Storage Devices

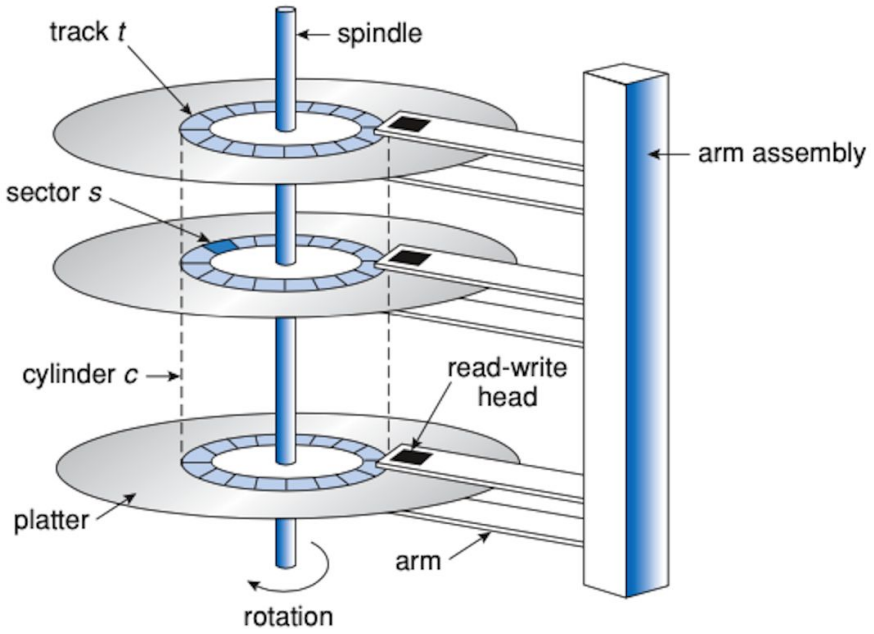
W4118 Operating Systems I

https://cs4118.github.io/www/2024-1/

Credits to Jae and David Mazières

Hard Disk Drive (HDD)

- 2-5 aluminum **platters** attached to a rotating **spindle**
- The surfaces have a thin magnetic coating that stores bits
- The platters are divided into **cylinders/tracks** and **sectors** (512 bytes each)
- The platters rotate fast and the **arm** moves back and forth to cover the surface.
- The **read-write** head is used to access the data.



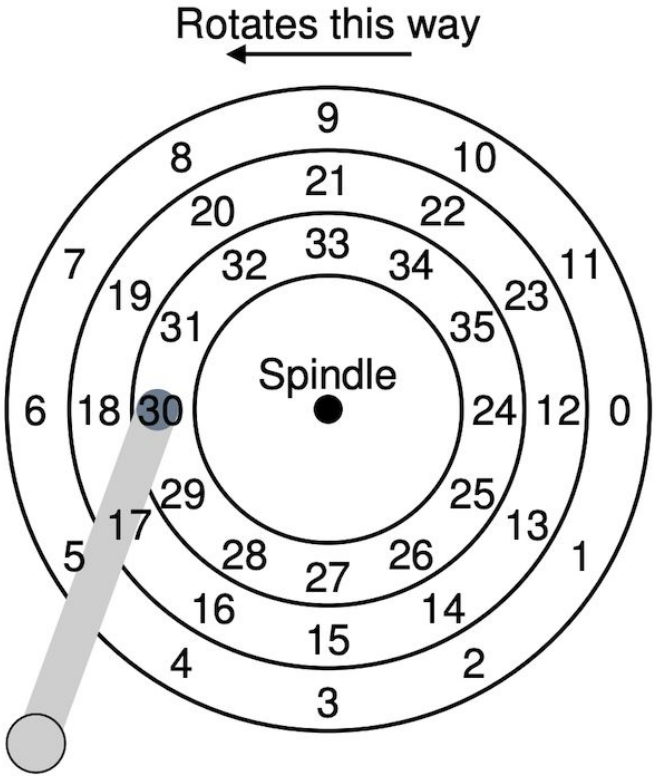
Accessing Data

Older HDDs required the OS to provide cylinder #, head #, sector # (**CHS**)

Newer devices expose a uniform **LBA** logical block address

The disk controller translates the LBA to CHS

OS views storage device as 1D-array of groups of sectors known as **blocks**



Performance Considerations

Reading a sector:

- **Seek time:** move disk arm to the correct track; the most expensive
  - requires acceleration, coasting, decelerating, and settling
- **Rotational delay:** wait for sectors to pass under the head
- **Transfer time:** read bits off the disk

Avoid expensive seek time by accessing disk sequentially (e.g. by interacting with adjacent sectors). Random access is 200-300x more expensive than sequential access. Data structures and algorithms are designed around this trade-off.

Seek Performance Considerations

Move head to specific track and keep it there

- Resist physical shocks, imperfect tracks, etc.

A seek consists of up to four phases:

- *speedup* – accelerate arm to max speed or half way point
- *coast* – at max speed (for long seeks)
- *slowdown* – stops arm near destination
- *settle* – adjusts head to actual desired track

Very short seeks dominated by settle time (~1 ms)

Short (200-400 cyl.) seeks dominated by speedup

- Accelerations of 40g

Seek Performance Considerations

Head switches comparable to short seeks

- May also require head adjustment
- Settles take longer for writes than for reads – Why?

Disk keeps table of pivot motor power

- Maps seek distance to power and time
- Disk interpolates over entries in table
- Table set by periodic “thermal recalibration”
- But, e.g., ~500 ms recalibration every ~25 min bad

Seek Performance Considerations

Head switches comparable to short seeks

- May also require head adjustment
- Settles take longer for writes than for reads – Why?
  - If read strays from track, catch error with checksum, retry
  - If write strays, you've just clobbered some other track

Disk keeps table of pivot motor power

- Maps seek distance to power and time
- Disk interpolates over entries in table
- Table set by periodic "thermal recalibration"
- But, e.g., ~500 ms recalibration every ~25 min bad

File Access Patterns

- **Sequential access**
  - Data read/written in order (e.g., copy files)
  - Can be made very fast

- **Random access**
  - Randomly address any block (e.g. update database record)
  - Difficult to make fast because of seek time and rotational delay

Disk Management

Placement & ordering of requests a huge issue

- Sequential I/O much, much faster than random
- Long seeks much slower than short ones
- Power might fail any time, leaving inconsistent state

Must be careful about order for crashes

- More on this in next lectures

Try to achieve contiguous accesses where possible

- E.g., make big chunks of individual files contiguous

Try to order requests to minimize seek times

- OS can only do this if it has multiple requests to order
- Requires disk I/O concurrency
- High-performance apps try to maximize I/O concurrency

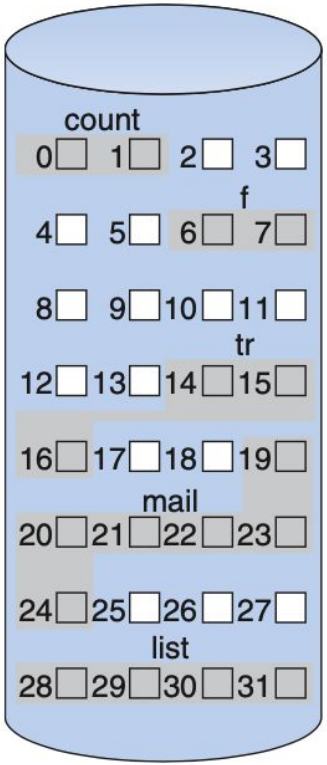
Next: How to place blocks and how to schedule concurrent requests

Allocation Strategies Considerations

- Internal/external fragmentation
- Easy to grow file over time?
- Fast to find data for sequential/random access?
- Easy to implement?
- Storage overhead of metadata and data structures?

Contiguous Allocation

- user specifies length, FS allocates space all at once
- find disk space by examining bitmap
- simple metadata: “base & limit” of file – starting location and size of file



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Contiguous Allocation

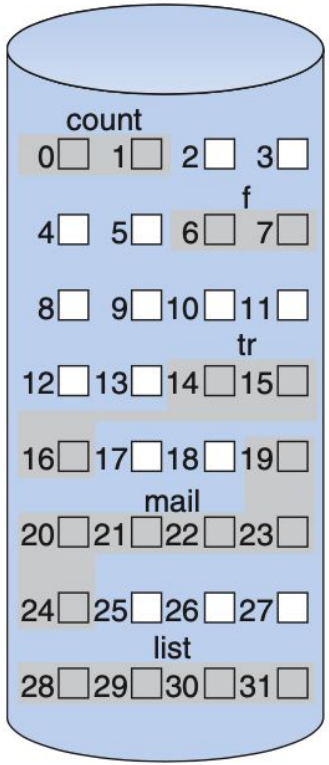
- user specifies length, FS allocates space all at once
- find disk space by examining bitmap
- simple metadata: “base & limit” of file – starting location and size of file

Pros

- Easy to implement
- Low storage overhead (start & length for each file)
- Fast sequential access because of contiguous blocks

Cons

- Suffers from external fragmentation as files are created and removed overtime
- Difficult to grow file because of contiguous requirement



directory

file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Extent-based Allocation

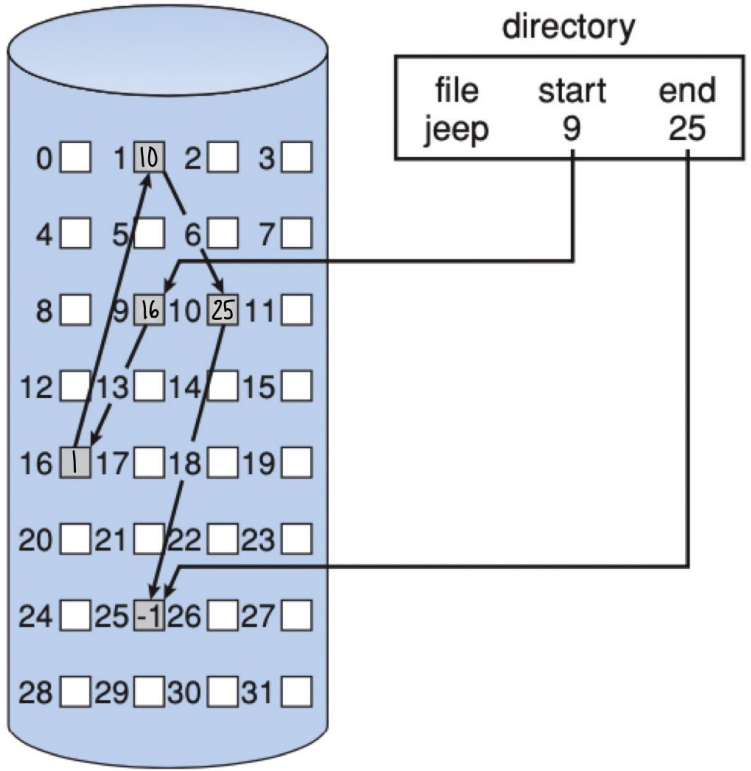
- user specifies length, FS allocates space all at once
- find disk space by examining bitmap
- simple metadata: “base & limit” of file – starting location and size of file

Linked Allocation

All data blocks are part of a linked list

Reserve some metadata bytes at the beginning of the data block for next pointer

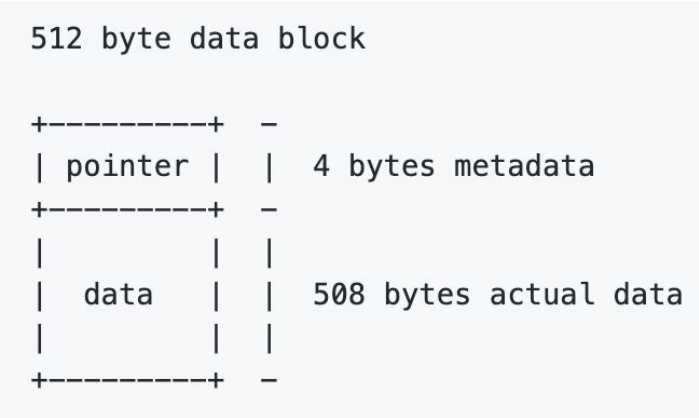
```
512 byte data block

+---------+  -
| pointer |  |  4 bytes metadata
+---------+  -
|         |  |
|  data   |  |  508 bytes actual data
|         |  |
+---------+  -
```



Linked Allocation

All data blocks are part of a linked list

Reserve some metadata bytes at the beginning of the data block for next pointer



Pros

- No external fragmentation – similar to paging
- File can easily grow without limits
- Easy to implement

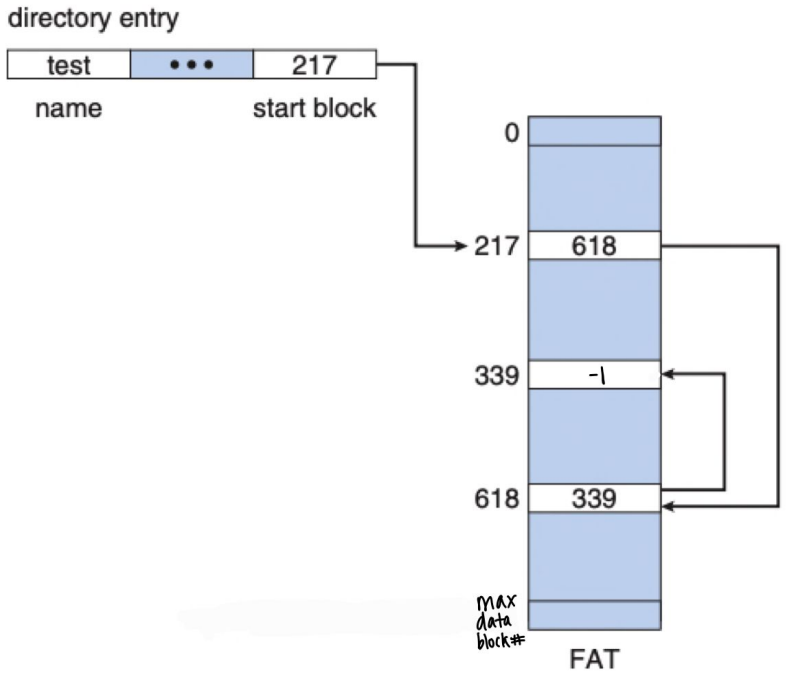
Cons

- Potentially slow sequential access: looks of seeking since blocks need not be contiguous
- Difficult to computer random access, need to walk the linked list
- Some storage overhead (one pointer per block)

File Allocation Table (FAT)

Store linked-list pointers outside of data block in a dedicated pointer table.

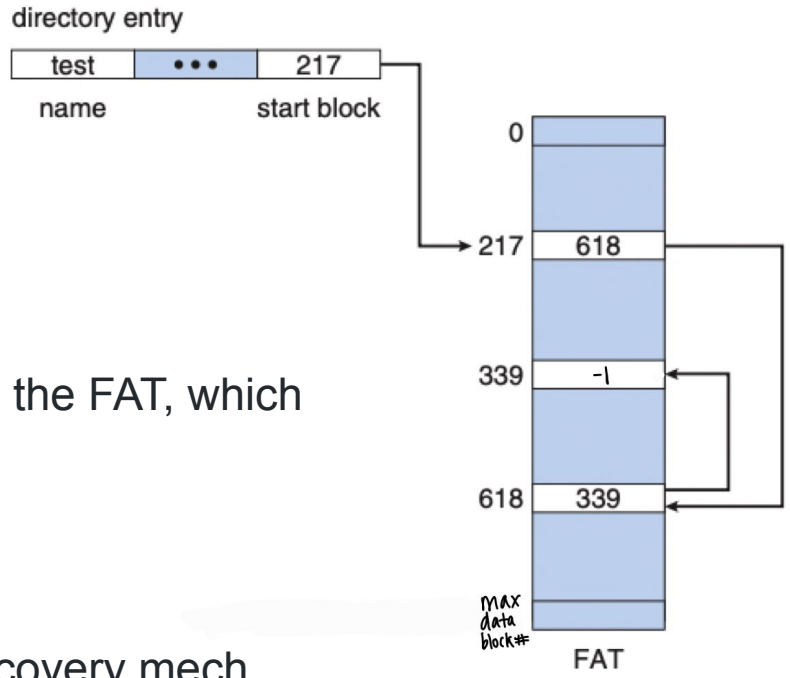
Used in MSDOS and Windows



File Allocation Table (FAT)

Store linked-list pointers outside of data block in a dedicated pointer table.

Used in MSDOS and Windows



Pros

- Better random access: only need to search the FAT, which can be cached in memory

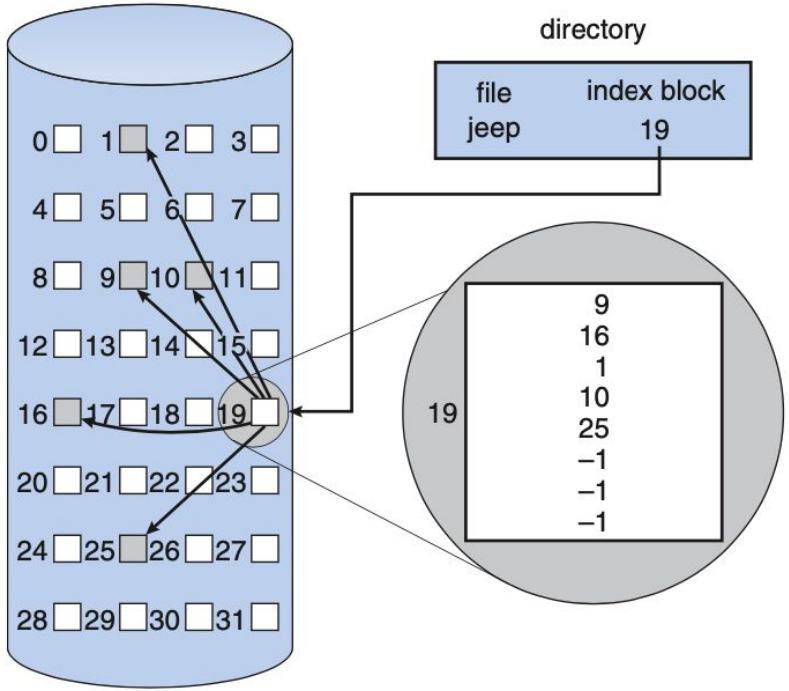
Cons

- Sequential access still slow
- What happens if system crashes? Need recovery mech.

Indexed Allocation

File metadata: array of pointers (index) to data blocks

- file size is limited by number of pointers
- allocate blocks on demand (pointers are marked as invalid in the meantime)



Indexed Allocation

File metadata: array of pointers (index) to data blocks

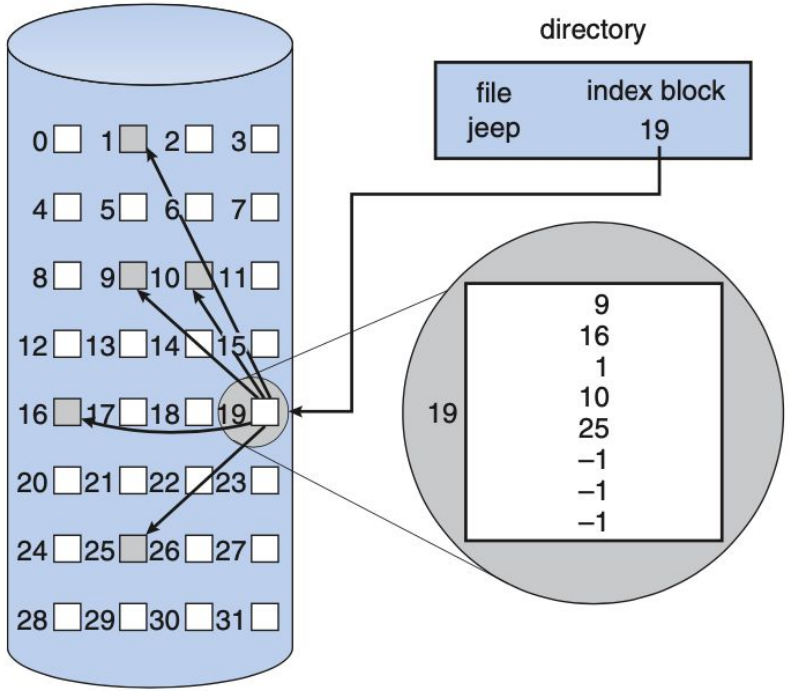
- file size is limited by number of pointers
- allocate blocks on demand (pointers are marked as invalid in the meantime)

Pros

- No external fragmentation
- File can easily grow within the limit
- Fast random access: consult index

Cons

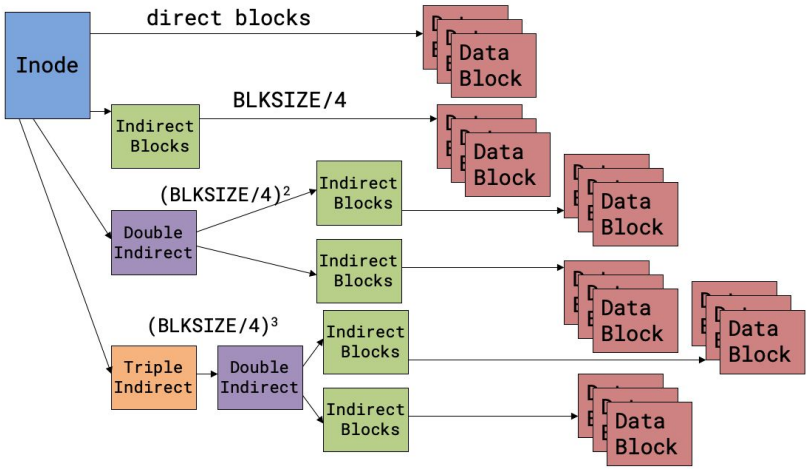
- Large storage overhead for index
- Sequential access can still be slow because of disk seeks



Multi-level indexed Allocation

- Instead of having a single index block of having direct data block pointers, maintain indirect pointers to have larger file size. Used in UNIX FFS, Linux ext2/ext3

- In addition to direct blocks, like we saw in indexed allocation, inodes (index nodes) can also refer to:
  - **indirect block:** contains pointers to other data blocks
  - **double indirect block:** contains pointers to other indirect blocks
  - **triple indirect block:** contains pointers to other double indirect blocks



Max file size: `(NDIRECT + BLKSIZE/4 + (BLKSIZE/4)^2 + (BLKSIZE/4)^3) * BLKSIZE`

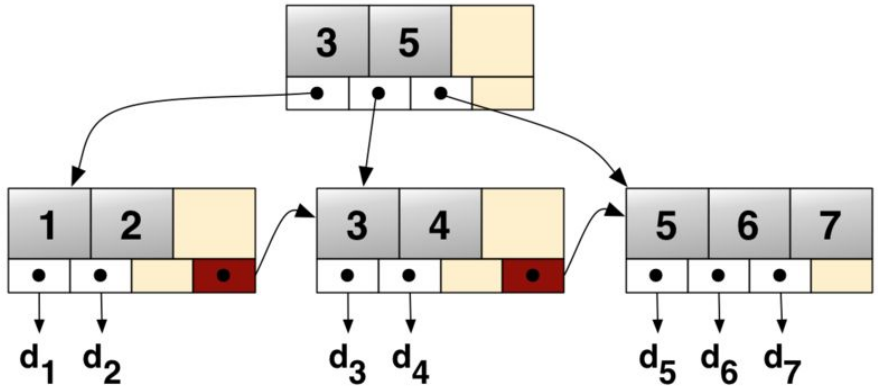
+ Huge file size limit

Advanced Data Structures

Combine indexes with extents/multiple cluster sizes

B+ trees:

- Used by many high performance for directories and/or data
- Can support very large and very sparse files
- Can give very good performance: minimize disk seeks to find blocks
- Used in ReiserFS, ext4, MSFT NTFS



Scheduling: FCFS

“First Come First Served”

- Process disk requests in the order they are received

Pros

Cons

Scheduling: FCFS

“First Come First Served”

- Process disk requests in the order they are received

Pros

- Easy to implement
- Good fairness

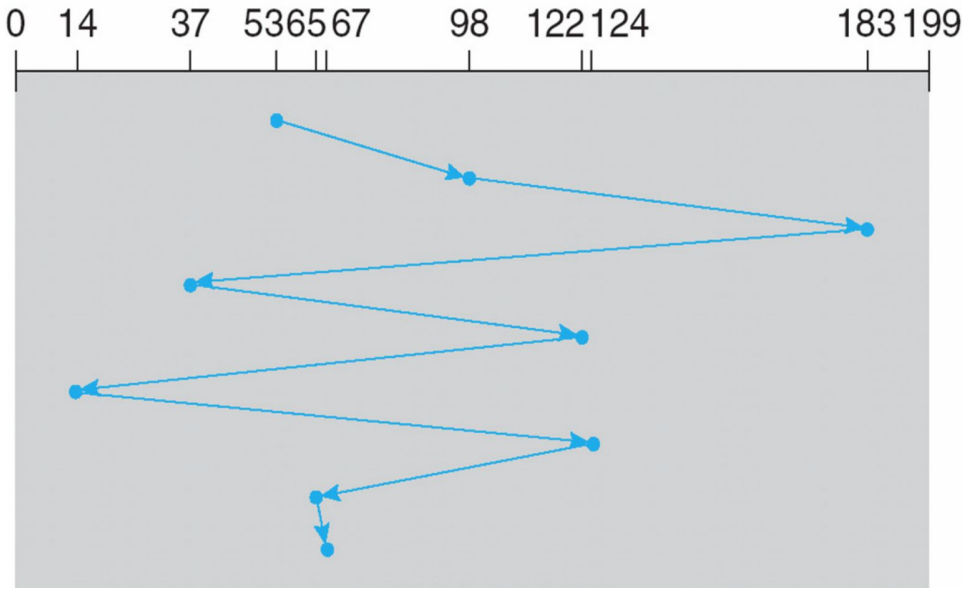
Cons

- Cannot exploit request locality
- Increases average latency, decreasing throughput

FCFS Example

queue = 98, 183, 37, 122, 14 , 124, 65, 67

head starts at 53



Shortest Positioning Time First (SPTF)

Shortest positioning time first (SPTF)

- Always pick request with shortest seek time

Also called Shortest Seek Time First (SSTF)

Pros

Cons

Shortest Positioning Time First (SPTF)

Shortest positioning time first (SPTF)

- Always pick request with shortest seek time

Also called Shortest Seek Time First (SSTF)

Improvement?

Pros

- Exploits locality of disk requests
- Higher throughput

Cons

- Starvation
- Don’t always know what request will be fastest

Shortest Positioning Time First (SPTF)

Shortest positioning time first (SPTF)

- Always pick request with shortest seek time

Also called Shortest Seek Time First (SSTF)

Pros

- Exploits locality of disk requests
- Higher throughput

Cons

- Starvation
- Don’t always know what request will be fastest

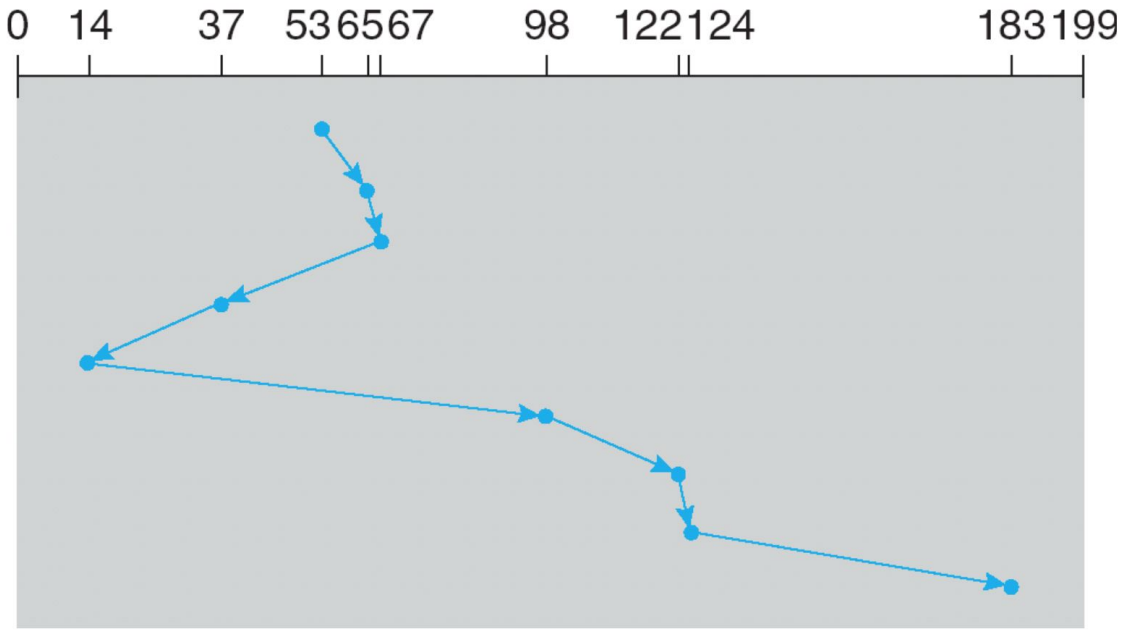
Improvement?

- Give older requests higher priority
- Adjust effective seek time with weighing factor

SPTF Example

queue = 98, 183, 37, 122, 14 , 124, 65, 67

head starts at 53



Elevator Scheduling

Sweep across disk, servicing all requests passed

- Like SPTF, but next seek must be in same direction
- Switch directions only if no further requests

Pros

Cons

Elevator Scheduling

Sweep across disk, servicing all requests passed

- Like SPTF, but next seek must be in same direction
- Switch directions only if no further requests

Pros

- Takes advantage of locality
- Bounded waiting

Cons

- Might miss locality SPTF could exploit

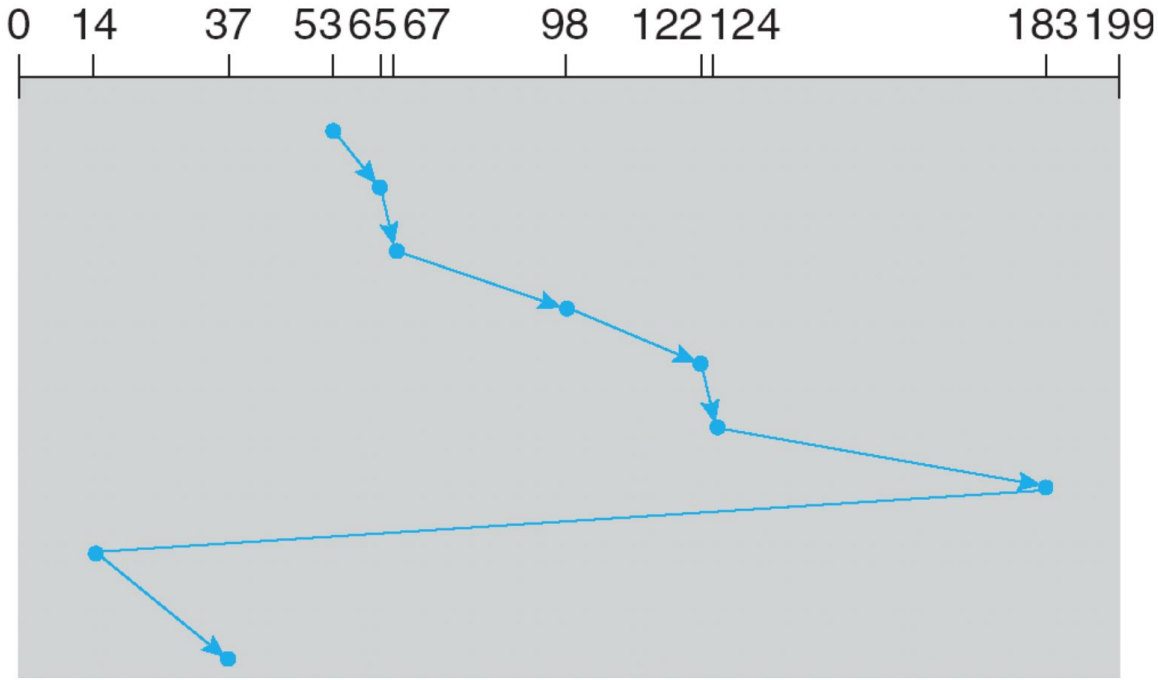
CSCAN: Only sweep in one direction – Commonly used in Linux

Also called LOOK/CLOOK

Elevator Scheduling Example

queue = 98, 183, 37, 122, 14 , 124, 65, 67

head starts at 53



Solid State Drive (SSD)

- Based on flash-NAND technology; similar to RAM but non-volatile

Composed of a grid of cells. Each cell stores a bit of data by charging the cell with low (0) or high (1) voltage. More complicated cells can store several bits by adding more granularity to the voltage charge

- Cells are grouped into pages (1KB-8KB).
- Pages are grouped into blocks (100-1000 pages).

Don’t confuse SSD terminology with memory management terminology, they are referring to different things!

Accessing an SSD – Reads :)

Reads: simply read the voltage levels of cells.

- No moving parts reduces access time and results in a less fragile and power-consumptive device
- Faster sequential read/writes, where random I/O is magnitudes faster

Takes ~25 µsec + time to get data off chip

Accessing an SSD – Writes :(

Writes: much much harder

- Back-up a block
- Erase the block
- Change some pages within the block
- Write-out the entire block

Very expensive (2 msec)

Programming pre-erased block requires moving data to internal buffer, then 200 –800 µsec

Repeated writes on the same cells will wear them out by leaving behind residual charge over time

Flash Translation Layer (FTL)

Translates LBA according to device geometry and helps prevent device wearout by “spreading out” the writes over the entire device by implementing wear-leveling.

Let’s say you have a “hot” file, how can you do this?

Flash Translation Layer (FTL)

Translates LBA according to device geometry and helps prevent device wearout by “spreading out” the writes over the entire device by implementing wear-leveling.

Let’s say you have a “hot” file, how can you do this?

- **log-structured I/O:** Instead of treating files as fixed regions of the device, store “snapshots” of the file across the device. The latest snapshot reflects the latest version of the file.
- periodically move data around (even read-only data) to help with wear-leveling.

Write amplification: writes issued to the device end up causing more internal writes.

Log-structured I/O

- This concept is not unique to SSDs – file systems were also designed in this way to maximize cheap sequential access on HDDs.
- For SSDs, this means don’t update blocks in-place, we copy them and write the new version somewhere else so that no one block is overused.
- Requires garbage collection – clearing out older snapshots so the blocks can be reused.

FTL straw man: in-memory map

Keep in-memory map of logical → physical page #

- On write, pick unused page, mark previous physical page free
- Repeated writes of a logical page will hit different physical pages

Idea: Put header on each page, scan all headers on power-up:

⟨logical page #, Allocated bit, ,Written bit, Obsolete bit⟩

- A-W-O = 0-0-0: free page
- A-W-O = 1-0-0: about to write page
- A-W-O = 1-1-0: successfully written page
- A-W-O = 1-1-1: obsolete page (can erase block without copying)

Why the 0-1-1 state?

What's wrong still?

FTL straw man: in-memory map

Keep in-memory map of logical → physical page #

- On write, pick unused page, mark previous physical page free
- Repeated writes of a logical page will hit different physical pages

Idea: Put header on each page, scan all headers on power-up:

⟨logical page #, Allocated bit, ,Written bit, Obsolete bit⟩

- A-W-O = 0-0-0: free page
- A-W-O = 1-0-0: about to write page
- A-W-O = 1-1-0: successfully written page
- A-W-O = 1-1-1: obsolete page (can erase block without copying)

Why the 0-1-1 state? After power failure partly written ≠ free

What's wrong still?

FTL straw man: in-memory map

Keep in-memory map of logical → physical page #

- On write, pick unused page, mark previous physical page free
- Repeated writes of a logical page will hit different physical pages

Idea: Put header on each page, scan all headers on power-up:

⟨logical page #, Allocated bit, ,Written bit, Obsolete bit⟩

- A-W-O = 0-0-0: free page
- A-W-O = 1-0-0: about to write page
- A-W-O = 1-1-0: successfully written page
- A-W-O = 1-1-1: obsolete page (can erase block without copying)

Why the 1-0-0 state? After power failure partly written ≠ free

What's wrong still?

- FTL requires a lot of RAM on device, plus time to scan all headers
- Some blocks still get erased more than others (w. long-lived data)
- Blocks with obsolete pages may also contain live pages

More realistic FTL

Store the FTL map in the flash device itself

- Add one header bit to distinguish map page from data page
- Logical read may miss map cache, require 2 flash reads
- Keep smaller “map-map” in memory, cache some map pages

Must garbage-collect blocks with obsolete pages

- Copy live pages to a new block, erase old block
- Always need free blocks, can’t use 100% physical storage

Problem: write amplification

- Small random writes punch holes in many blocks
- If small writes require garbage-collecting a 90%-full block. . . means you are writing 10× more physical than logical data!

Must also periodically re-write even blocks w/o holes

- *Wear leveling* ensures active blocks don’t wear out first

Summary: SSDs vs. HDDs

- **Pros**
  - No moving parts – less fragile, less power hungry
  - Faster sequential read/write
  - Orders of magnitude faster random read/write
- **Cons**
  - Expensive
  - Slow erase
  - Limited life span