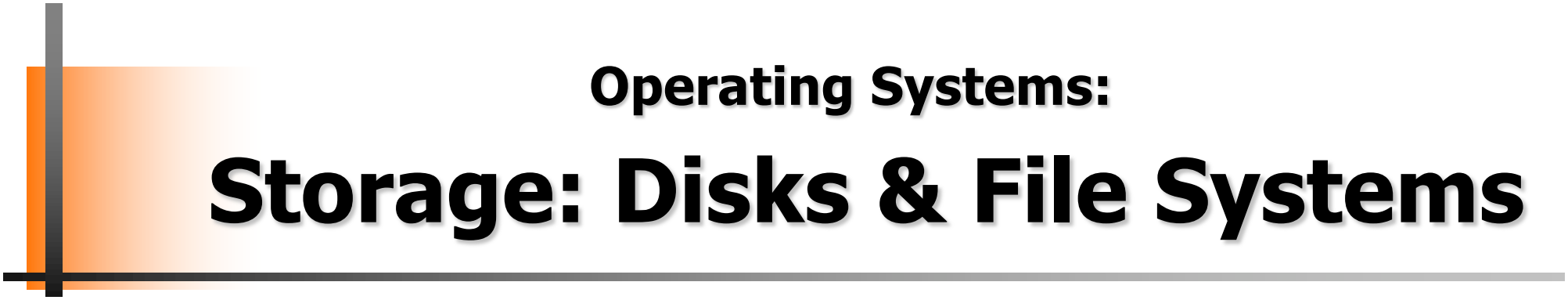
IN2140:
Introduction to Operating Systems and Data Communication



Operating Systems:

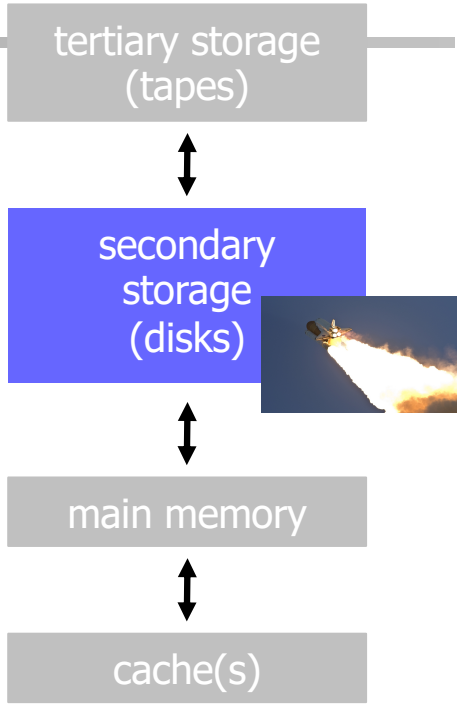
Storage: Disks & File Systems

Overview

- (Mechanical) Disks
- Disk scheduling
- Memory/buffer caching
- File systems
- Some trends and "new" technologies ...

Disks

- Disks ...
  - are used to have a **persistent system**
  - ☺ are cheaper compared to main memory
  - ☺ have more capacity
  - ☹ are orders of magnitude slower
- Two resources of importance
  - storage space
  - I/O bandwidth
- We must look closer on how to manage disks, because...
  - ...there is a large speed-mismatch (ms vs. ns) compared to main memory
  - ...disk I/O is often the main performance bottleneck

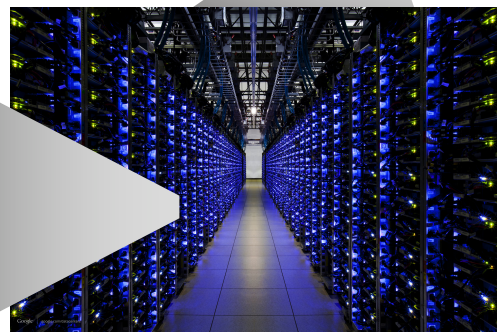


Why spend a lecture talking about HDDs?

- SSDs are persistent and
  - "almost like memory"
    (no mechanical parts)
  - much faster
    (ms vs µs)
  - but, more expensive
    (price per byte, but also shorter lifetime)
- Many devices:
  - **Google 2012**
    - ✓ 417,600 servers - Douglas County, USA
    - ✓ 204,160 servers - The Dalles, USA
    - ✓ 241,280 servers - Council Bluffs, USA
    - ✓ 139,200 servers – Lenoir, USA
    - ✓ 250,560 servers - Moncks Corner, USA
    - ✓ 296,960 servers - St. Ghislain, Belgium
    - ✓ 116,000 servers - Hamina, Finland
    - ✓ 125,280 servers - Mayes C
  - **Go**
    - ✓ , Ireland
    - ✓ vest, Singapore (projected estimate)
    - ✓ Kowloon, Hong Kong (projected estimate)
    - ✓ servers - Mayes County, USA
  - Estimated grand total: 2,376,640
    (early 2013 – 3 more centers in 2019)
  - one 0.5 TB SSD in each
    - Seagate HDD a 1.4 billion NOK
    - Intel P 17.7 billion NOK
  - one 4 T each
    - Seagate HDD at Komplett: 4.1 billion NOK
    - Samsung SSD at komplett: 17.8 billion NOK (2 TB)
    - Intel P3608 SSD: 160 billion NOK

~10x price

Google data center locations (2019):



Mechanical HDDs will exist for a long time....!

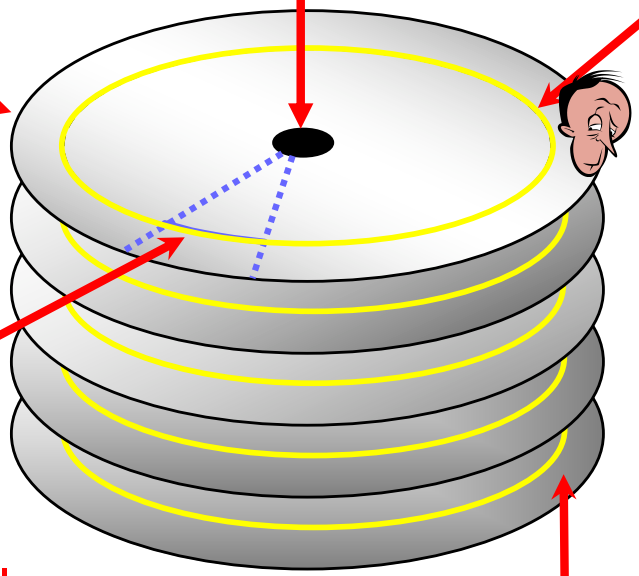
Mechanics of Disks

Mechanics of Disks

Platters
circular platters covered with magnetic material to provide nonvolatile storage of bits

Spindle
of which the platters rotate around

Tracks
concentric circles on a single platter



Disk heads
read or alter the magnetism (bits) passing under it. The heads are attached to an arm enabling it to move across the platter surface

Sectors
segment of the *track* circle – usually each contains 512 bytes – separated by non-magnetic gaps. The gaps are often used to identify beginning of a sector

Cylinders
corresponding tracks on the different platters are said to form a cylinder

Disk Capacity

- The size (storage space) of the disk is dependent on
  - the number of platters
  - whether the platters use one or both sides
  - number of tracks per surface
  - (average) number of sectors per track
  - number of bytes per sector

- Example (Cheetah X15.1):
  - 4 platters using both sides: 8 surfaces
  - 18497 tracks per surface
  - 617 sectors per track (average)
  - 512 bytes per sector
  - Total capacity = 8 x 18497 x 617 x 512 ≈ 4.6 x 10^{10} = 42.8 GB
  - Formatted capacity = 36.7 GB

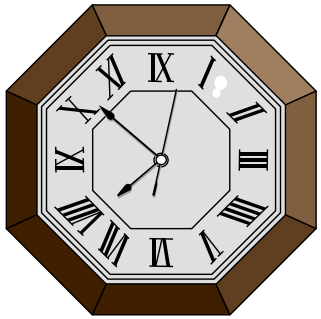
Note:
there is a difference between formatted and total capacity. Some of the capacity is used for storing checksums, **spare tracks**, etc.

Disk Access Time

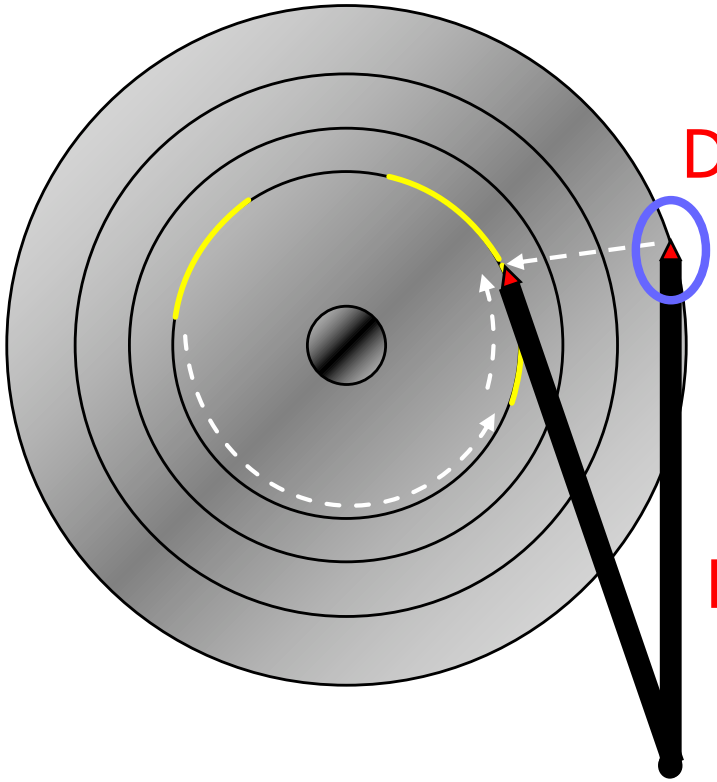
- How do we retrieve data from disk?
  - position head over the cylinder (track) on which the block (consisting of one or more sectors) is located
  - read (or write) the data block as the sectors are moved under the head when the platters rotate

- The time between the moment issuing a disk request and the time the block is resident in memory is called *disk latency* or *disk access time*

Disk Access Time

I want block X →  → block x in memory

Disk platter



Disk access time =

Seek time

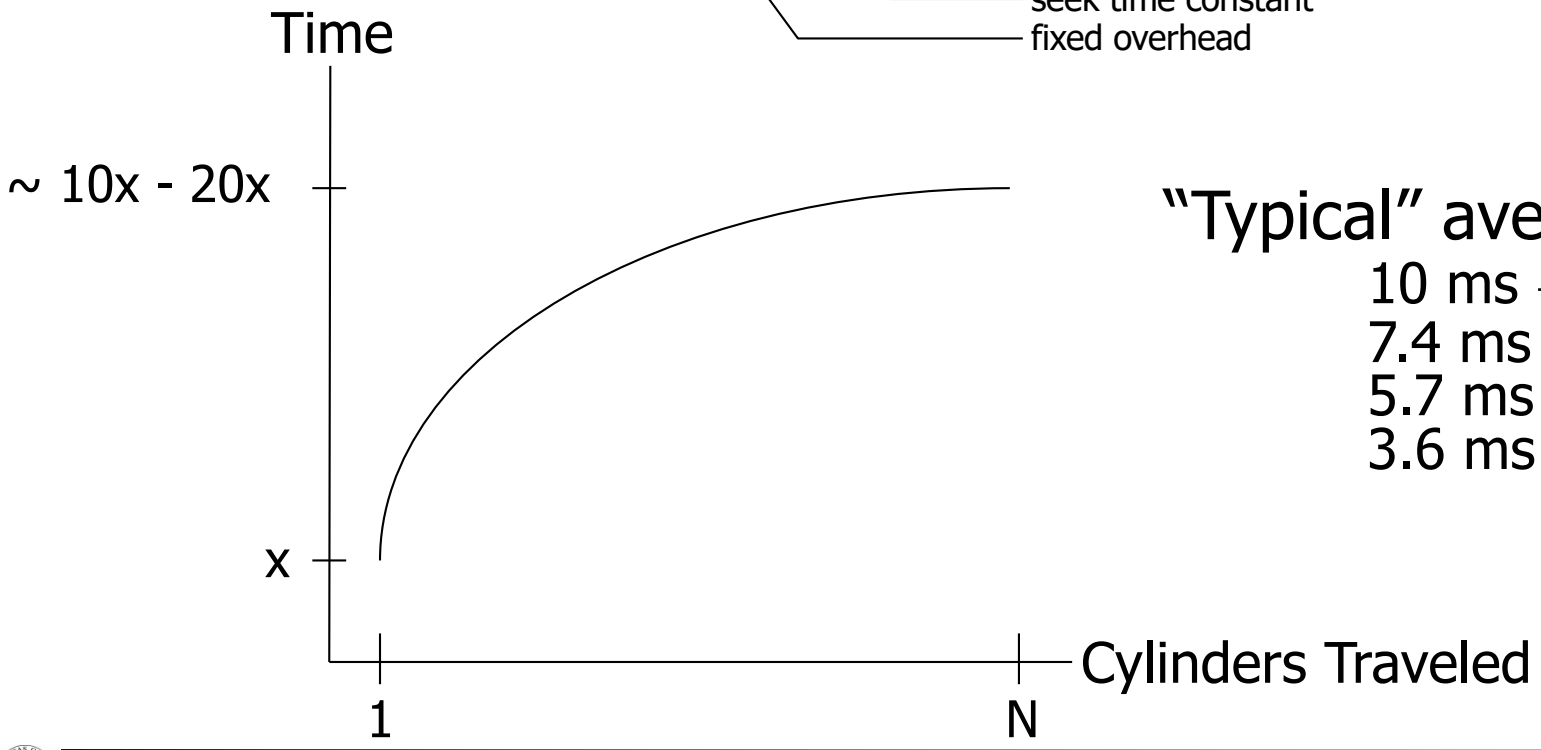
\+ Rotational delay

\+ Transfer time

\+ Other delays

Disk Access Time: Seek Time

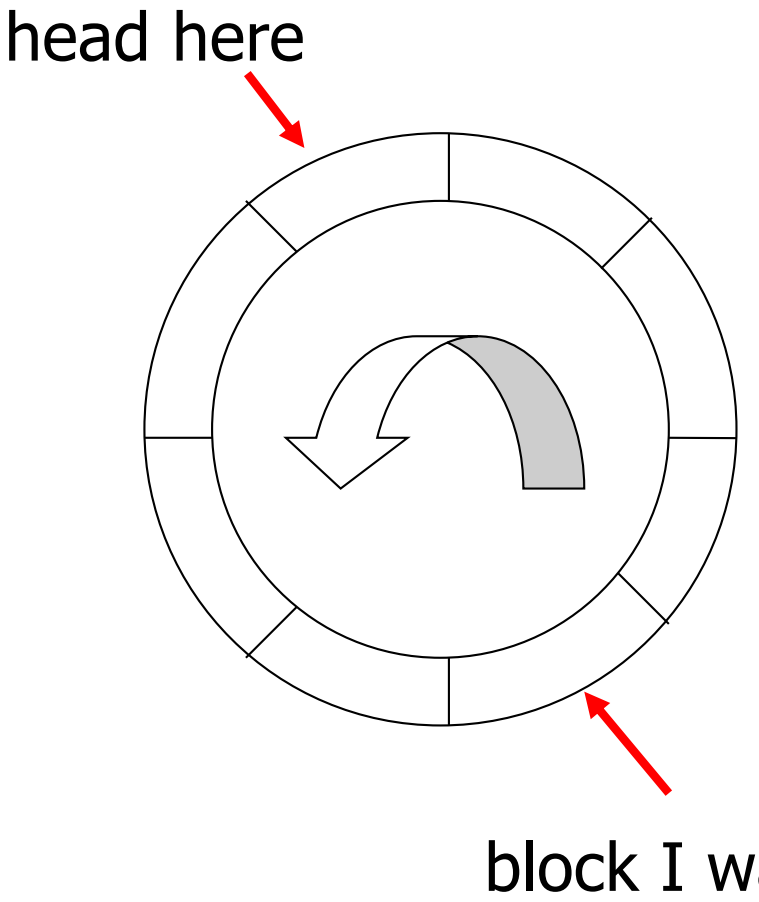
- Seek time is the time to position the head
  - time is used for actually moving the head – roughly proportional to the number of cylinders traveled
  - time to start and stop moving the head
  - time to move head: $\alpha + \beta\sqrt{n}$
    - n: number of tracks
    - β: seek time constant
    - α: fixed overhead



"Typical" average:
10 ms → 40 ms (old)
7.4 ms (Barracuda 180)
5.7 ms (Cheetah 36)
3.6 ms (Cheetah X15)

Disk Access Time: Rotational Delay

- Time for the disk platters to rotate so the first of the required sectors are under the disk head



Average delay is 1/2 revolution

"Typical" average:

8.33 ms	(3.600 RPM)
5.56 ms	(5.400 RPM)
4.17 ms	(7.200 RPM)
3.00 ms	(10.000 RPM)
2.00 ms	(15.000 RPM)

Disk Access Time: Transfer Time

- Time for data to be read by the disk head, i.e., time it takes the sectors of the requested block to rotate under the head
- Transfer time is dependent on data density and rotation speed
- $\text{Transfer rate} = \frac{\text{amount of data per track}}{\text{time per rotation}}$
- $\text{Transfer time} = \frac{\text{amount of data to read}}{\text{transfer rate}} = \frac{\text{amount of data to read} * \text{time per rotation}}{\text{amount of data per track}}$
- Transfer rate example
  - *Barracuda 180:*
    406 KB per track x 7.200 RPM ≈ 47.58 MB/s
  - *Cheetah X15:*
    306 KB per track x 15.000 RPM ≈ 77.15 MB/s

Note:
one might achieve these transfer rates reading continuously on disk, but time must be added for seeks, etc.

- If we have to change track, time must also be added for moving the head

Disk Access Time: Other Delays

- There are several other factors which might introduce additional delays:
  - CPU time to issue and process I/O
  - contention for controller, bus, memory
  - verifying block correctness with checksums (retransmissions)
  - waiting in scheduling queue
  - ...

- Typical values: "**0**"
  (maybe except from waiting in the scheduling queue)

Disk Specifications

- Some existing (Seagate) disks:

Note 1:
disk manufacturers usually denote GB as 10^9 whereas computer quantities often are powers of 2, i.e., GB is 2^{30}

	Barracuda 180	*Cheetah 36*	*Cheetah X15.3*
Capacity (GB)	181.6	36.4	73.4
Spindle speed (RPM)	7200	10.000	15.000
#cylinders	24.247	9.772	18.479
average seek time (ms)	7.4	5.7	3.6
min (track-to-track) seek (ms)	0.8	0.6	0.2
max (full stroke) seek (ms)	16	12	7
average latency (ms)	4.17	3	2
internal transfer rate (Mbps)	282 – 508	520 – 682	609 – 891

Note 2:
there is a difference between internal and formatted transfer rate. ***Internal*** is only between platter. ***Formatted*** is after the signals interfere with the electronics (cabling loss, interference, retransmissions, checksums, etc.)

Note 3:
there is usually a trade off between speed and capacity

Writing and Modifying Blocks

- A write operation is analogous to read operations
  - must potentially add time for block allocation
  - a complication occurs if the write operation has to be *verified* – must usually wait another rotation and then read the block again
  - Total write time ≈ read time (+ time for one rotation)

- A modification operation is similar to read and write operations
  - cannot modify a block directly:
    - **read** block into main memory
    - modify the block
    - **write** new content back to disk
  - Total modify time ≈ read time (+ time to modify) + write time

Disk Controllers

- To manage the different parts of the disk, we use a *disk controller*, which is a small processor capable of:
  - controlling the actuator moving the head to the desired track
  - selecting which head (platter and surface) to use
  - knowing when the right sector is under the head
  - transferring data between main memory and disk

Efficient Secondary Storage Usage

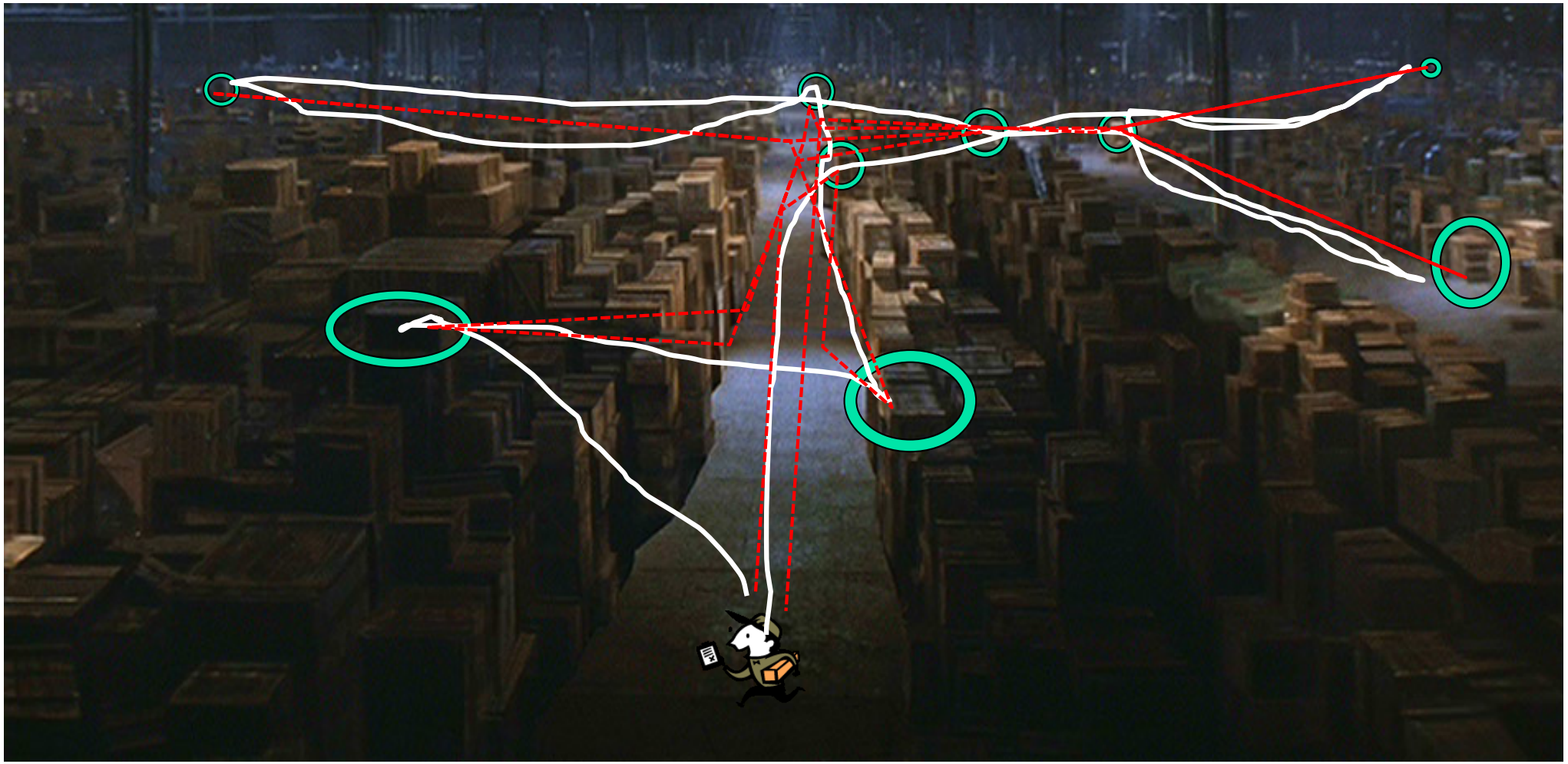
- Must take into account the use of secondary storage
  - large gaps in access times between disks and memory,
    i.e., a disk access will probably dominate the total execution time
  - huge performance improvements if we reduce the number of disk accesses
  - a "slow" algorithm with few disk accesses will probably outperform
    a "fast" algorithm with many disk accesses

- **Several ways to optimize .....**
  - block size - 4 KB
  - disk scheduling - SCAN derivate
  - file management / data placement - various
  - memory caching / replacement algorithms - LRU variant
  - prefetching - read-ahead
  - multiple disks - a specific RAID level
  - ...

Disk Scheduling

Disk Scheduling

- How to most efficiently fetch the parcels I want?



Disk Scheduling

- **Seek time is the dominant factor of the total disk I/O time**

- **IDEA:** Let the **operating system** or disk controller choose which request to serve next depending on the *head's current position* and *requested block's position* on disk (disk scheduling)

- Note that disk scheduling ≠ CPU scheduling
  - a mechanical device – hard to determine (accurate) access times
  - disk accesses can/should *not be preempted* – run until they finish

- General goals
  - short response time
  - high overall throughput
  - fairness (equal probability for all blocks to be accessed in the same time)

- Tradeoff: seek efficiency vs. maximum response time

Disk Scheduling

- Several traditional algorithms
  - First-Come-First-Serve (FCFS)
  - Shortest Seek Time First (SSTF)
  - SCAN (and variations)
  - Look (and variations)
  - ...
- A **LOT** of different algorithms exist depending on expected access pattern

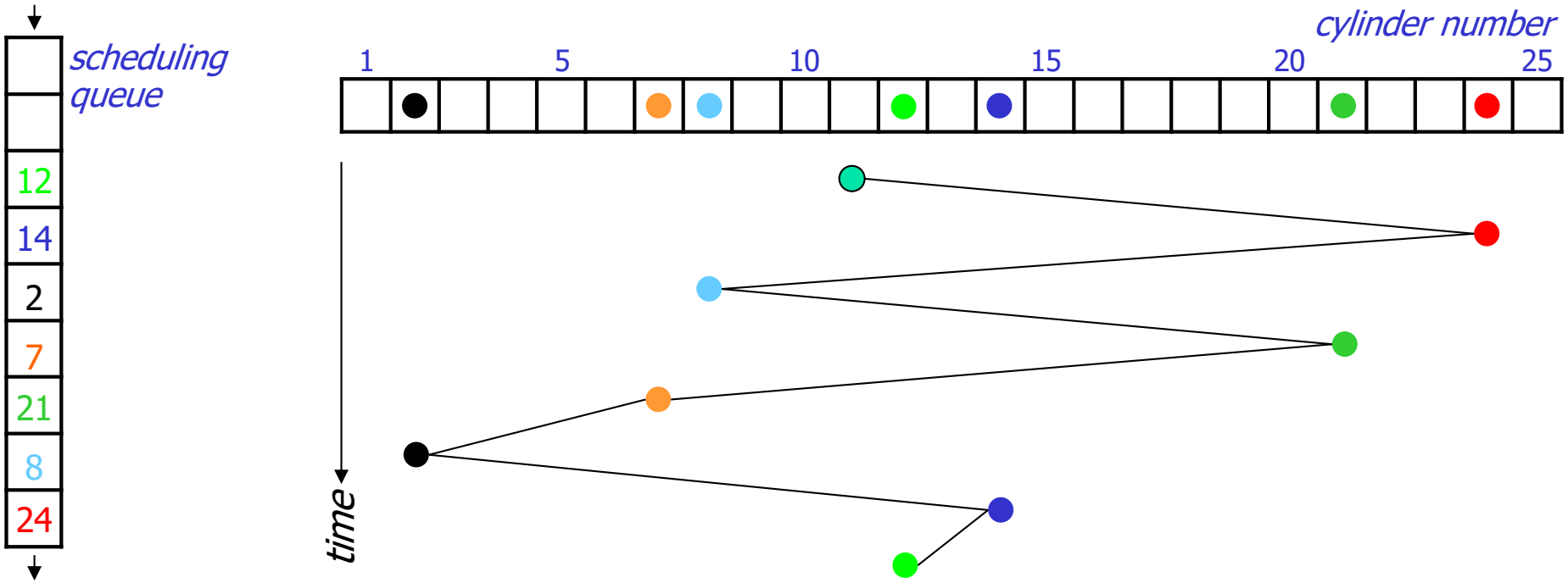
First–Come–First–Serve (FCFS)

FCFS (FIFO) serves the first arriving request first:

- Long seeks
- "Short" response time for all

incoming requests (in order of arrival, denoted by cylinder number):

12 14 2 7 21 8 24



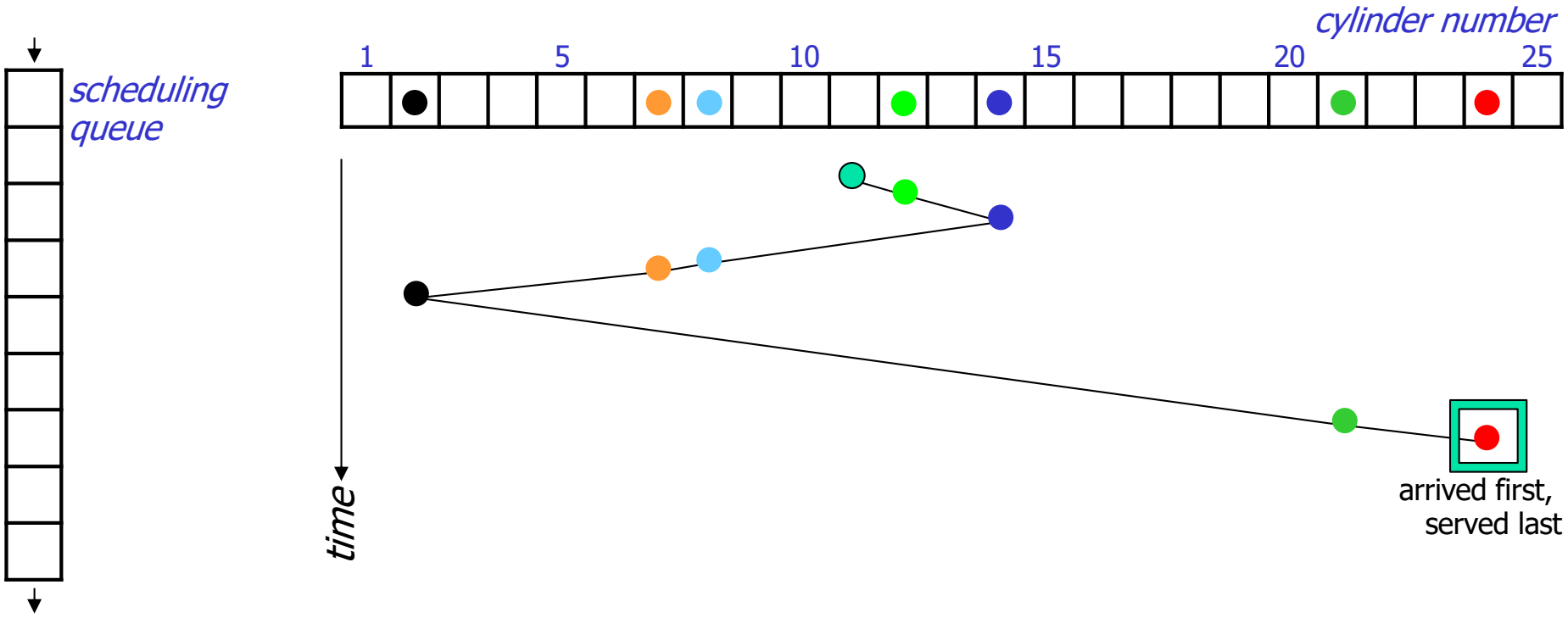
Shortest Seek Time First (SSTF)

SSTF serves closest request first:

- short seek times
- longer maximum response times – **may even lead to starvation**

incoming requests (in order of arrival):

12 14 2 7 21 8 24



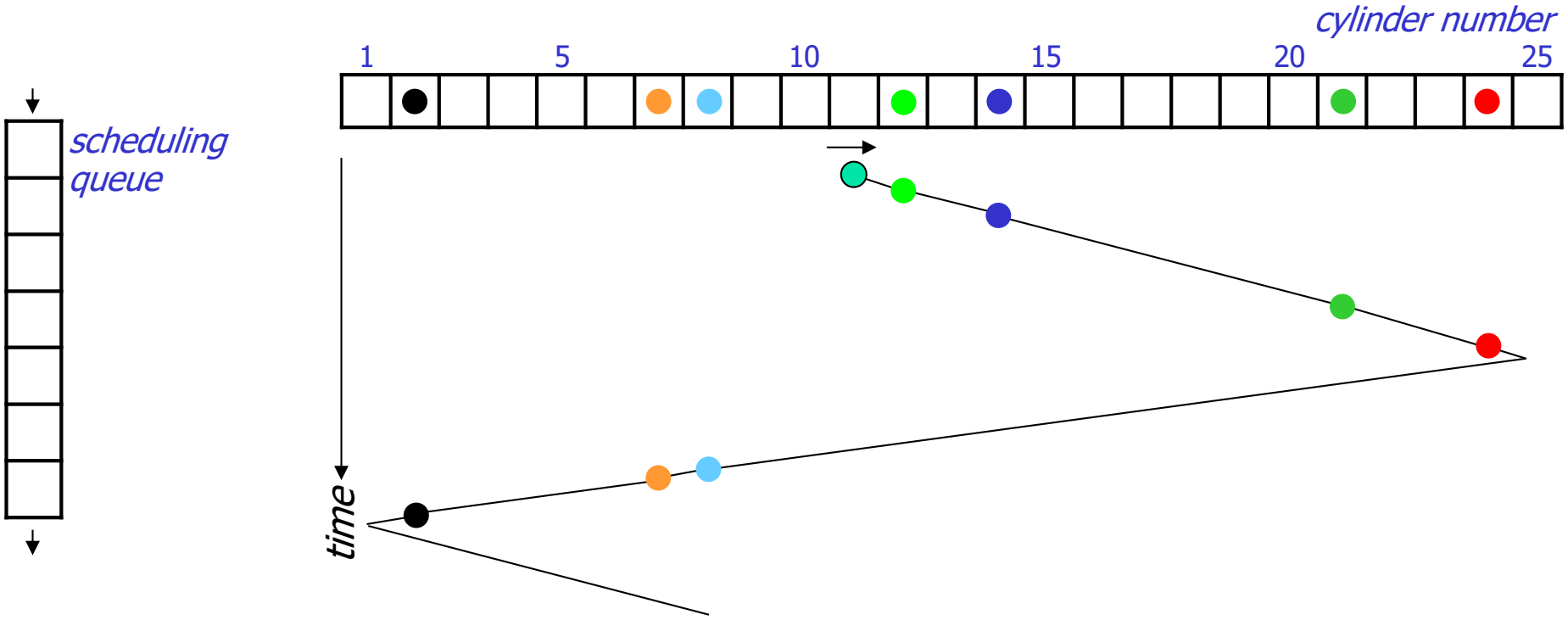
SCAN

SCAN (elevator) moves head edge to edge and serves requests on the way:

- bi-directional
- compromise between response time and seek time optimizations

incoming requests (in order of arrival):

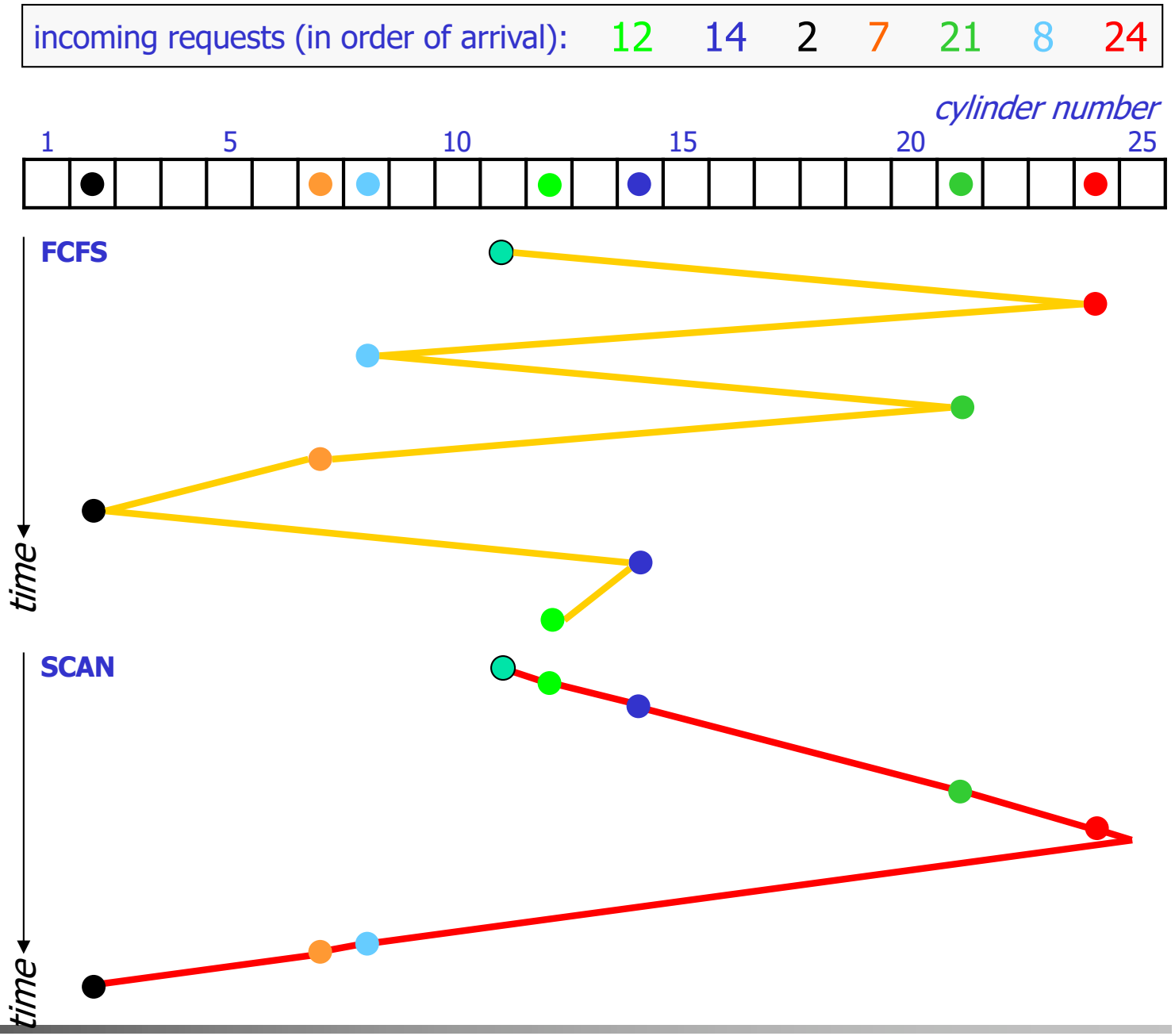
12 14 2 7 21 8 24



SCAN vs. FCFS

- Disk scheduling makes a difference!

- In this case, we see that SCAN requires much less head movement compared to FCFS
  - here 37 vs. 75 tracks
  - imagine having
    - 20.000++ tracks
    - many users
    - many files
    - …



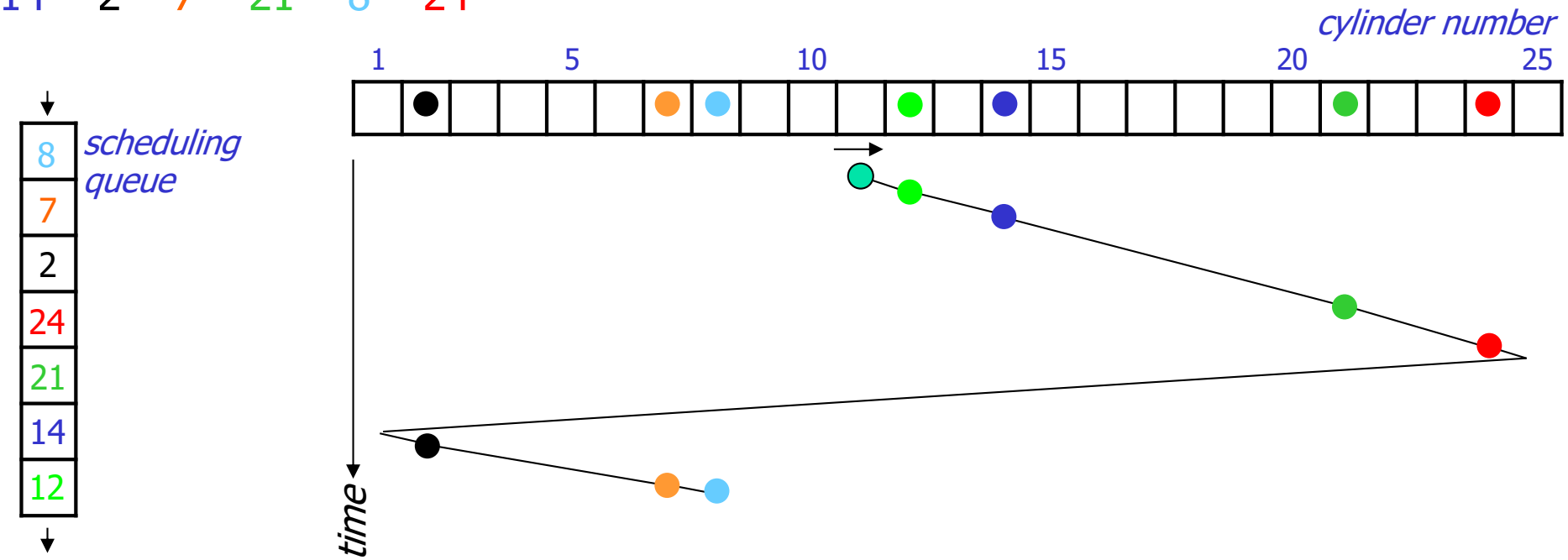
C–SCAN

Circular-SCAN moves head from edge to edge

- optimization of SCAN
- serves requests on one way – uni-directional
- improves response time (fairness)

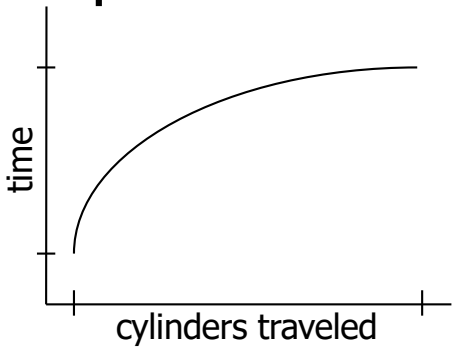
incoming requests (in order of arrival):

12 14 2 7 21 8 24



SCAN vs. C–SCAN

- Why is C-SCAN in average better in reality than SCAN when both service the same number of requests in two passes?
  - modern disks must accelerate (speed up and down) when seeking
  - head movement formula: $\alpha + \beta\sqrt{c}$
    - c: number of cylinders
    - β: seek time constant
    - α: fixed overhead



SCAN	C-SCAN
bi-directional 	uni-directional 
requests: n avg. dist: 2x (spread over both directions) total cost: $n \times \sqrt{2x} = (n \times \sqrt{2}) \times \sqrt{x}$	requests: n avg. dist: x (over one direction only + one full pass) total cost: $\sqrt{n \times x} + n \times \sqrt{x} = (\sqrt{n} + n) \times \sqrt{x}$
if n is large: $n \times \sqrt{2} > \sqrt{n} + n$	

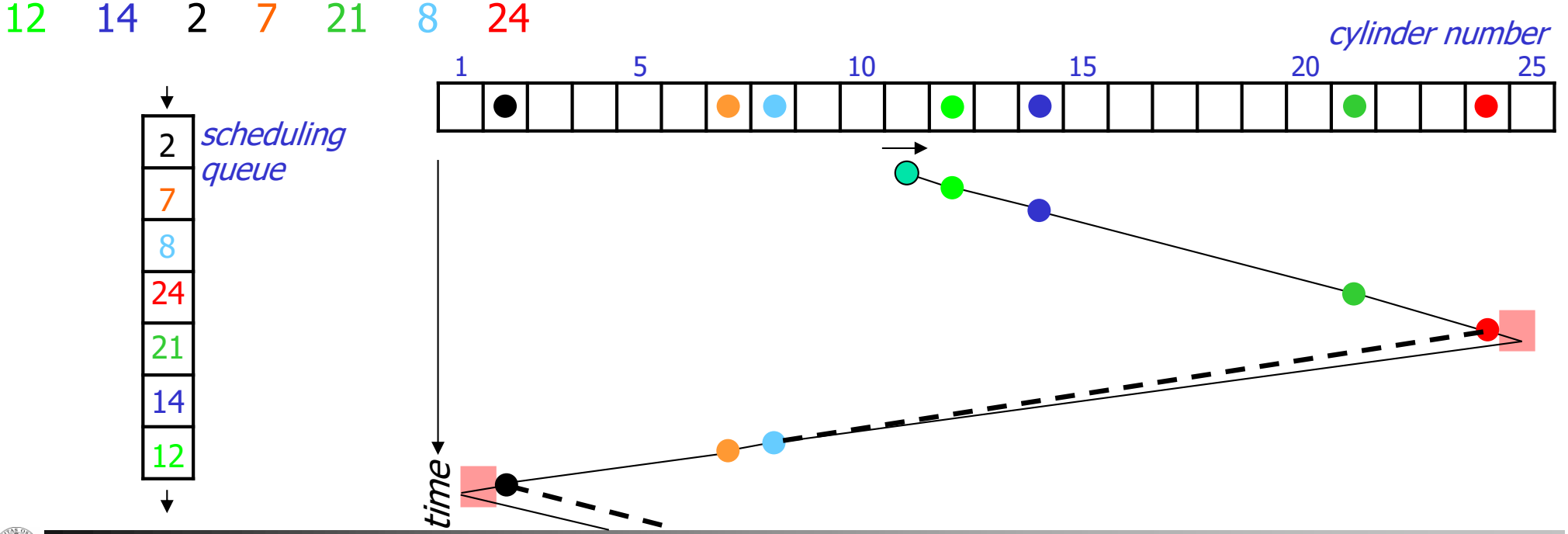
LOOK and C–LOOK

LOOK (C-LOOK) is a variation of SCAN (C-SCAN):

- same schedule as SCAN
- does not run to the edges
- stops and returns at outer- and innermost requests
- increased efficiency
- SCAN vs. LOOK example:

incoming requests (in order of arrival):

12 14 2 7 21 8 24



Modern Disk Scheduling

- Disk used to be simple devices and disk scheduling used to be performed by OS (file system or device driver) only…

- … but, new disks are more complex
  - hide their true layout, e.g.,
    - only logical block numbers
    - different number of surfaces, cylinders, sectors, etc.

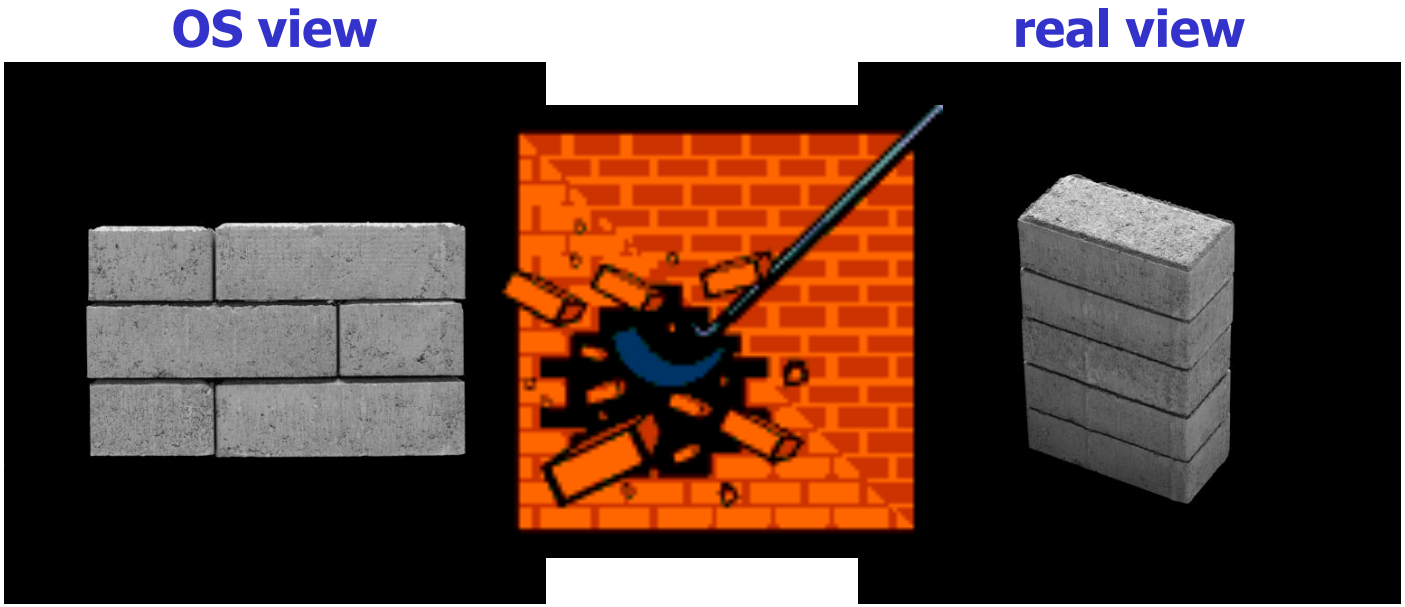
OS view

real view

Modern Disk Scheduling

- Disk used to be simple devices and disk scheduling used to be performed by OS (file system or device driver) only…

- … but, new disks are more complex
  - hide their true layout
  - transparently move blocks to spare cylinders
    - e.g., due to bad disk blocks



Modern Disk Scheduling

- Disk used to be simple devices a
performed by OS (file system or

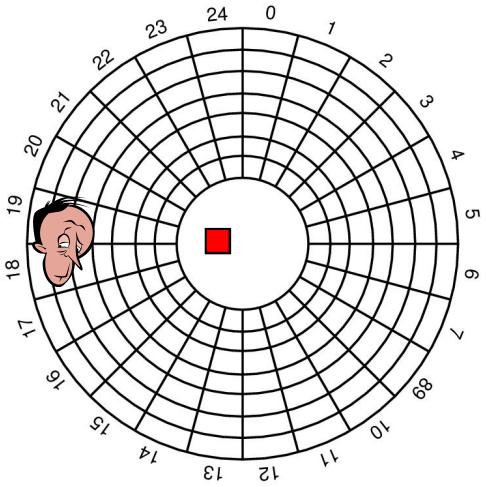
- … but, new disks are more com
  - hide their true layout
  - transparently move blocks to spare
  - have different zones

Seagate X15.3:

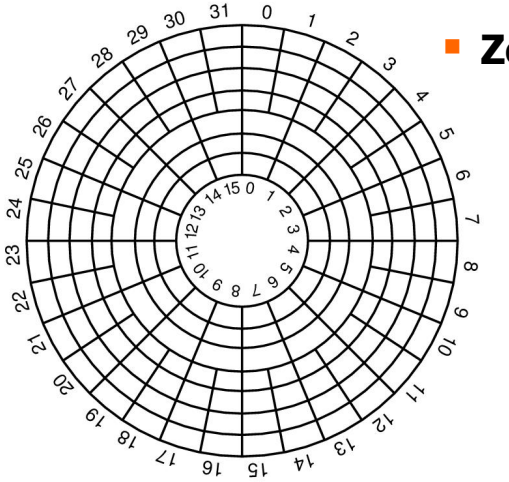
Zone	Cylinders per Zone	Sectors per Track	Zone Transfer Rate (MBps)	Sectors per Zone	Efficiency	Formatted Capacity (MB)
1	**3544**	**672**	**890,98**	19014912	77,2%	**9735,635**
2	3382	652	878,43	17604000	76,0%	9013,248
3	3079	624	835,76	15340416	76,5%	7854,293
4	2939	595	801,88	13961080	76,0%	7148,073
5	2805	576	755,29	12897792	78,1%	6603,669
6	2676	537	728,47	11474616	75,5%	5875,003
7	2554	512	687,05	10440704	76,3%	5345,641
8	2437	480	649,41	9338880	75,7%	4781,506
9	2325	466	632,47	8648960	75,5%	4428,268
10	**2342**	**438**	**596,07**	8188848	75,3%	**4192,690**

OS view

- **Constant angular velocity (CAV) disks**
  - constant rotation speed
  - equal amount of data in each track
  - ⇨ thus, constant transfer time



real view

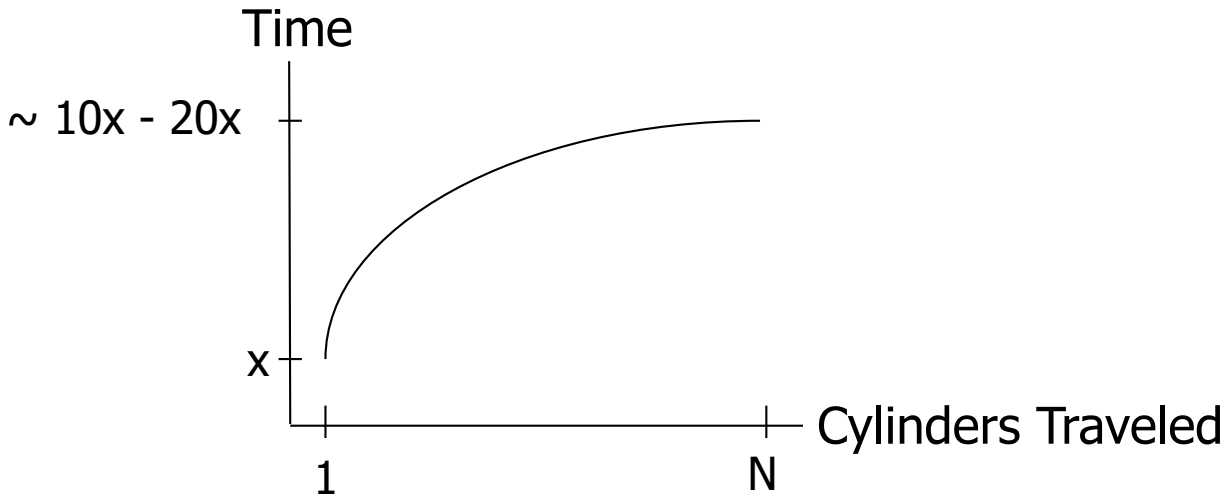


- **Zoned CAV disks**
  - constant rotation speed
  - zones are ranges of tracks
  - typical few zones
  - the different zones have different amount of data, i.e., more better on outer tracks
  - ⇨ thus, variable transfer time

Modern Disk Scheduling

- Disk used to be simple devices and disk scheduling used to be performed by OS (file system or device driver) only…

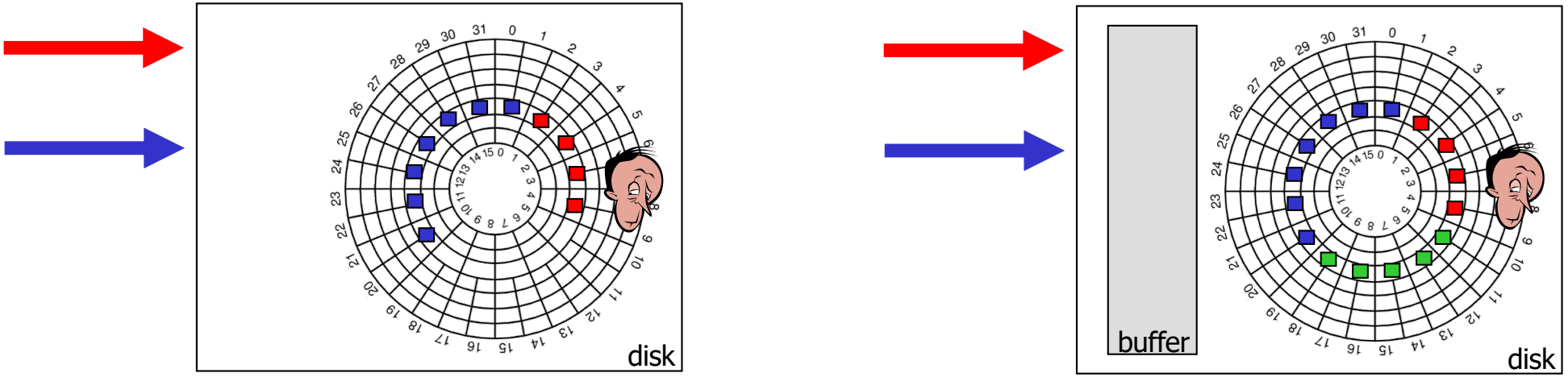
- … but, new disks are more complex
  - hide their true layout
  - transparently move blocks to spare cylinders
  - have different zones
  - head accelerates – most algorithms assume linear movement overhead



Modern Disk Scheduling

- Disk used to be simple devices and disk scheduling used to be performed by OS (file system or device driver) only…

- … but, new disks are more complex
  - hide their true layout
  - transparently move blocks to spare cylinders
  - have different zones
  - head accelerates – most algorithms assume linear movement overhead
  - on device buffer caches may use read-ahead prefetching



Modern Disk Scheduling

- Disk used to be simple devices and disk scheduling used to be performed by OS (file system or device driver) only…

- … but, new disks are more complex
  - hide their true layout
  - transparently move blocks to spare cylinders
  - have different zones
  - head accelerates – most algorithms assume linear movement overhead
  - on device buffer caches may use read-ahead prefetching

  ⇨ "smart" with build-in low-level scheduler (usually SCAN-derivate)

  ⇨ we cannot fully control the device (black box)

- OS could (should?) focus on high level scheduling only!??

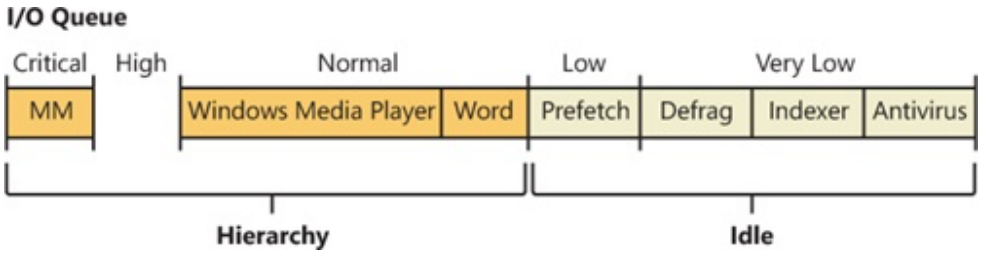
Schedulers today (Linux)?

- Elevator – SCAN
- NOOP
  - FCFS with request merging
- Deadline I/O
  - C-SCAN based
  - 3 queues: 1 sorted (elevator) queue, and 2 deadline queues (one for read and one for write)
- Anticipatory
  - same queues as in Deadline I/O
  - delays decisions to be able to merge more requests
- Completely Fair Queuing (CFQ)
  - 1 queue per process (periodic access, but period length depends on load)
  - gives time slices and ordering according to priority level (real-time, best-effort, idle)
  - selects requests from queues in RR for the final elevator sorting
  - work-conserving

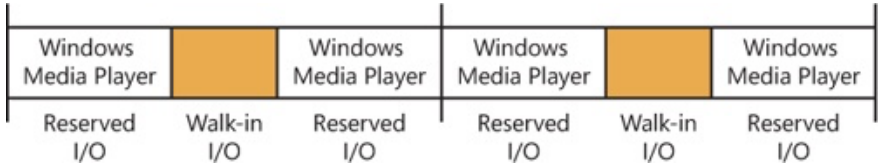
```
$> more /sys/block/sda/queue/scheduler
noop deadline [cfq]
```

Schedulers today (Windows)?

- Well, a bit hard to say, but…
- I/O priorities
  - individual I/O operations
    - FIFO within each queue
    - to avoid low-priority starvation, a timer enforce ONE I/O per time unit (.5 sec)



  - reservations



  - many special functions: fast I/O, I/O boosts and bumps, …

https://www.microsoftpressstore.com/articles/article.aspx?p=2201309&seqNum=3

Cooperative user-kernel space scheduling

- Some times the kernel does not have enough information to make an efficient schedule
  - File tree traversals
    - *processing one file after another*
    - `tar`, `zip`, ...
    - recursive copy (`cp -r`)
    - search (`find`)
    - ...
- Only application knows access pattern
  - use `ioctl FIEMAP` (`FIBMAP`) to retrieve extent/block locations
  - sort in user space
  - send I/O request according to sorted list

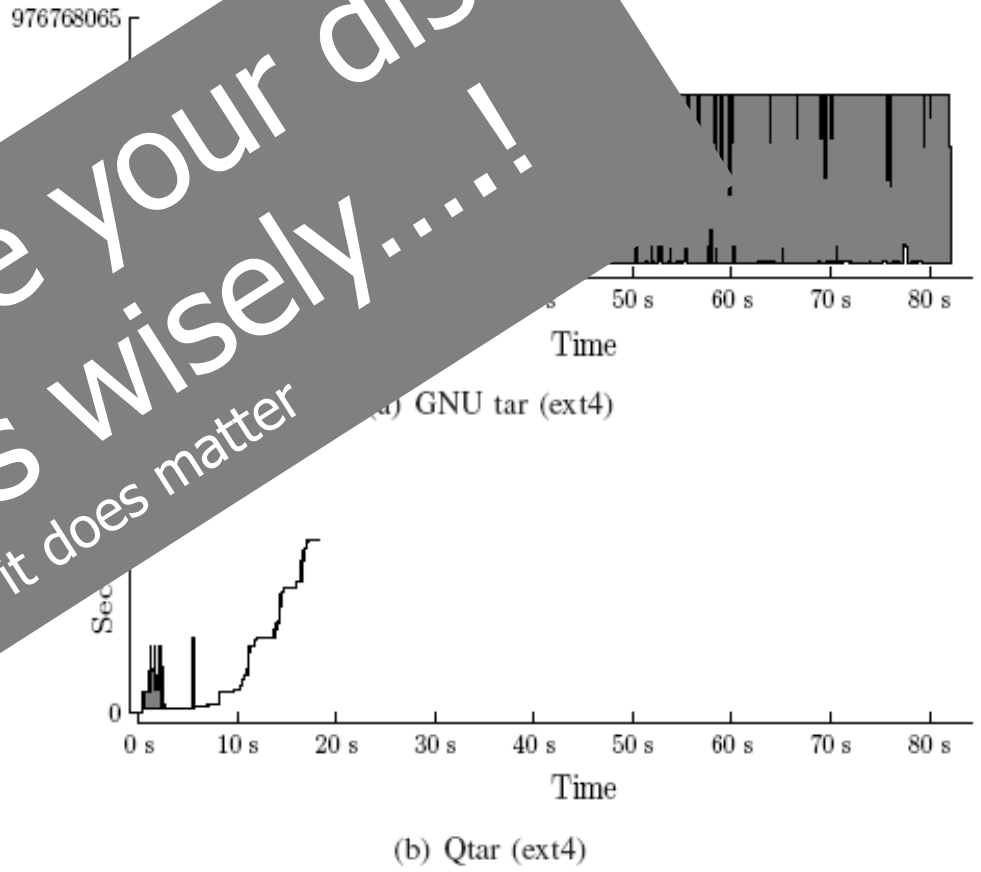
⇨ **GNU/BSD Tar** vs. **QTAR**

Cooperative user-kernel space scheduling

- Some times the kernel does not have enough information to make an efficient schedule
  - File tree traversals
    - *processing one file after another*
    - `tar`, `zip`, …
    - recursive copy (`cp -r`)
    - search (`find`)
    - …
- Only application
  access patt
    - use `ioct`
      to retrieve
    - sort in user sp
    - send I/O reque ing to
      sorted list

So, schedule your disk requests wisely....!

→ it does matter

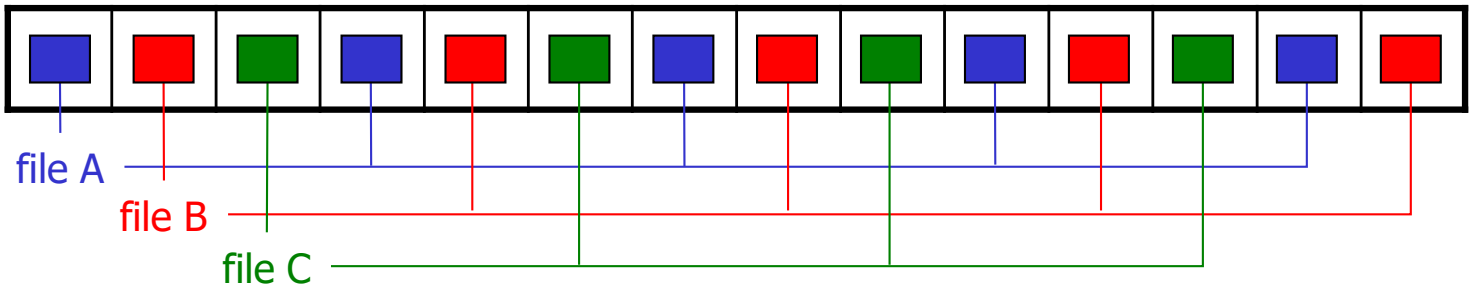


⇨ **GNU/BSD Tar** vs. **QTAR**

Data Placement

Data Placement on Disk

- Interleaved placement tries to store blocks from a file with a fixed number of other blocks in-between each block

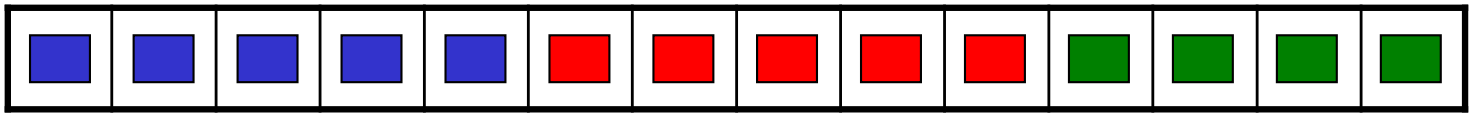


  - minimal disk arm movement reading the files A, B and C (starting at the same time)
  - fine for predictable workloads reading multiple files
  - no gain if we have unpredictable disk accesses

- Non-interleaved (or even random) placement can be used for highly unpredictable workloads

Data Placement on Disk

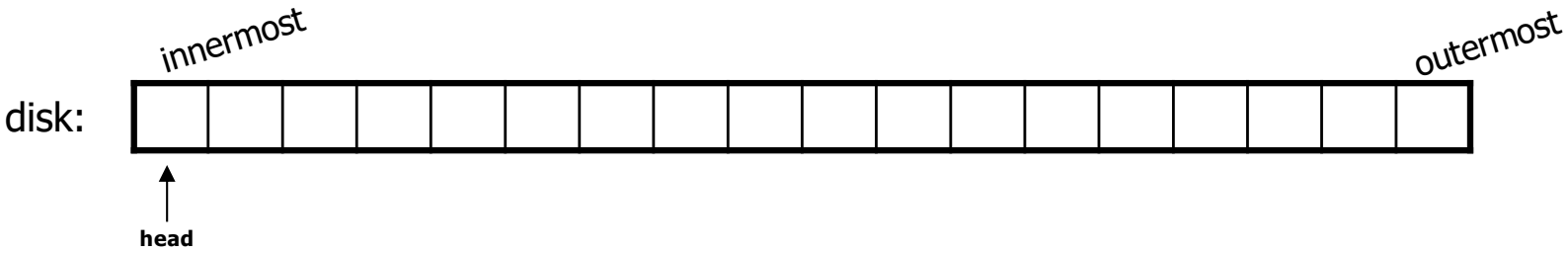
- Contiguous placement stores disk blocks contiguously on disk



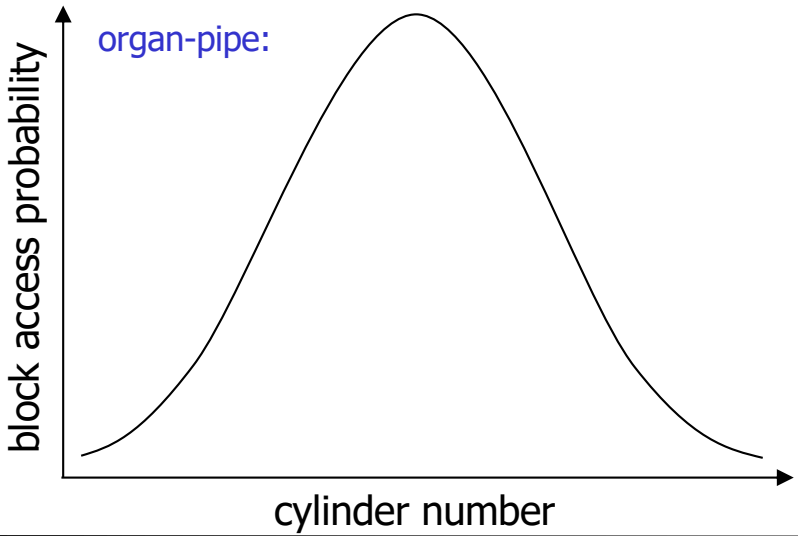
  - minimal disk arm movement reading the whole file (no intra-file seeks)
  - pros/cons
    - ☺ within a file, head must not move between reads - no seeks / rotational delays
    - ☺ can approach theoretical transfer rate
    - ☹ but usually we read other files as well (giving possible large inter-file seeks)
  - real advantage
    - whatever amount to read, at most track-to-track seeks are performed within one request
  - no inter-operation gain if we have unpredictable disk accesses (but still not worse than random placement)

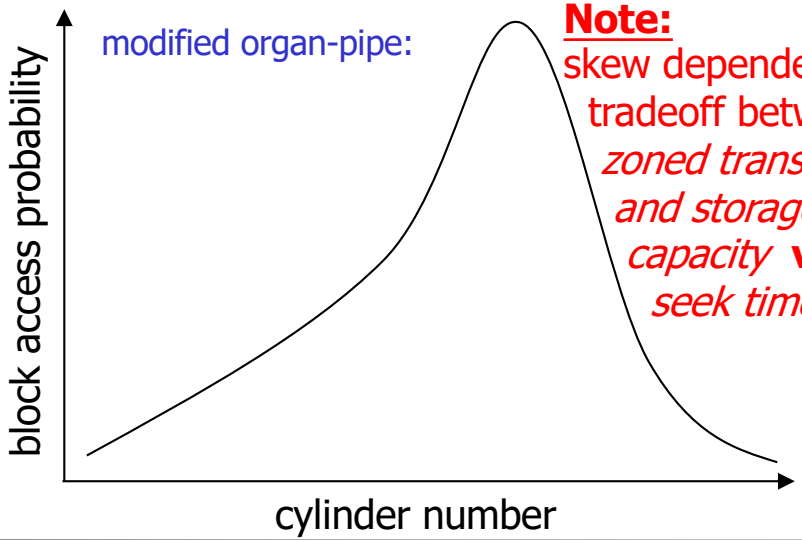
Data Placement on Disk

- Organ-pipe placement consider the 'average' disk head position
  - place most popular data where head is most often

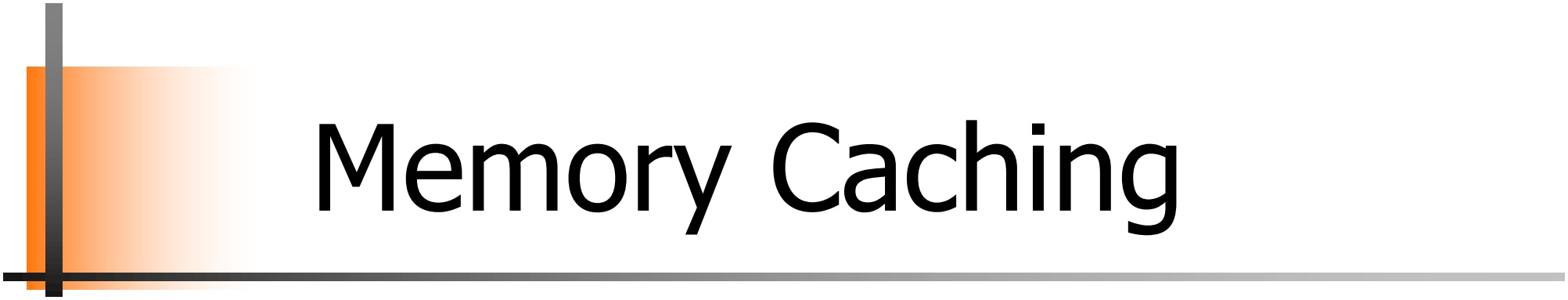


  - center of the disk is in average "closest" to the head
  - but, a bit outward for *zoned* disks (modified organ-pipe)

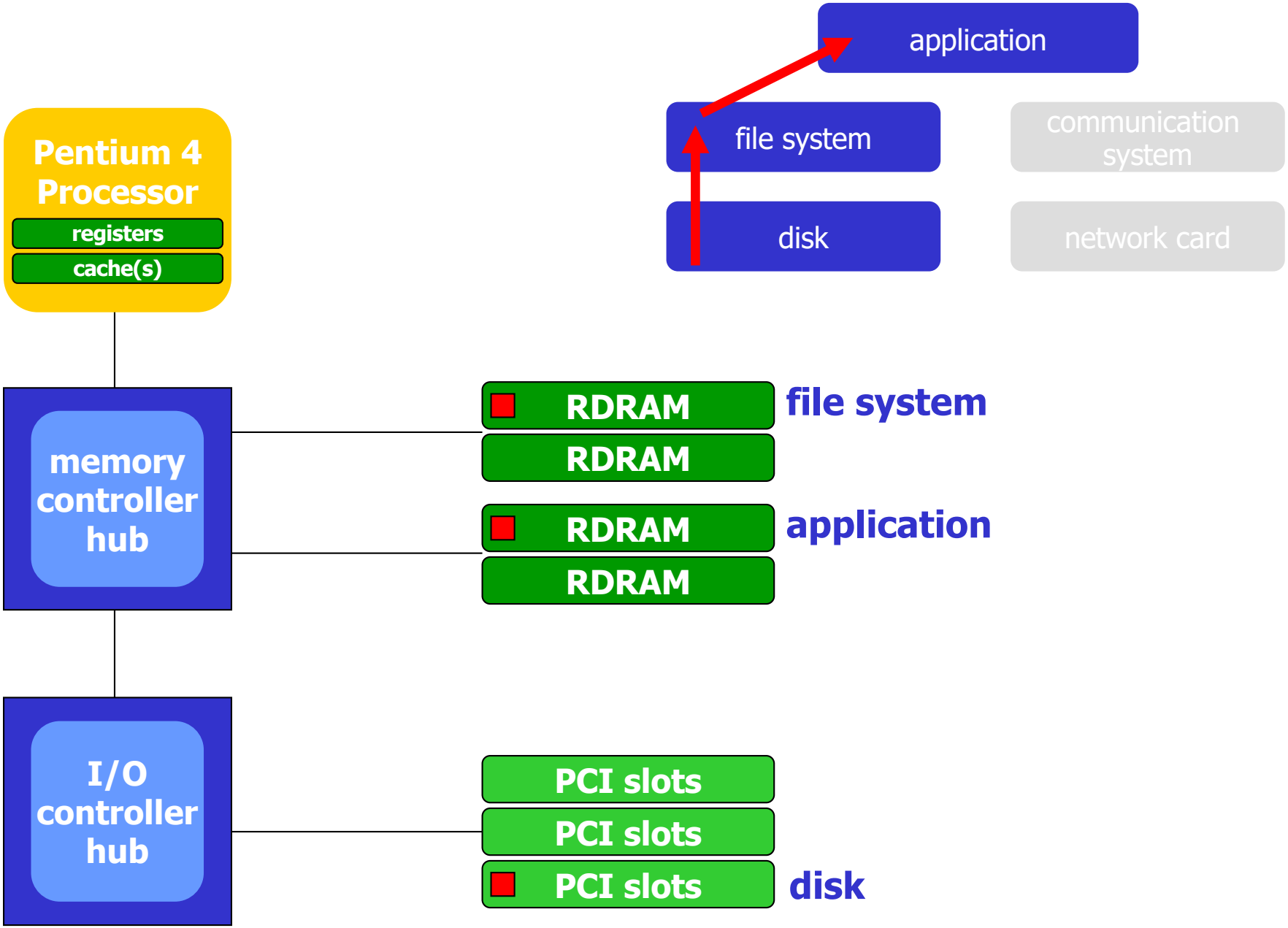




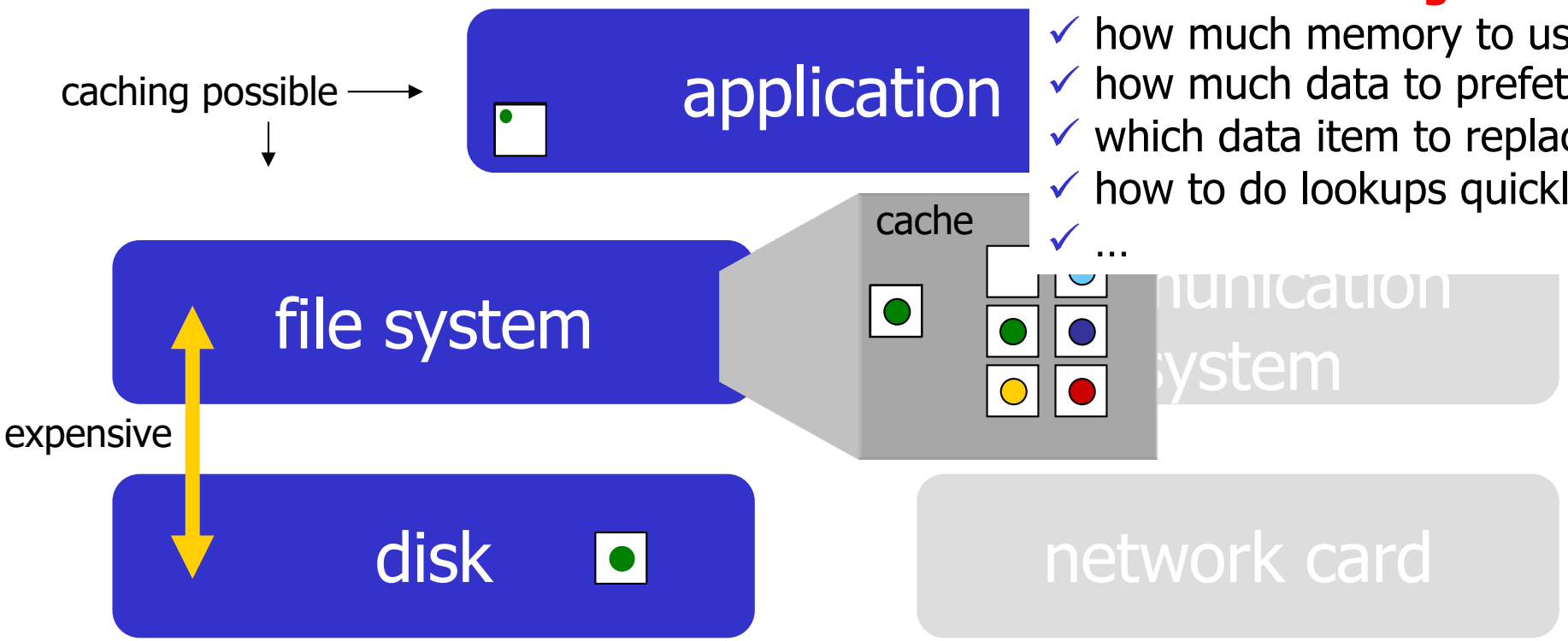
Note:
skew dependent on tradeoff between *zoned transfer time and storage capacity* **vs.** *seek time*


Memory Caching

Data Path (Intel Hub Architecture)



Buffer Caching

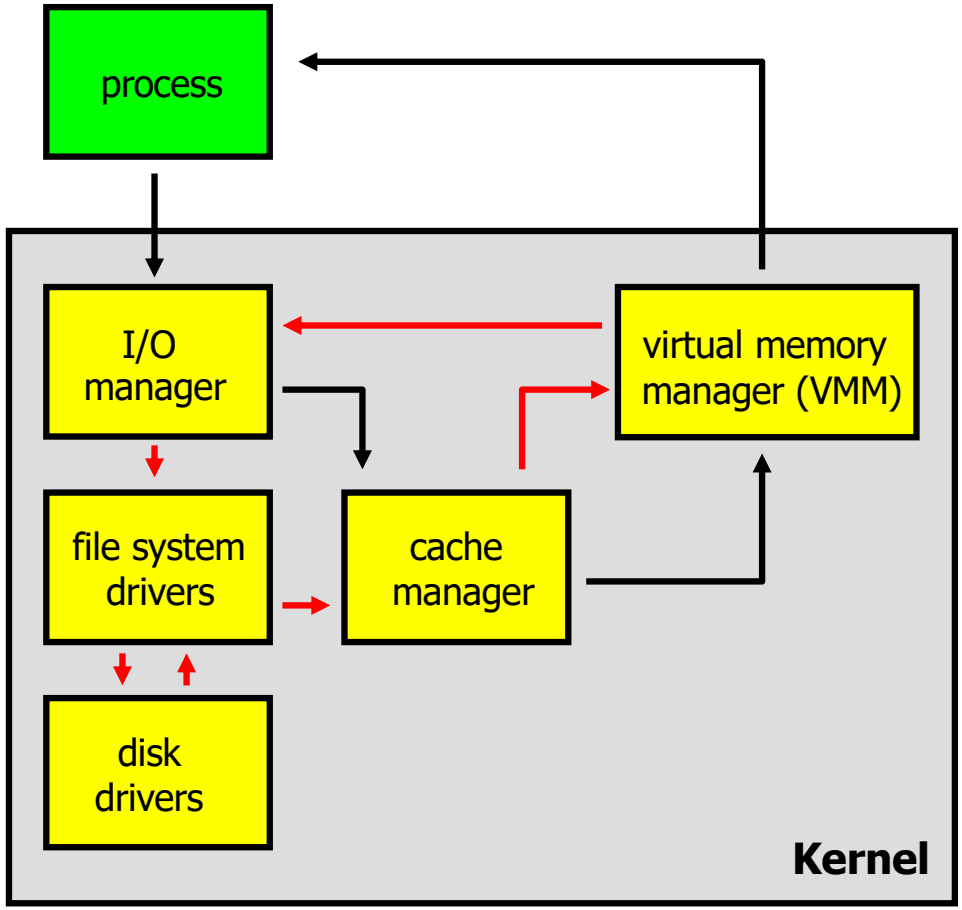


How do we manage a cache?

- ✓ how much memory to use?
- ✓ how much data to prefetch?
- ✓ which data item to replace?
- ✓ how to do lookups quickly?
- ✓ ...

Buffer Caching: Windows (XP ++)

- An I/O manager performs caching
  - centralized facility to all components (not only file data)

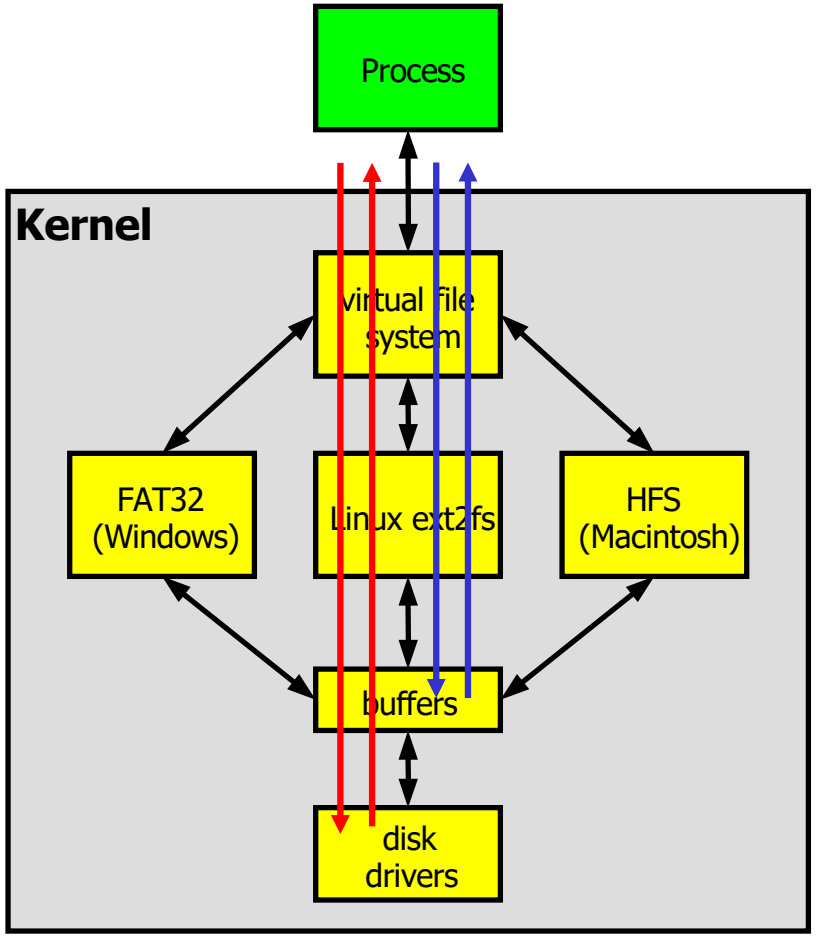


- I/O request processing:
  1. I/O request from process
  2. I/O manager forwards to cache manager
- **in cache**:
  3. cache manager locates and copies data to process buffer via VMM
  4. VMM notifies process
- **on disk**:
  3. cache manager generates a page fault
  4. VMM makes a non-cached service request
  5. I/O manager makes request to file system
  6. file system forwards to disk
  7. disk finds data
  8. reads into cache
  9. cache manager copies data to process buffer via VMM
  10. VMM notifies process

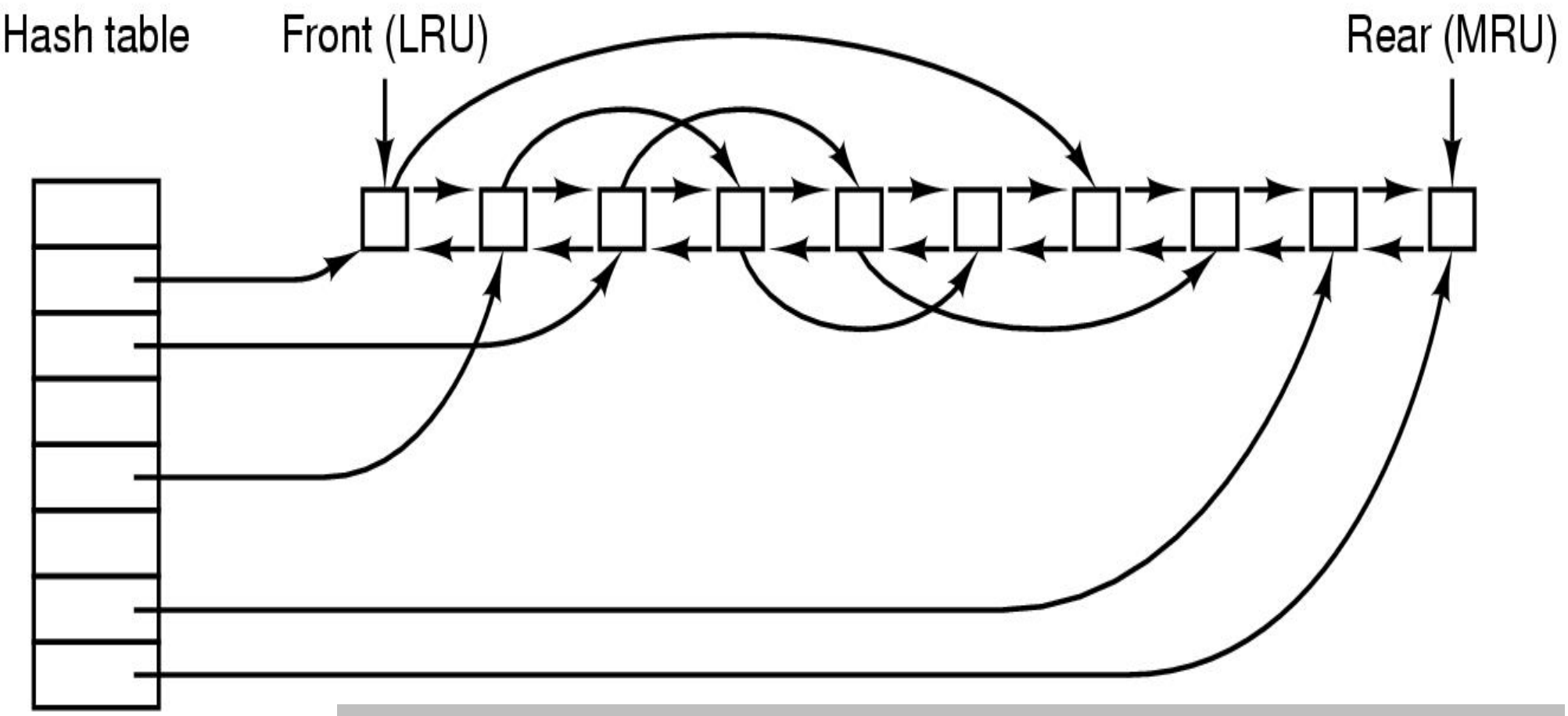
Buffer Caching: Linux / Unix

- A file system performs caching
  - caches disk data (blocks) only
  - may hint on caching decisions
  - prefetching

- I/O requests processing:
  1. I/O request from process
  2. virtual file system forwards to local file system
  3. local file system finds requested block number
  4. requests block from buffer cache
  5. data located…
     - **… in cache**:
       a. return buffer memory address
     - **… on disk**:
       a. make request to disk driver
       b. data is found on disk and transferred to buffer
       c. return buffer memory address
  6. file system copies data to process buffer
  7. process is notified



Buffer Caching Structure



Many different algorithms for replacement, similar to page replacement...

File Systems

Files??

- A file is a collection of data – often for a specific purpose
  - unstructured files, e.g., Unix and Windows
  - structured files, e.g., early MacOS (to some extent) and MVS

- In this course, we consider unstructured files
  - for the operating system, a file is only a sequence of bytes
  - it is up to the application/user to interpret the meaning of the bytes
  - ➥ simpler file systems

File Systems

- File systems organize data in files and manage access regardless of device type:
  - storage management (bottom-up view) –
    allocating space for files on secondary storage
  - file management (top-down view) –
    mechanisms for files to be stored, referenced, shared, secured, …
    - file integrity mechanisms – ensuring that information is not corrupted, intended content only
    - access methods – provide methods to access stored data

File & Directory Operations

- File:
  - create
  - delete
  - open
  - close
  - read
  - write
  - append
  - seek
  - get/set attributes
  - rename
  - link
  - unlink
  - …
- Directory:
  - create
  - delete
  - opendir
  - closedir
  - readdir
  - rename
  - link
  - unlink
  - …

Example: `open()`, `read()` and `close()`

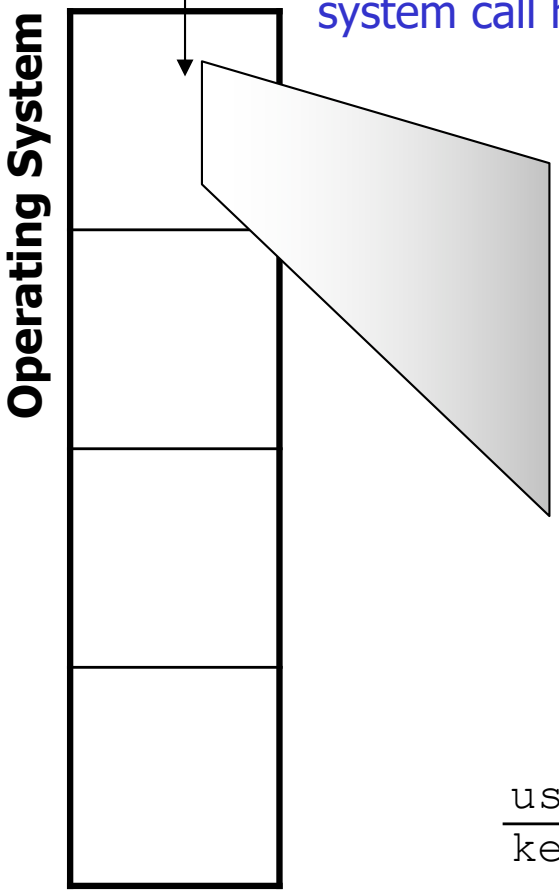
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        int fd, n;
        char buffer[BUFSIZE];
        char *buf = buffer;

        if ((fd = open( "my.file" , O_RDONLY , 0 )) == -1) {
                printf("Cannot open my.file!\n");
                exit(1); /* EXIT_FAILURE */
        }

        while ((n = read(fd, buf, BUFSIZE) > 0) {
                <<USE DATA IN BUFFER>>
        }

        close(fd);

        exit(0); /* EXIT_SUCCESS */
}
```

Open

BSD example

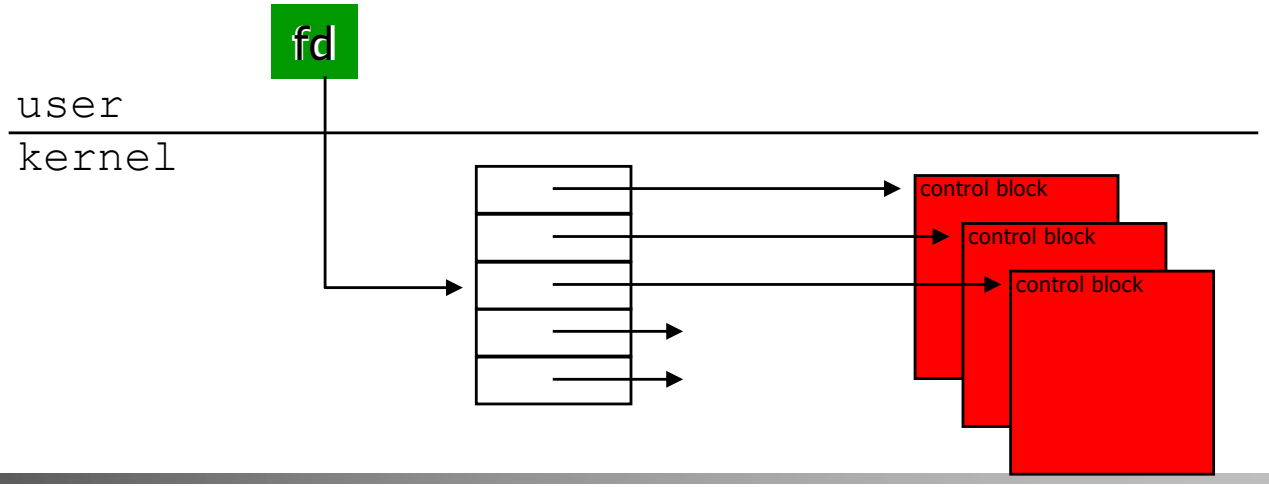
`open(name,oflags,mode)`

system call handling as described earlier



sys_open() → vn_open():

1. Check if valid call
2. Allocate file descriptor
3. If file exists, open for read (remember `O_RDONLY`).
   Must get directory inode (?). May require disk I/O.
4. Set access rights, flags and pointer to vnode (?)
5. Return index to file descriptor table

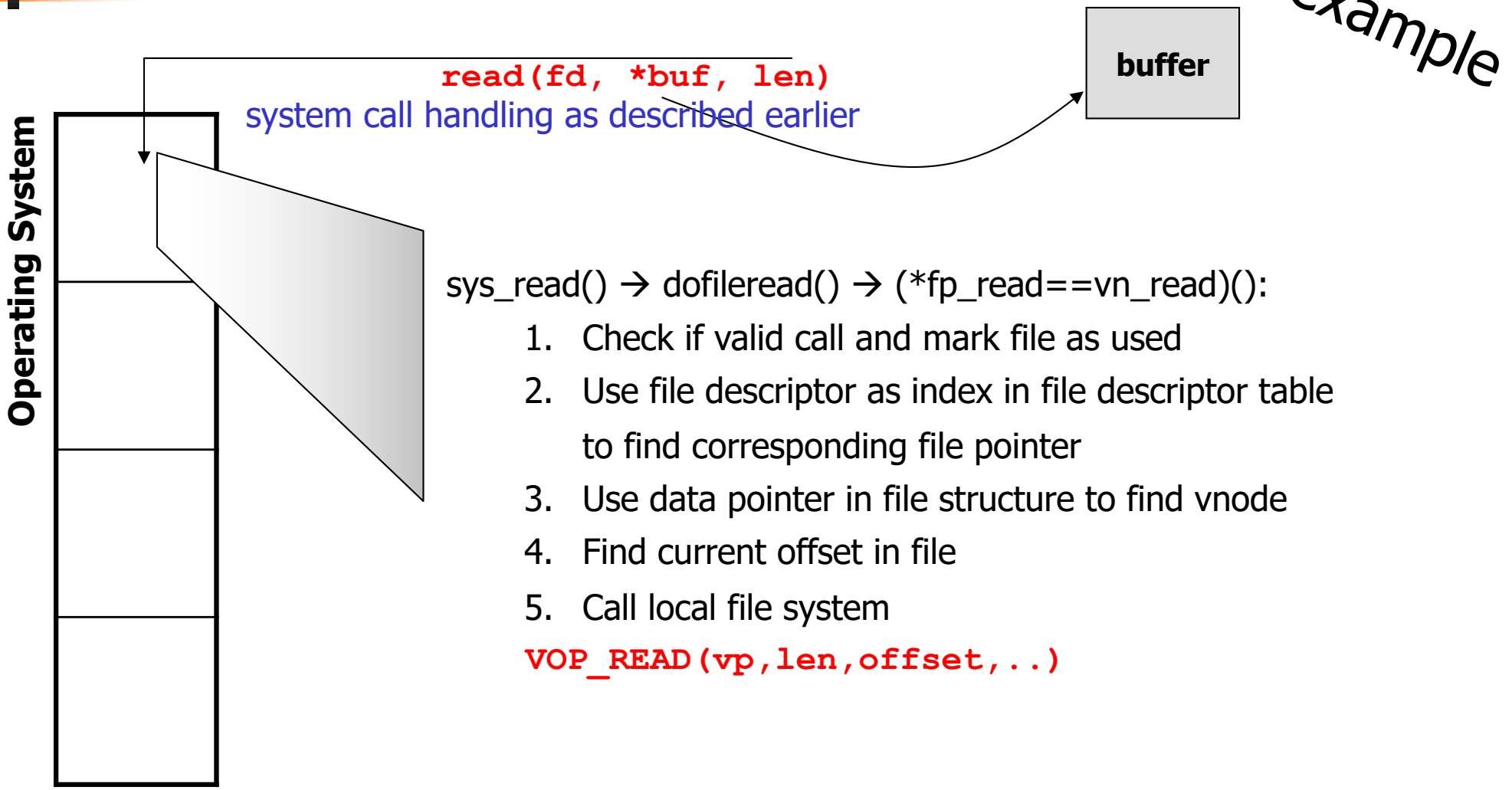


Example: `open()`, `read()` and `close()`

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        int fd, n;
        char buffer[BUFSIZE];
        char *buf = buffer;

        if ((fd = open( "my.file" , O_RDONLY , 0 )) == -1) {
                printf("Cannot open my.file!\n");
                exit(1); /* EXIT_FAILURE */
        }

        while ((n = read(fd, buf, BUFSIZE) > 0) {
                <<USE DATA IN BUFFER>>
        }

        close(fd);

        exit(0); /* EXIT_SUCCESS */
}
```

Read

BSD example

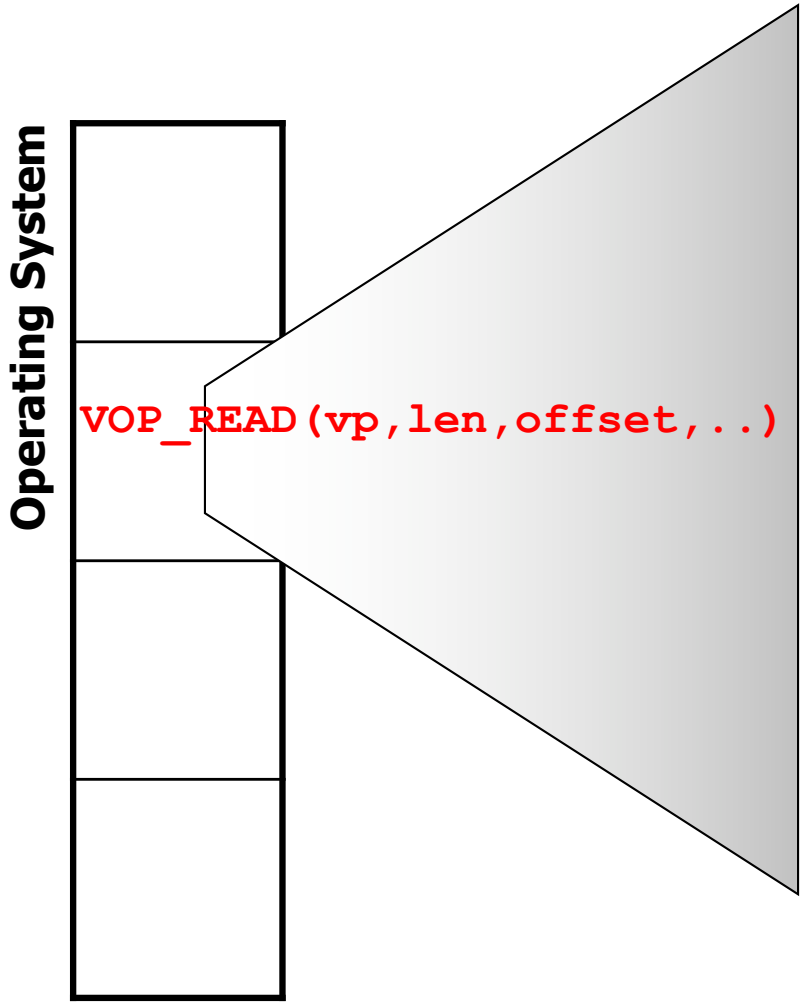


sys_read() → dofileread() → (*fp_read==vn_read)():

1. Check if valid call and mark file as used
2. Use file descriptor as index in file descriptor table to find corresponding file pointer
3. Use data pointer in file structure to find vnode
4. Find current offset in file
5. Call local file system

```
VOP_READ(vp,len,offset,..)
```

Read



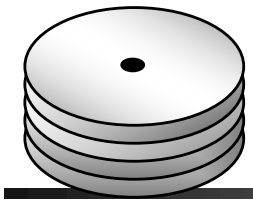
VOP_READ(...) is a pointer to a read function in the corresponding file system, e.g., Fast File System (FFS)

READ():

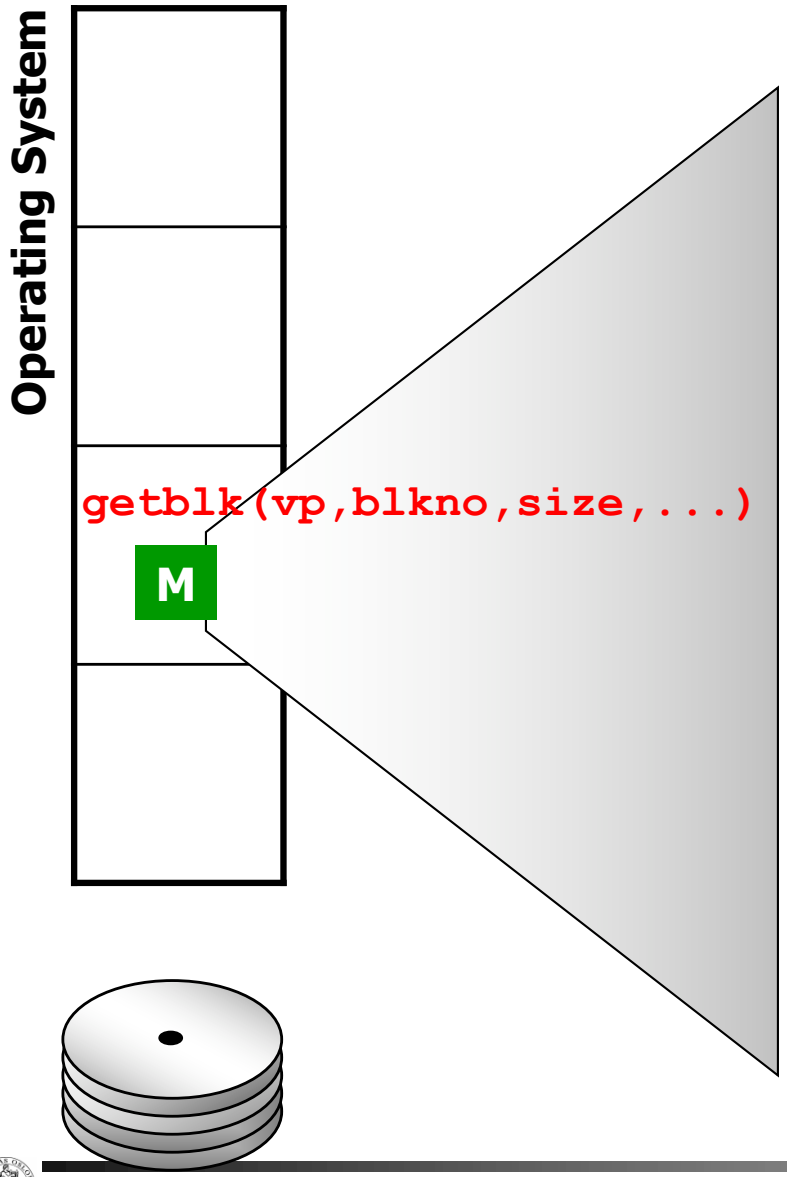
1. Find corresponding inode
2. Check if valid call: len + offset ≤ file size
3. Loop and find corresponding blocks
   - find logical blocks from inode, offset, length
   - do block I/O, fill buffer structure
     e.g., bread(...) → bio_doread(...) → getblk()

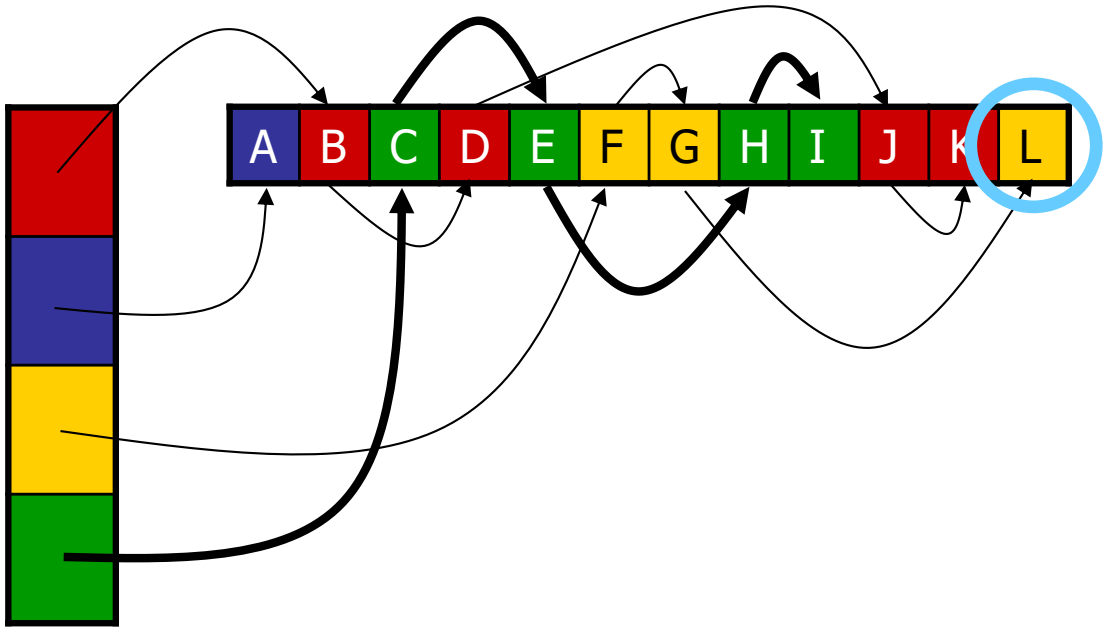
     `getblk(vp,blkno,size,...)`

   - return and copy block to user



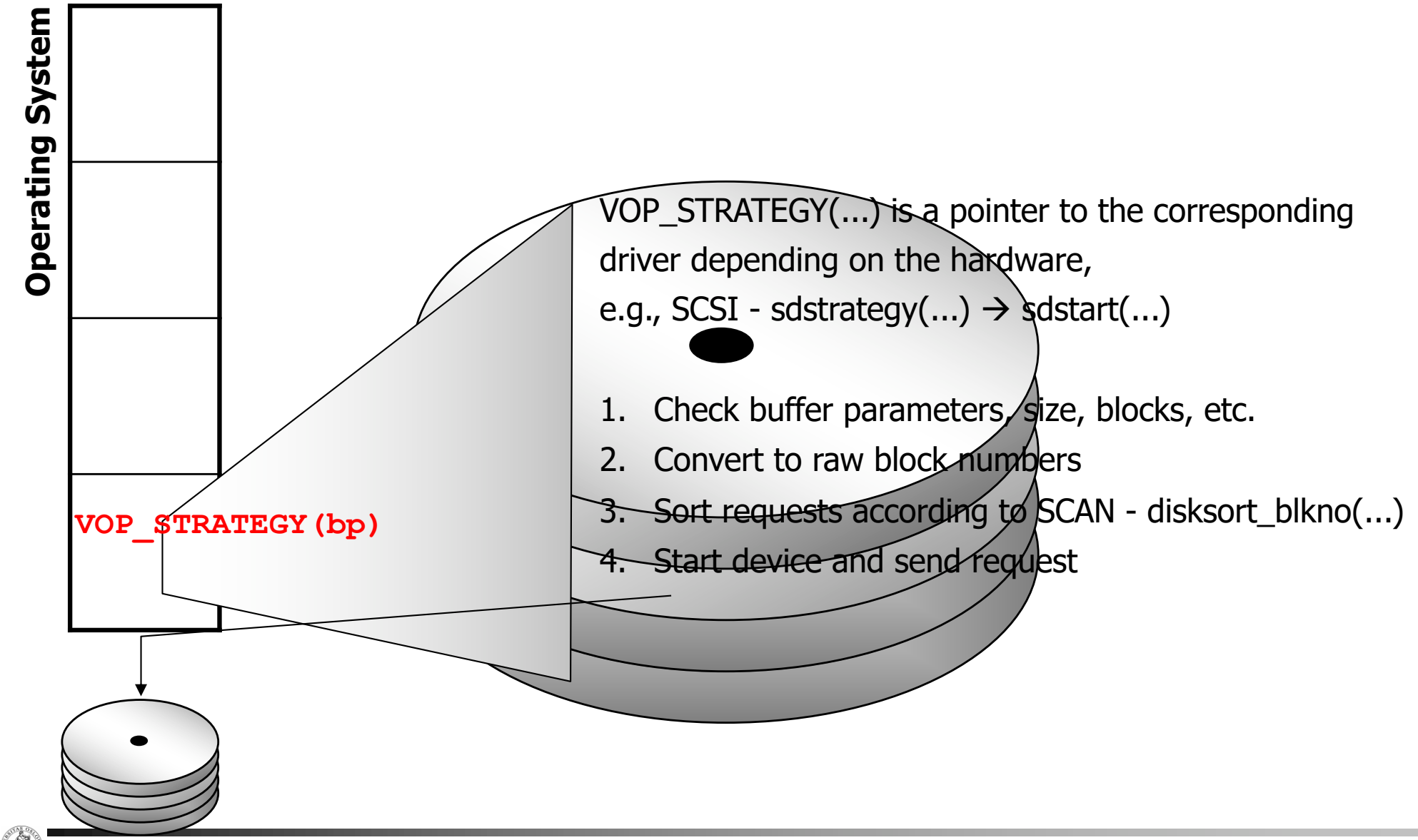
Read



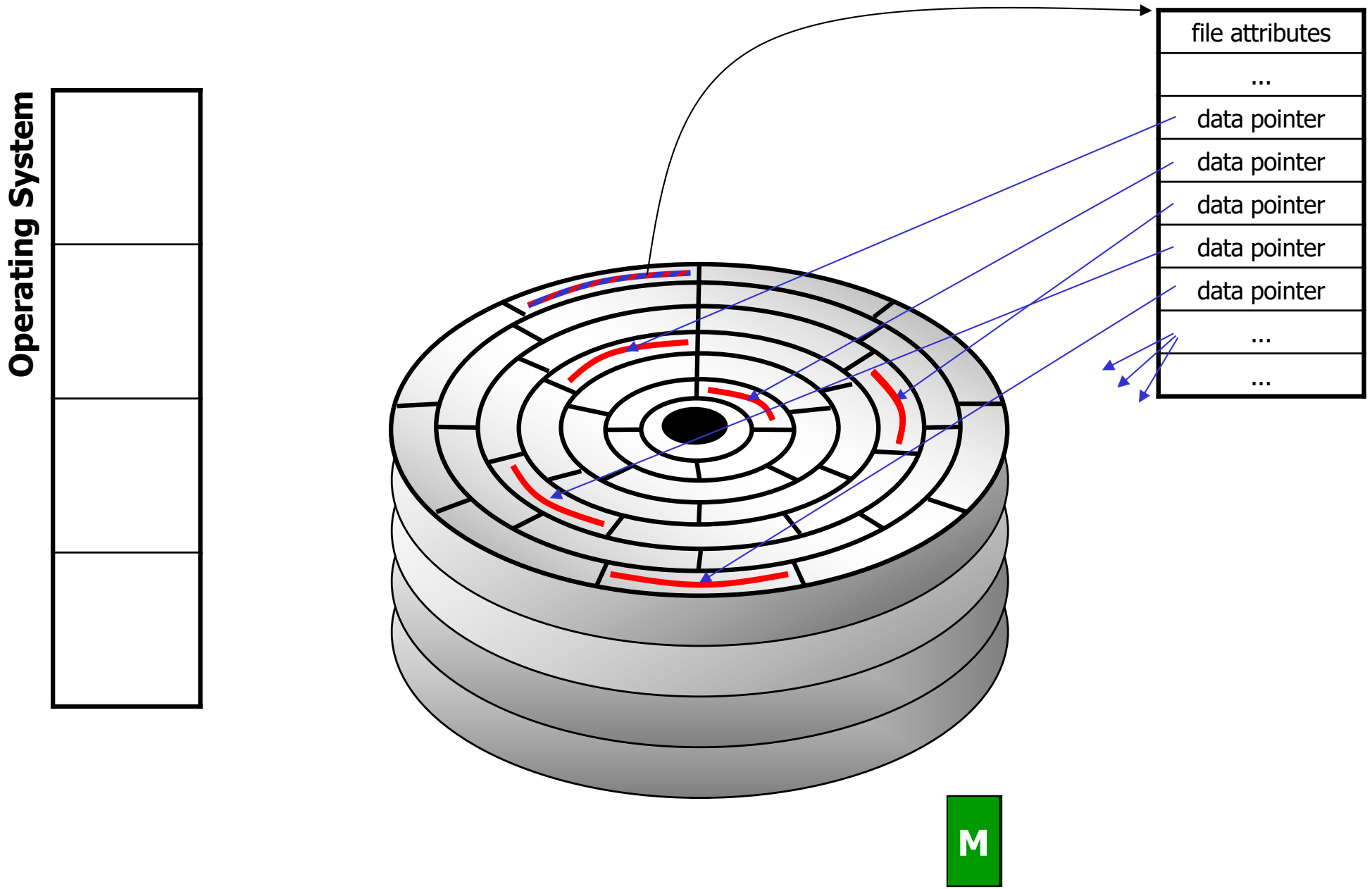


1. Search for block in buffer cache, return if found
   (hash vp and blkno and follow linked hash list)
2. Get a new buffer (LRU, age)
3. Call disk driver - sleep or do something else
   `VOP_STRATEGY(bp)`
4. Reorganize LRU chain and return buffer

Read


Operating System
VOP_STRATEGY(bp)
VOP_STRATEGY(...) is a pointer to the corresponding driver depending on the hardware,
e.g., SCSI - sdstrategy(...) → sdstart(...)
1. Check buffer parameters, size, blocks, etc.
2. Convert to raw block numbers
3. Sort requests according to SCAN - disksort_blkno(...)
4. Start device and send request

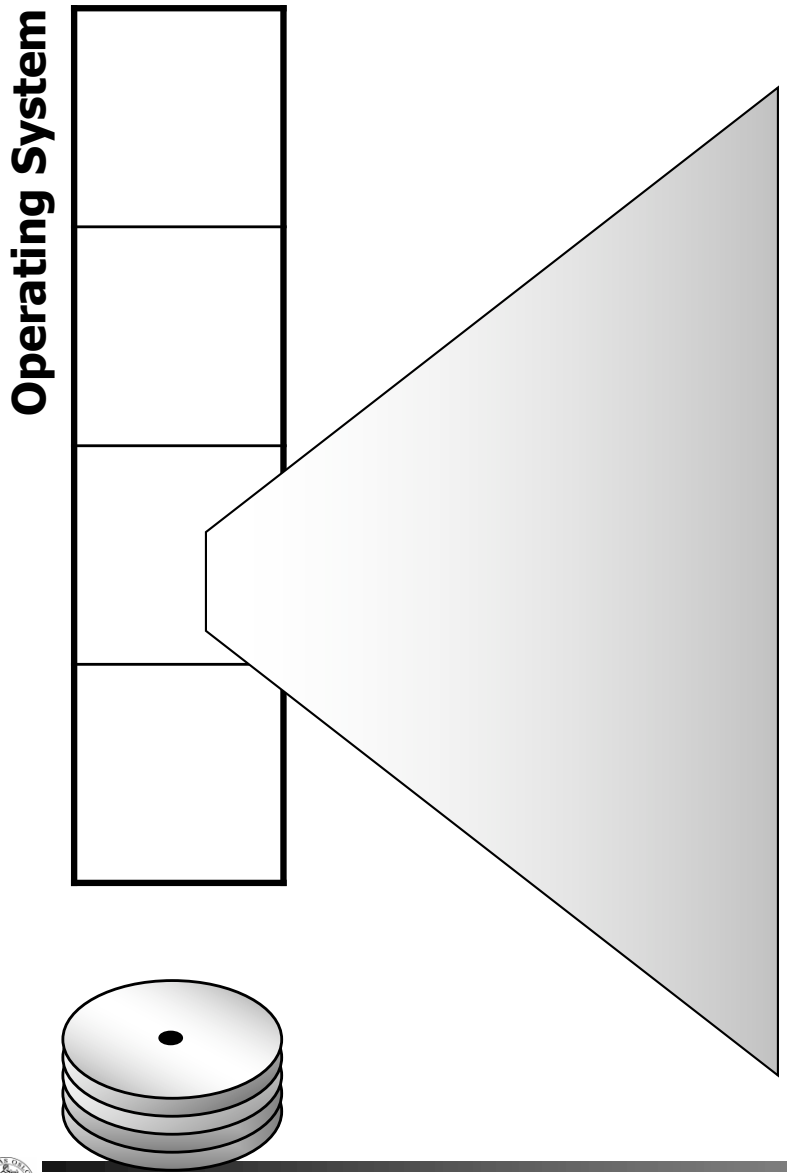
Read

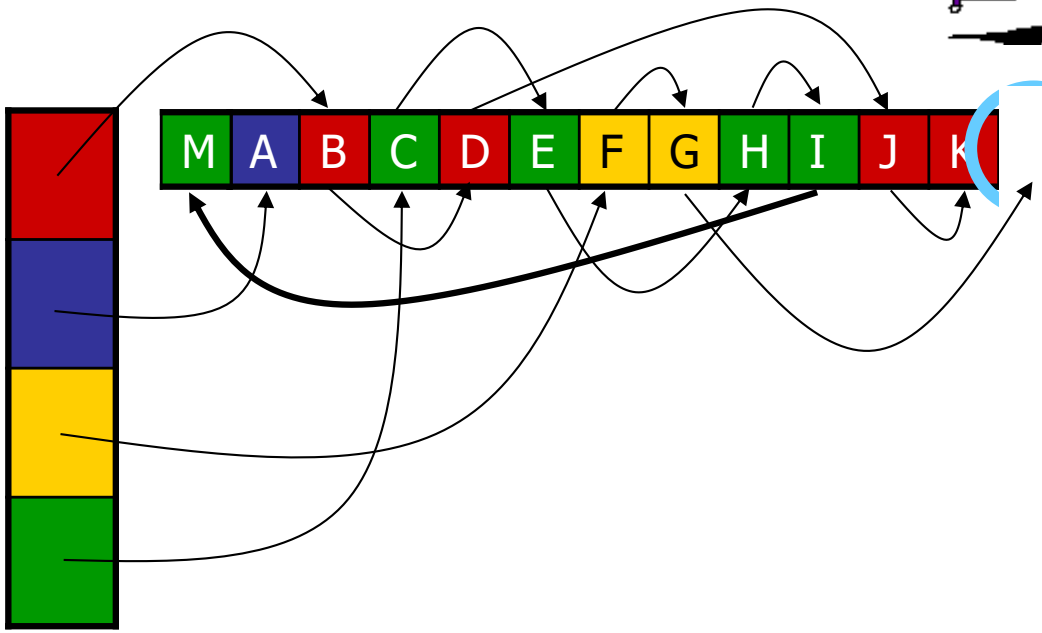

Operating System
file attributes
...
data pointer
data pointer
data pointer
data pointer
data pointer
...
...
M

Read

Interrupt to notify end of disk IO

Kernel may awaken sleeping process

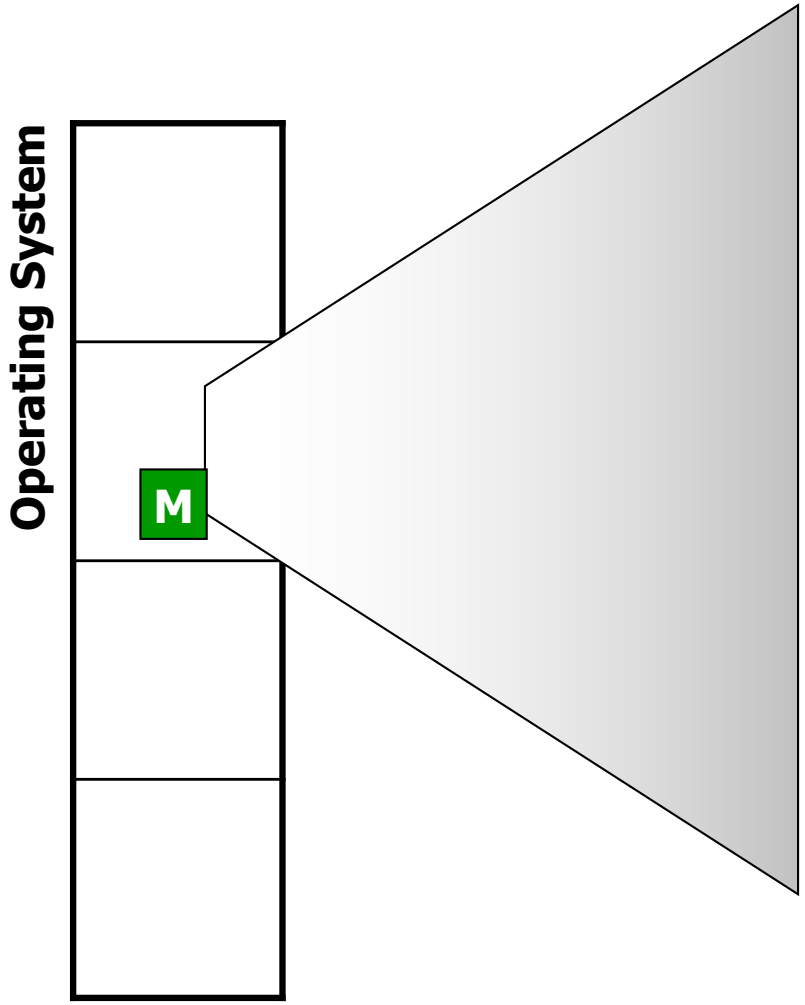




1. Search for block in buffer cache, return if found (hash vp and blkno and follow linked hash list)
2. Get a new buffer (LRU, age)
3. Call disk driver - sleep or do something else

4. Reorganize LRU chain (M) return buffer

Read

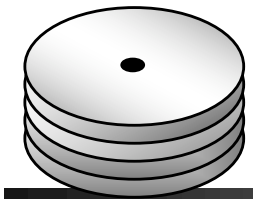


buffer

READ():

1. Find corresponding inode
2. Check if valid call - file size vs. len + offset
3. Loop and find corresponding blocks
   - find logical blocks from inode, offset, length
   - do block I/O,
     e.g., bread(...) → bio_doread(...) → getblk()

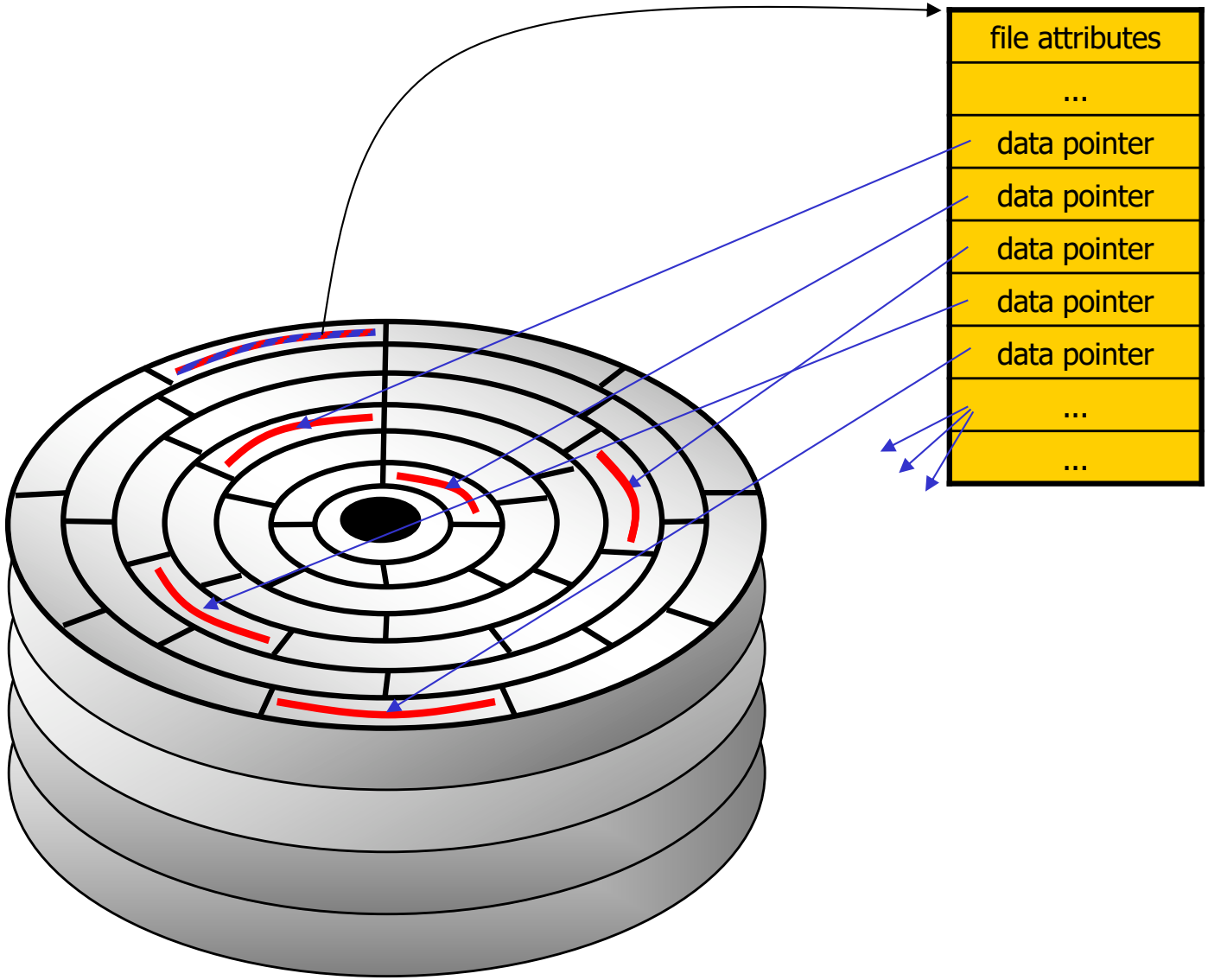
- return and copy block to user



Example: open(), read() and close()

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
        int fd, n;
        char buffer[BUFSIZE];
        char *buf = buffer;

        if ((fd = open( "my.file" , O_RDONLY , 0 )) == -1) {
                printf("Cannot open my.file!\n");
                exit(1); /* EXIT_FAILURE */
        }

        while ((n = read(fd, buf, BUFSIZE) > 0) {
                <<USE DATA IN BUFFER>>
        }

        close(fd);

        exit(0); /* EXIT_SUCCESS */
}
```

Management of File Blocks

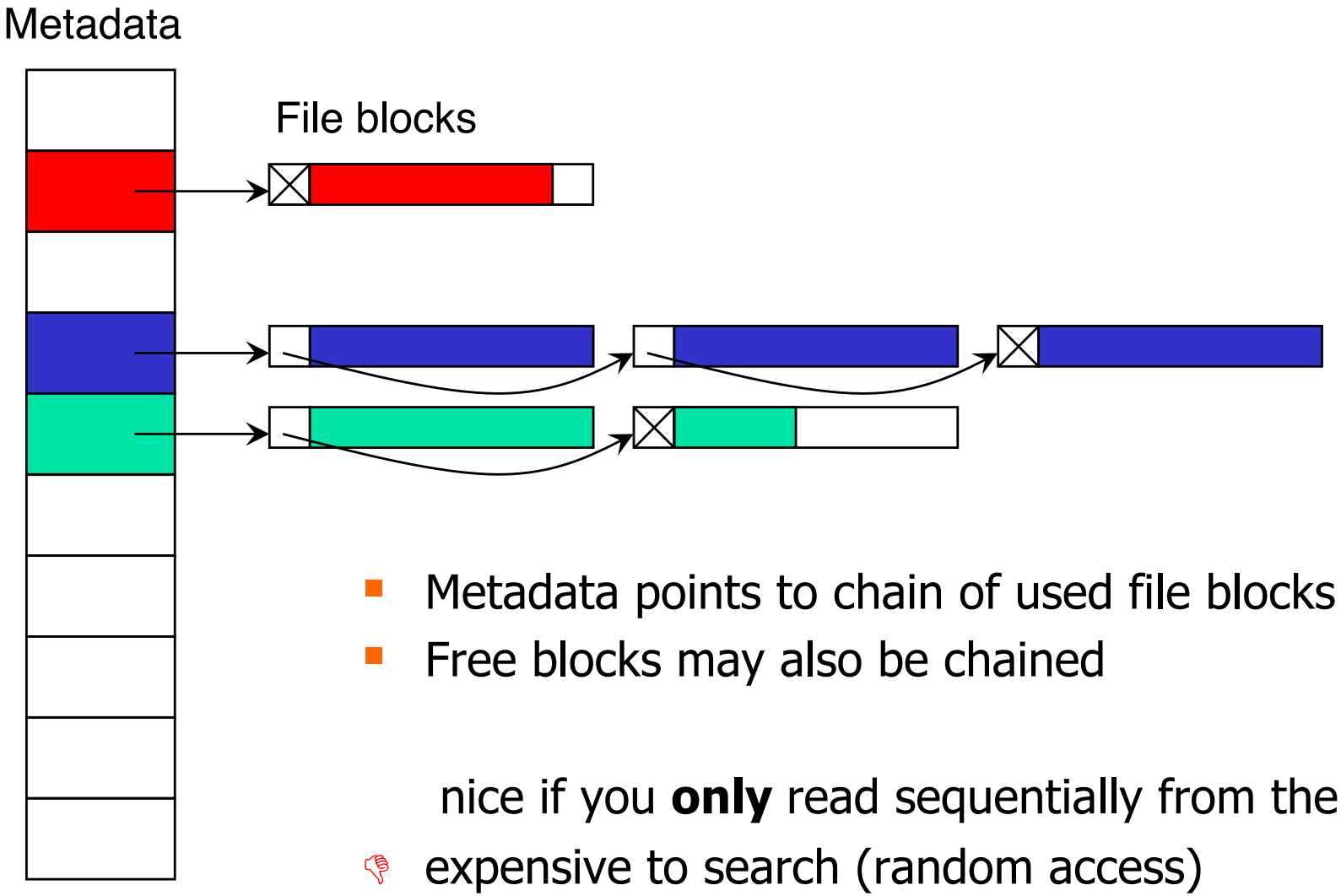


Management of File Blocks

- Many files consist of several blocks
  - relate blocks to files
  - how to locate a given block
  - maintain order of blocks

- Approaches
  - chaining in media
  - chaining in a map
  - table of pointers
  - extent-based allocation

Chaining in the Media

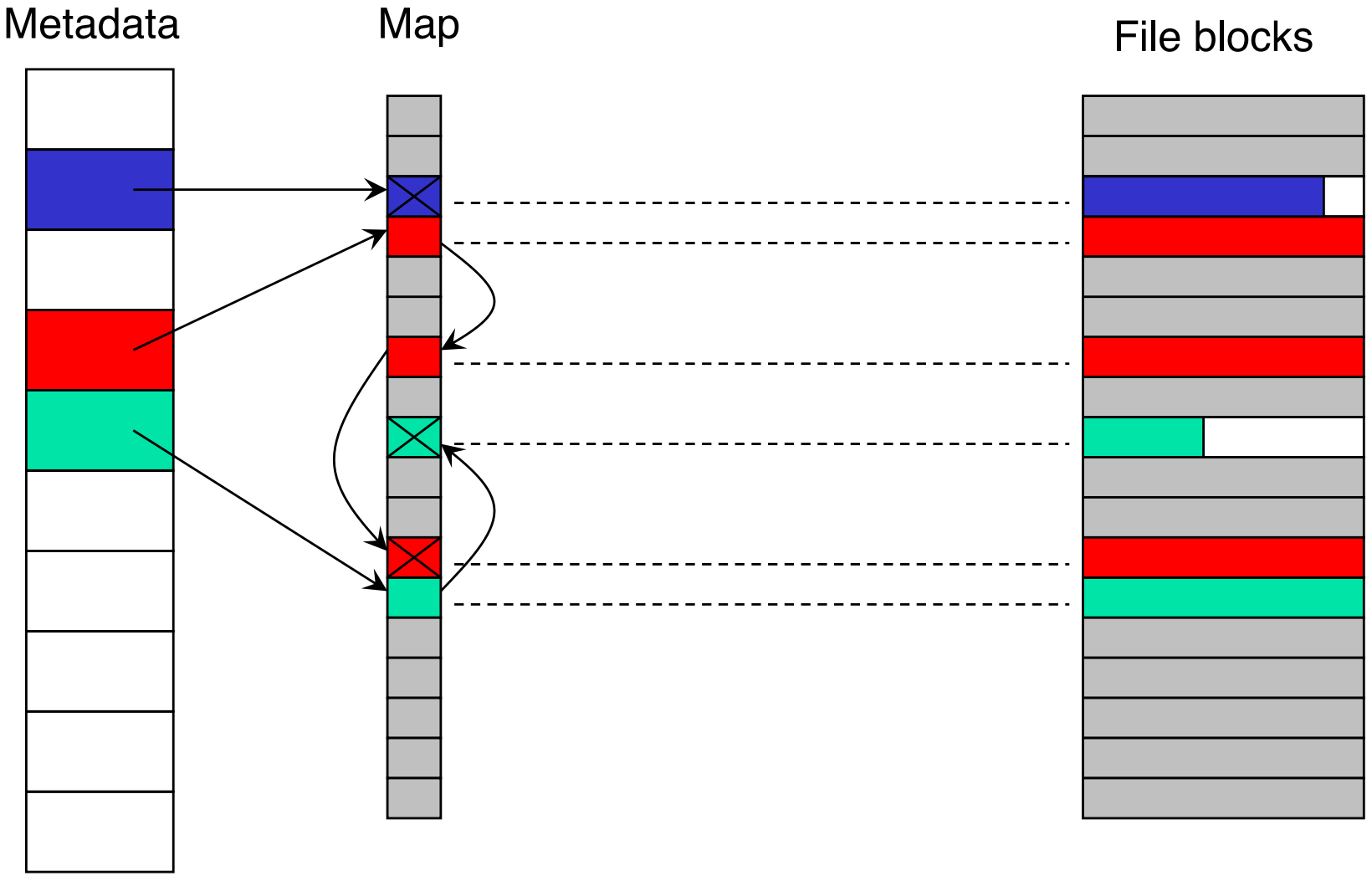


- Metadata points to chain of used file blocks
- Free blocks may also be chained

nice if you **only** read sequentially from the start

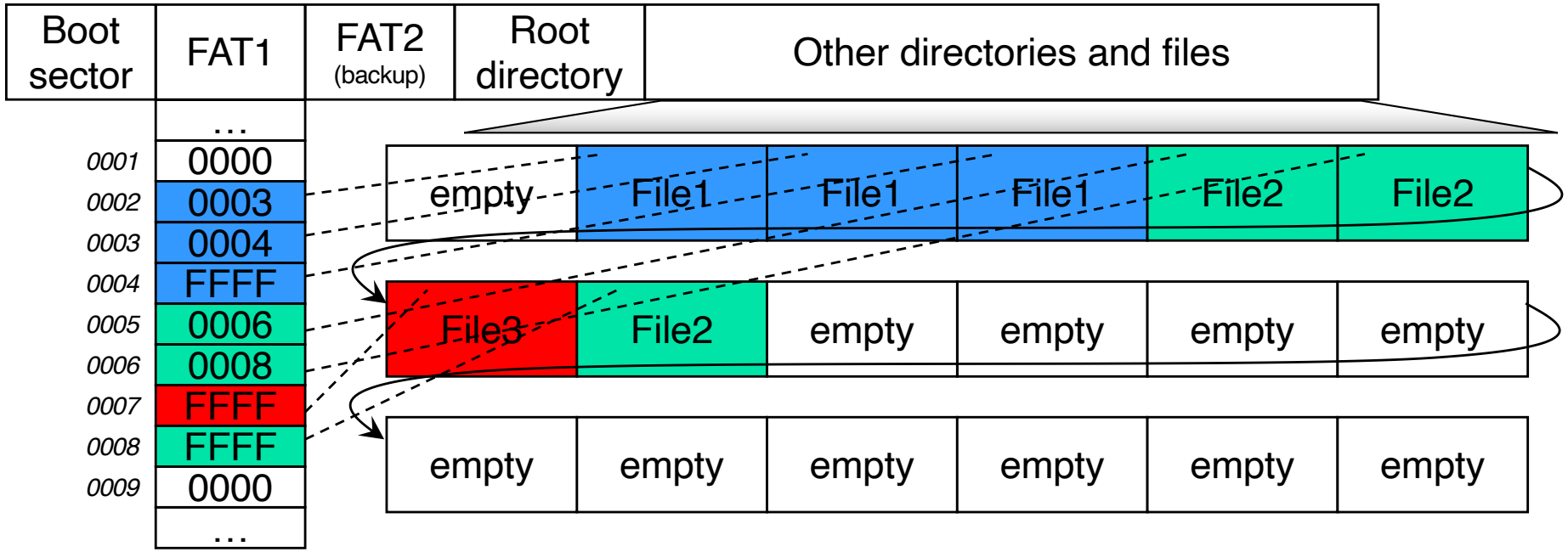
- expensive to search (random access)
- must read block by block

Chaining in a Map



Chaining in a Map: FAT Example

- FAT: File Allocation Table
- Versions FAT12, FAT16, FAT32
  - number indicates number of bits used to identify blocks in partition (2^{12},2^{16},2^{32})
  - FAT12: Block sizes 512 bytes – 8 KB: max 32 MB partition size
  - FAT16: Block sizes 512 bytes – 64 KB: max 4 GB partition size
  - FAT32: Block sizes 512 bytes – 64 KB: max 2 TB partition size



Kom hit!

Table of Pointers

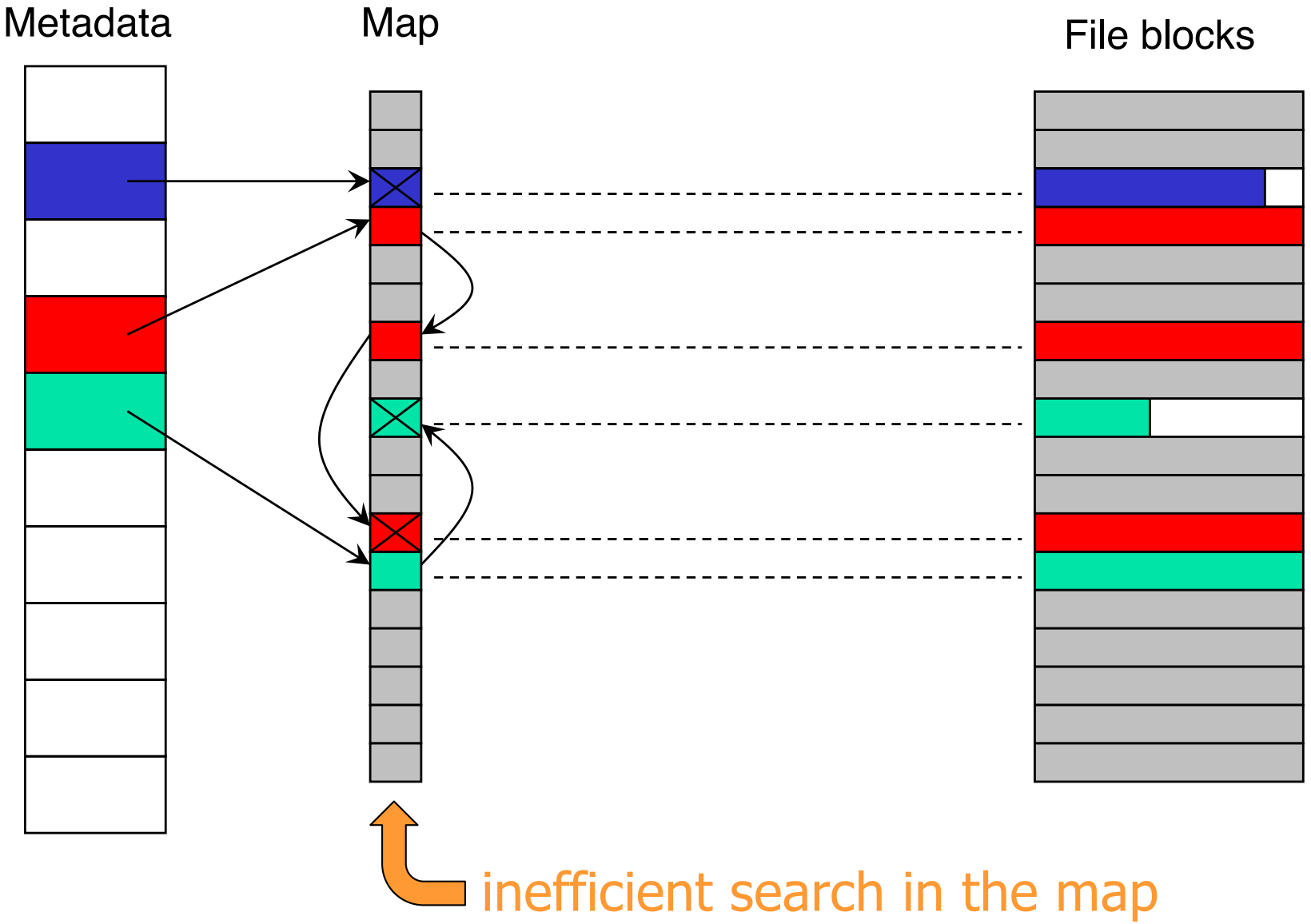
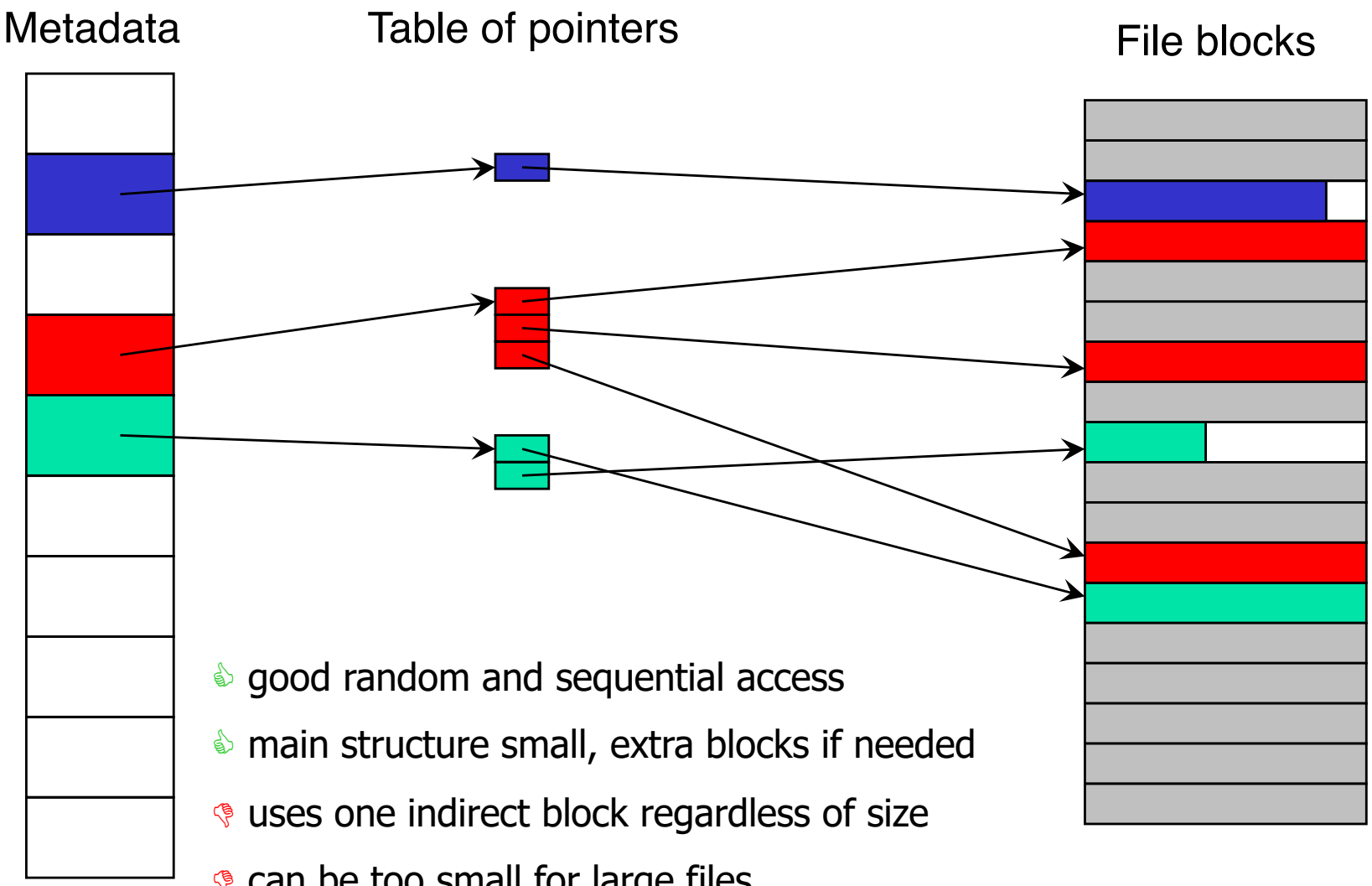
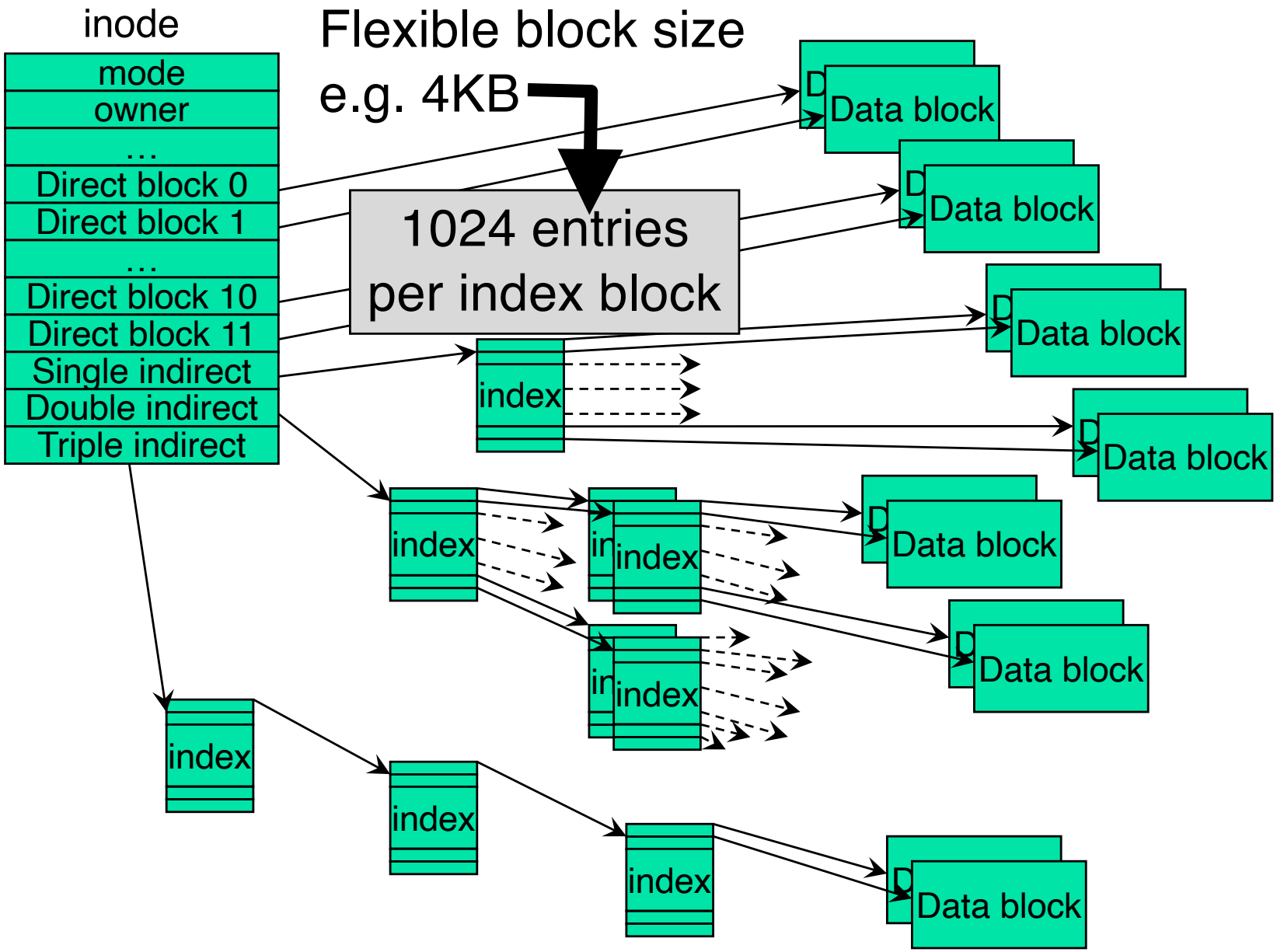


Table of Pointers



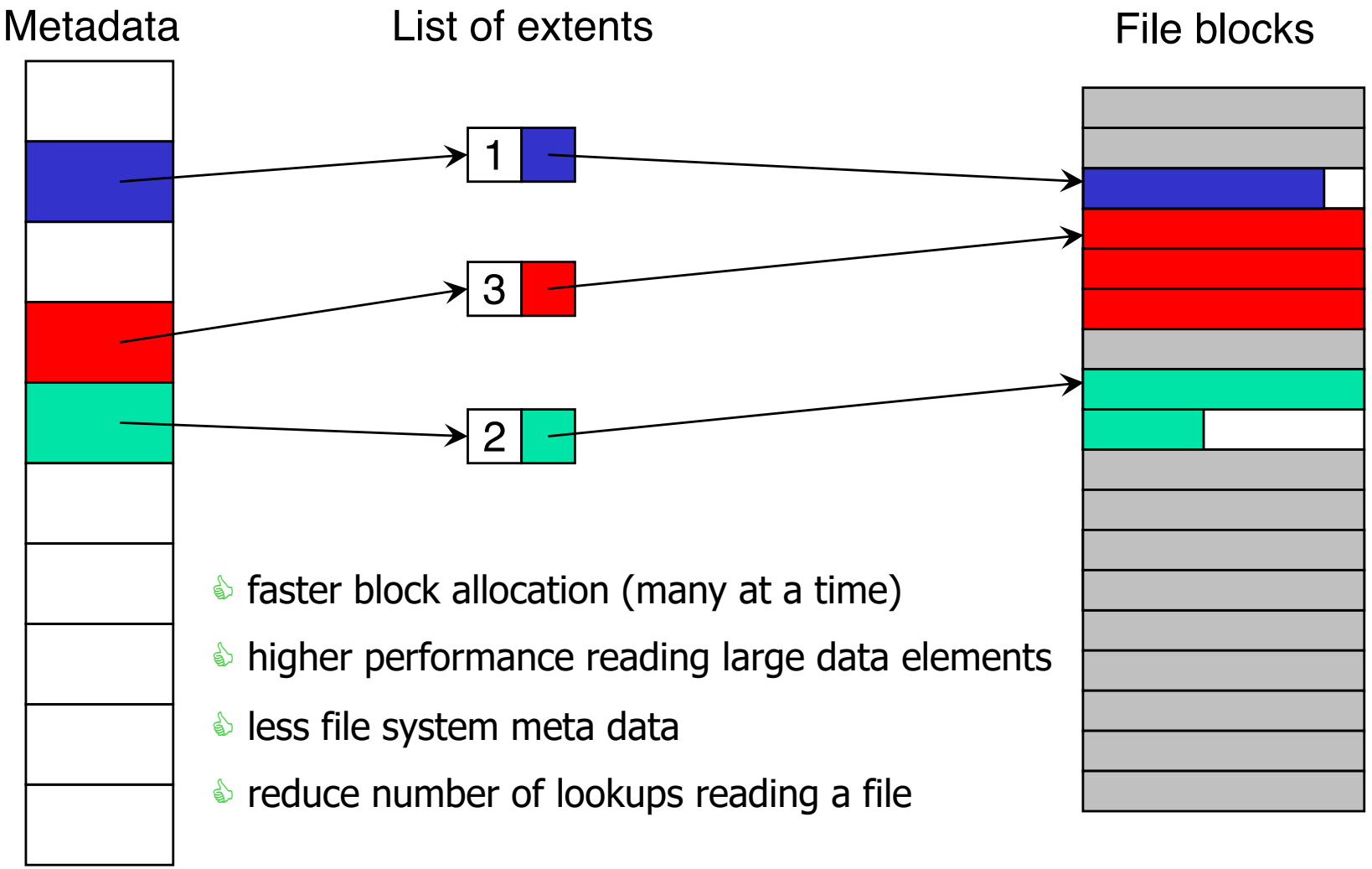
- good random and sequential access
- main structure small, extra blocks if needed
- uses one indirect block regardless of size
- can be too small for large files

Unix/Linux Example: FFS, UFS, ...



Extent-based Allocation

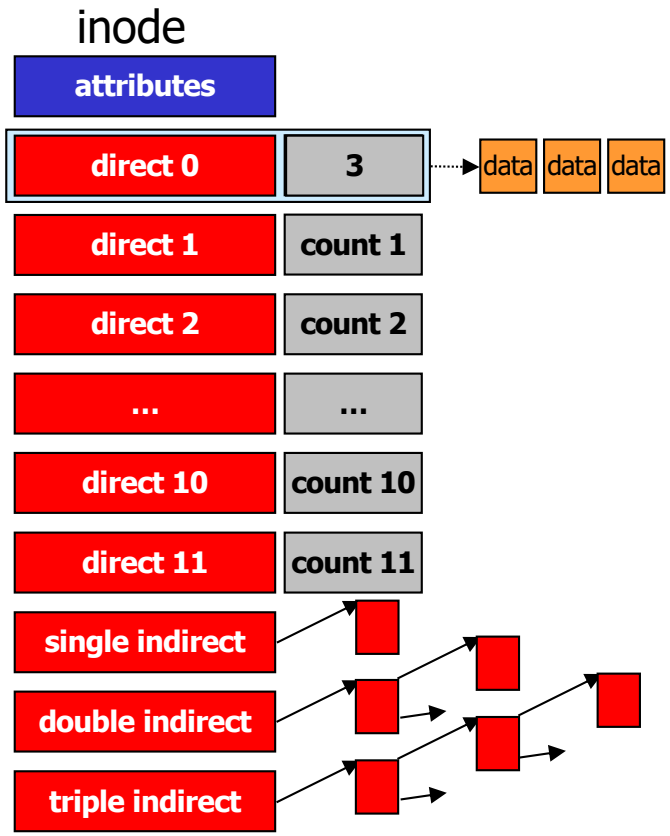
- ✓ Observation:
  indirect block reads introduce disk I/O and break access locality



- faster block allocation (many at a time)
- higher performance reading large data elements
- less file system meta data
- reduce number of lookups reading a file

Linux Example: XFS, JFS, EXT4...

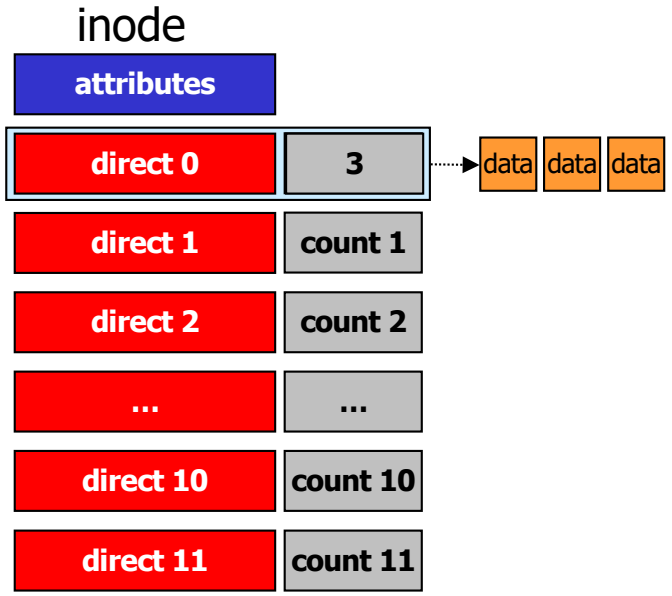
- Count-augmented address indexing in the extent sections
- Introduce a new inode structure
  - add counter field to original direct entries
    - direct points to a disk block
    - count indicated how many other blocks is following the first block (contiguously)



ext4_inode

```
struct ext4_inode {
    __le16 i_mode;              /* File mode */
    __le16 i_uid;               /* Low 16 bits of Owner Uid */
    __le32 i_size_lo;           /* Size in bytes */
    __le32 i_atime;             /* Access time */
    __le32 i_ctime;             /* Inode Change time */
    __le32 i_mtime;             /* Modification time */
    __le32 i_dtime;             /* Deletion Time */
    __le16 i_gid;               /* Low 16 bits of Group Id */
    __le16 i_links_count;       /* Links count */
    __le32 i_blocks;            /* Blocks count */
    __le32 i_flags;             /* File flags */
    ...
    __le32 i_block[EXT4_N_BLOCKS];/* Pointers to blocks */
    __le32 i_generation;        /* File version (for NFS) */
    __le32 i_file_acl;          /* File ACL */
    __le32 i_dir_acl;           /* Directory ACL */
    __le32 i_faddr;             /* Fragment address */
    ...

    __le32 i_ctime_extra;       /* extra Change time (nsec << 2 | epoch) */
    __le32 i_mtime_extra;       /* extra Modification */
    __le32 i_atime_extra;       /* extra Access time  */
    __le32 i_crtime;            /* File Creation time */
    __le32 i_crtime_extra;      /* extra */
};
```

i_block [15]



ext4_inode

```
...
__le16 eh_depth;
...
```

↳Tree of extents organized using an HTREE

i_block [NUM]

ext4_extent_header
ext4_extent
ext4_extent
ext4_extent
ext4_extent

```
struct ext4_extent {
    __le32 ee_block;      /* first logical block extent covers */
    __le16 ee_len;        /* number of blocks covered by extent */
    __le16 ee_start_hi;   /* high 16 bits of physical block */
    __le32 ee_start;      /* low 32 bits of physical block */
};
```

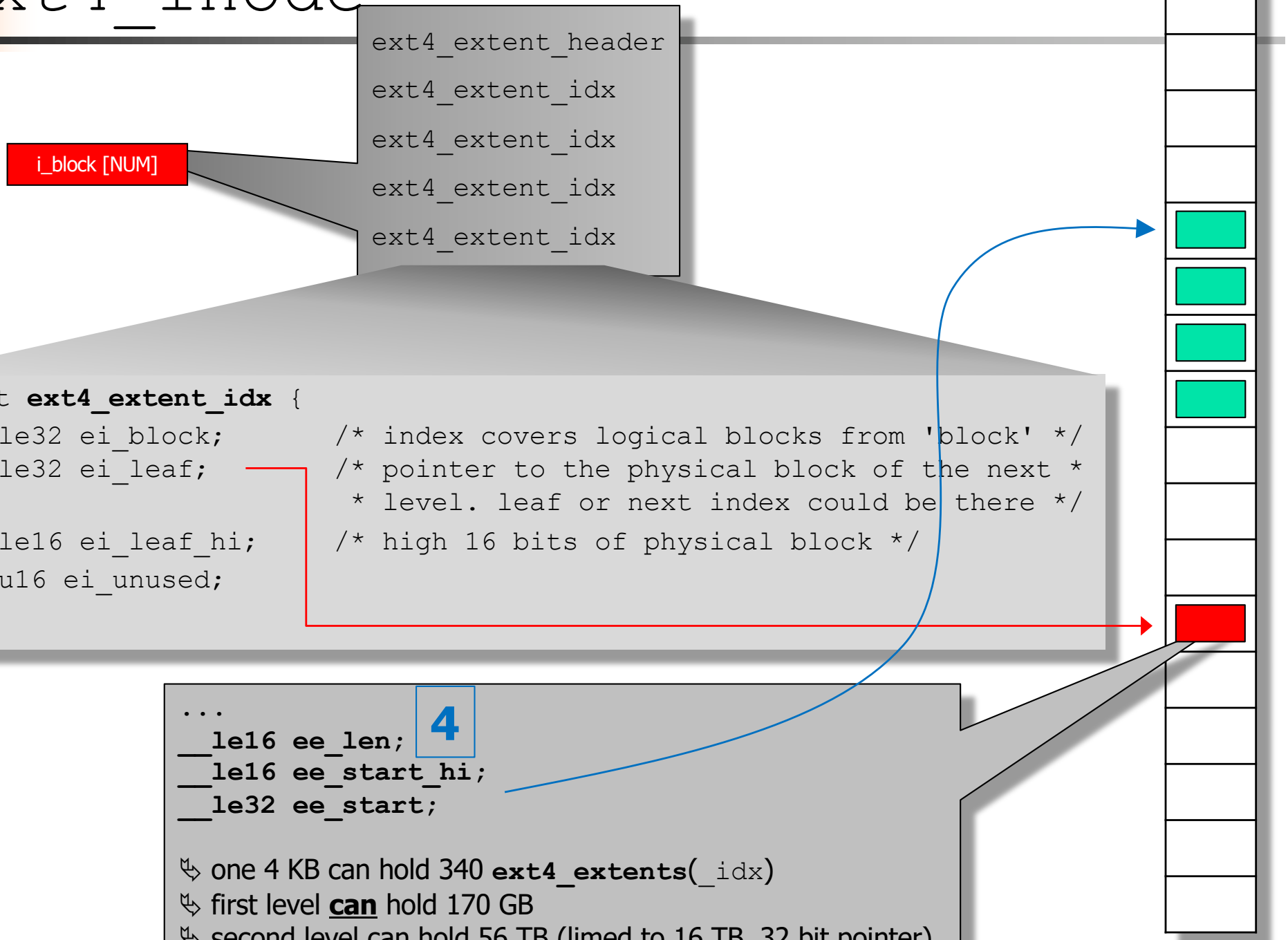
4

Theoretically, each extent can have 2^{16} - 1 continuous blocks, i.e., 64 GB data using a 4KB block size, but limited to 128 MB

Max size of 4 x 128 = **512 MB files**?

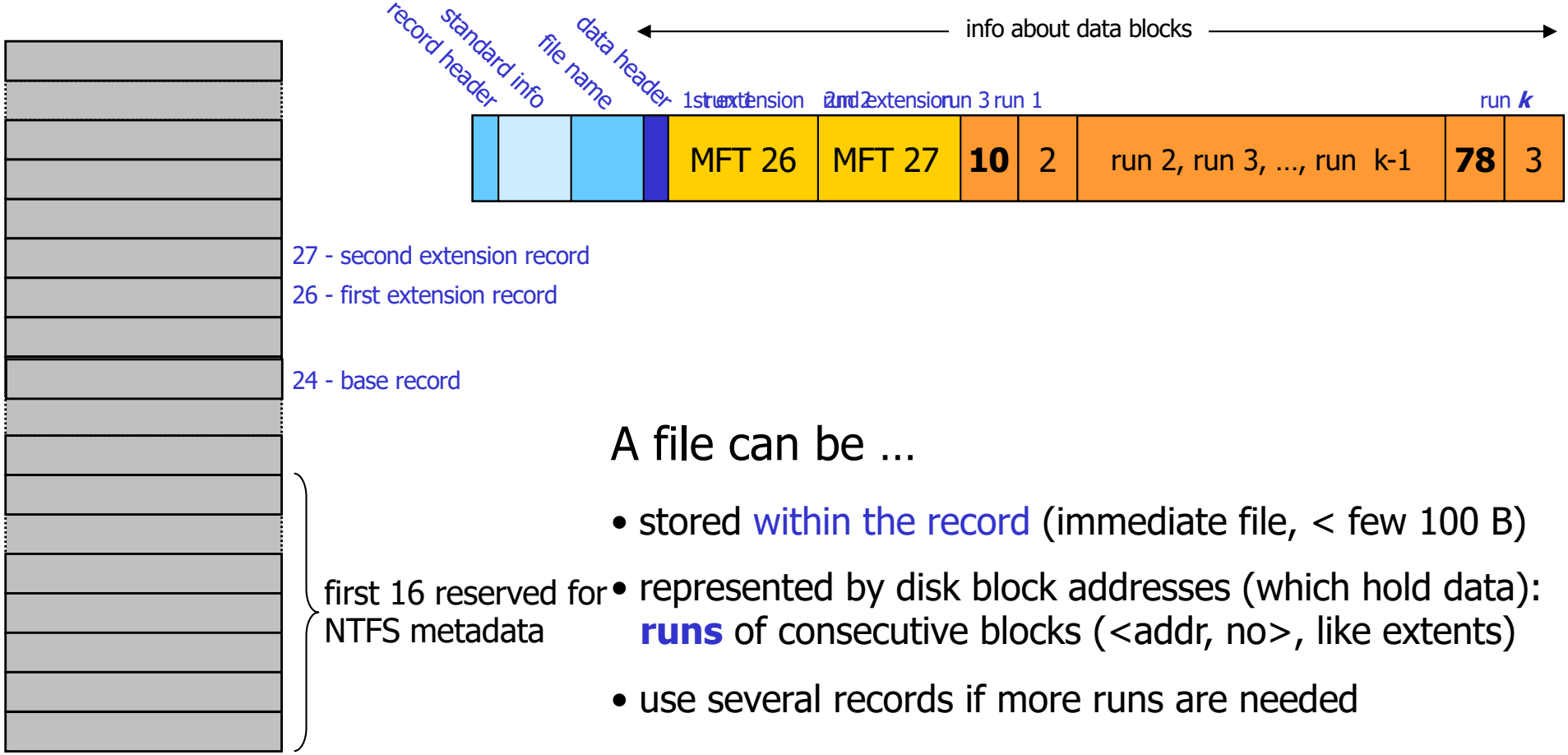
What about **fragmented disks**??

ext4_inode


ext4_extent_header
ext4_extent_idx
ext4_extent_idx
ext4_extent_idx
ext4_extent_idx
i_block [NUM]
struct ext4_extent_idx {
__le32 ei_block; /* index covers logical blocks from 'block' */
__le32 ei_leaf; /* pointer to the physical block of the next *
* level. leaf or next index could be there */
__le16 ei_leaf_hi; /* high 16 bits of physical block */
__u16 ei_unused;
};
...
__le16 ee_len;
4
__le16 ee_start_hi;
__le32 ee_start;
one 4 KB can hold 340 ext4_extents(_idx)
first level can hold 170 GB
second level can hold 56 TB (limed to 16 TB, 32 bit pointer)

Windows Example: NTFS

- Each partition contains a master file table (MFT)
  - a linear sequence of 1 KB records
  - each record describes a directory or a file (attributes and disk addresses)



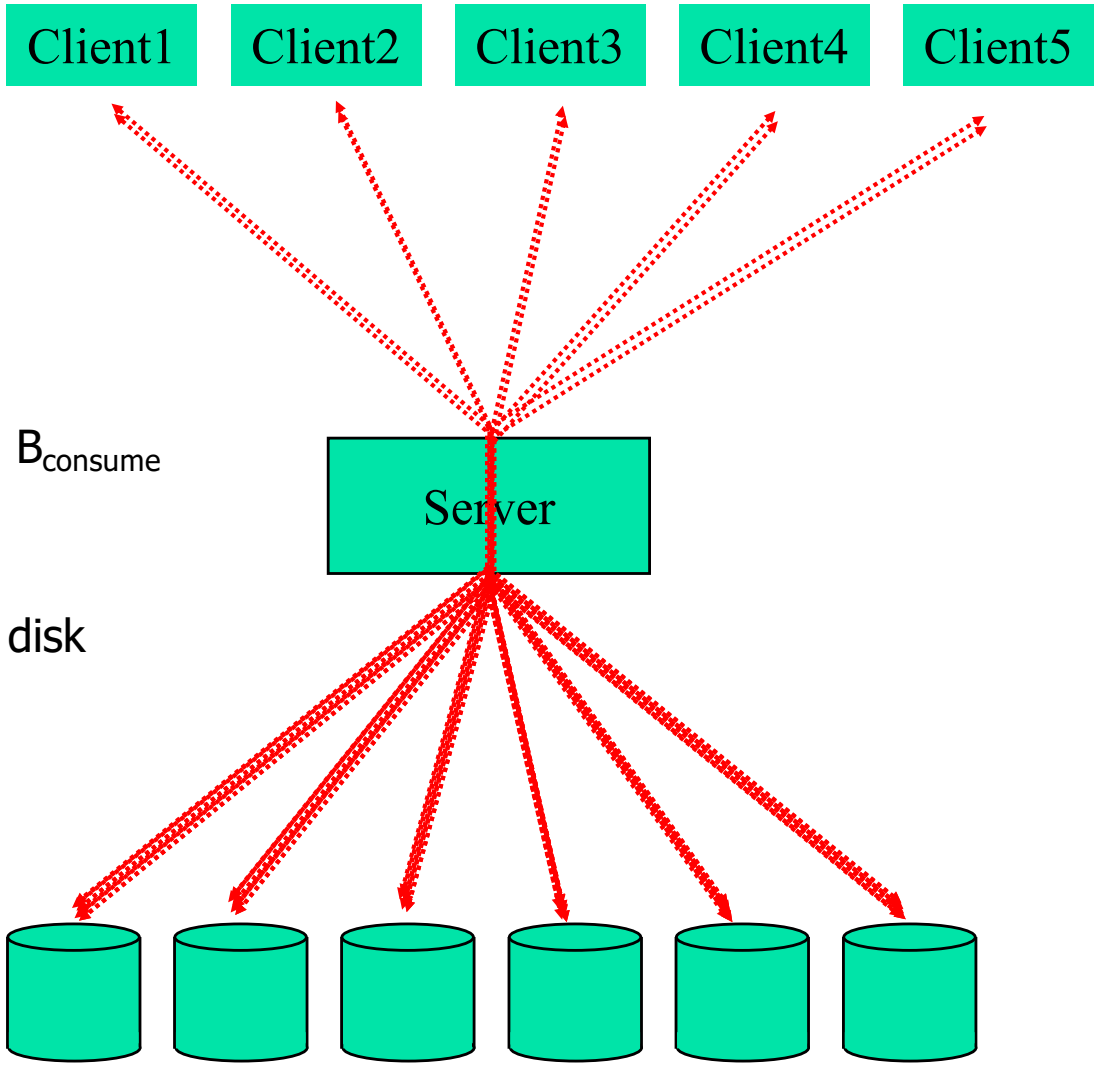
A file can be ...

- stored within the record (immediate file, < few 100 B)
- represented by disk block addresses (which hold data): **runs** of consecutive blocks (<addr, no>, like extents)
- use several records if more runs are needed

Multiple Disks

Striping

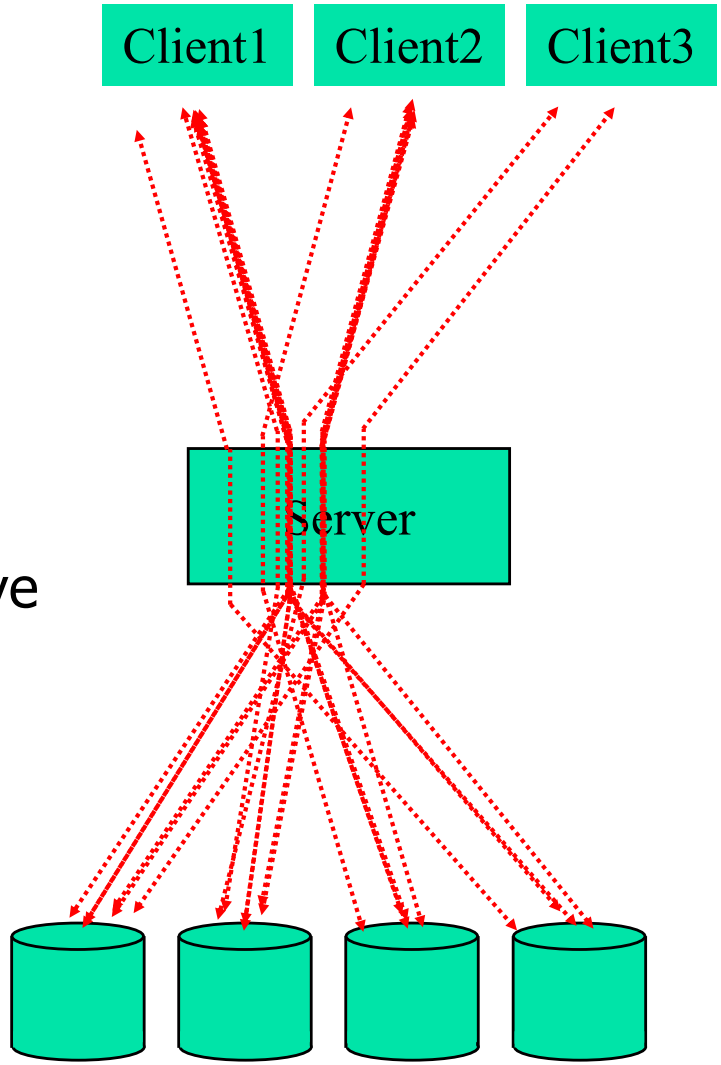
- A reason to use multiple disks is when one disk cannot deliver requested data rate
- In such a scenario, one might use several disks for striping:
  - bandwidth disk: B_{disk}
  - required bandwidth: $B_{consume}$
  - $B_{disk} < B_{consume}$
  - read from *n* disks in parallel: $n\,B_{disk} > B_{consume}$
- Advantages
  - higher transfer rate compared to one disk
- Drawbacks
  - can't serve multiple clients in parallel
  - positioning time increases (i.e., reduced efficiency)



Interleaving (Compound Striping)

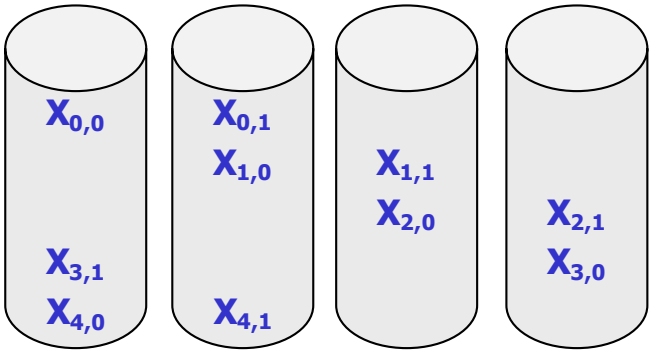
- Full striping usually not necessary today:
  - faster disks
  - better compression algorithms

- Interleaving lets each client be serviced by only a set of the available disks
  - make groups
  - ”stripe” data in a way such that a consecutive request arrive at next group
  - one disk group example:



Interleaving (Compound Striping)

- Divide traditional striping group into sub-groups, e.g., staggered striping



- Advantages
  - multiple clients can still be served in parallel
  - more efficient disks operations
  - potentially shorter response time

- Potential drawback/challenge
  - load balancing (all clients access same group)

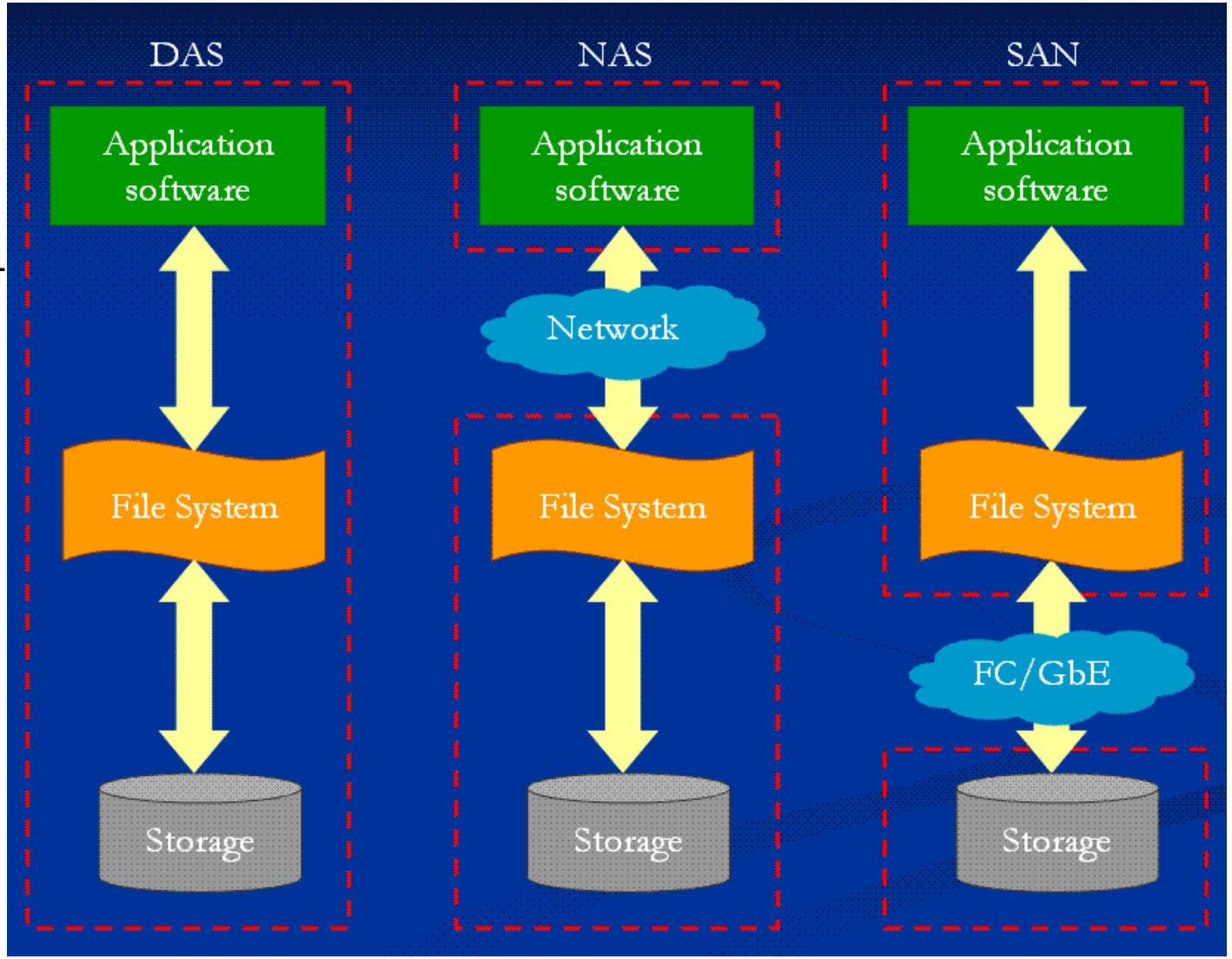
Redundant Array of Inexpensive Disks

- The various RAID levels define different disk organizations to achieve higher performance and more reliability
  - RAID 0 - striped disk array without fault tolerance (non-redundant)
  - RAID 1 - mirroring
  - RAID 2 - memory-style error correcting code (Hamming Code ECC)
  - RAID 3 - bit-interleaved parity
  - RAID 4 - block-interleaved parity
  - RAID 5 - block-interleaved distributed-parity
  - RAID 6 - independent data disks with two independent distributed parity schemes (P+Q redundancy)
  - RAID 10 - striped disk array (RAID level 0) whose segments are mirrored (RAID level 1)
  - RAID 0+1 - mirrored array (RAID level 1) whose segments are RAID 0 arrays
  - RAID 03 - striped (RAID level 0) array whose segments are RAID level 3 arrays
  - RAID 50 - striped (RAID level 0) array whose segments are RAID level 5 arrays
  - RAID 53, 51, ...

DAS vs. NAS vs. SAN??

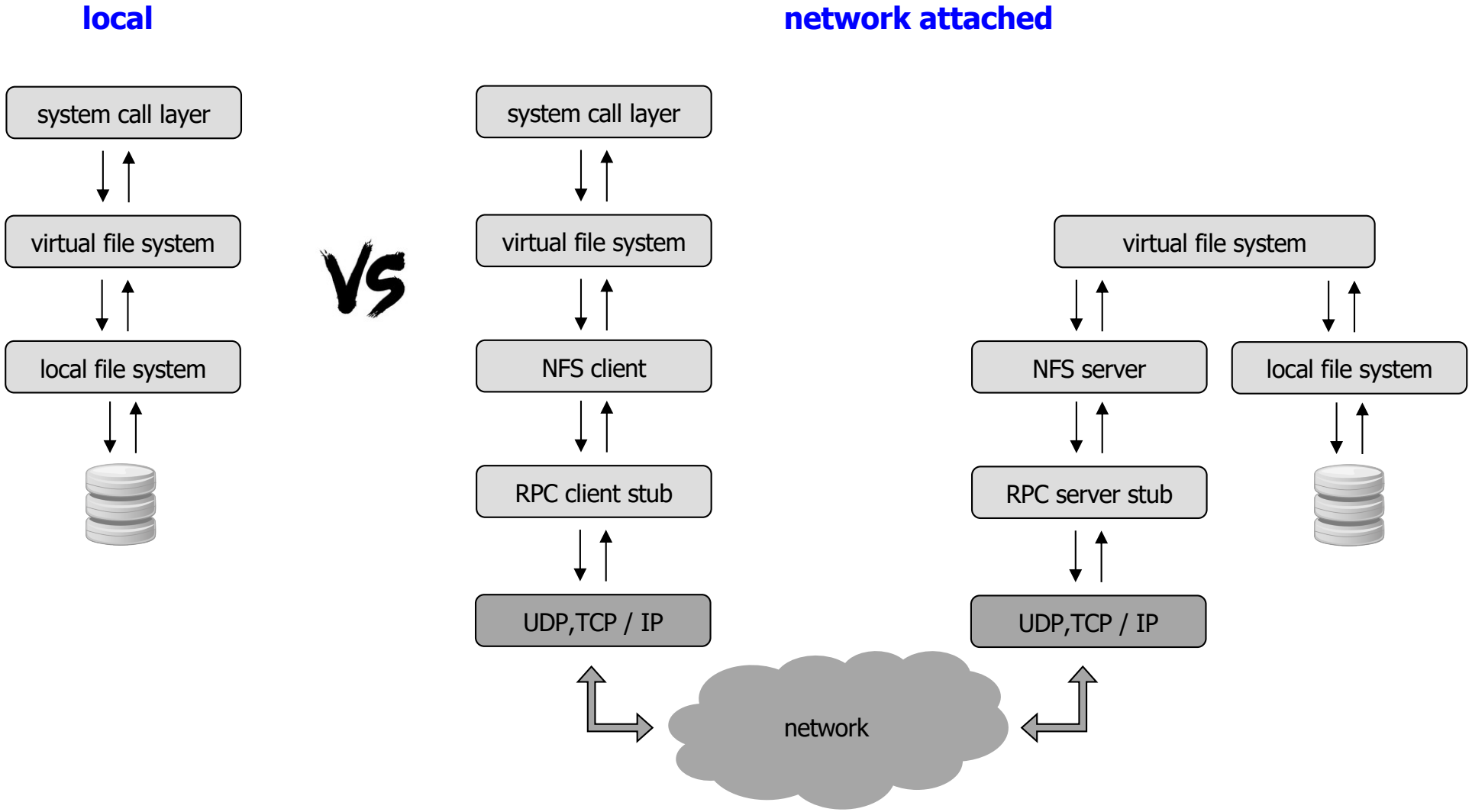
- How will the introduction of **network attached disks** influence storage?

- Direct attached storage
- Network attached storage
  - uses some kind of file-based protocol to attach remote devices non-transparently
  - NFS, SMB, CIFS
- Storage area network
  - transparently attach remote storage devices
  - iSCSI (SCSI over TCP/IP), iFCP (SCSI over Fibre Channel), HyperSCSI (SCSI over Ethernet), ATA over Ethernet



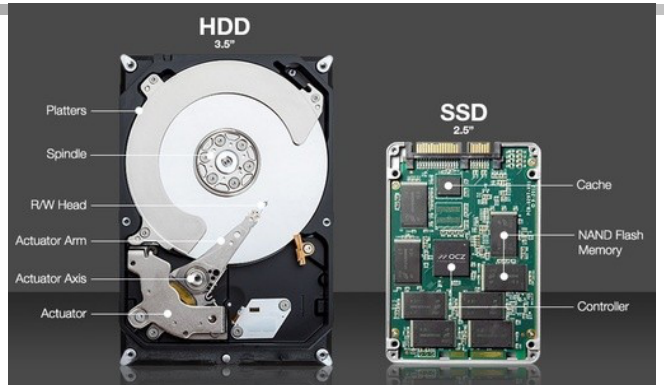
Example: Network File System (NFS)

- Distributed file system – allowing a client to access a file over a network



Mechanical Disks vs. Solid State Disks???

- How will the introduction of **SSDs** influence storage?



	Storage	Average (seek) / latency (ms)	Sustained transfer rate (MBps)	Interface (Gbps)
Seagate Cheetah X15		**3.4** (0.2)	110 - **171**	SAS (3) FC (4)
Seagate Savvio 15K (2.5"		**2.9** (track to track 0.2)	29 - **112**	SAS (3)
Sagate Barracuda Pro	14000	**8.5** (track to track 1.0)	**250**	SATA (6)
Intel X25-	64	**0.075**	**250**	SATA (3)
Intel Drive	800	**< 0.065**	**2000**	SAS (6)
Intel DC S3700 Se	2000	**0.020**	**2000**	PCIe, v3
Intel DC P3608	4000	**0.020**	**5000**	PCIe, v3

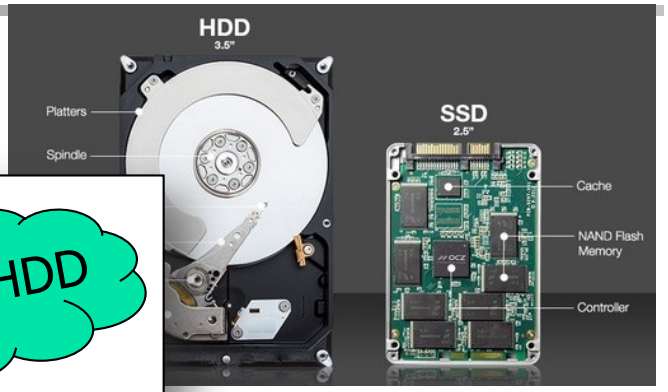
Komplett.no ~5 KNOK (2019):
- Standard HDD: 12 TB
- Standard SSD: 2 TB

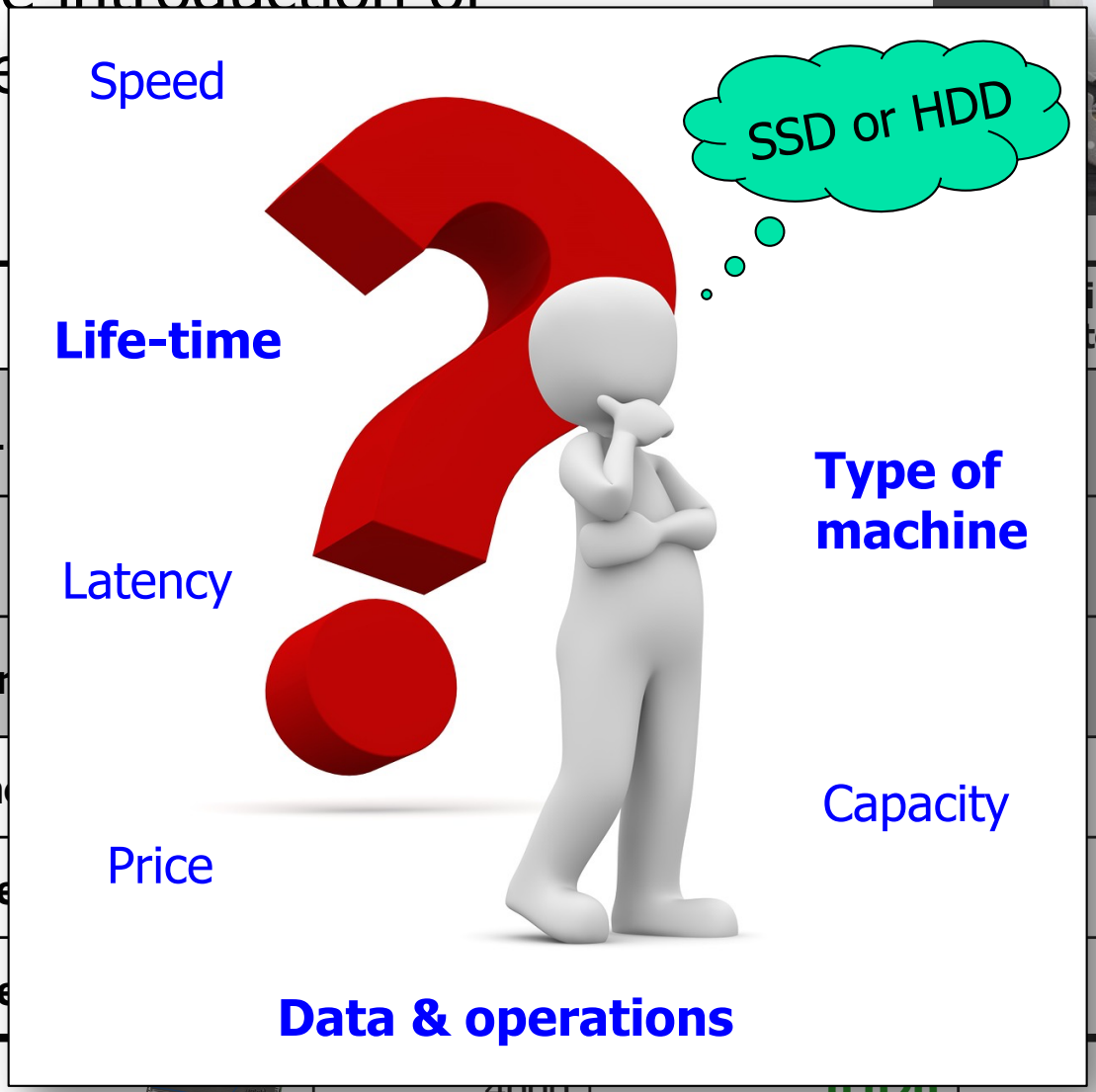
2017 Price? **$ 8.999**

2019 (3.2TB) = 80 KNOK

Mechanical Disks vs. Solid State Disks???

- How will the introduction of **SSDs** influe





				ined transfer te (MBps)	Interface (Gbps)
Seagate Cheetah X1				110 - **171**	SAS (3) FC (4)
Seagate Savvio 15K				29 - **112**	SAS (3)
Sagate Barracuda Pr				**250**	SATA (6)
Intel X25-E (extrem				**250**	SATA (3)
Intel Drive 910 Serie				**2000**	SAS (6)
Intel DC S3700 Serie				**2000**	PCIe, v3
Intel DC P3608		4000	0.020	**5000**	PCIe, v3

Summary

- Disks are the main persistent secondary storage device
- The main bottleneck is often disk I/O performance due to disk mechanics: seek time and rotational delays
- Much work has been performed to optimize disks performance
  - scheduling algorithms try to minimize seek overhead (most systems use SCAN derivates)
  - memory caching can save disk I/Os
  - additionally, many other ways (e.g., block sizes, placement, prefetching, striping, …)
  - world today more complicated (both different access patterns, unknown disk characteristics, …)
    → new disks are "smart", we cannot fully control the device
- File systems provide
  - file management – store, share, access, …
  - storage management – of physical storage
  - access methods – functions to read, write, seek, …
  - …
- New non-mechanical storage **may** change the way of thinking…!!??