Georgia Tech Computer Science

CS2200
Systems and Networks
Spring 2022

Lecture 27:
Disk Scheduling and File Systems

Alexandros (Alex) Daglis
School of Computer Science
Georgia Institute of Technology
adaglis@gatech.edu

Lecture slides adapted from Bill Leahy, Charles Lively, Umakishore Ramachandran of Georgia Tech

Agenda

- Disk capacity, performance, and scheduling
  - Chapter 10
- Last cs2200 topic: File Systems
  - Chapter 11

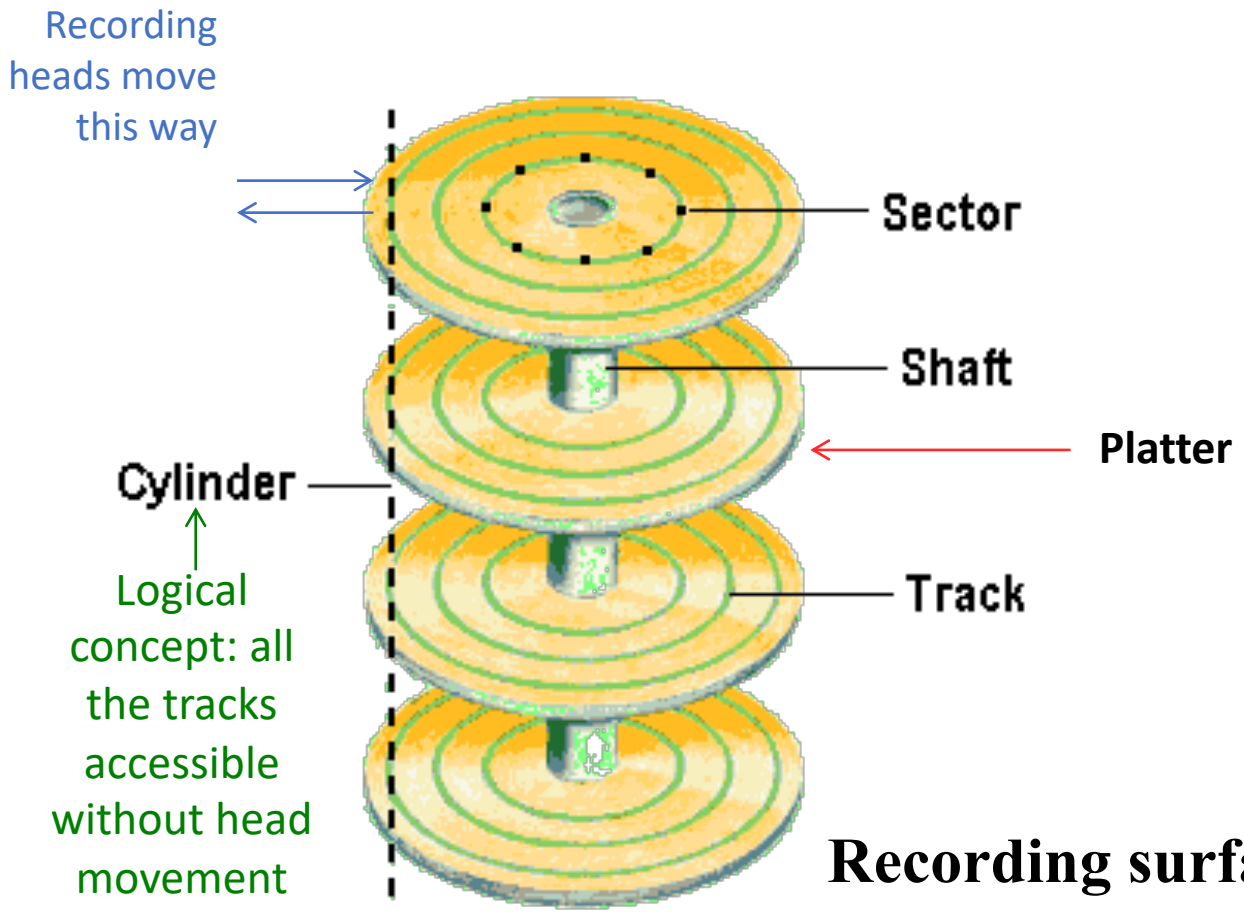
CIOS is open

- Your feedback helps me improve cs2200!
  - Your future peers will be thankful
  - What worked well? Suggestions for improvement?

- CIOS participation incentive:

  Entire class gets **1% bonus** to total grade if we reach **95% participation**

More disk drive terminology



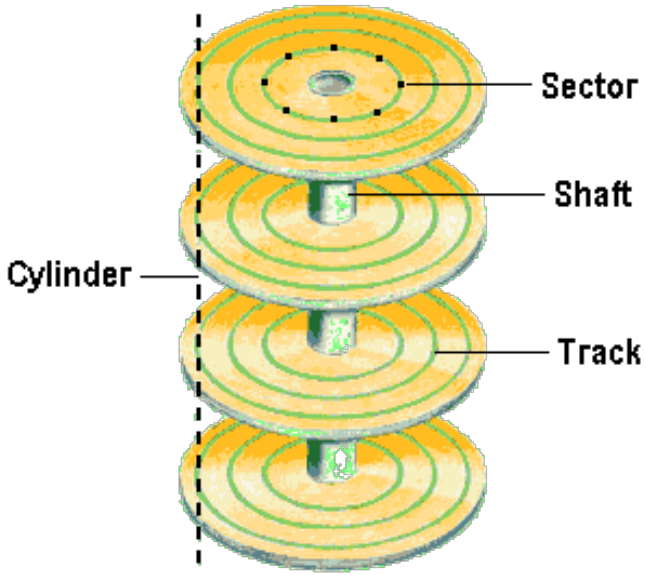
Capacity Metrics

Let,

p – number of platters,
n – number of surfaces per platter (1 or 2),
t – number of tracks per surface,
s – number of sectors per track,
b – number of bytes per sector,

The total capacity of the disk:

Capacity = (p * n * t * s* b) bytes



Example

b

s

n

Suppose a disk drive has 512 bytes per sector, 12 sectors per track, 100 tracks per surface, 2 surfaces per platter, and 6 platters.

t

What is the total capacity of such a drive in bytes?

How many cylinders does the device have?

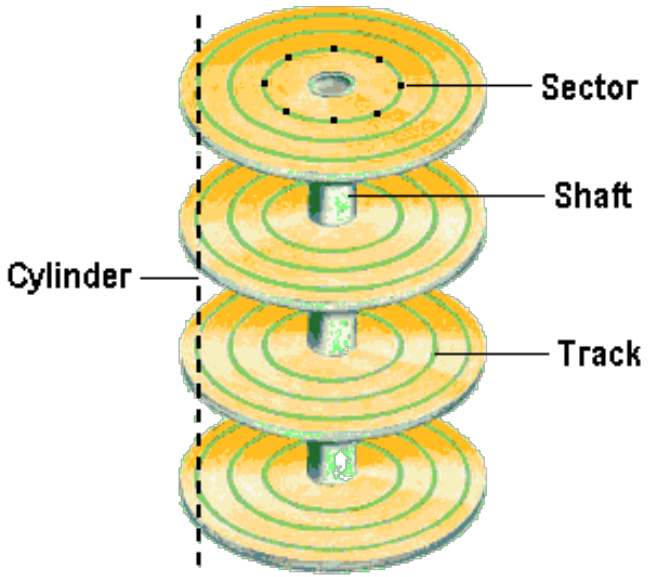
p

Capacity = ($p * n * t * s * b$) bytes

= 6 * 2 * 100 * 12 * 512 bytes

= 7200 KB (KB = 2^{10} bytes)

And it has t cylinders, or 100



Transfer metrics

Let,

r – rotational speed of the disk in RPM,
s – number of sectors per track,
b – number of bytes per sector,

Time for one revolution = $60/r$ seconds

Amount of data read in one revolution = $s * b$ bytes

The data transfer rate of the disk:

(Amount of data in track) / (time for one revolution)

$= (s * b)/(60/r)$

Data transfer rate

$= (s * b * r) / 60$ bytes/second

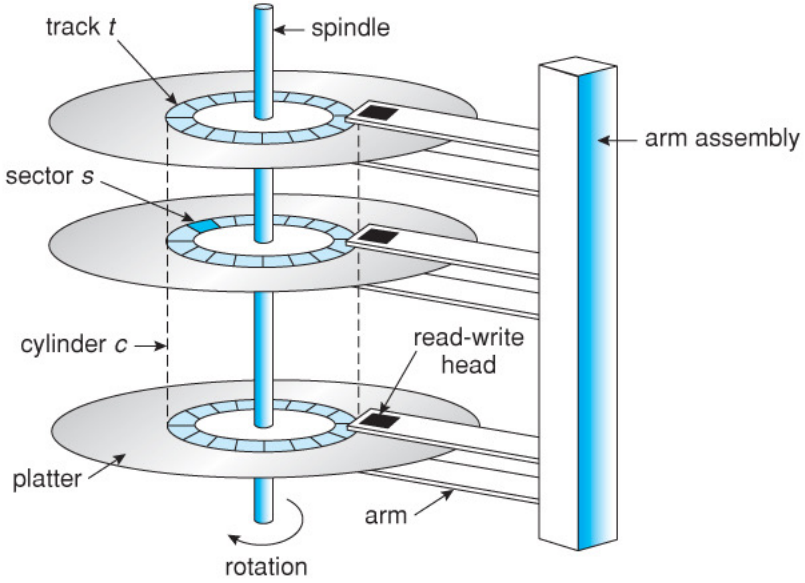
Example

Given the following specifications for a disk drive:

- b 512 bytes per sector
- s 400 sectors per track

  6000 tracks per surface

  3 platters
- r Rotational speed 15000 RPM



What is the transfer rate of the disk?

Data transfer rate = (s * b * r) / 60 bytes/second

= (400 * 512 * 15000) /60

= 51,200,000 bytes/second

Only one head transfers data at a time!

Which of these attributes of a hard disk drive

is conceptual rather than physical?

A. I just want the participation credit
B. p – number of platters
C. n – number of surfaces per platter (1 or 2)
D. t – number of tracks per surface
E. s – number of sectors per track
F. b – number of bytes per sector
G. c – number of cylinders

Early 60's disk pack: 5MB

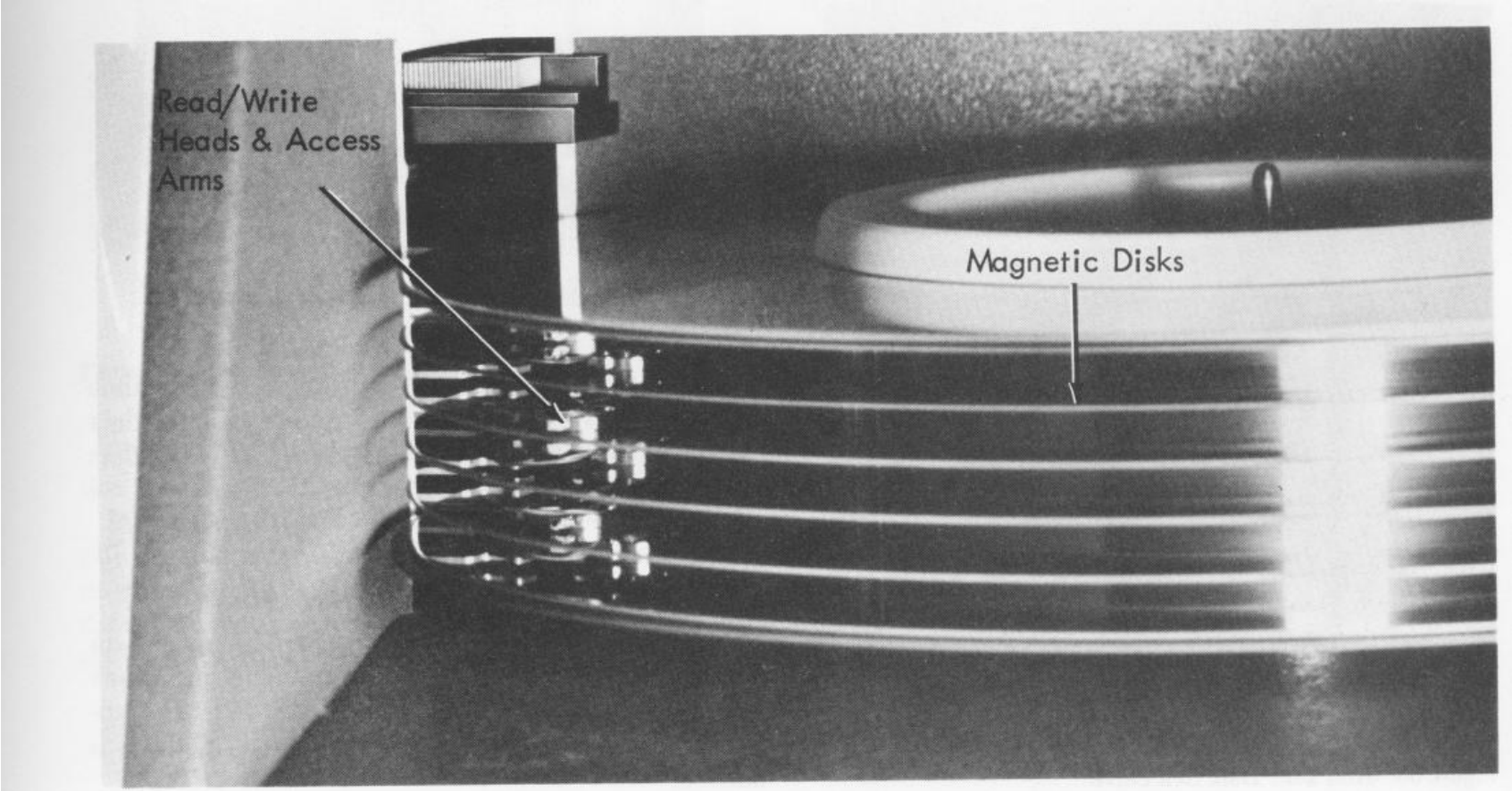
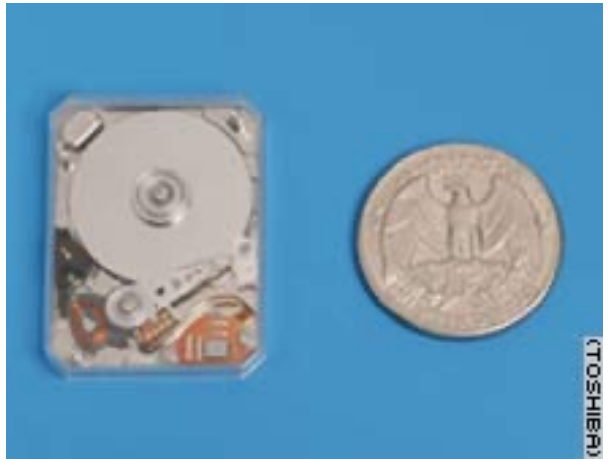


Figure 54. IBM 2311 Disk Storage Drive showing mounted disk pack and access arms

When computers were big…

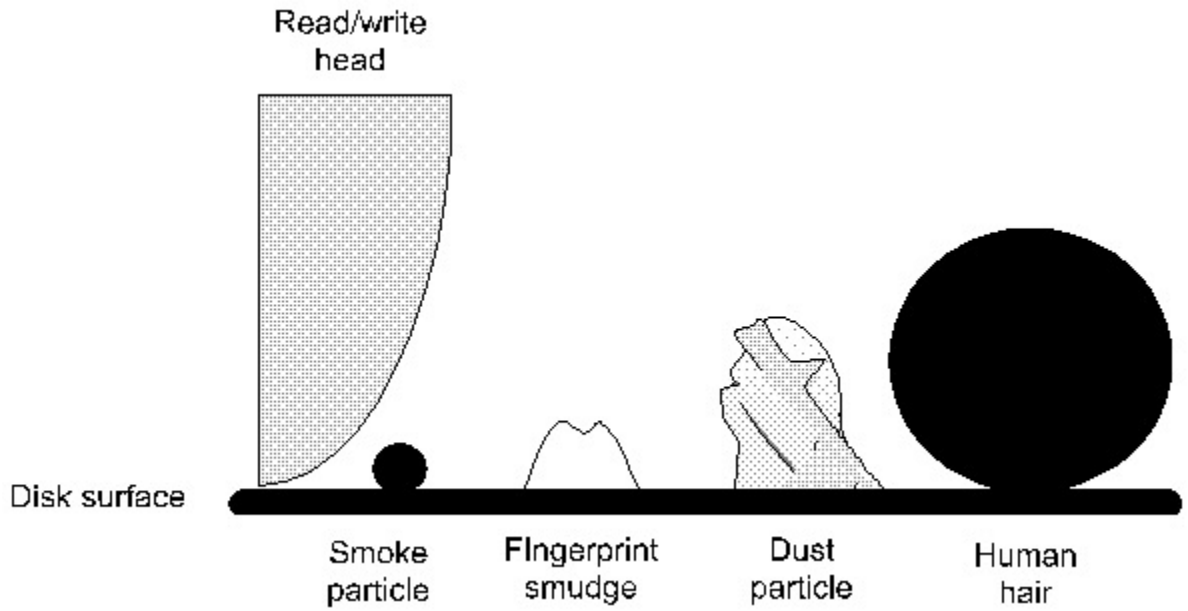
Welcome to modern times



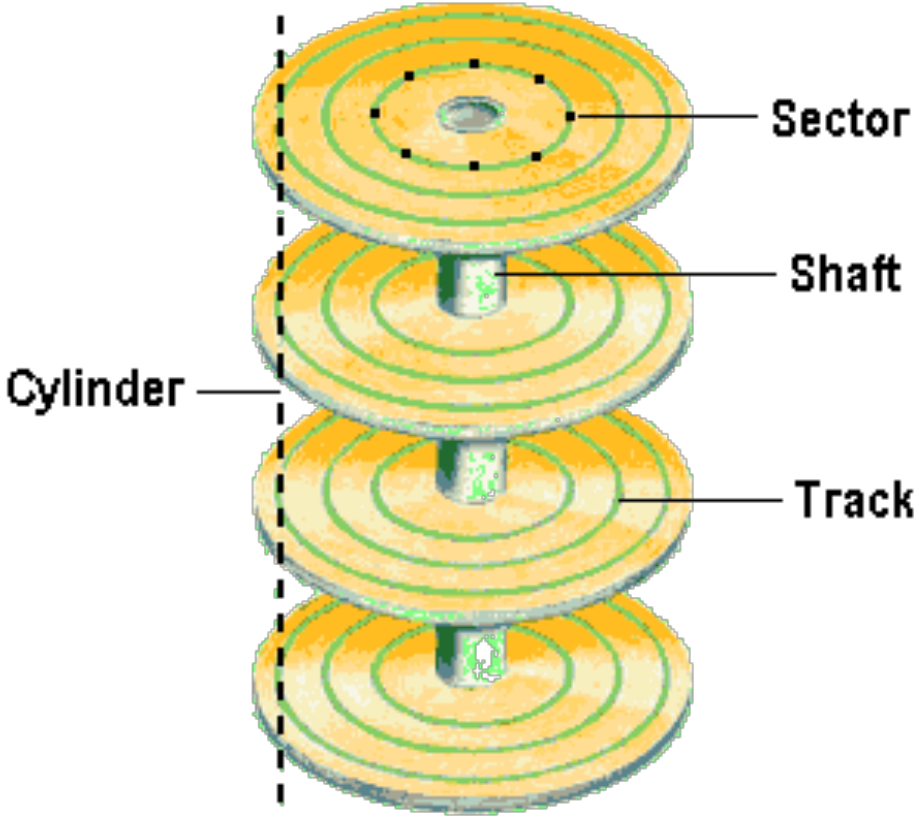
4 GB drive

Why are disk drives sealed?

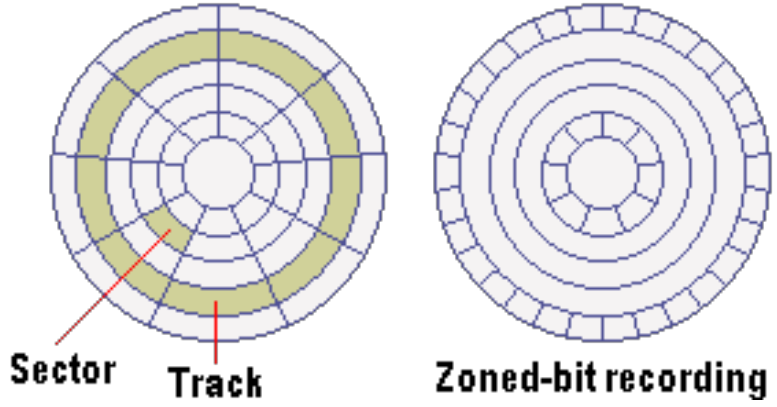
Since the early 60s, read/write heads are aerodynamically shaped so they fly on a very thin cushion of air. Obstacles are not well tolerated.



Zoned recording

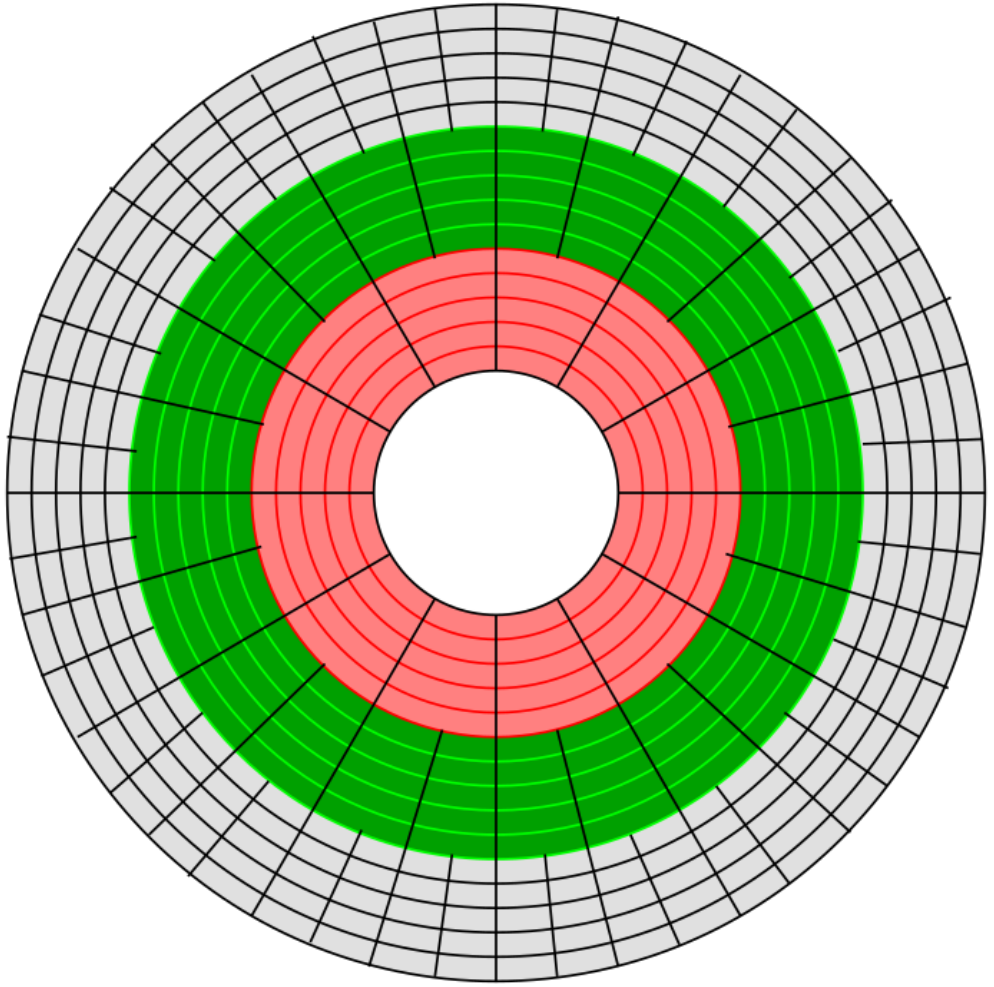


- The outside track of a platter has a larger surface and moves faster than an inside track!
- This allows the outside tracks to hold more data and transfer faster
- Zoned recording (of which there are many schemes) assigns more sectors to the outer tracks



Example of Zoned Recording

- Physical layout of sectors in a zone-bit disc
- As distance from the center increases, the number of sectors in a given angle increases from one (red) to two (green) to four (grey)
- The red zone has 12 sectors/track; green has 24; grey has 48.



https://en.wikipedia.org/wiki/Zone_bit_recording#/media/File:DiskStructure.svg

With zoned recording,

z – number of zones,

t_{zi} – number of tracks at zone z_i,

s_{zi} – number of sectors per track at zone z_i,

The total capacity of the disk with zoned recording:

$$\text{Capacity} = (p * n * (\sum (t_{zi} * s_{zi}), \text{ for } 1 \leq i \leq z) * b) \text{ bytes}$$

Total access latency

Let,

a – average seek time in seconds

r – rotational speed of the disk in RPM

s – number of sectors per track

Rotational latency = $60/r$ seconds

Average rotational latency = $(60 / (r * 2))$ seconds

Sector read time = rotational latency / number of sectors per track

Sector read time = $(60 / (r * s))$ seconds

Time to get to the desired track

= Average seek time

= a seconds

Time to get the head over the desired sector

= Average rotational latency

= (60 / (r * 2)) seconds

Time to read a sector

= Sector read time

= (60 / (r * s)) seconds

Time to read a random sector on the disk

= Time to get to the desired track +
Time to get the head over the desired track +
Time to read a sector

= a + (60 / (r * 2)) + (60 / (r * s))
seconds

Example

Given the following specifications for a disk drive:

256 bytes per sector
12 sectors per track
20 tracks per surface
3 platters
Average seek time of 20 ms
Rotational speed 3600 RPM

What would be the time to read 6 contiguous sectors from the same track?

1 seek + 1 sector location + 6 sector transfers = 20 + 60 / (3600 * 2) + 6 * (60 / (3600*12)) = 36.7ms

What would be the time to read 6 sectors at random?

(1 seek + 1 sector location + 1 sector transfers) * 6 = 29.7 * 6 = 178.2ms

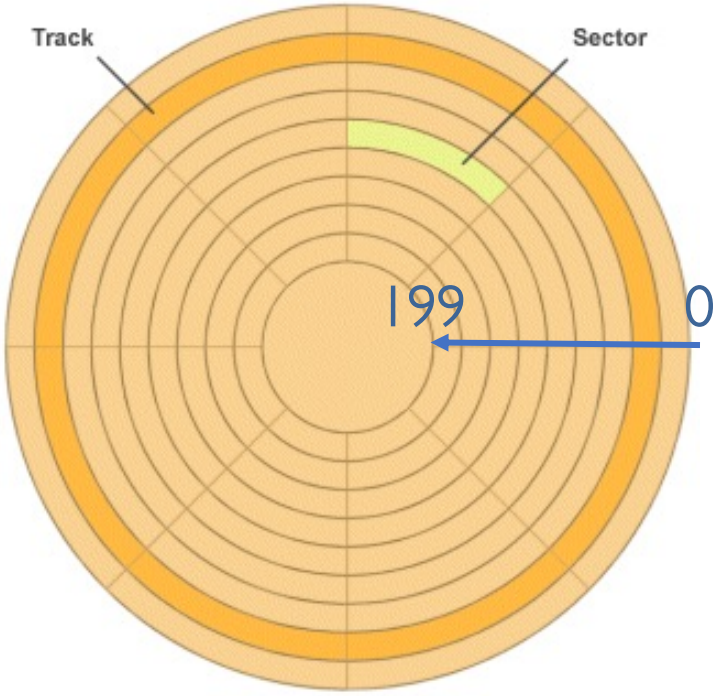
Disk scheduling

- Ordering of disk requests is quite important for I/O performance just as job scheduling is important for CPU performance
- Seeks and rotational delay (on the order of 5ms each) are mechanical and hence huge compared to the speeds of memory (100 ns) and CPU instructions (1 ns). That's 5×10^{-3} compared with 1×10^{-9}, or more than 6 orders of magnitude·
- However, as disk controllers have become more capable, I/O requests are queued for them by the OS as soon as they come in
- Disk controllers may have tens of pending I/O operations queued and perform their own scheduling on them

Disk scheduling

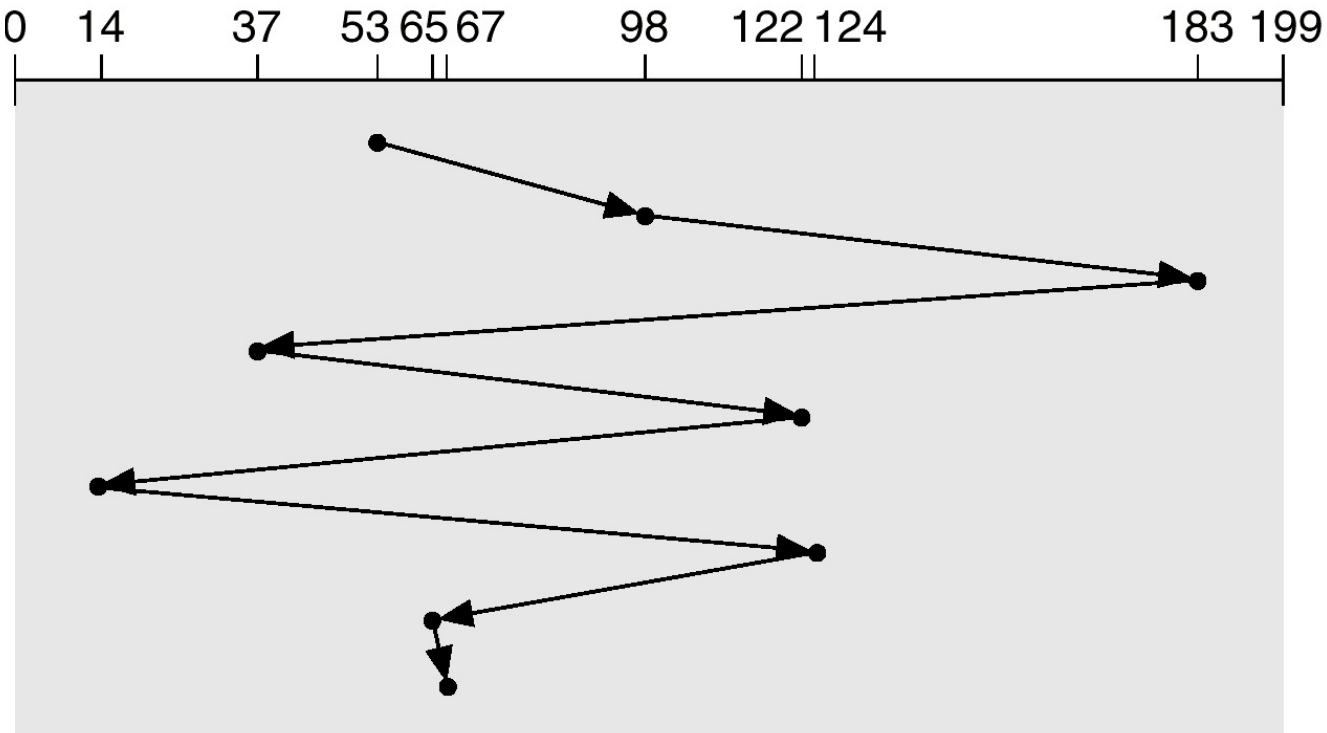
- We're going to bypass discussing the performance metrics for disk scheduling, but know there are many algorithms similar to CPU scheduling
- Some prominent algorithms
  - FCFS – self explanatory
  - SSTF – shortest seek time first
  - LOOK – keep going the same direction until no more requests in that direction (more like a real elevator)
  - SCAN – "elevator algorithm": like LOOK but always go to the top and bottom floors on each trip
  - C-SCAN – always go "up" and don't service any requests on the down cycle
  - C-LOOK – like LOOK but only service requests in one direction

FCFS

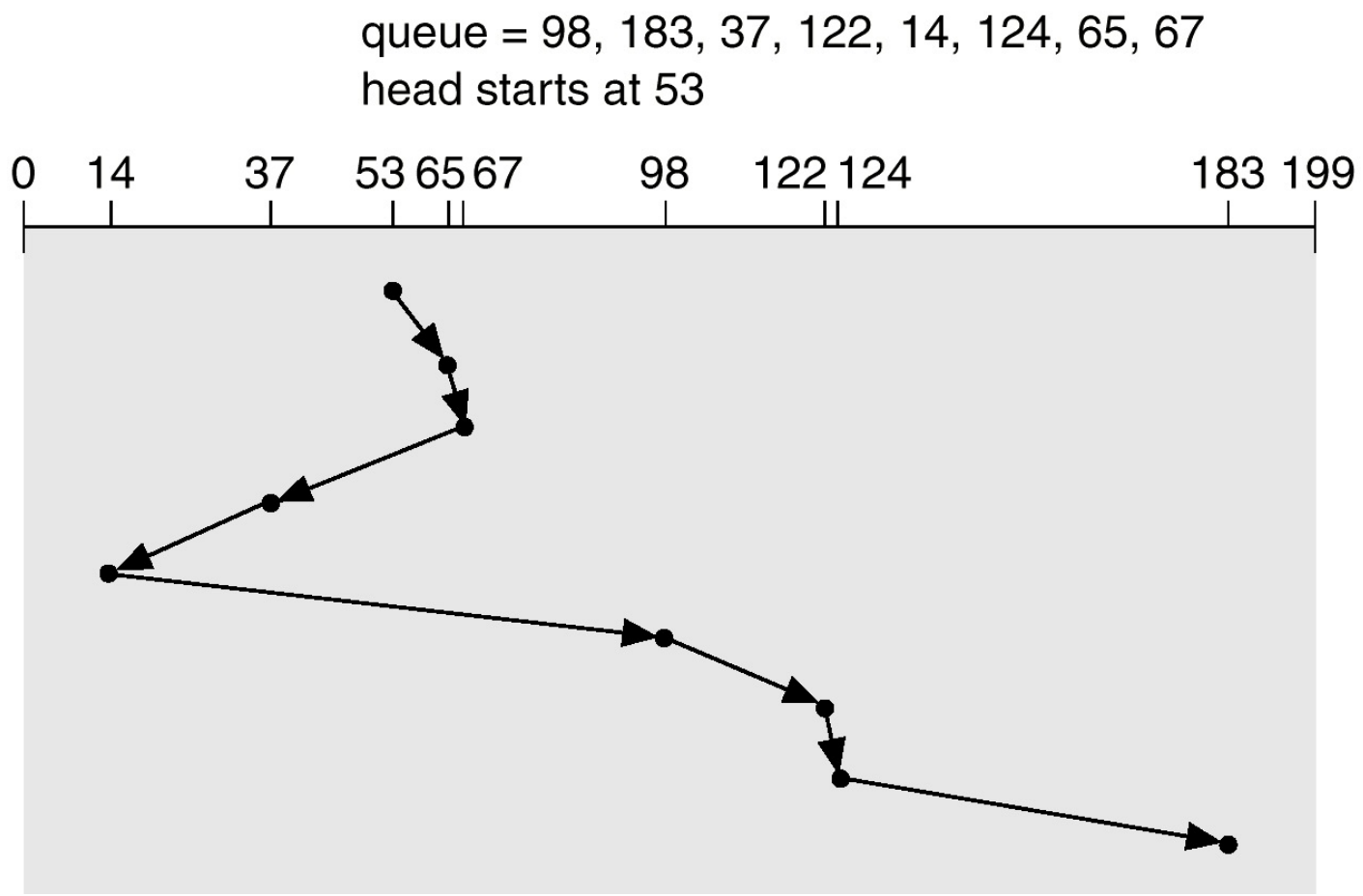


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53

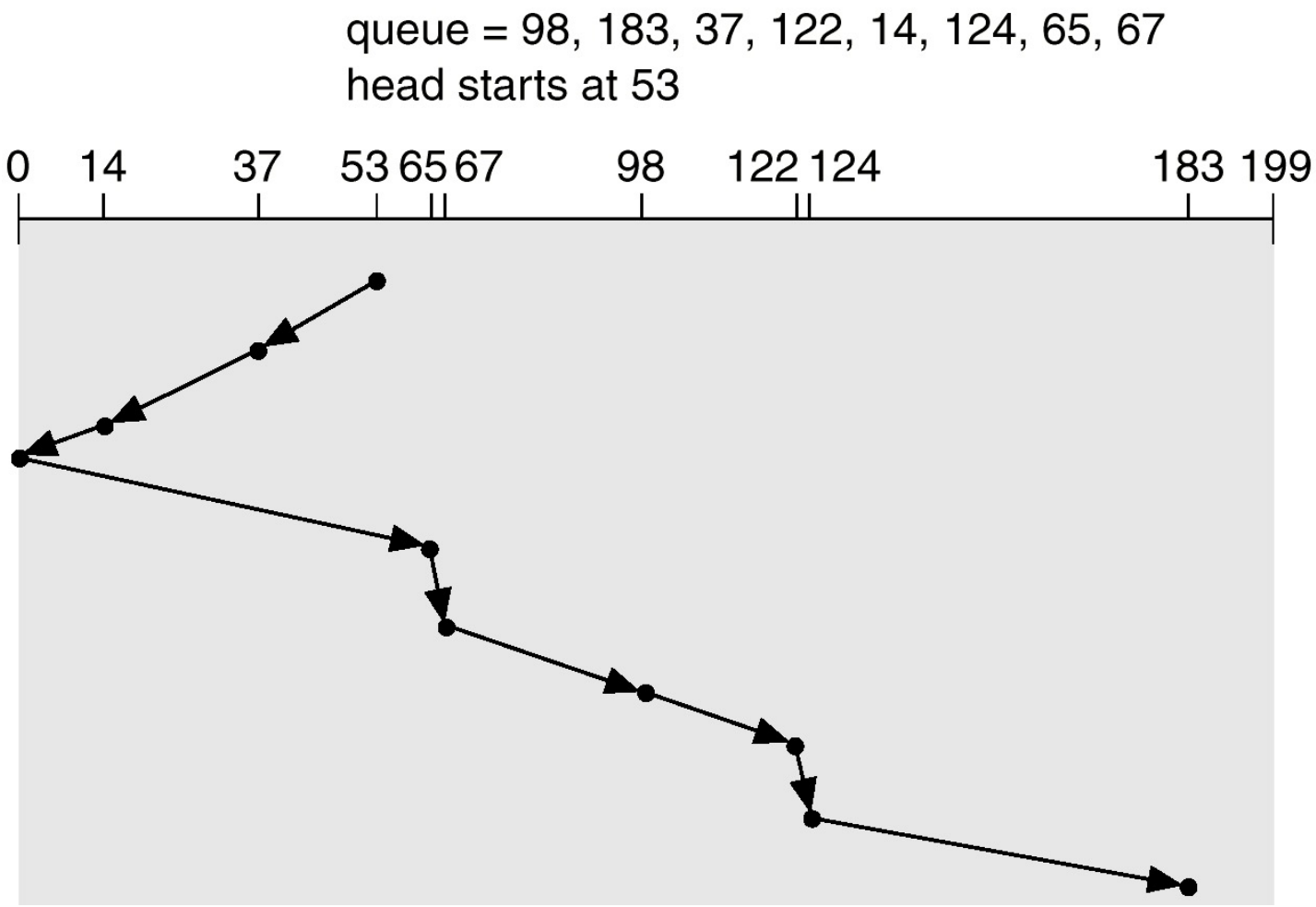
Tracks:



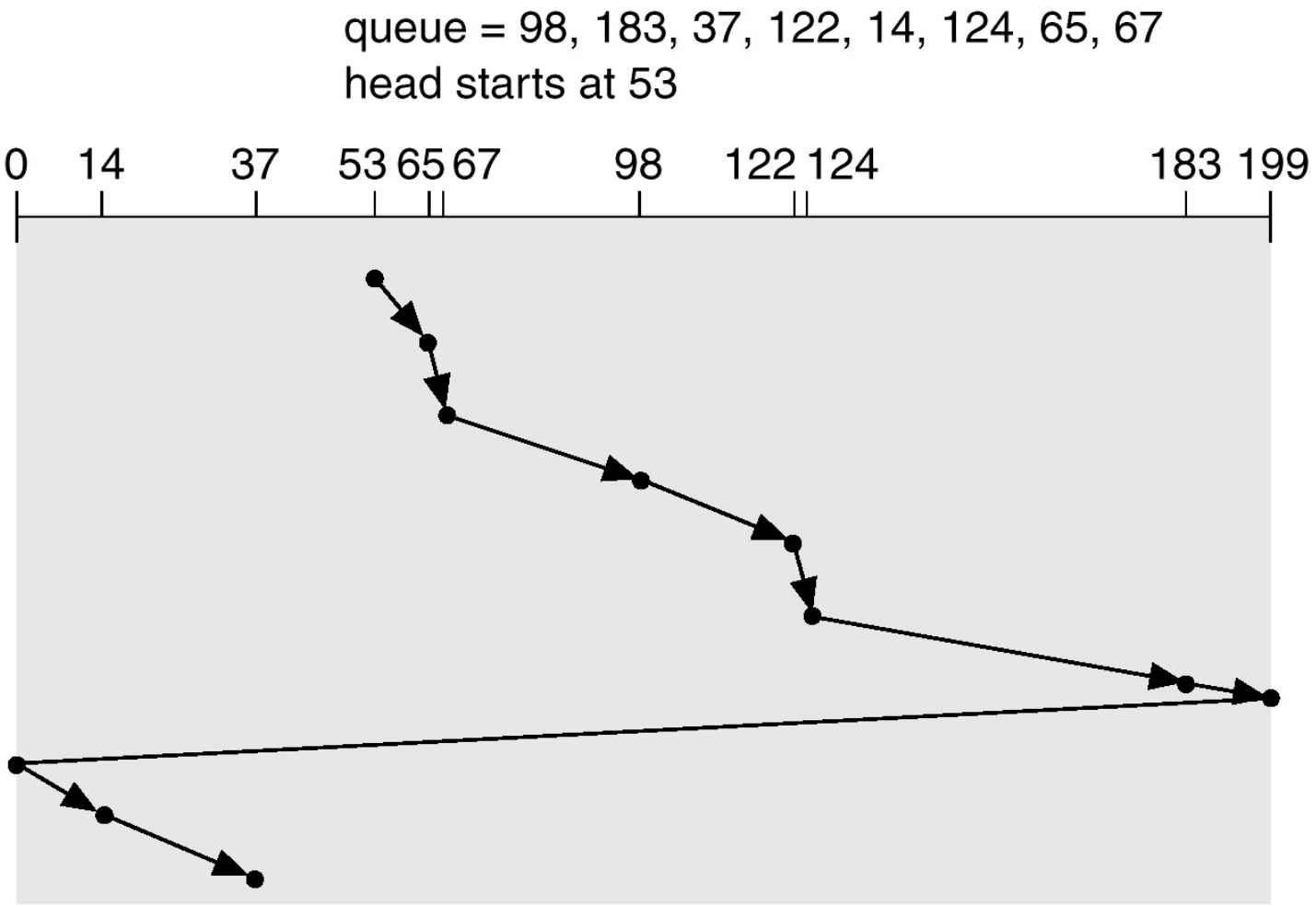
SSTF


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

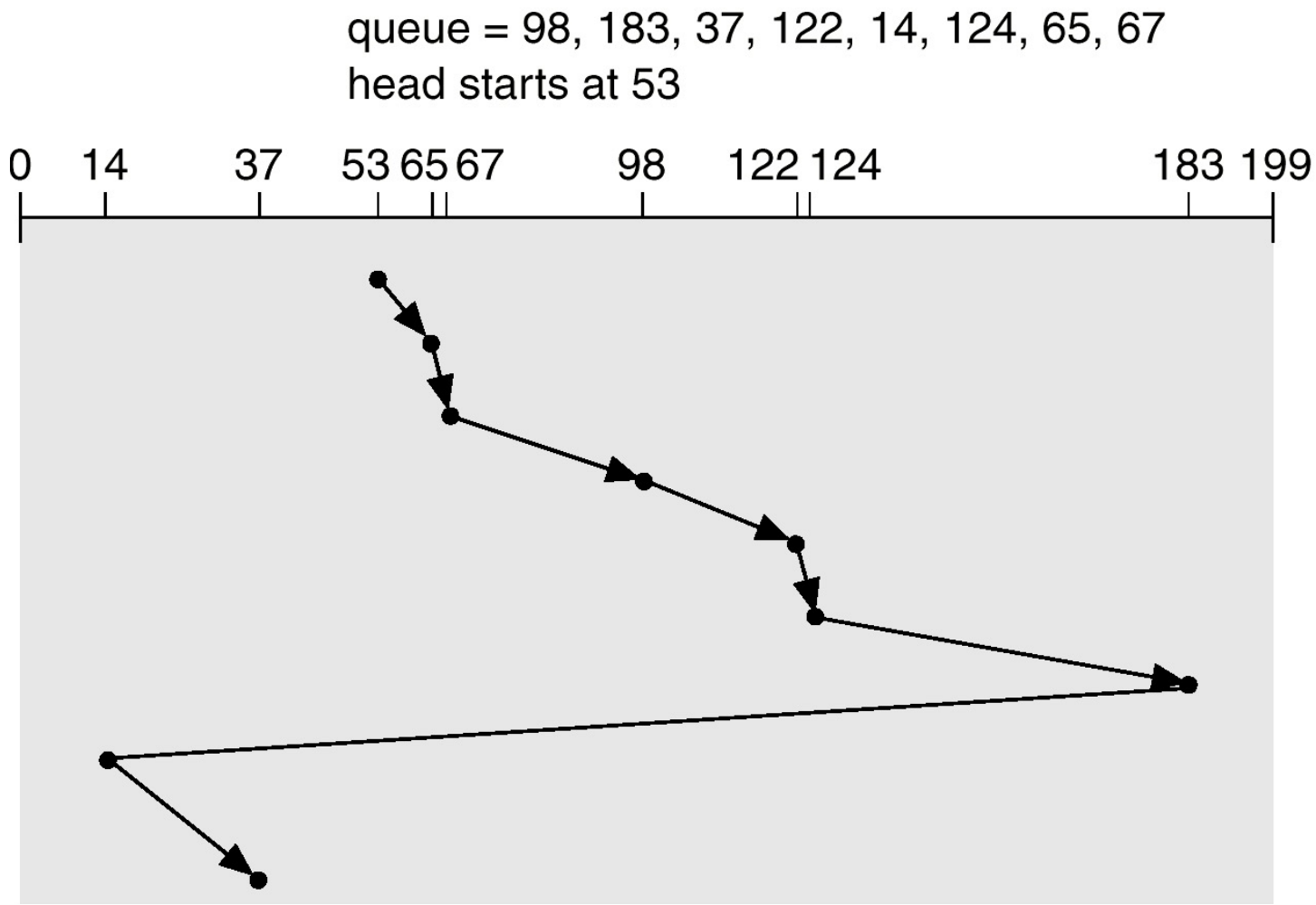
SCAN


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

C-SCAN


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

C-LOOK


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

Session ID: dag2200

Question

Which disk scheduling algorithm isn't called the "elevator algorithm" but works more like a modern elevator?

A. I just want the participation credit
B. SCAN
C. LOOK
D. C-SCAN
E. C-LOOK

Solid-state storage devices (SSDs)

- These have rapidly become effective replacements for small & fast storage applications
- They can act like HDDs and are faster but more expensive

	HDD	SSD
Capacity	1 - 18TB	0.25 - 15TB
IOPS (I/O per second)	400	50,000-1,000,000
Max transfer rate	Up to 500 MB/sec	Up to 7 GB/sec
Seek time	8 ms	0.1 ms
Energy use	6-16 W	2-5 W
Cost ($/GB)	0.02	0.10

Agenda

- ~~Disk capacity, performance, and scheduling~~
  - ~~Chapter 10~~

- File Systems
  - Chapter 11
  - Last cs2200 topic 🙌🏻

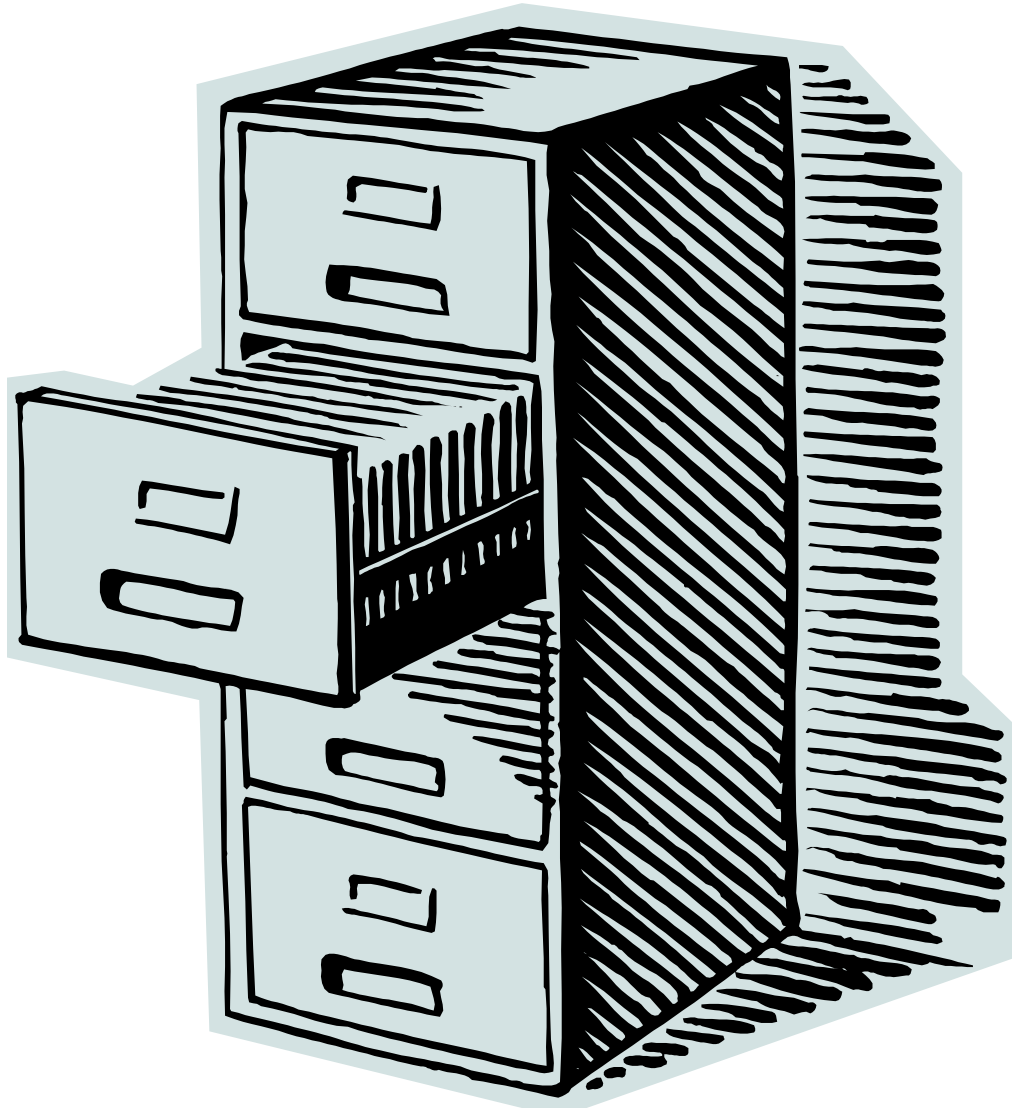
In this chapter

- Disk space allocation policies
  - Figures of merit
  - Unix file system
- Similarities to memory management

	Namespace	Physical entity
Virtual memory	Virtual address space	Physical memory
File system	File name	Disk blocks

- Disk block: unit of disk management for the OS
  - May consist of one or more disk sectors

File System Abstraction



Logical block addresses

- We're going to adopt the modern convention of converting cylinder/track/sector into a **logical block address** (LBA)
- That is, we calculate an integer 0 to n-1 so that the disk appears to be a simple array of disk blocks
- Modern disk controllers present this abstraction to the CPU to prevent it from having to know the geometry of the physical disk drives

Disk block addresses

- In the past, including the first 10 years of PCs, we used the ordered triple **(cylinder#, head#, sector#)** as an address because we needed to know the physical attributes of the drive to optimize performance
- In modern times, we've added intelligent disk controllers that do the scheduling for us as well as zoned recording schemes on the disks to pack more data
- This makes the (cyl, hd, sct) addressing from the 1950s pretty much uncalculatable without a significant table of data from the specific HDD model being addressed
- The solution to all this is to move to addressing the disk blocks as an array: 0 – n-1 and letting the controller and the drive do the optimization.
- This is the state of modern SATA, SAS, and NVMe drive protocols, so if you see the (cyl, hd, sct) notation, just realize that's now sent as a simple integer from 0 to the number of blocks on the device.

File allocation schemes on the disk

- Contiguous allocation
- Contiguous allocation with overflow
- Linked allocation
- FAT
- Indexed allocation
- Multilevel indexed
- Hybrid indexed

Functionality is similar to memory management

Implemented by "storage manager" or "file system" part of OS

Most OSs support more than one file system.

Figures of merit

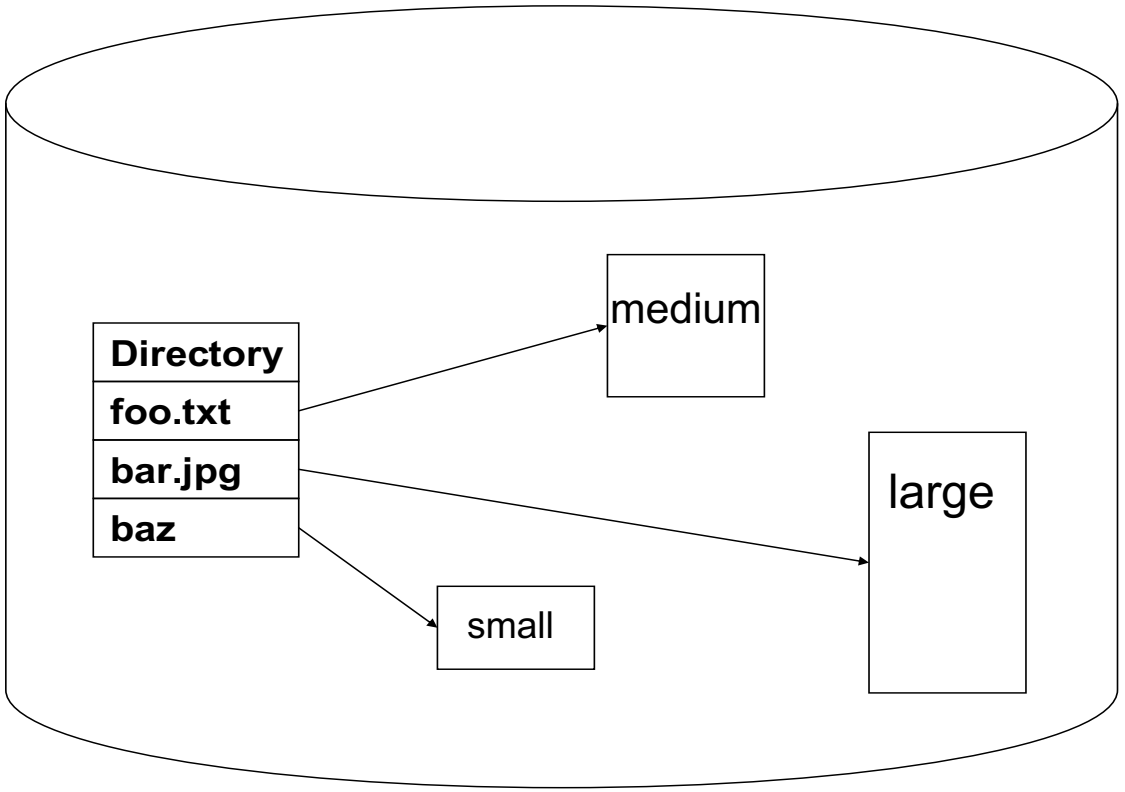
Specific to the file system

- Fast **sequential** access → Use case?
- Fast **random** access → Use case?
- Ability to **grow** the file → Flexibility
- **Easy allocation** of storage → Time
- Efficiency of **space utilization** on the disk → Concerns of internal and external fragmentation

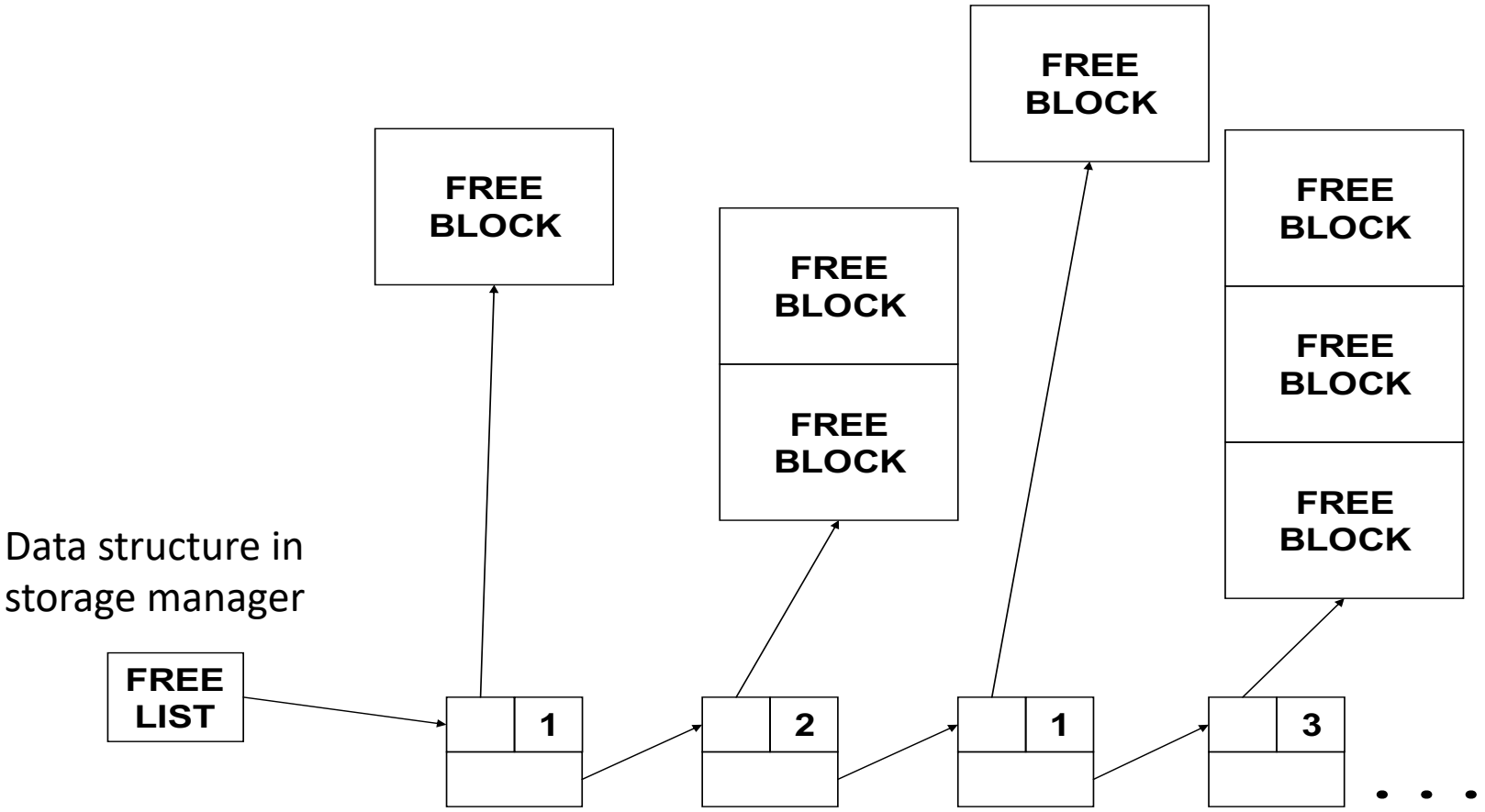
Common concerns to VM and file system

Contiguous allocation

Similar to variable-sized partitioning in memory management



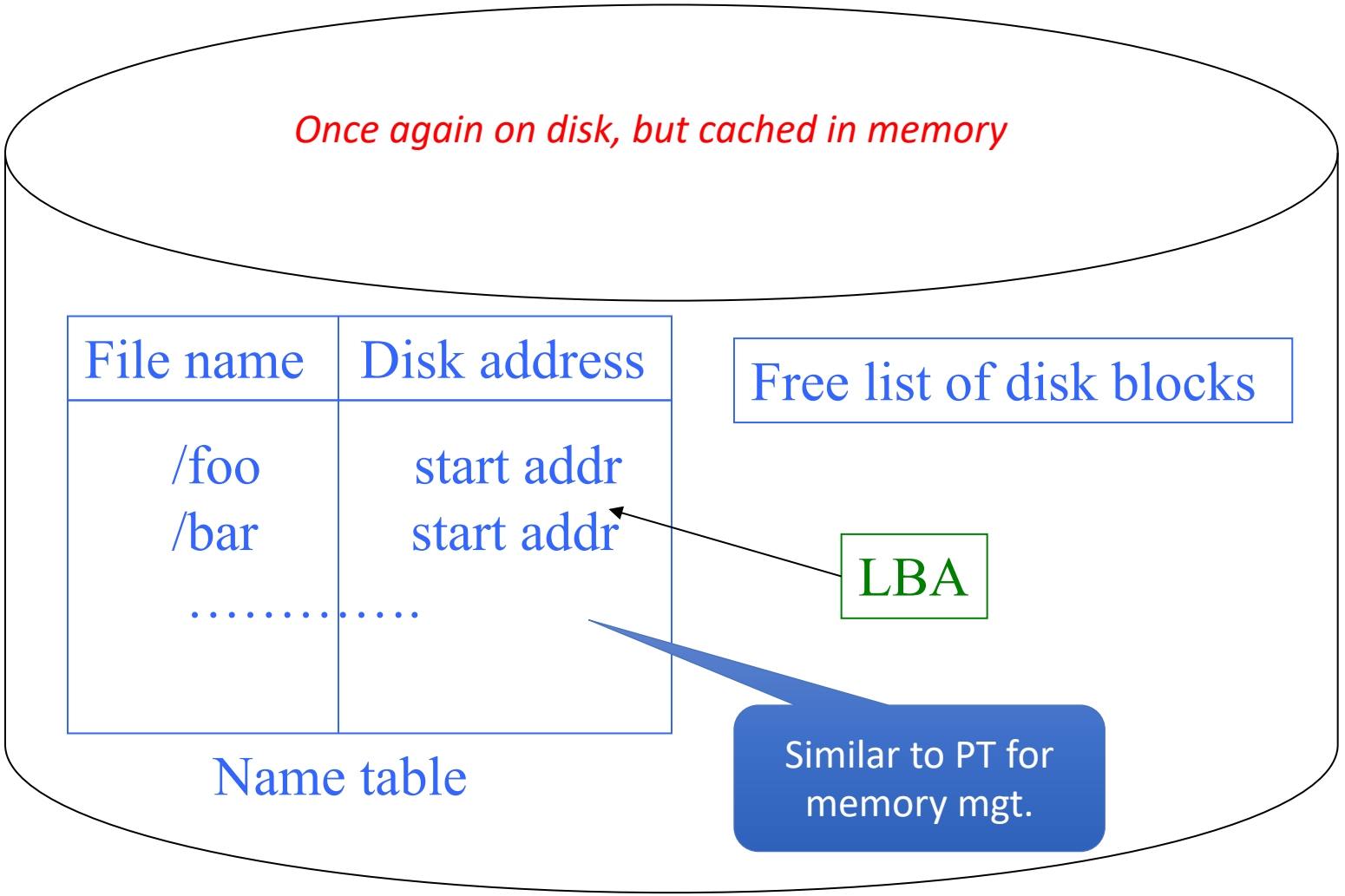
Free list for contiguous allocation



Where does this data structure live?

➔On disk but "cached" in physical memory

Data structures for storage manager



Contiguous allocation

- X Growth?
- X Quickness of allocation (time)?
- X Fragmentation (space)?
- ✓ Sequential access?
- ✓ Random access?

Can be fixed with "overflow" areas…
➔ Contiguous allocation with overflow

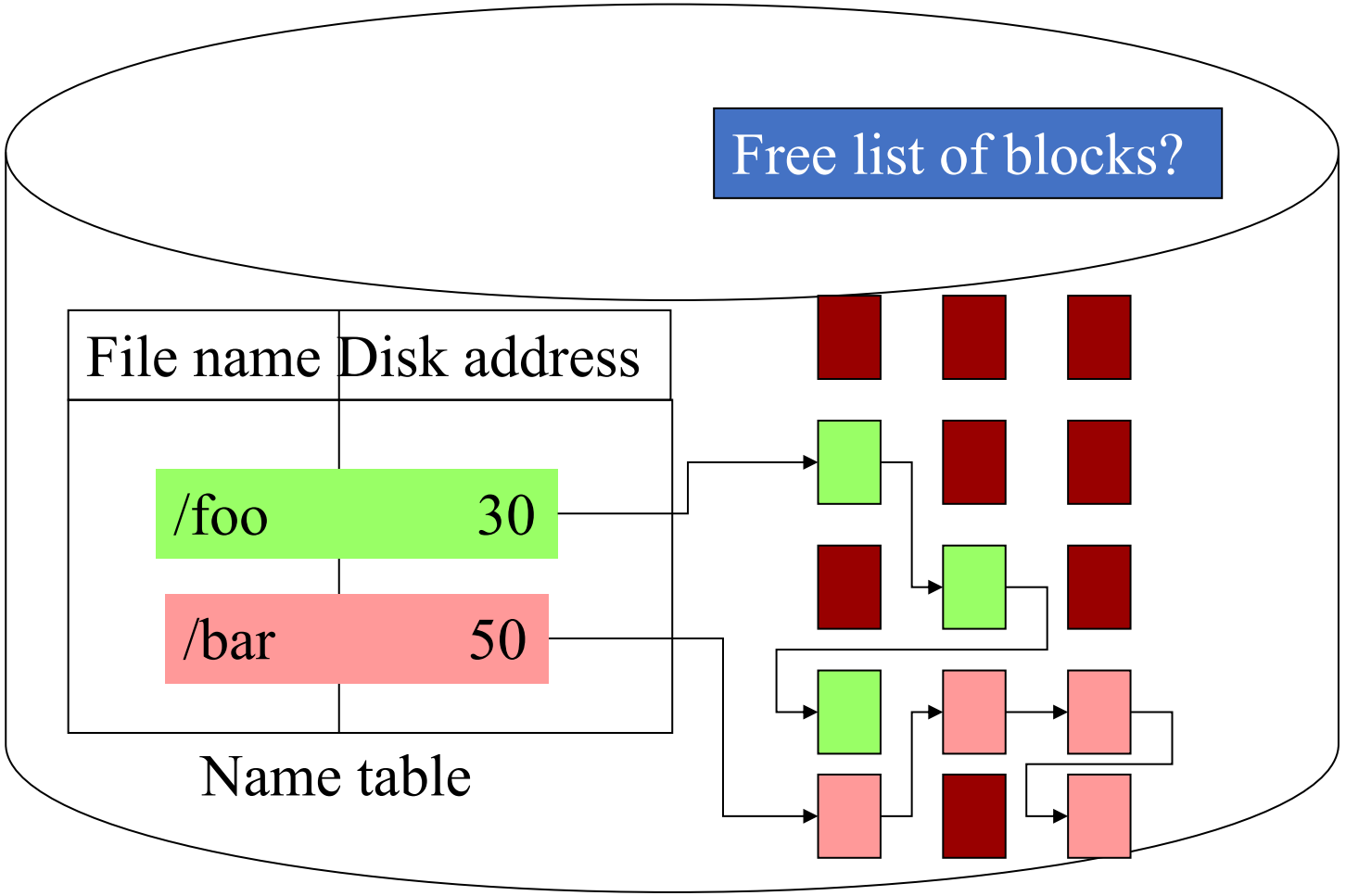
Both internal and external

Similar issues to variable-sized partitioned memory management
- First fit
- Best fit
- Compaction

File allocation schemes on the disk

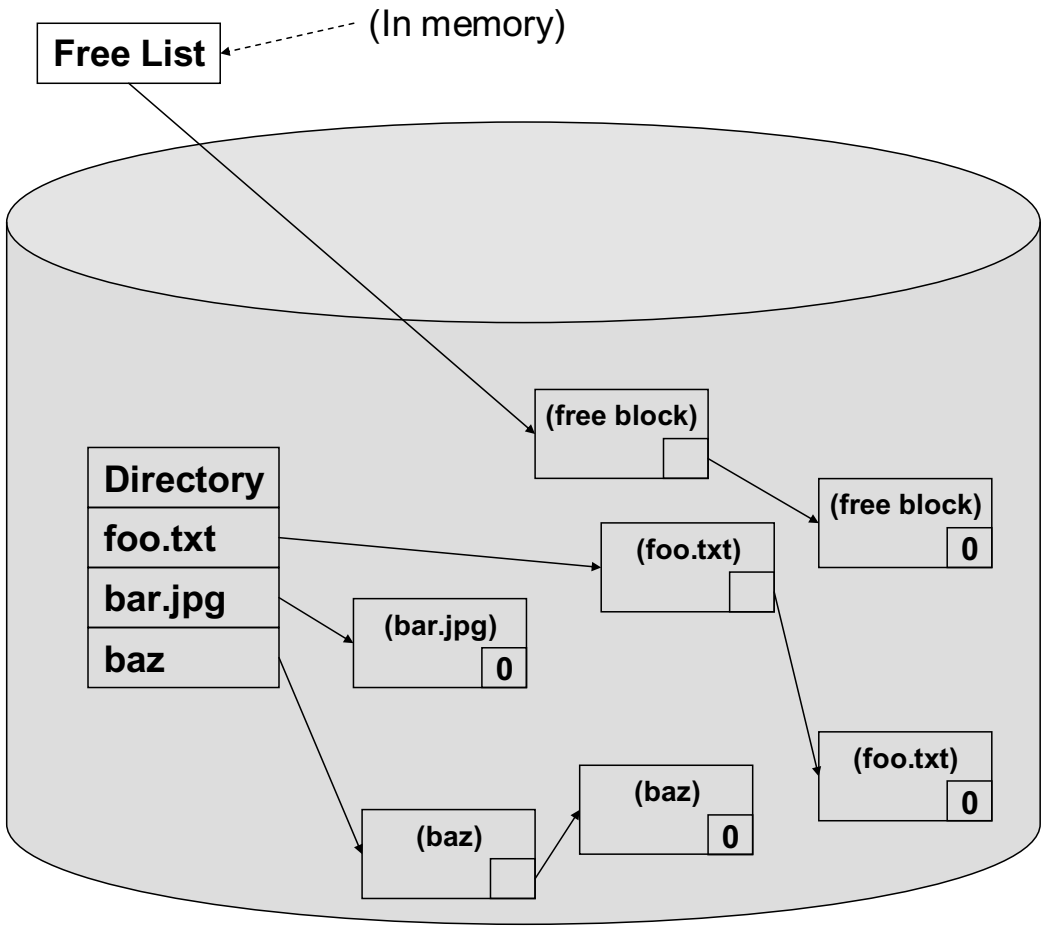
- ✓ Contiguous allocation
- ✓ Contiguous allocation with overflow
- ➔ Linked allocation
- FAT
- Indexed allocation
- Multilevel indexed
- Hybrid indexed

Linked List Allocation



Linked allocation

As before, data structures are on disk but cached in memory



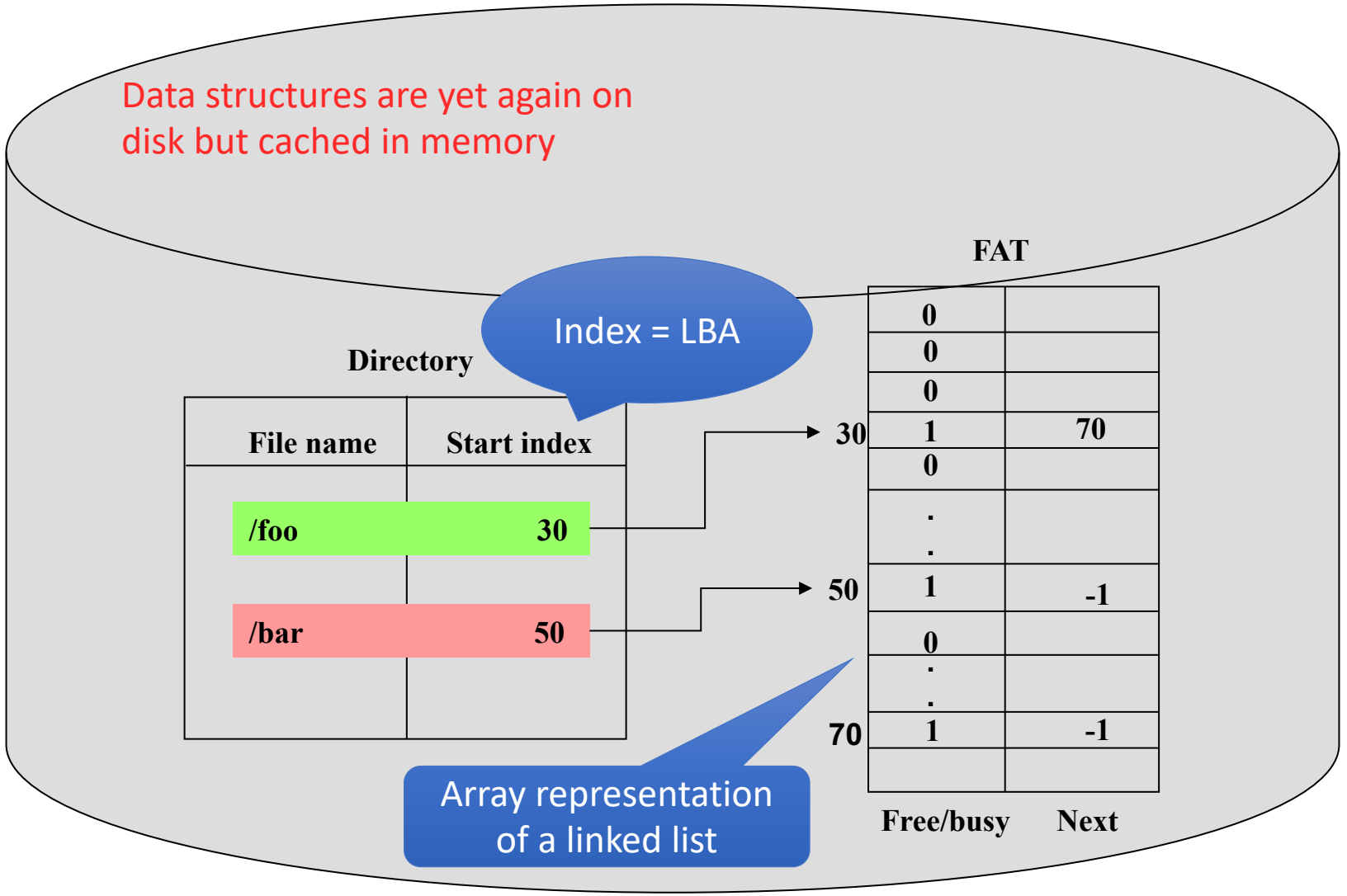
Linked list allocation

- ✓ Growth?
- ✓ Quickness of allocation (time)? Block at a time
- ✓ Fragmentation (space)? No external
- ✓ Sequential access?
- X Random access? Very dependent on seek time

File allocation schemes on the disk

- ✓ Contiguous allocation
- ✓ Contiguous allocation with overflow
- ✓ Linked allocation
- ➔ FAT
- Indexed allocation
- Multilevel indexed
- Hybrid indexed

File allocation table



File allocation table

- ✓ Growth?
- ✓ Quickness of allocation (time)? Better than linked; FAT cached in memory
- ✓ Fragmentation (space)? Same as linked
- ✓ Sequential access? Similar to linked
- ✓ Random access? Still depends on seek time, but next-block pointers stored in FAT, so less serialization

Less error prone than linked allocation – we can write data blocks before FAT

FAT table size limits max supported disk capacity. Must resort to disk partitioning , which becomes a burden to user

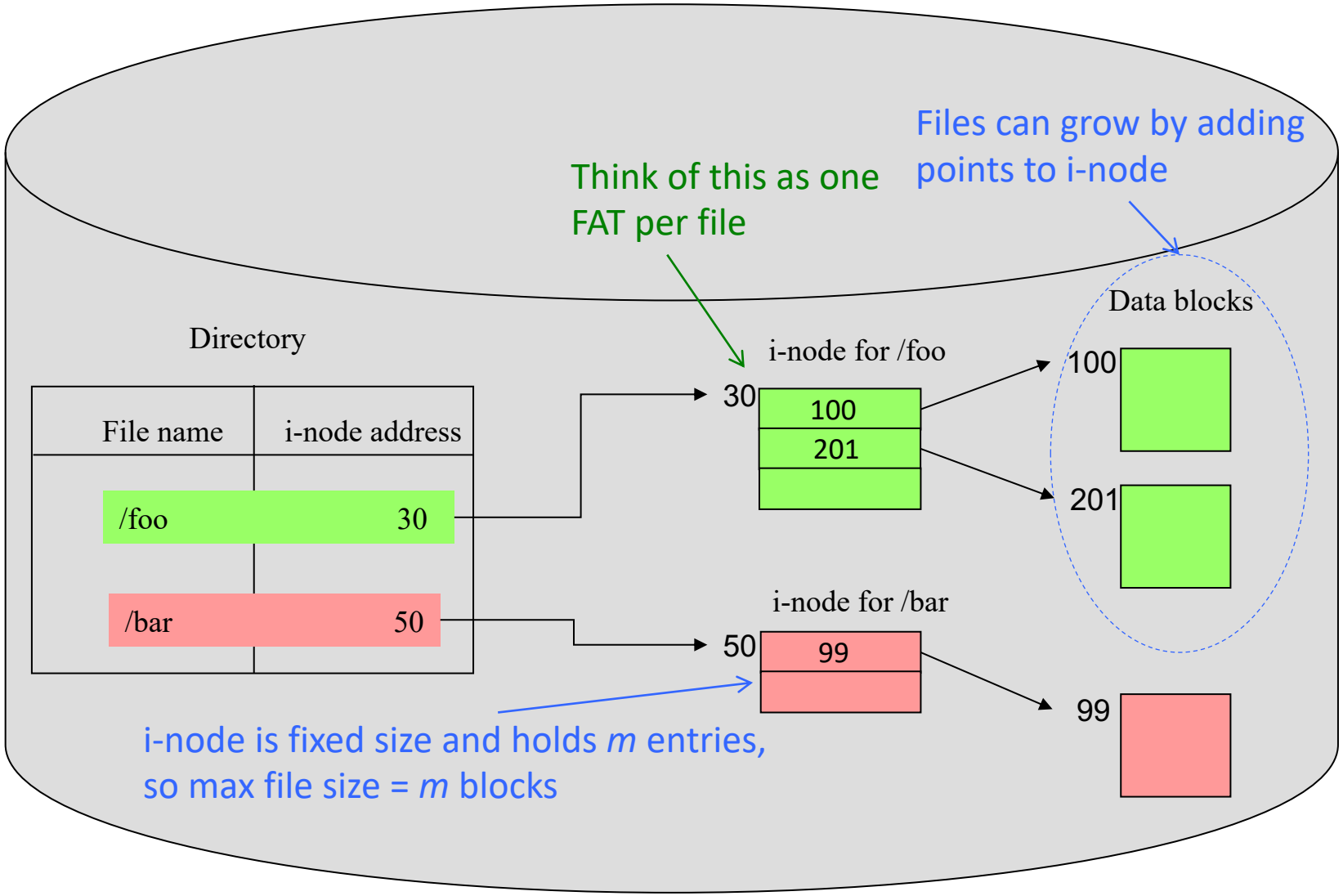
File allocation schemes on the disk

- ✓ Contiguous allocation
- ✓ Contiguous allocation with overflow
- ✓ Linked allocation
- ✓ FAT
- ➔ Indexed allocation
- Multilevel indexed
- Hybrid indexed

Some Unix terminology

- The Unix file system stored the metadata for a file separately from the directory entry; this was a novel idea at the time
- This technique has been used in a number of subsequent file systems but the Unix terminology is often used to describe it
- i-node (modern usage *inode*)
  - All of the information about a file except name was stored in a fixed-length entry on disk called an i-node
  - The i-nodes were kept in an array on the file system so that the disk location of an i-node could be calculated from its index, often called the i-number
  - i-nodes can be cached by the OS in memory when a file is in use
- Directory
  - Directories contained only file names and the i-number (i-node index number)
  - It is possible for two directory entries to contain the same i-number (more on that later)

Indexed allocation



Indexed allocation

- X Growth?
- ✓ Quickness of allocation (time)?
- ✓ Fragmentation (space)?
- ✓ Sequential access?
- ✓ Random access?

Big problems

Same as FAT

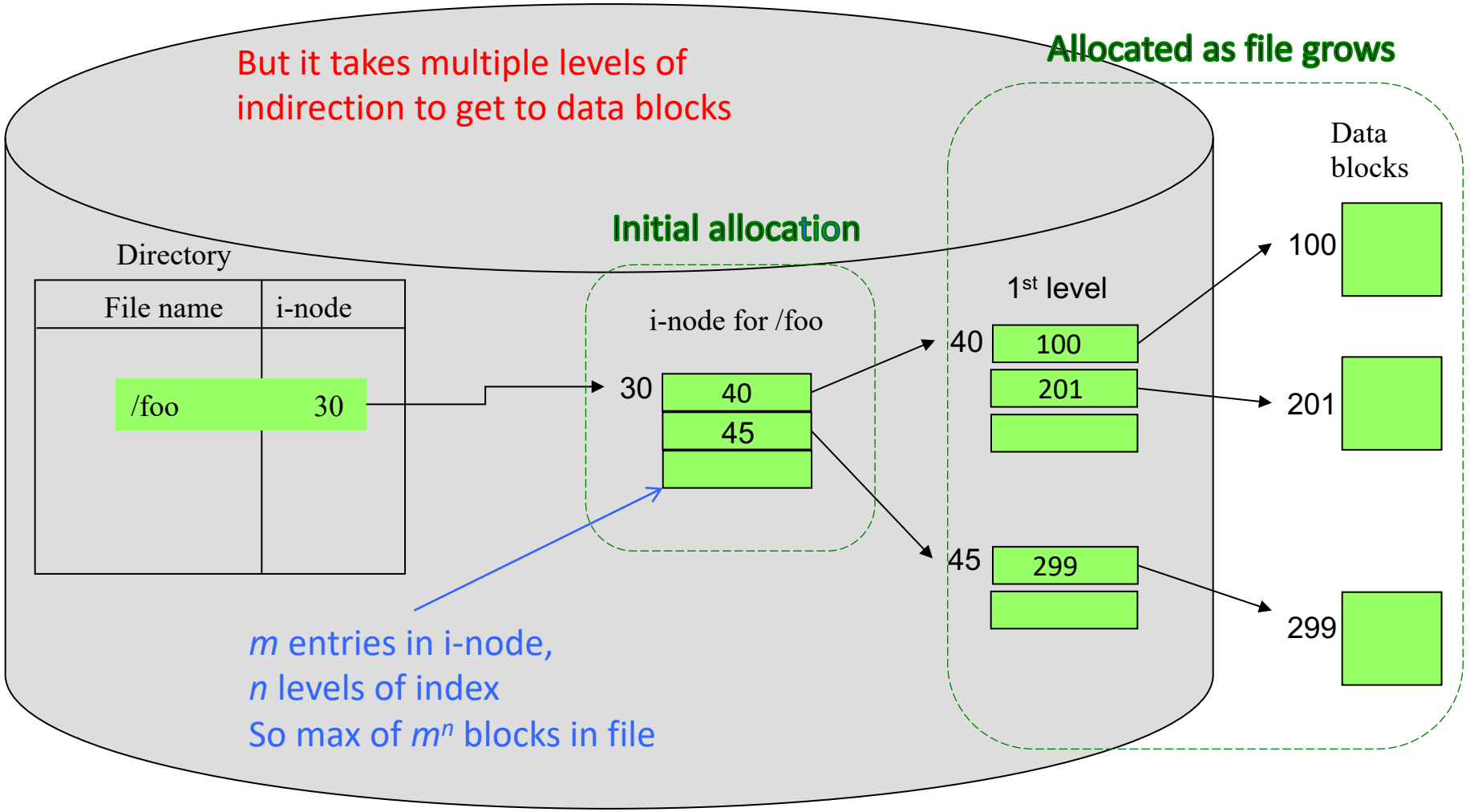
Potentially better than FAT -- Still depends on seek time, but can cache i-node in memory

File allocation schemes on the disk

- ✓ Contiguous allocation
- ✓ Contiguous allocation with overflow
- ✓ Linked allocation
- ✓ FAT
- ✓ Indexed allocation
- ➔ Multilevel indexed
- ➔ Hybrid indexed

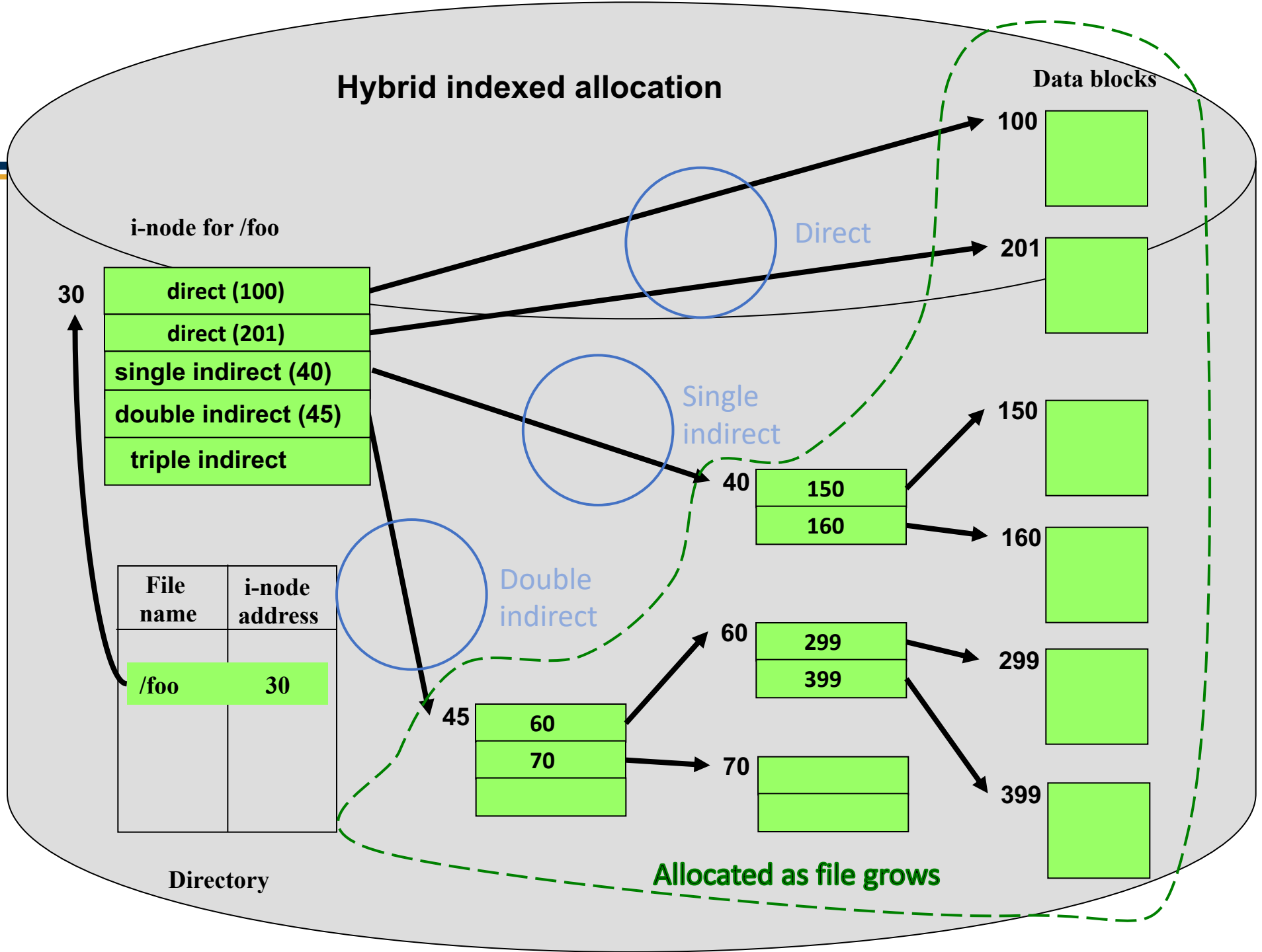
Multilevel indexed allocation

"Every problem in CS can be solved with a layer of indirection"



Different file sizes

- Small files are a big problem with multilevel indexed allocation
- Why?
- The index block(s) take up as much or more space than the file!
- Enter Hybrid Indexed Allocation


Hybrid indexed allocation
Data blocks
i-node for /foo
30
direct (100)
direct (201)
single indirect (40)
double indirect (45)
triple indirect
Direct
Single indirect
Double indirect
100
201
40
150
160
150
160
45
60
70
60
70
299
399
299
399
File name
i-node address
/foo
30
Directory
Allocated as file grows

Hybrid indexed allocation

- ✓ Growth?
- ✓ Quickness of allocation (time)?
- ✓ Fragmentation (space)?
- ✓ Sequential access? Same as FAT
- ✓ Random access? Same as indexed allocation

Best of both worlds: indexed allocation and multilevel indexed allocation

Allocation Strategy	File represen-tation	Free list mainte-nance	Sequential Access	Random Access	File growth	Allocation Overhead	Space Efficiency
Contiguous	Contiguous blocks	complex X	Very good ✓	Very good ✓	messy X	Medium to high X	Internal and external fragmentation X
Contiguous With Overflow	Contiguous blocks for small files	complex X	Very good for small files ✓	Very good for small files ✓	OK ✓	Medium to high X	" X
Linked List	Non-contiguous blocks	Bit vector ✓	Good but dependent on seek time ✓	Not good X	Very good ✓	Small to medium	Excellent ✓
FAT Partitioning not good for user	"	FAT ✓	" ✓	Good but dependent on seek time ✓	" ✓	Small ✓	" ✓
Indexed	"	Bit vector ✓	" ✓	" ✓	limited X	" ✓	" ✓
Multilevel Indexed Small Files ☹	"	Bit vector ✓	" ✓	" ✓	good ✓	" ✓	" ✓
Hybrid	"	Bit vector ✓	" ✓	" ✓	" ✓	" ✓	" ✓

Example

Given the following:

Size of index block = 512 bytes
Size of data block = 2048 bytes
Size of pointer = 8 bytes (to index or data blocks)

The i-node consists of

2 direct data block pointers,
1 single indirect pointer, and
1 double indirect pointer.

An index block is used for the i-node as well as for the index blocks that store pointers to other index blocks and data blocks. Note that the index blocks and data blocks are allocated on a need basis.

(a) What is the maximum size (in bytes) of a file that can be stored in this file system?
(b) How many data blocks are needed for storing the same data file of 266 KB?
(c) How many index blocks are needed for storing a data file of size 266 KB?

Example: solution

Size of index block = 512 bytes
Size of data block = 2048 bytes
Size of pointer = 8 bytes (to index or data blocks)

The i-node consists of
2 direct data block pointers,
1 single indirect pointer, and
1 double indirect pointer.

(a) Maximum file size:

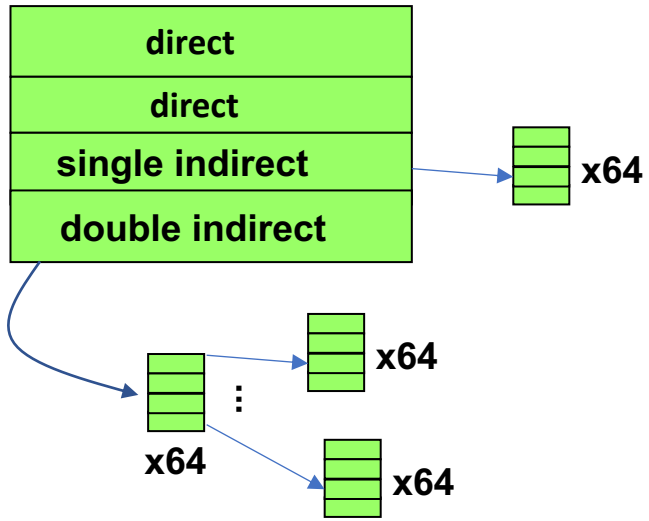
- An index block can hold 512/8 = 64 entries
- 2 direct data block pointers in i-node
- 64 entries in first-level index
- 64 * 64 entries in second-level index
- Total: 4162 blocks * 2048 bytes = 8,523,776 bytes

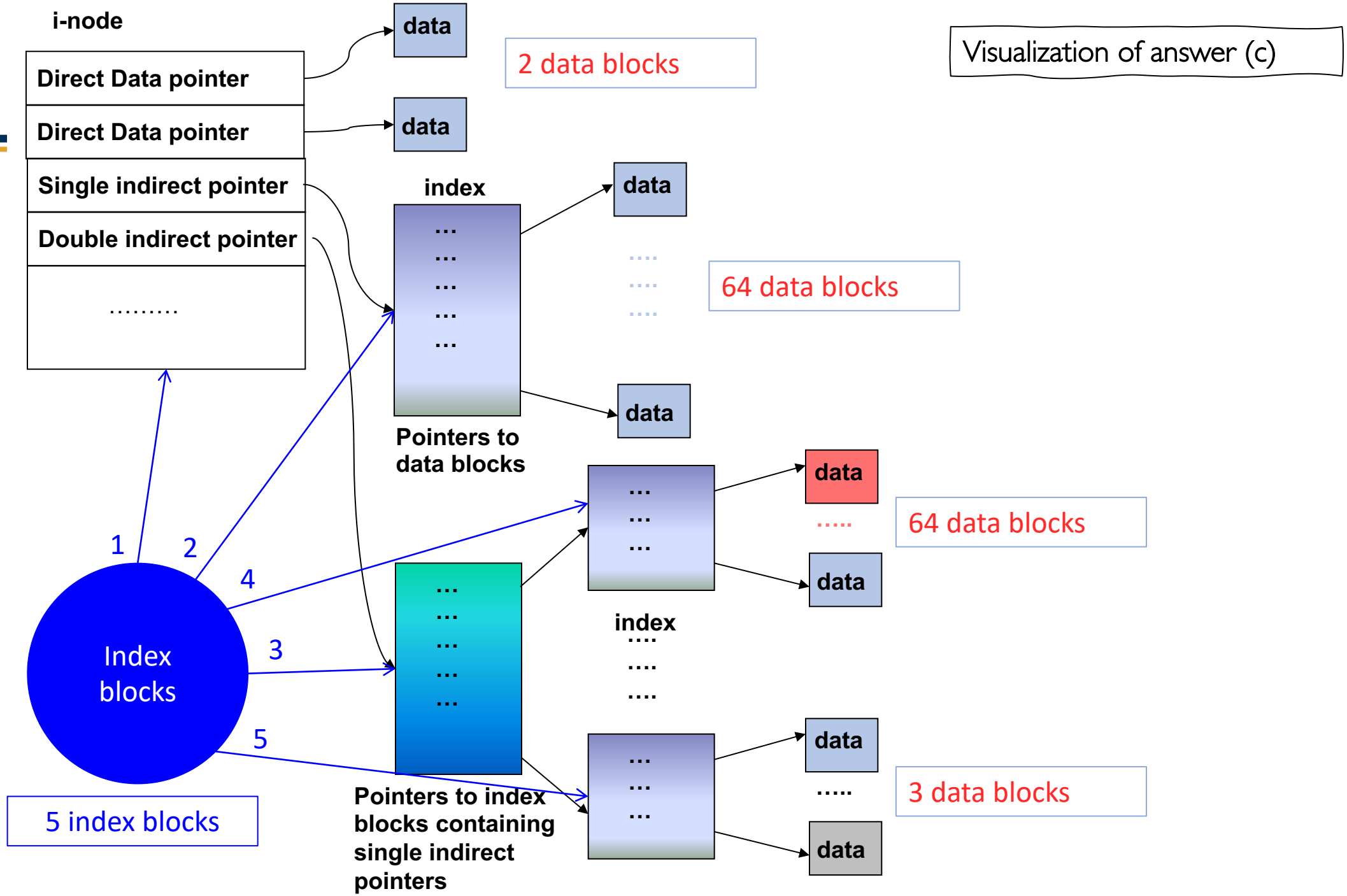
(b) Number of data blocks to hold 266KB:

- $266 * 2^{10} / 2048 = 266 / 2 = 133$ blocks

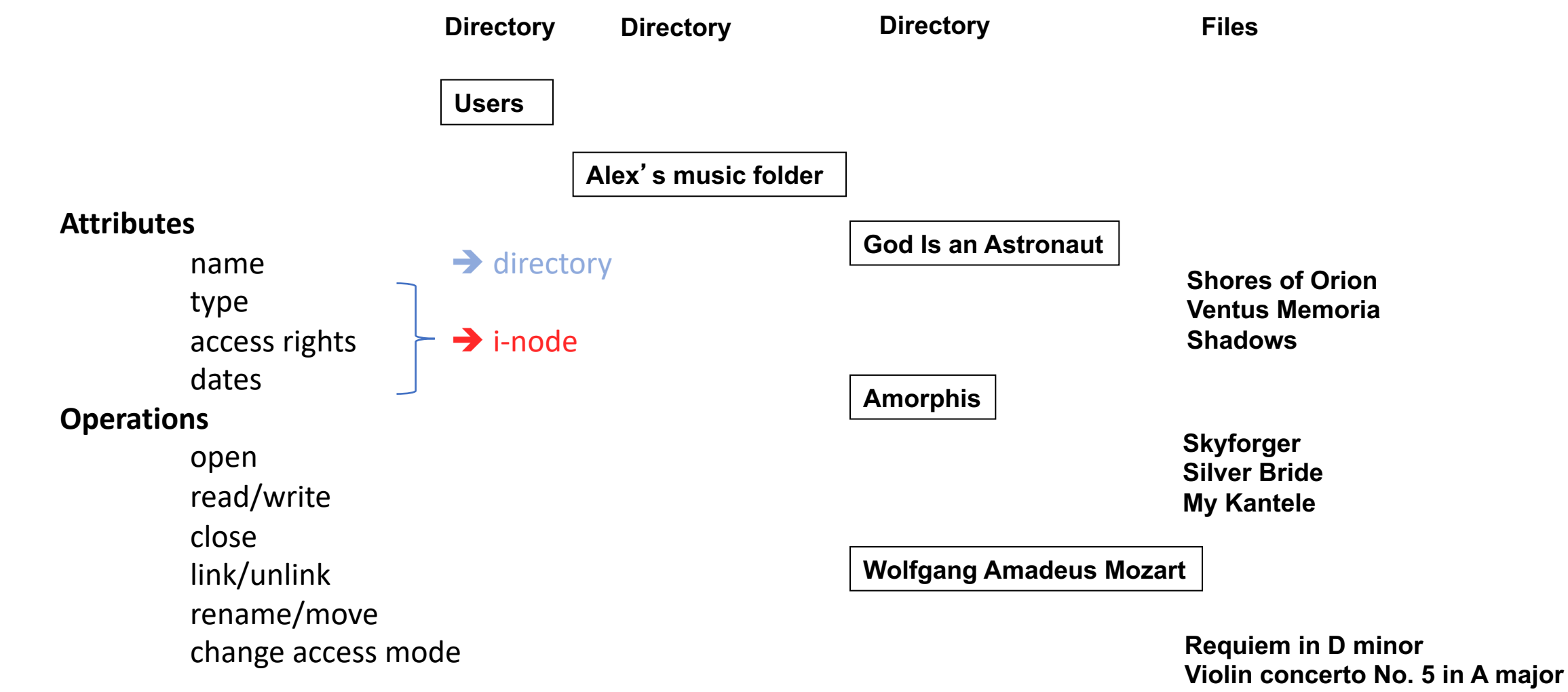
(c) How many index blocks to hold 266KB (133 blocks):

- 2 direct + 64 first-level blocks + (64 + 3) second-level blocks
- 1 i-node + 1 first-level index + 1 + 2 second-level index



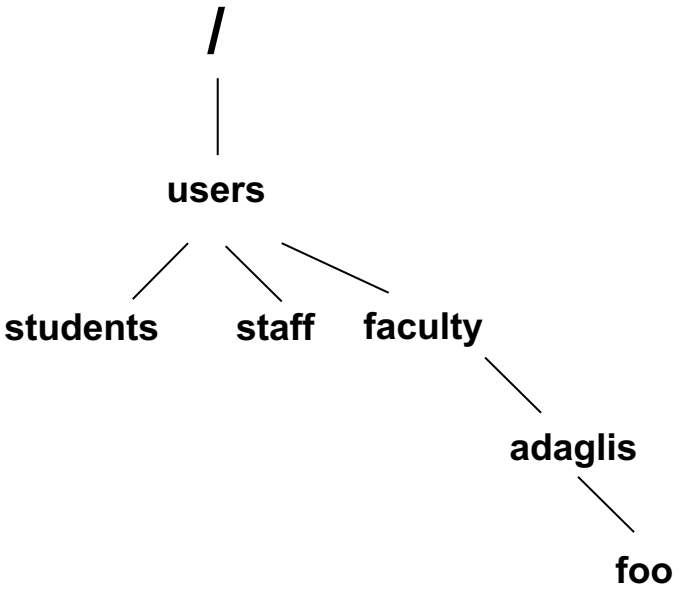

i-node
Direct Data pointer
Direct Data pointer
Single indirect pointer
Double indirect pointer
.....
data
data
2 data blocks
index
data
64 data blocks
data
Pointers to data blocks
Visualization of answer (c)
1
2
4
3
5
Index blocks
5 index blocks
data
64 data blocks
data
index
Pointers to index blocks containing single indirect pointers
data
3 data blocks
data

Unix file system



Unix file system

- Metadata in the i-node
  - Access rights: U G O
  - User name (uid)
  - Group name (gid)
  - Size
  - Modification date
  - Data/index block addrs



"d" if directory

User Group Other

Links

Name (in directory)

```
-rw-r--r--   1 adaglis  faculty 475  Apr 25  2014 foo
```

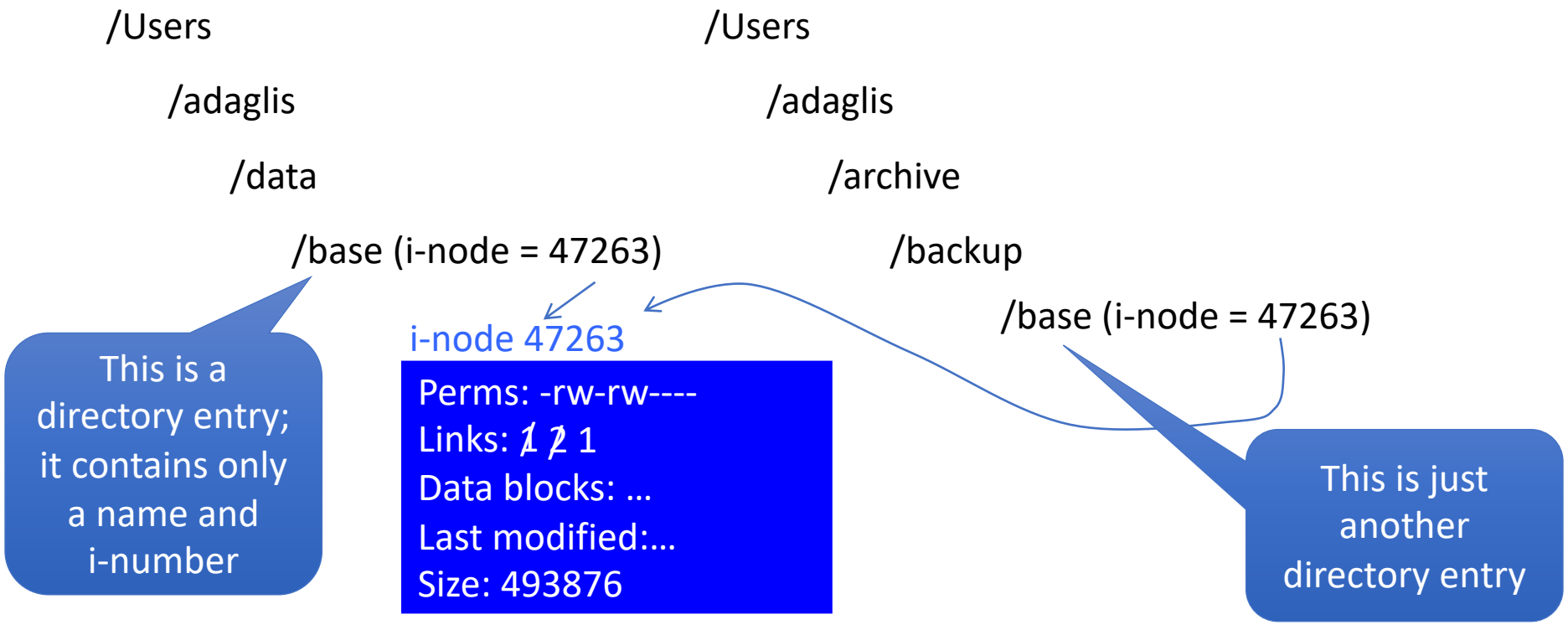
Links

- Remember that Unix-style file system separates the directory entry from the i-node; this means more than one directory entry can index the same i-node.
- When you create a file, you initialize an i-node and create a directory entry; this sets the link count in the i-node to 1
- You can also create additional links to the file with the link() system call or *ln* command; this increments the link count in the i-node for each link; this is called a "hard link"
- Curiously, there is no such thing as a primary or secondary link; both directory entries are names for the file and neither link has precedence over the other
- There is no system call to "remove" a file; you can only unlink a directory entry
- The file is deleted from the file system only when its link count reaches zero; you remove a link with the unlink() system call or the *rm* command
- Even better, each i-node also has an in-use count which counts the number processes that have the file open; the i-node is not deallocated until the link count AND the in-use count both reach zero, so you can have an open file that has an i-node but no directory entry and hence no name in the file system

Symbolic links

- Unix also provides a way to create file aliases; these are called symbolic links
- A symbolic link occupies an i-node (contrast with a hard link doesn't take an additional i-node)
- The i-node is marked as a symbolic link and the data blocks contain a path name instead of user data
- When a symbolic link is encountered while following a path name, the path is replaced by the contents of the symbolic link and the search continues by following the path in the symbolic link
- The file system doesn't promise that symbolic links point to anything; the presence of a symbolic link pointing to a file name doesn't prevent that file from being unlinked
- Symlinks work across file systems contrasted with hard links that do not

Unix file and links

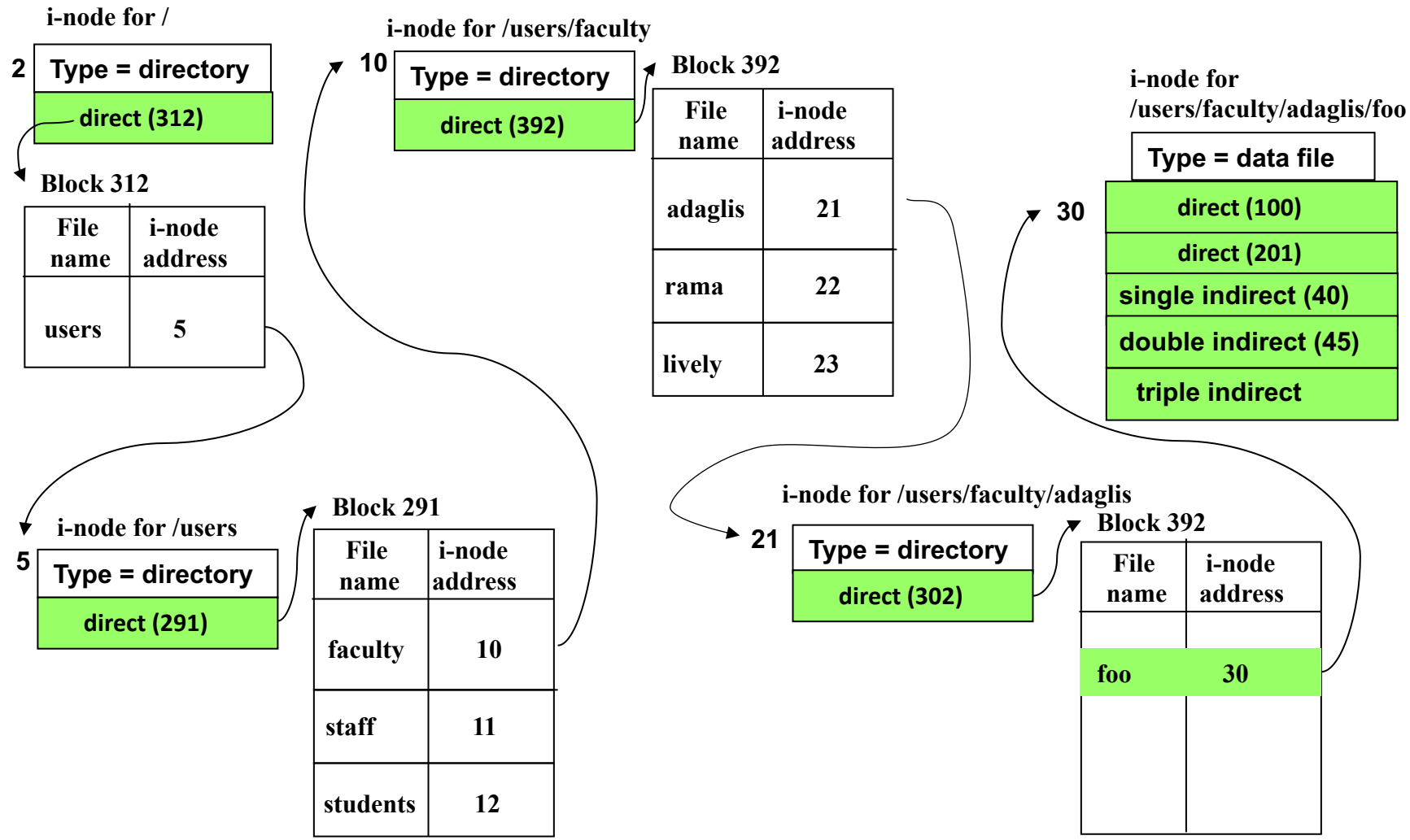



```
ln /Users/adaglis/data/base /Users/adaglis/archive/backup/base
rm /Users/adaglis/data/base # remember this calls unlink()
```

Attribute	Meaning	Elaboration
Name	Name of the file	Attribute set at the time of creation or renaming
Alias	Other names that exist for the same physical file	Attribute gets set when an alias is created; system such as Unix provide explicit commands for creating aliases for a given file; Unix supports aliasing at two different levels (physical or hard, and symbolic or soft)
Owner	Usually the user who created the file	Attribute gets set at the time of creation of a file; systems such as Unix provide mechanism for the file's ownership to be changed by the superuser
Creation time	Time when the file was created first	Attribute gets set at the time a file is created or copied from some other place
Last write time	Time when the file was last written to	Attribute gets set at the time the file is written to or copied; in most file systems the creation time attribute is the same as the last write time attribute; Note that moving a file from one location to another preserves the creation time of the file
Privileges •Read •Write •Execute	The permissions or access rights to the file specifies who can do what to the file	Attribute gets set to default values at the time of creation of the file; usually, file systems provide commands to modify the privileges by the owner of the file; modern Linux and Windows file systems such as ext4 and NTFS also provide an access control list (ACL) to give more granular access to different users
Size	Total space occupied on the file system	Attribute gets set every time the size changes due to modification to the file

Unix command	Semantics	Elaboration
touch <name>	**Create a file with the name <name>**	Creates a zero byte file with the name <name> and a creation time equal to the current wall clock time
mkdir <sub-dir>	**Create a sub-directory <sub-dir>**	The user must have write privilege to the current working directory (if <sub-dir> is a relative name) to be able to successfully execute this command
rm <name>	**Remove (or delete) the file named <name>**	Only the owner of the file (and/or superuser) can delete a file
rmdir <sub-dir>	**Remove (or delete) the sub-directory named <sub-dir>**	Only the owner of the <sub-dir> (and/or the superuse) can remove the named sub-directory
ln –s <orig> <new>	**Create a name <new> and make it symbolically equivalent to the file <orig>**	This is name equivalence only; so if the file <orig> is deleted, the storage associated with <orig> is reclaimed, and hence <new> will be a dangling reference to a non-existent file
ln <orig> <new>	**Create a name <new> and make it physically equivalent to the file <orig>**	Even if the file <orig> is deleted, the physical file remains accessible via the name <new>
chmod <rights> <name>	**Change the access rights for the file <name> as specified in the mask <rights>**	Only the owner of the file (and/or the superuser) can change the access rights
chown <user> <name>	**Change the owner of the file <name> to be <user>**	Only superuser can change the ownership of a file
chgrp <group> <name>	**Change the group associated with the file <name> to be <group>**	Only the owner of the file (and/or the superuser) can change the group associated with a file
cp <orig> <new>	**Create a new file <new> that is a copy of the file <orig>**	The copy is created in the same directory if <new> is a file name; if <new> is a directory name, then a copy with the same name <orig> is created in the directory <new>
mv <orig> <new>	**Renames the file <orig> with the name <new>**	Renaming happens in the same directory if <new> is a file name; if <new> is a directory name, then the file <orig> is moved into the directory <new> preserving its name <orig>
cat/more/less <name>	**View the file contents**	

Directory Structure



File system integrity

- File systems are prone to corruption at a system crash
  - Some of the cached blocks may not get written out
  - Metadata on disk is inconsistent
  - E.g. if the free space list is inconsistent, subsequent allocations may write over good data
- Requires an extensive, time-consuming consistency check before the file system can be used again
  - Historically, fsck on Unix, scandisk on DOS
- Many modern file systems attempt to avoid this situation

File system integrity: journaling

- Journaling generally means writing a journal of metadata changes before committing them to the file system
  - Replaying the last few minutes of journaling is much faster than doing a full consistency check
  - Linux ext3/ext4 and Windows NTFS are examples
- Journaling doesn't guarantee user data is consistent
  - E.g. some blocks may get written out of order
- Other modern file systems use a copy-on-write model
  - Data blocks are never re-written, but copied to free space
  - Blocks are rewritten in an order that guarantees that the disk file system is always consistent
  - No need for consistency checking
  - Examples include ZFS and Btrfs

A simple hybrid-indexed file system

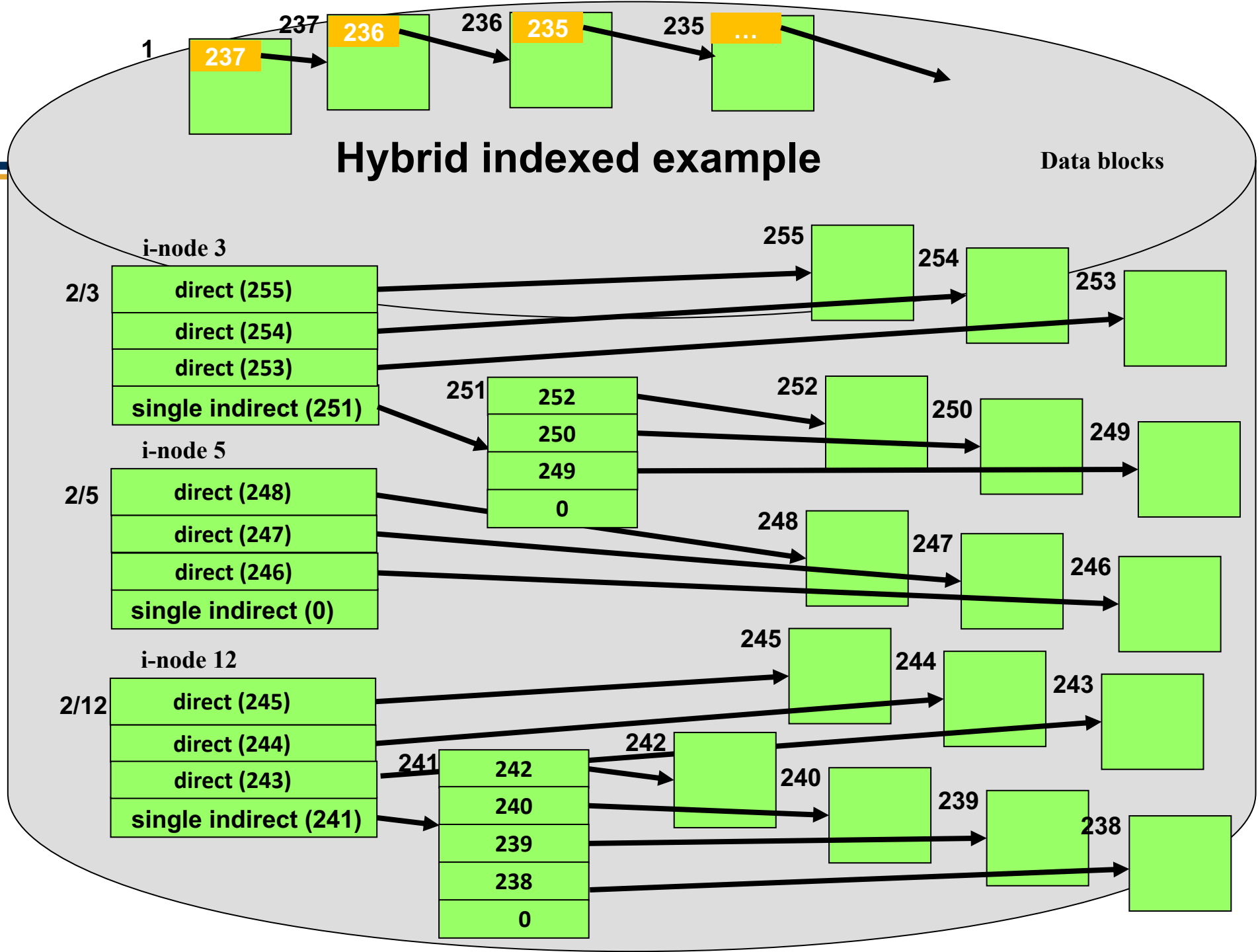
- We've provided two files - fs.h and fs.c. Between the two files is an implementation of a small hybrid-indexed file system.
- Our file system divides the disk into blocks of size BLOCK SIZE. Each block can be one of four types:
  - **An i-node.** This is a top-level data structure of a file and contains pointers to blocks that hold the file's data.
  - **An index block.** An index block simply contains pointers to data blocks. An index block will be used for the i-node's indirect data pointer.
  - **A data block.** A data block contains a file's data.
  - **A free block.** A free block is any block which isn't currently being used. It is a part of a linked list, so it contains a pointer to the next block in the freelist.
- Each of these types has a corresponding struct definition in fs.h.

Simple example characteristics

- Our i-nodes
  - have NUM_DIRECT_BLOCKS direct data pointers
  - one single indirect data pointer
- Both the direct data pointers and the indirect data pointer are stored in the i-node as their block numbers, with which the file system can find the correct block on disk.
- Even though these types of file systems have directories for naming files, this small implementation doesn't include them to make it more approachable.
- In this implementation, the file system interface allows you to reference files by their i-node number (inumber) instead of names, which means the implementation of directories can wait for the next version.
- For now, recognize that directories are not implemented in this version.

Functions of note

- main() – test program to create and exercise the file system
- fsread()/fswrite – functions that allow reading and writing of blocks in a file
- disk_read()/disk_write() – functions that pretend to be device-driver read and write block functions (but use a file named "diskimage" to act as the pretend disk drive)


1
237
237
236
236
235
235
...
Hybrid indexed example
Data blocks
i-node 3
2/3
direct (255)
direct (254)
direct (253)
single indirect (251)
255
254
253
251
252
250
249
0
252
250
249
i-node 5
2/5
direct (248)
direct (247)
direct (246)
single indirect (0)
248
247
246
i-node 12
2/12
direct (245)
direct (244)
direct (243)
single indirect (241)
245
244
243
241
242
240
239
238
0
242
240
239
238