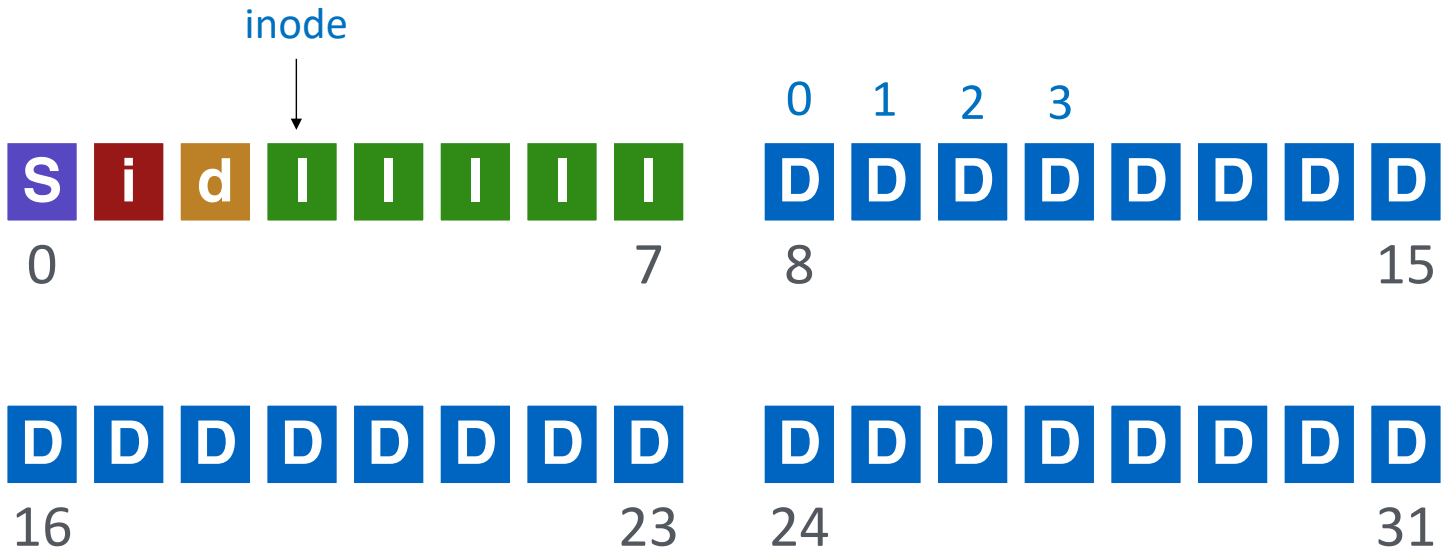
Policy: Choose Inode, Data Blocks



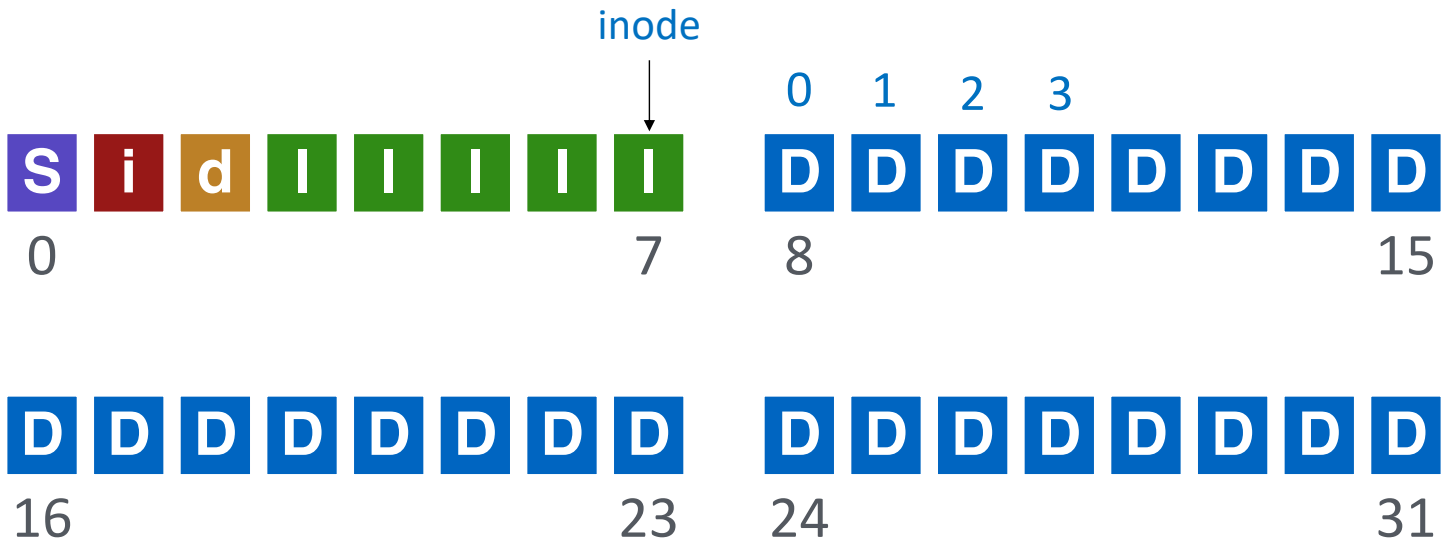
Bad File Layout



Better File Layout

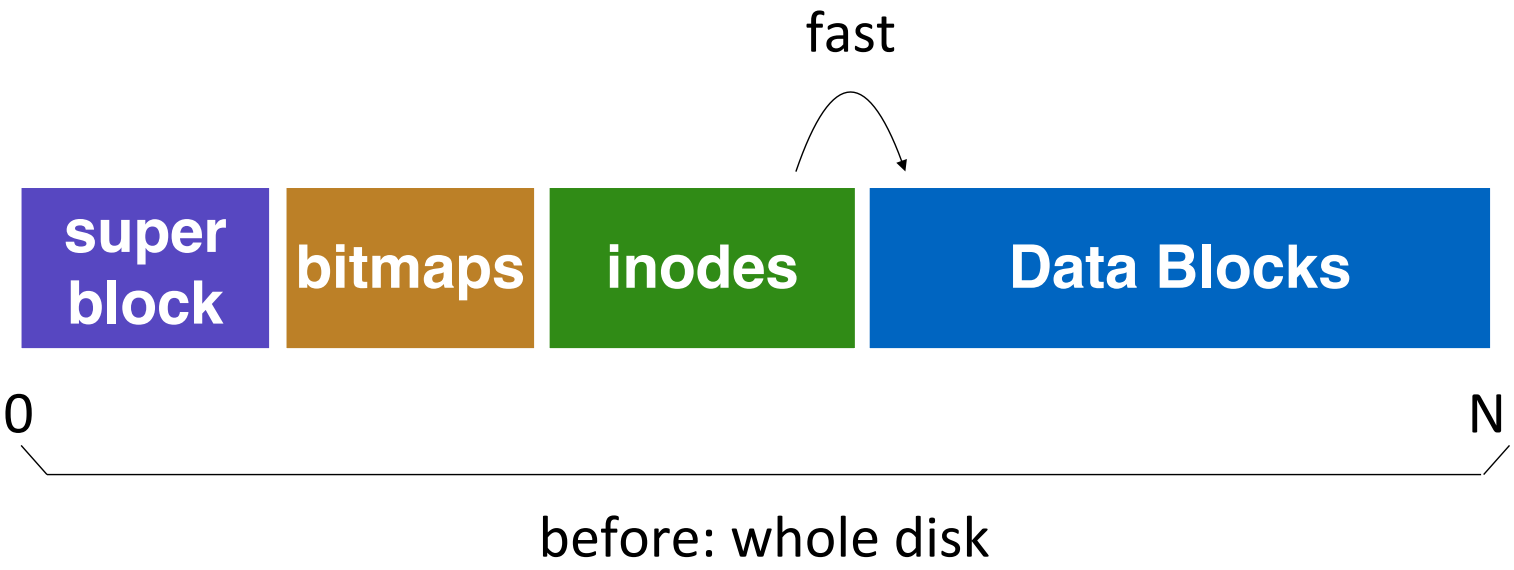


Best File Layout



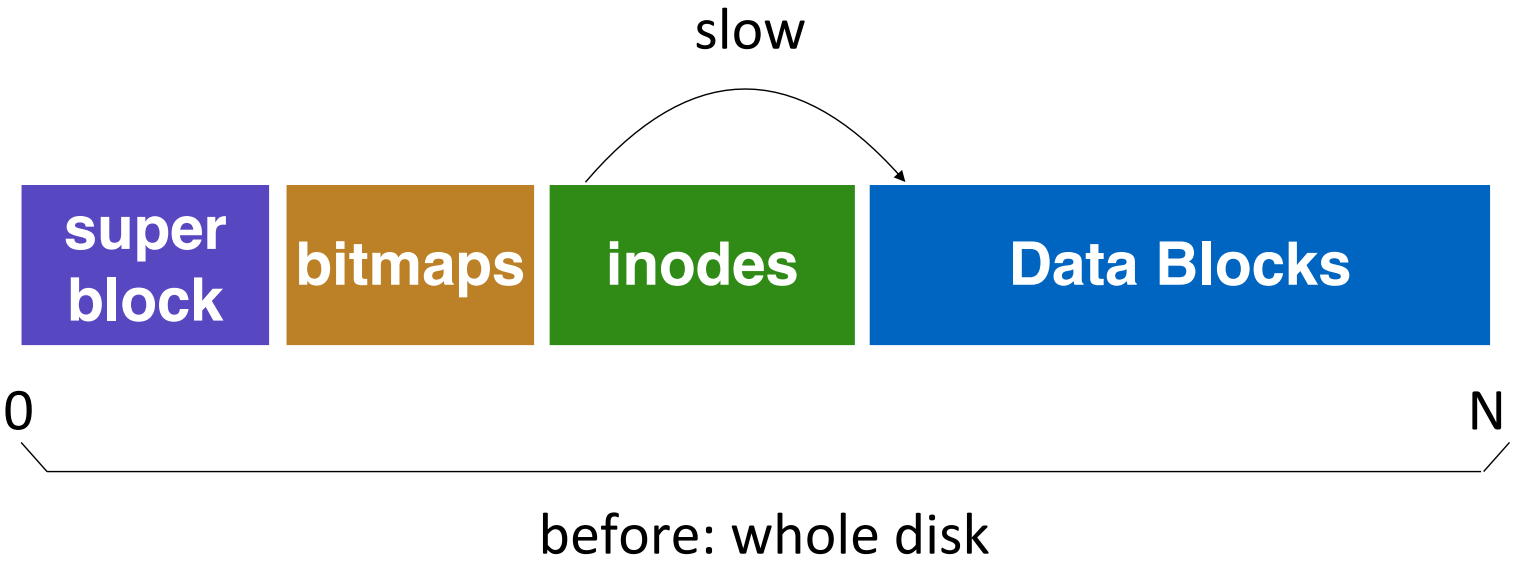
Can’t do this for all files ☹

Placement Technique: Groups



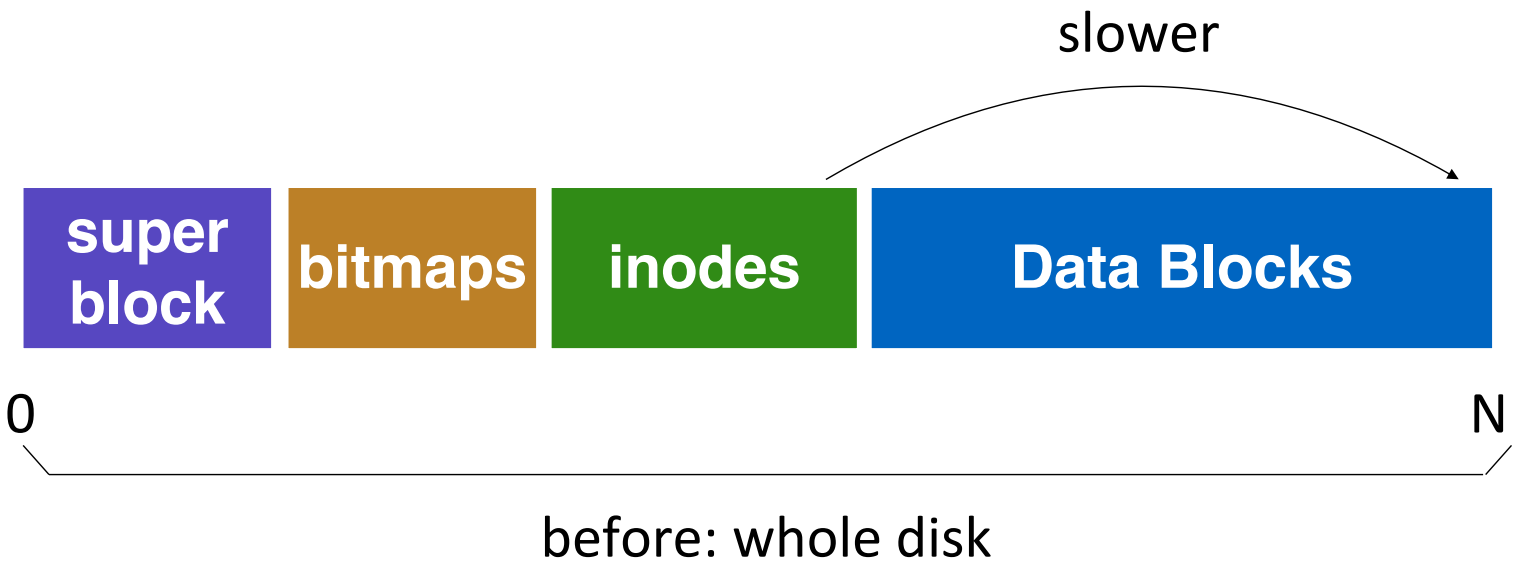
How can we keep the inode close to data?

Placement Technique: Groups



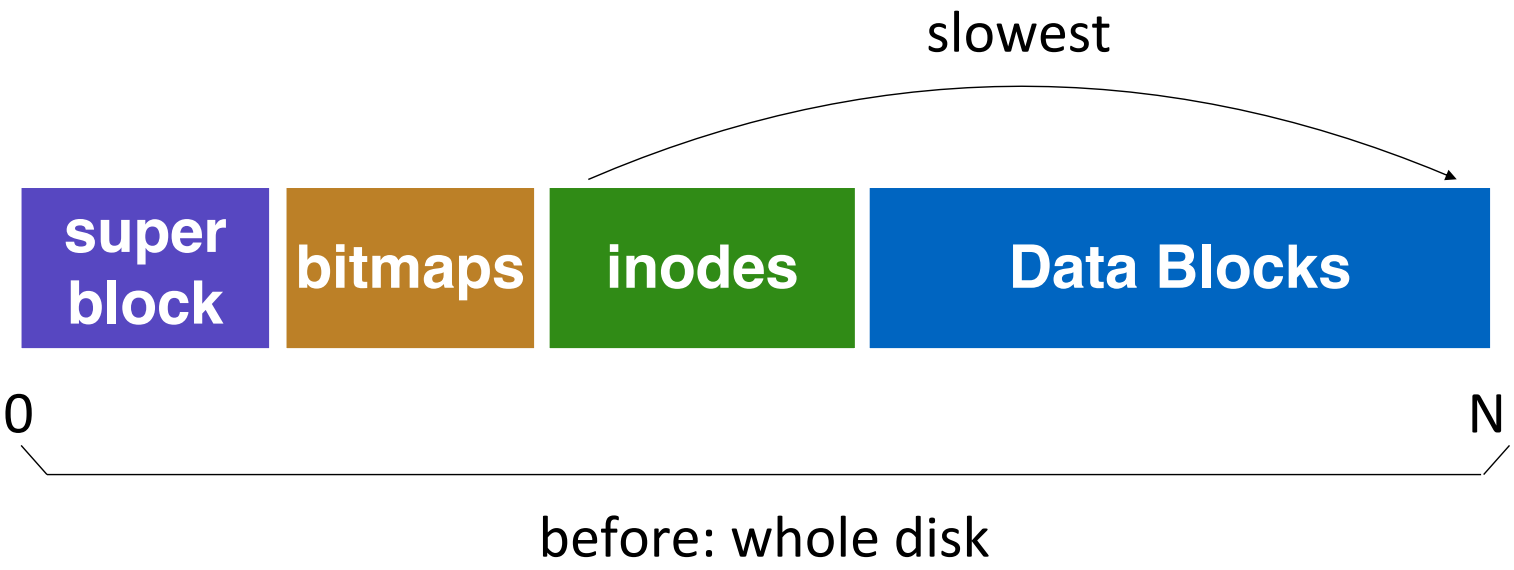
How can we keep the inode close to data?

Placement Technique: Groups



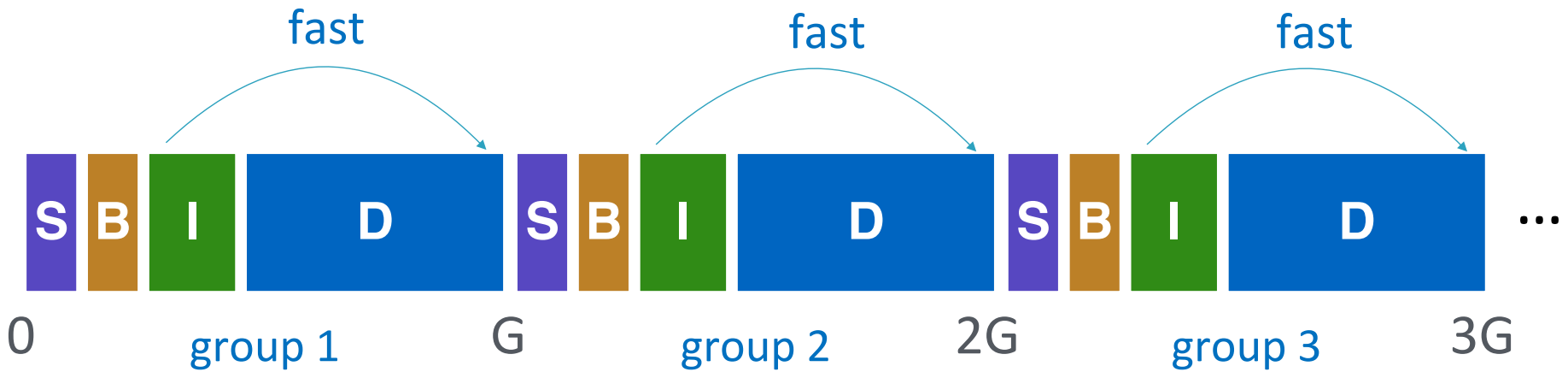
How can we keep the inode close to data?

Placement Technique: Groups



How can we keep the inode close to data?

Technique: Groups



How can we keep the inode close to data?

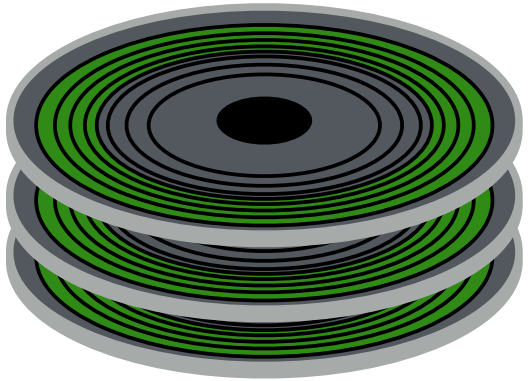
Answer: Use groups across disks;

Try to place inode and data in the same group

Groups

- In FFS (Fast File System), groups were ranges of cylinders
  - called cylinder group
  - Large files are spread across multiple groups.

- In ext2-4, groups are ranges of blocks
  - called block group

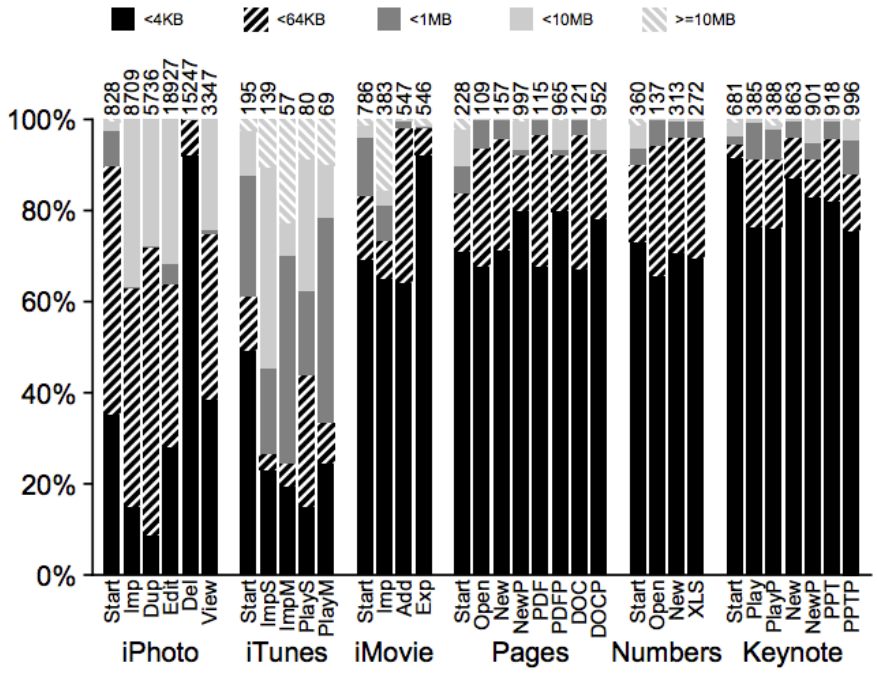


Technique: Larger Blocks

- Observation: Doubling block size for old FS over doubled performance
  - Logically adjacent blocks not physically adjacent
  - Only half as many seeks+rotations now required
- Why more than double the performance?
  - Smaller blocks require more indirect blocks

Why not make blocks huge?

- Most files are very small, even today!
  - Lots of Waste due to internal fragmentation in most of blocks.
- Time vs. Space tradeoffs...



Fragments

- Hybrid – combine best of large blocks and best of small blocks

- Use large block when file is large enough
- Introduce “fragment” for files that use parts of blocks
  - Only tail of file uses fragments

Fragment Example

- Block size = 4096
- Fragment size = 1024

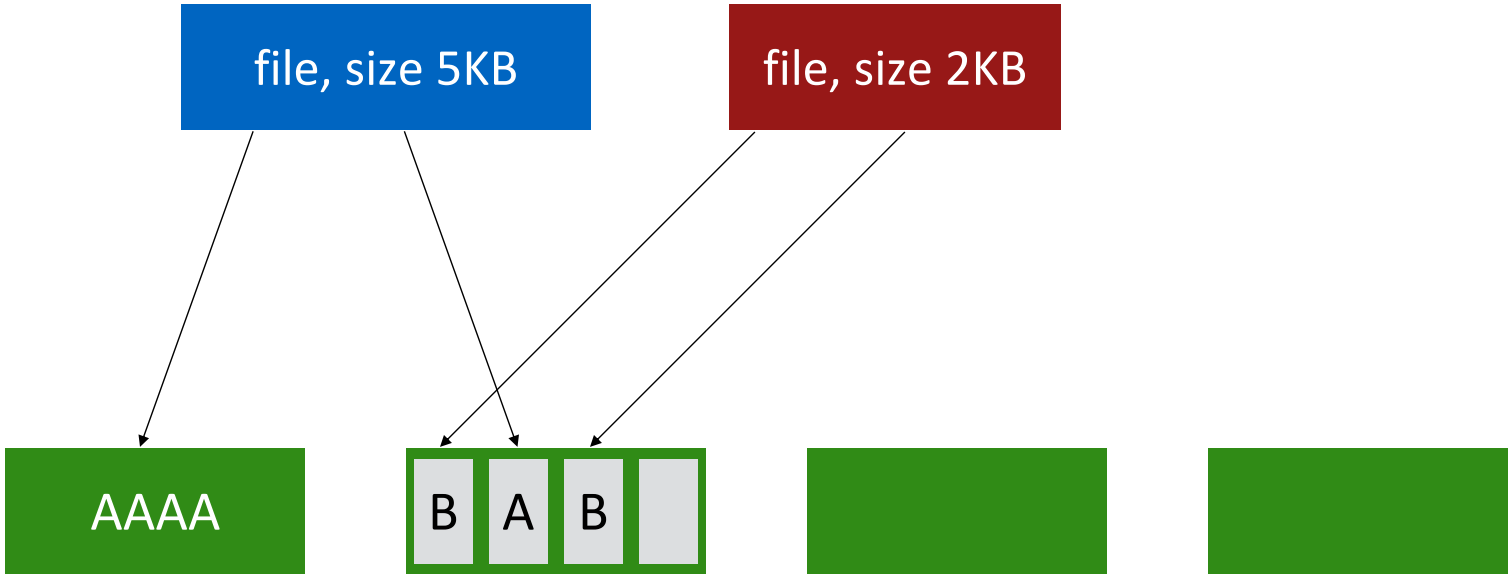
bits: 0000 0000 1111 0010
blk1 blk2 blk3 blk4

Whether addr refers to block or fragment is inferred by file offset

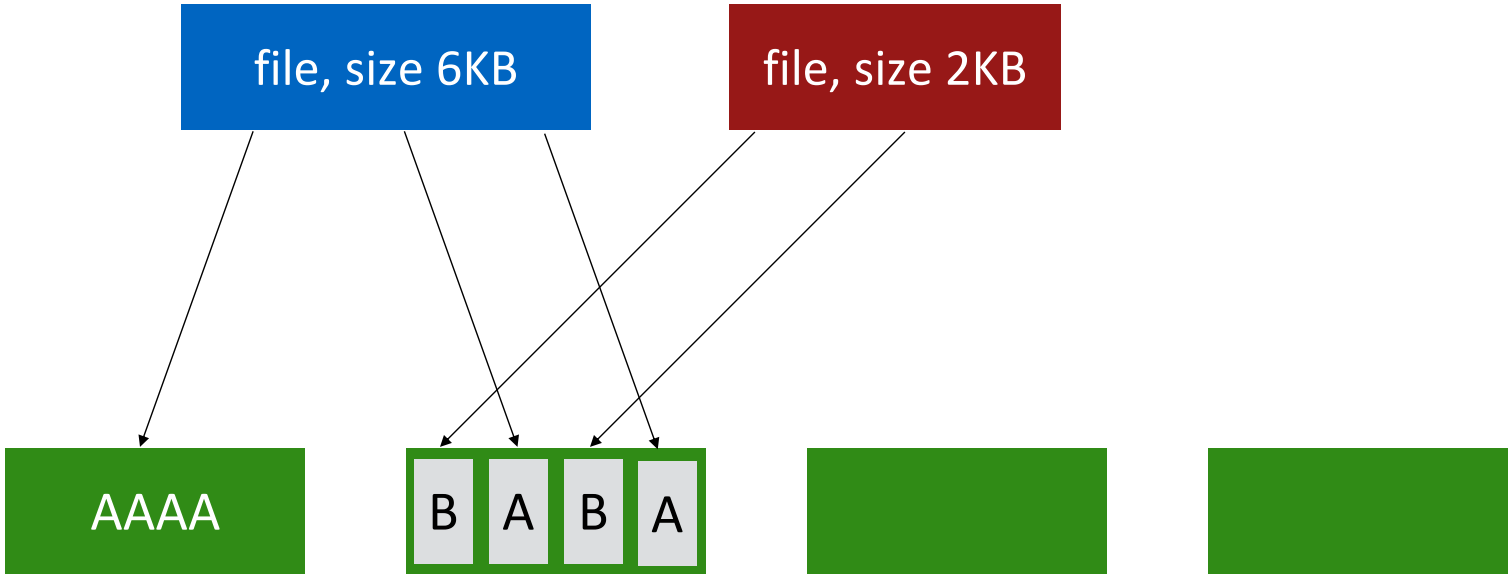
What about when files grow?

Must copy fragments to new block if no room to grow

Fragment Example

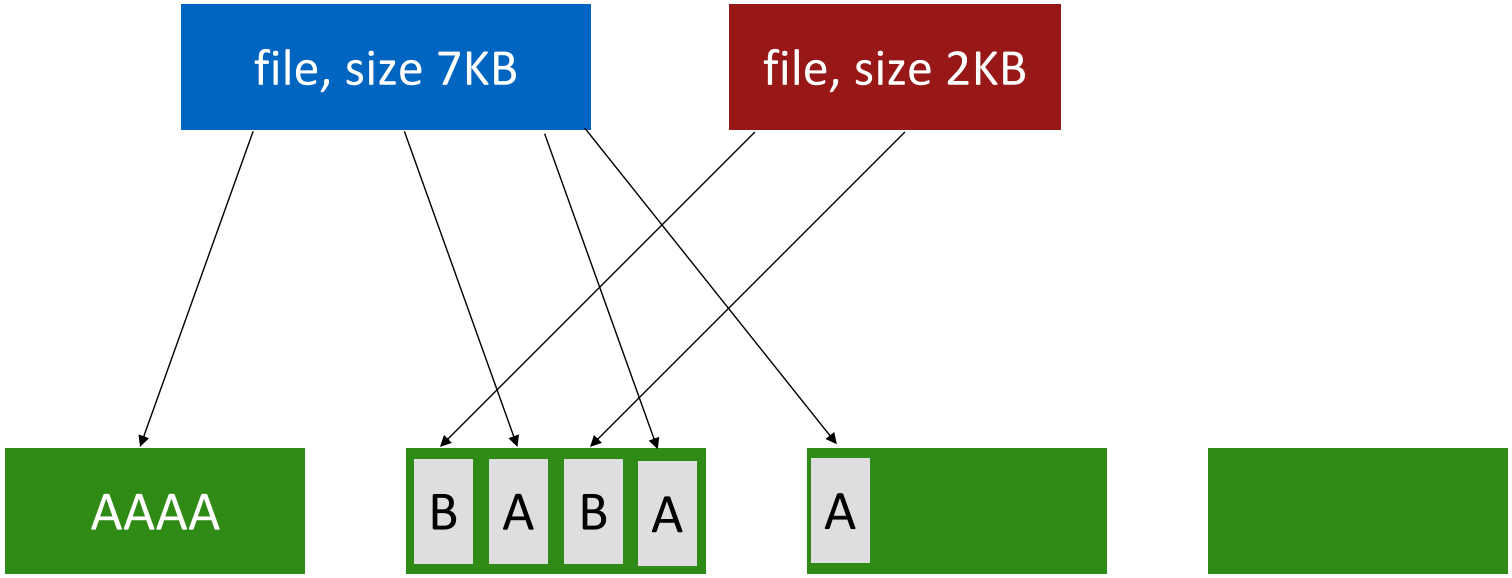


Fragment Example



append A to first file

Fragment Example

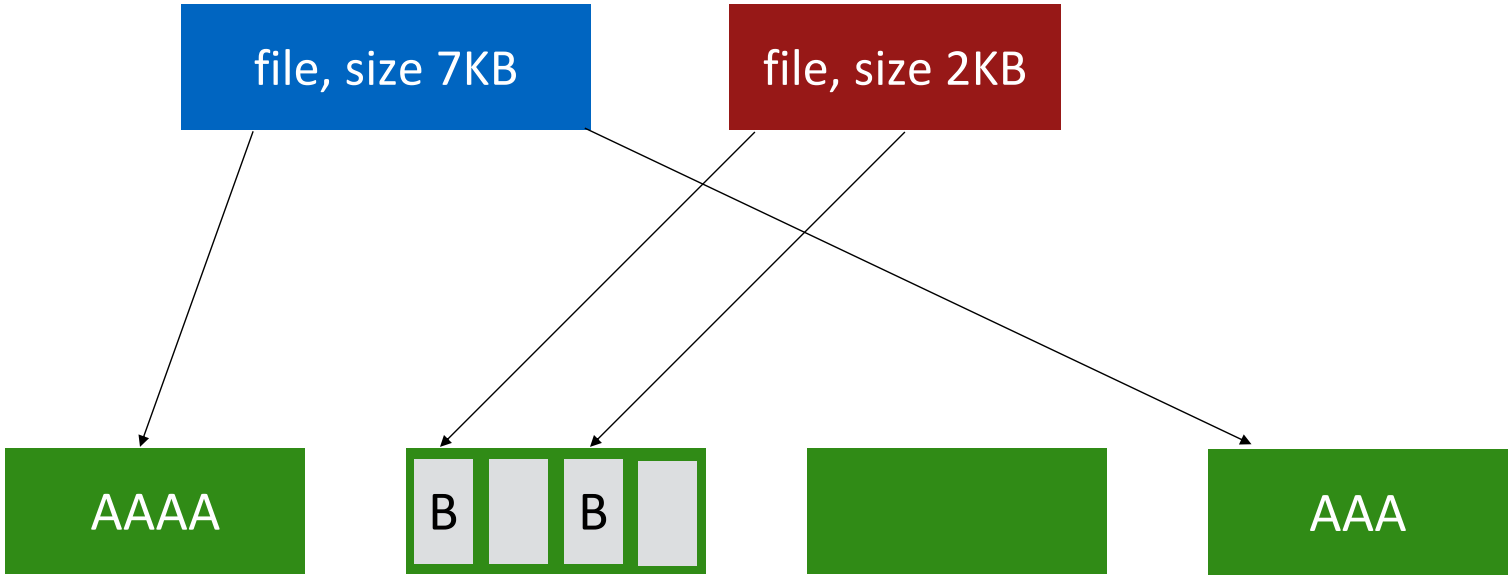


append A to first file

Not allowed to use fragments across multiple blocks!

What can be done instead?

Fragment Example



Copy fragments to a new block

NOTE: Unix disks can be divided into five parts

- Boot block
  - can boot the system by loading from this block
- Superblock
  - specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- i-node area
  - contains descriptors (i-nodes) for each file on the disk; all inodes are the same size; head of freelist is in the superblock
- File contents area
  - fixed-size blocks; head of freelist is in the superblock
- Swap area
  - holds processes that have been swapped out of memory

5. Persistence
Exercises

Exercise #1

- There is a 4GB disk with a simple file system like the one seen in class. Each inode in this file system has 12 direct pointers and 1 indirect pointer. Assuming that the block size is 4KB and an address (pointer) to a block in the disk is 32 bits, what is the maximum file size in this file system? Explain your answer.

Exercise #2 (part I)

- There is a simple file system like the one seen in class. It is implemented on a disk with 16-byte blocks and 20 blocks in total (it is pretty small!).
  - The first block is the superblock. The next 9 have one inode each, and the last 10 are data blocks.
  - The root inode in this file system is the inode 2 (block 3 in the diagram).
  - Assume there aren't any blocks loaded in memory and there is no need to read the superblock.

- Each inode has the following format (each parameter is 4 bytes in size):

  type: 0 means regular file, 1 means directory
  size: number of blocks in file (can be 0, 1, or 2)
  direct pointer: pointer to first block of file (if there is one)
  direct pointer: pointer to second block of file (if there is one)

- A directory has the following format (each parameter is 4 bytes in size):

  name of file
  inode number of file
  name of next file
  inode number of next file

Exercise #2 (part II)

Super Block			Root Inode						
0	1	1	1	0	0	0	0	0	1
1	1	1	1	1	2	1	1	1	2
2	10	18	14	11	12	3	15	19	0
3	0	0	0	17	13	0	0	0	8
Block0	Block1	Block2	Block3	Block4	Block5	Block6	Block7	Block8	Block9
foo	3	7	hi	a	10	11	i	cs	0
3	4	8	10	0	goo	bar	luv	6	0
bar	5	9	you	b	11	oof	cs	537	0
4	6	10	12	1	goo	da	537	7	0
Block10	Block11	Block12	Block13	Block14	Block15	Block16	Block17	Block18	Block19

Exercise #2 (part III)

- To read the contents of the root directory, which blocks do you need to read?
- Which files and directories are in the root directory? List the names of each file/directory as well as their type (e.g., file or directory).
- How many files are in the system (that can be reached starting at the root)?
- What is the biggest file in the file system?
- What blocks are free in this file system? (that is, which inodes/data blocks are not in use?)

Exercise #3

- Assume that you have a RAID-4 with 4 disks + one parity disk (5 in total), and the following block structure:

0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

...and so on...

- Obtain the accesses that must be done to the RAID blocks. Put in parentheses the accesses that can be done in parallel.
  - i.e.: (r4, r8), w0, (w3, w8) means that we first read in parallel blocks 4 and 8, and then write in parallel blocks 3 and 8.

(a) Assuming that we want to read blocks 0, 5, 10 and 15, what are the accesses to the RAID blocks?

b) If we want to read blocks 0, 4, 8 and 12, what are the accesses to the RAID blocks in this case?

c) What are the accesses to the RAID blocks if we want to write blocks 5 and 10?

Exercise #4

- We have a disk with FIFO scheduling and just one track. The rotational delay of the disk is R (one full round), there is no seek time (just one track) and the transfer time is so quick, that we consider it irrelevant.
  - a) How long does it take (approximately) to fulfill three requests to three different blocks in the worst case?
  - b) If the disk uses Shortest-Access-Time-First (SATF) scheduling, how long does it take to fulfill the three requests in the worst case?
  - c) If the disk now has three tracks, and seek time S to access contiguous tracks (and 2S to go from the innermost track to the outermost one), with FIFO scheduling what is the worst-case access time for accessing three different blocks?
  - d) What happens in the previous question if we have SATF scheduling?