

INF1060:

Introduction to Operating Systems and Data Communication



Operating Systems:

Storage: Disks & File Systems

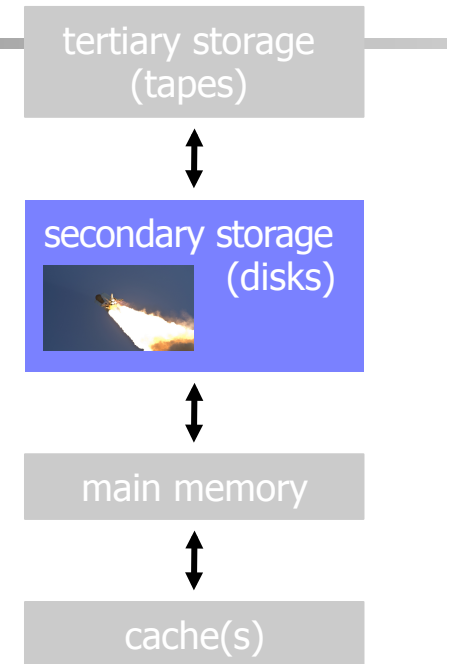
Wednesday 8 October 14

Overview

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

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



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

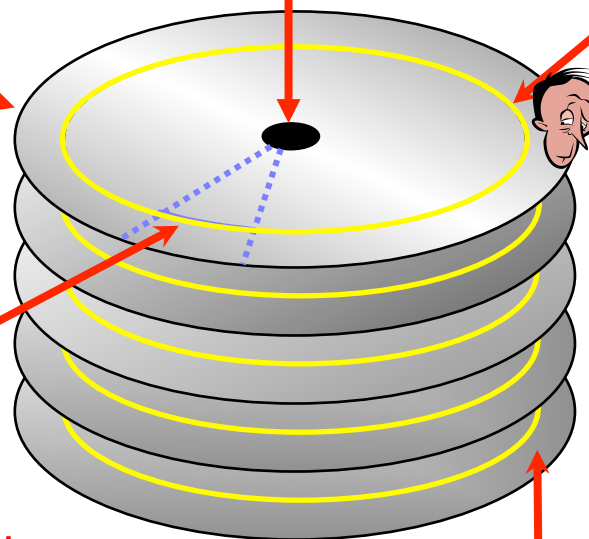
read or alter the magnetism (bits) passing under it. The heads are attached to an arm enabling it to move across the platter surface

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 = $8 \times 18497 \times 617 \times 512 \approx 4.6 \times 10^{10} = 42.8 \text{ GB}$
 - **Formatted** capacity = **36.7 GB**

Note:

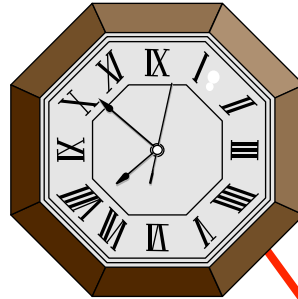
there is a difference between formatted and total capacity. Some of the capacity is used for storing checksums, **spare tracks**, etc.

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) are 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*

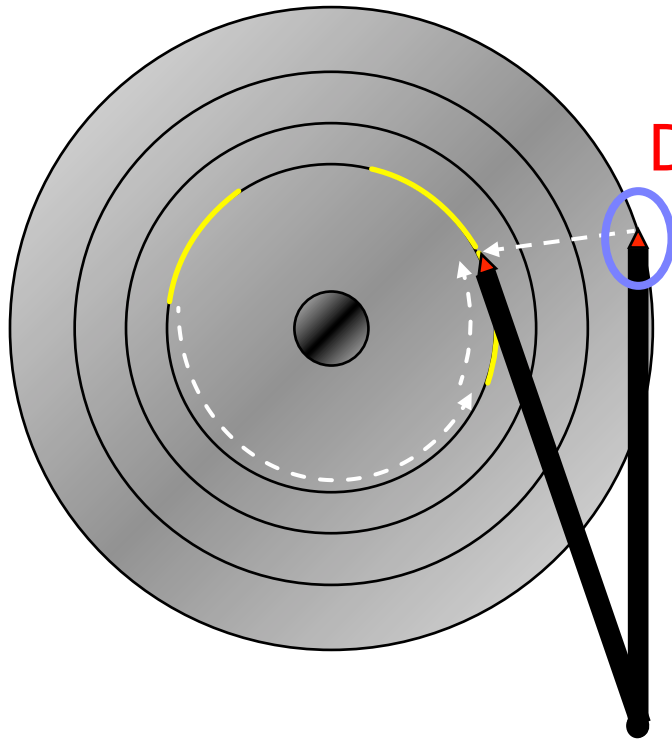
Disk Access Time

I want
block X



block x
in memory

Disk platter



Disk head

Disk arm

Disk access time =

Seek time

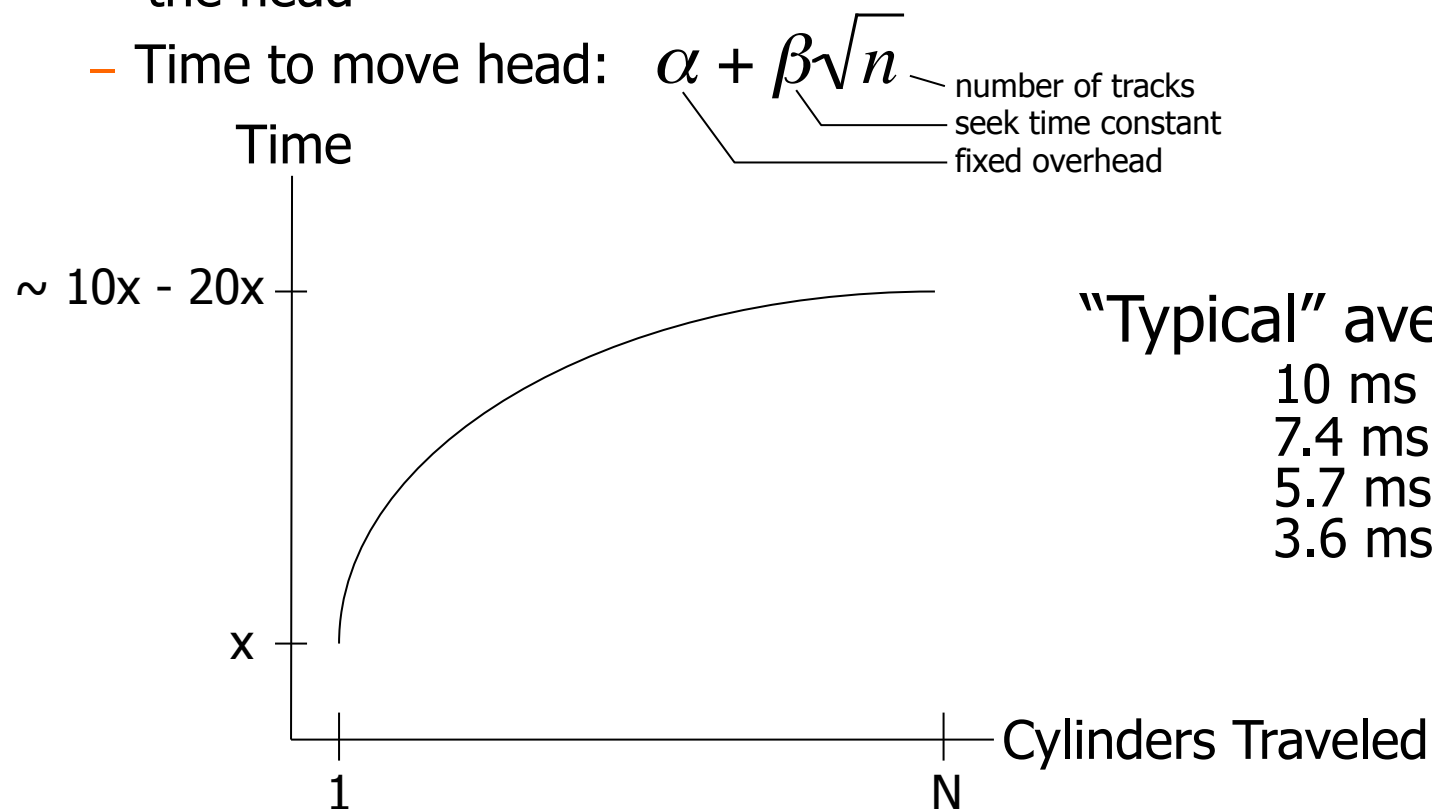
+ Rotational delay

+ Transfer time

+ Other delays

Disk Access Time: Seek Time

- Seek time is the time to position the head
 - some time is used for actually moving the head – roughly proportional to the number of cylinders traveled
 - the heads require a minimum amount of time to start and stop moving the head
 - Time to move head: $\alpha + \beta\sqrt{n}$



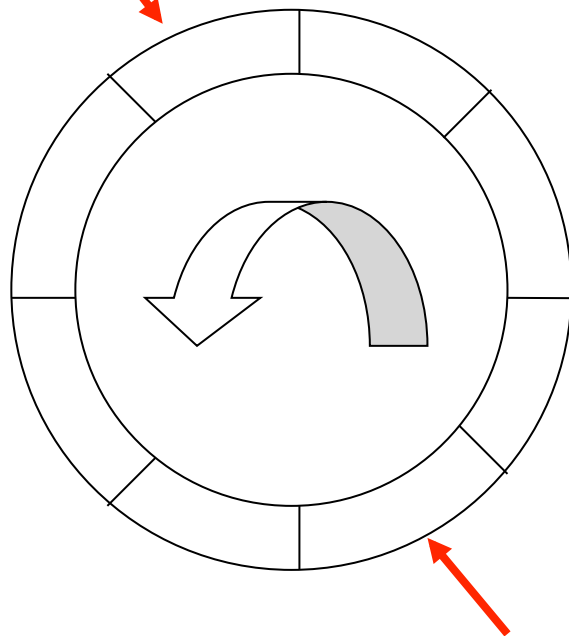
“Typical” average:

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

head here



block I want

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**
- Transfer rate = $\frac{\text{amount of data per track}}{\text{time per rotation}}$
- Transfer time = amount of data to read / transfer rate
- Transfer rate example
 - *Barracuda 180*:
406 KB per track x 7.200 RPM \approx 47.58 MB/s
 - *Cheetah X15*:
306 KB per track x 15.000 RPM \approx 77.15 MB/s
- If we have to change track, time must also be added for **moving the head**

Note:

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

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
 - contention for bus
 - contention for memory
 - verifying block correctness with checksums (retransmissions)
 - **waiting in scheduling queue**
 - ...
- Typical values: “0”
(maybe except from waiting in the 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}

	<i>Barracuda 180</i>	<i>Cheetah 36</i>	<i>Cheetah X15.3</i>
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
disk buffer cache	16 MB	4 MB	8 MB

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** \approx 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** \approx 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
 - there are large gaps in access times between disks and memory, i.e., a disk access will probably dominate the total execution time
 - there may be 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
 - file management / data placement
 - disk scheduling
 - multiple disks
 - prefetching
 - memory caching / replacement algorithms
 - ...

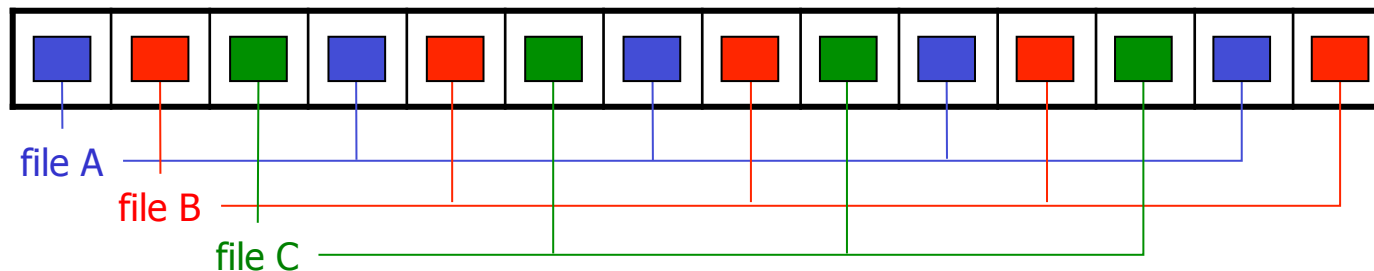
- 4 KB
- various
- SCAN derivate
- a specific RAID level
- read-ahead
- LRU variant



Data Placement

Data Placement on Disk

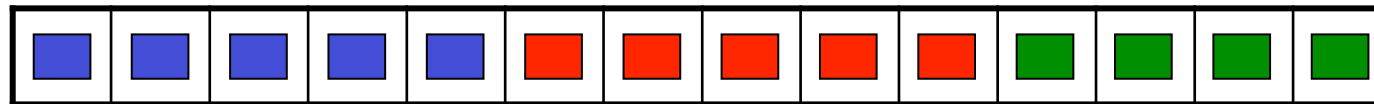
- **Interleaved** placement tries to store blocks from a file with a fixed number of other blocks in-between each block



- 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



file A

file B

file C

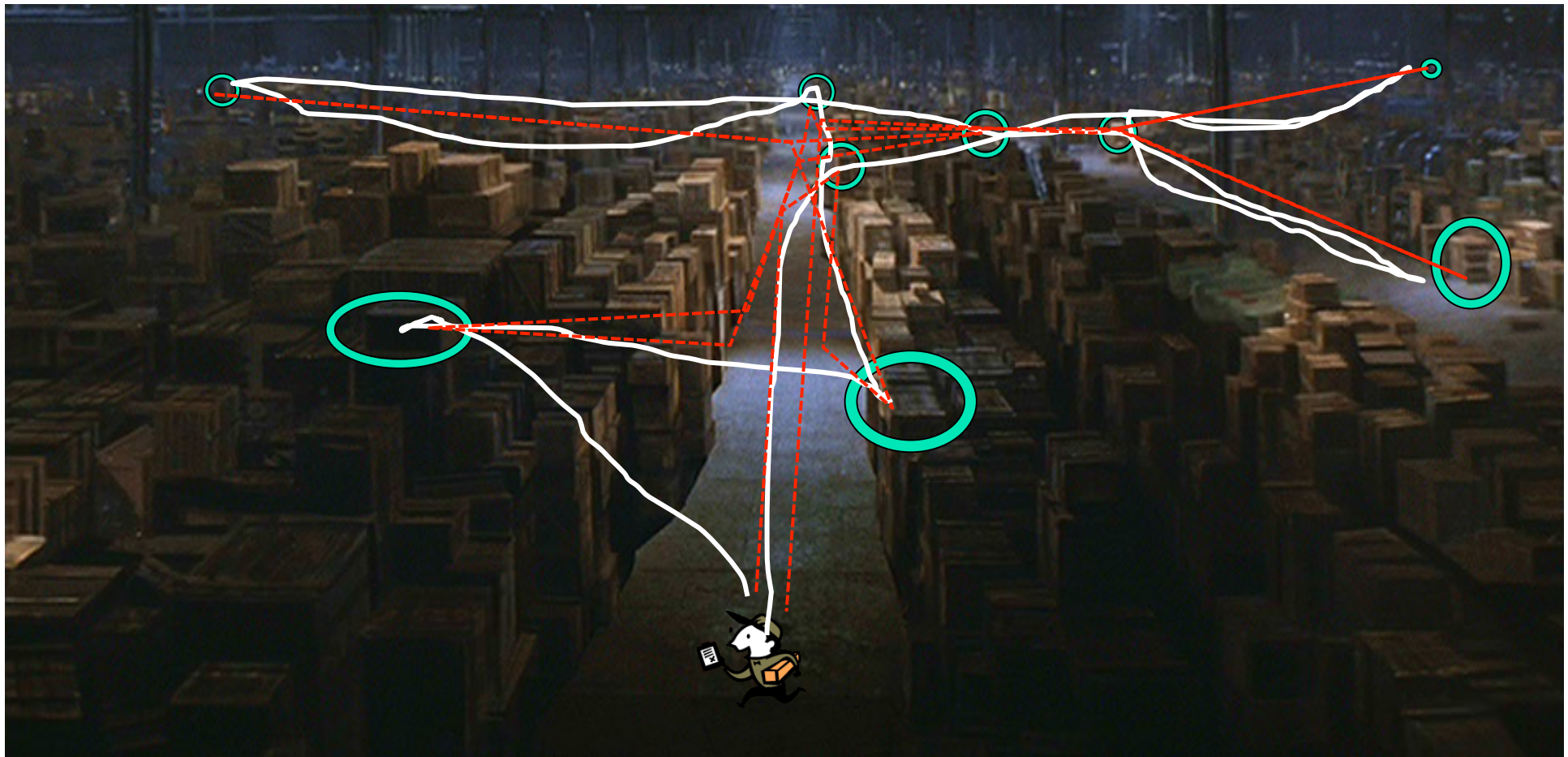
- minimal disk arm movement reading the whole file (no intra-file seeks)
- pros/cons
 - 😊 head must not move between read operations - no seeks / rotational delays
 - 😊 can approach theoretical transfer rate
 - 😞 but usually we read other files as well (giving possible large inter-file seeks)
- real 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



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** \neq **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 and rotational delay** 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

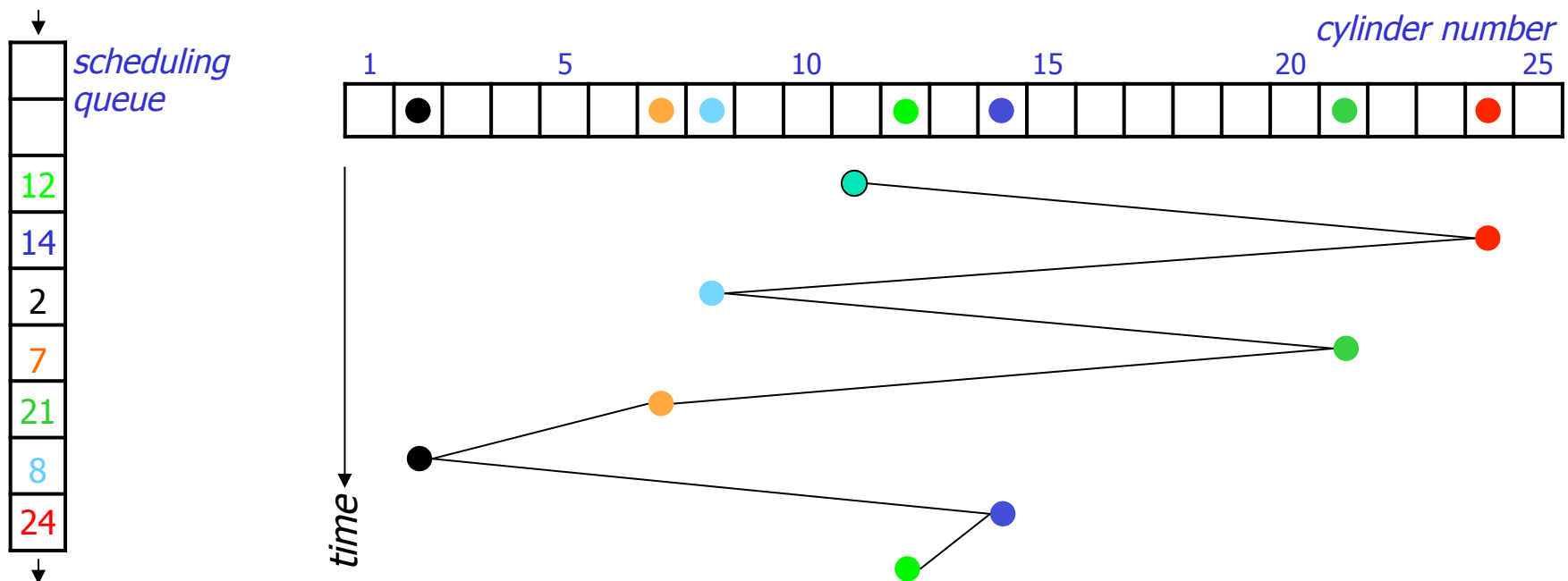
First-Come-First-Serve (FCFS)

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



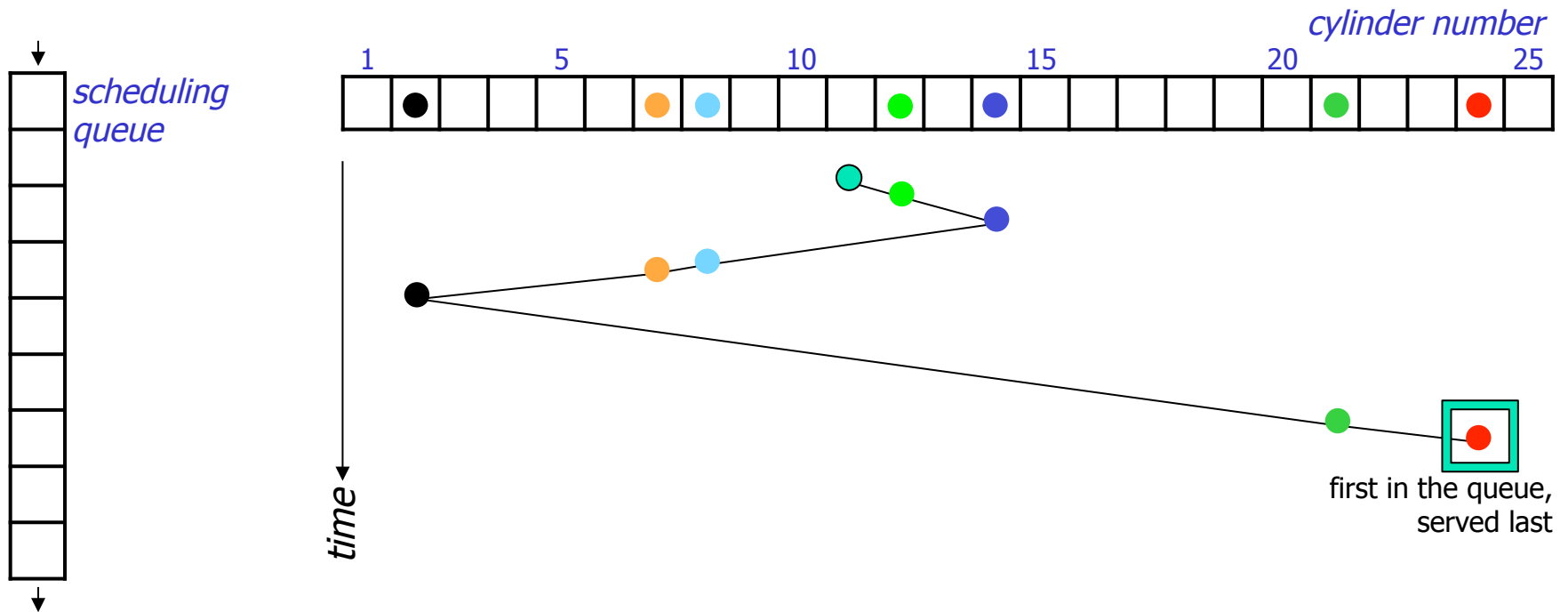
Shortest Seek Time First (SSTF)

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



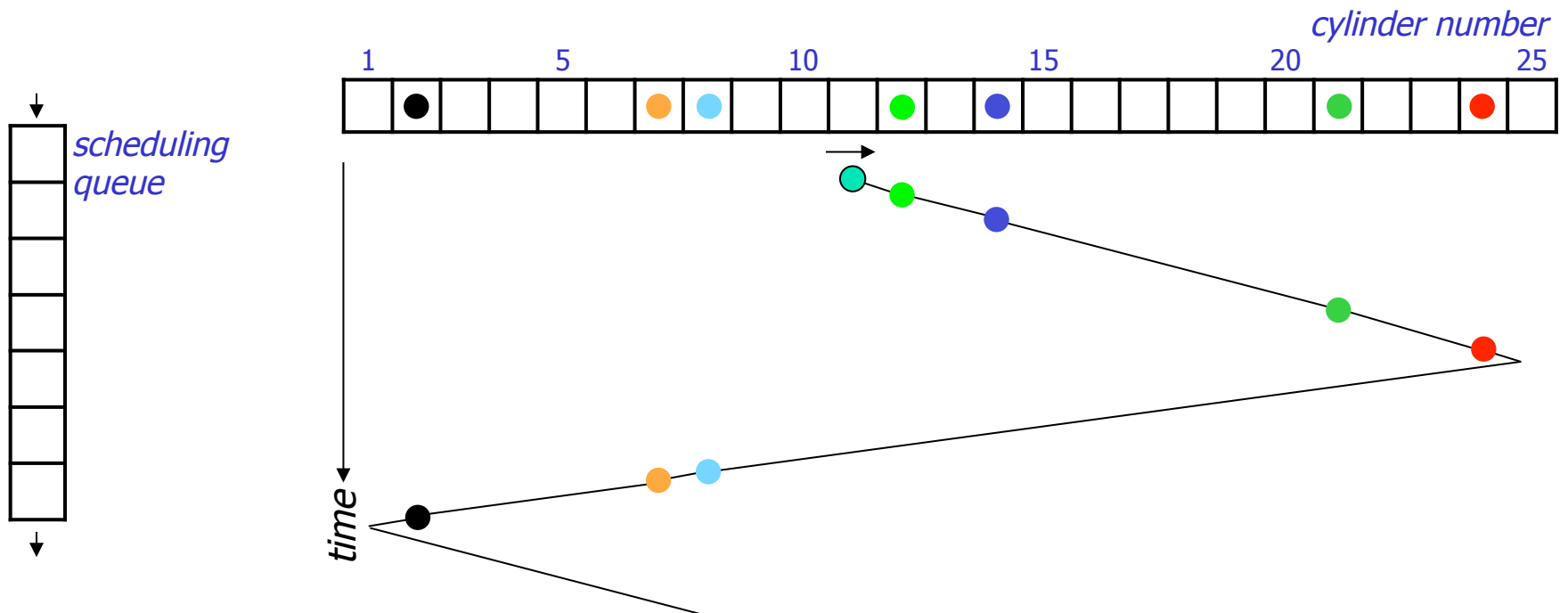
SCAN

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):

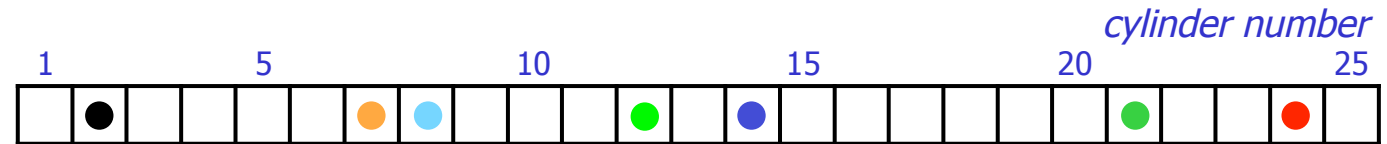
12 14 2 7 21 8 24



SCAN vs. FCFS

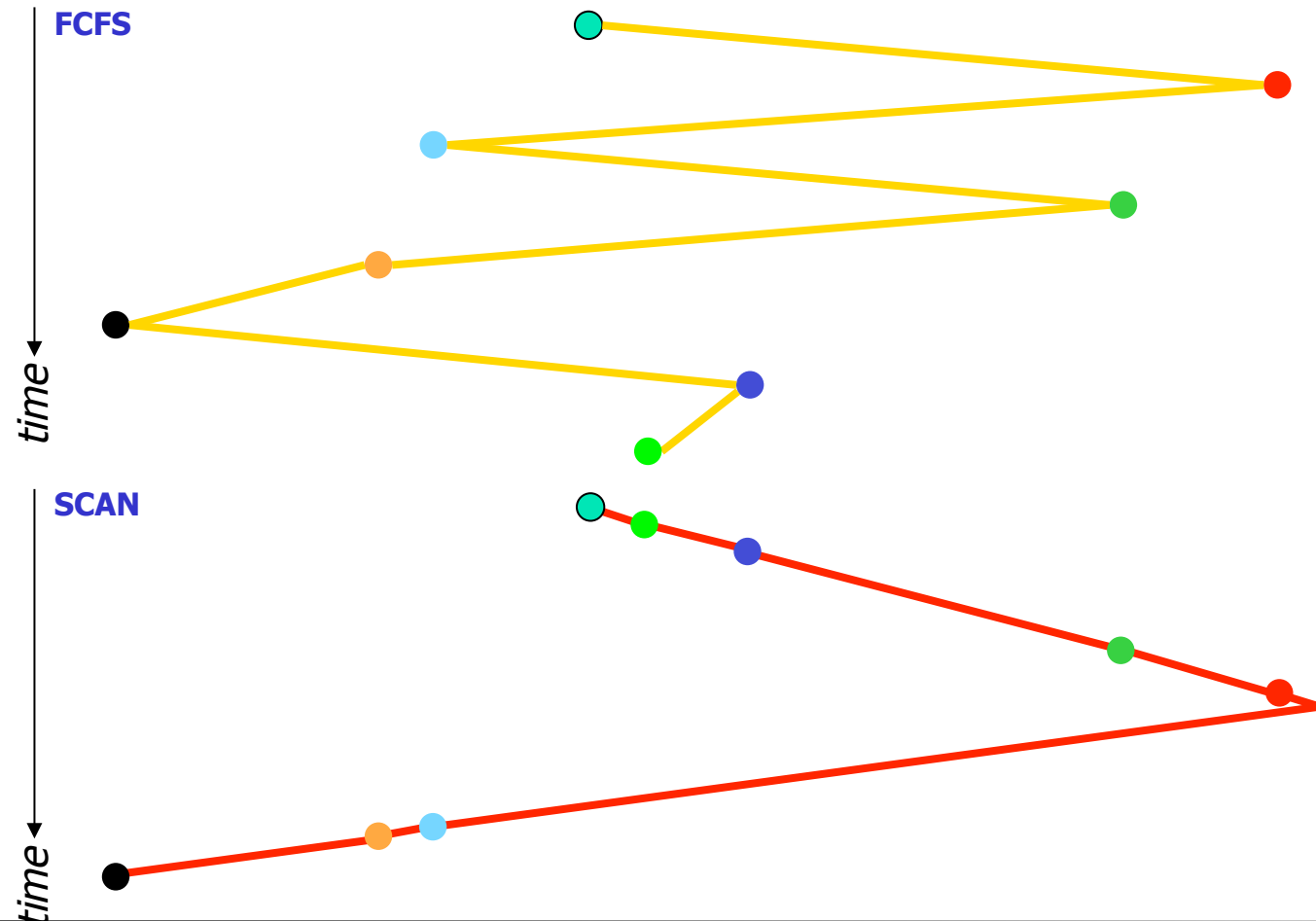
incoming requests (in order of arrival): 12 14 2 7 21 8 24

- 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
 - ...



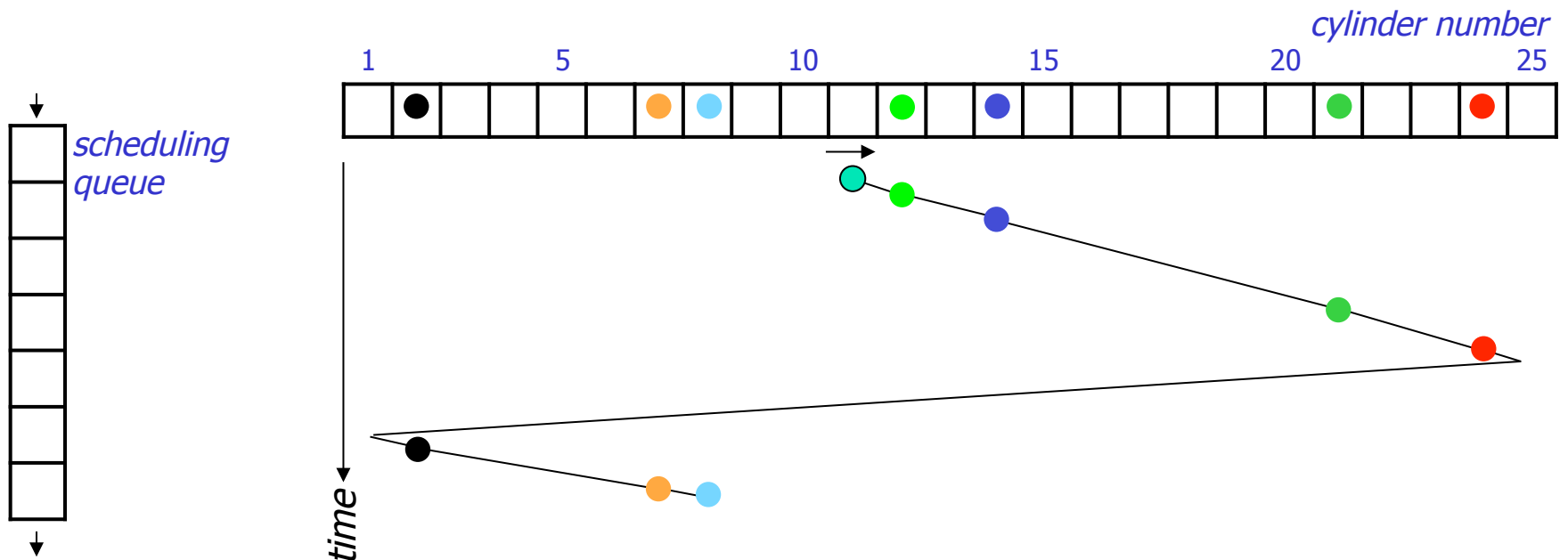
C-SCAN

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**

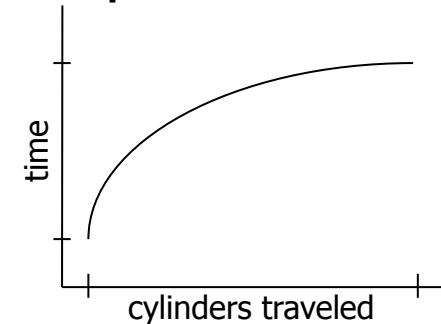




SCAN vs. C-SCAN

- 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: $\alpha + \beta\sqrt{n}$
 - fixed overhead
 - seek time constant
 - number of cylinders



SCAN	C-SCAN
bi-directional 	uni-directional 
requests: n avg. dist: $2x$ total cost: $n \times \sqrt{2x} = (n \times \sqrt{2}) \times \sqrt{x}$	requests: n avg. dist: x total cost: $\sqrt{n \times x} + n \times \sqrt{x} = (\sqrt{n} + n) \times \sqrt{x}$
<p>if n is large:</p> $n \times \sqrt{2} > \sqrt{n} + n$	

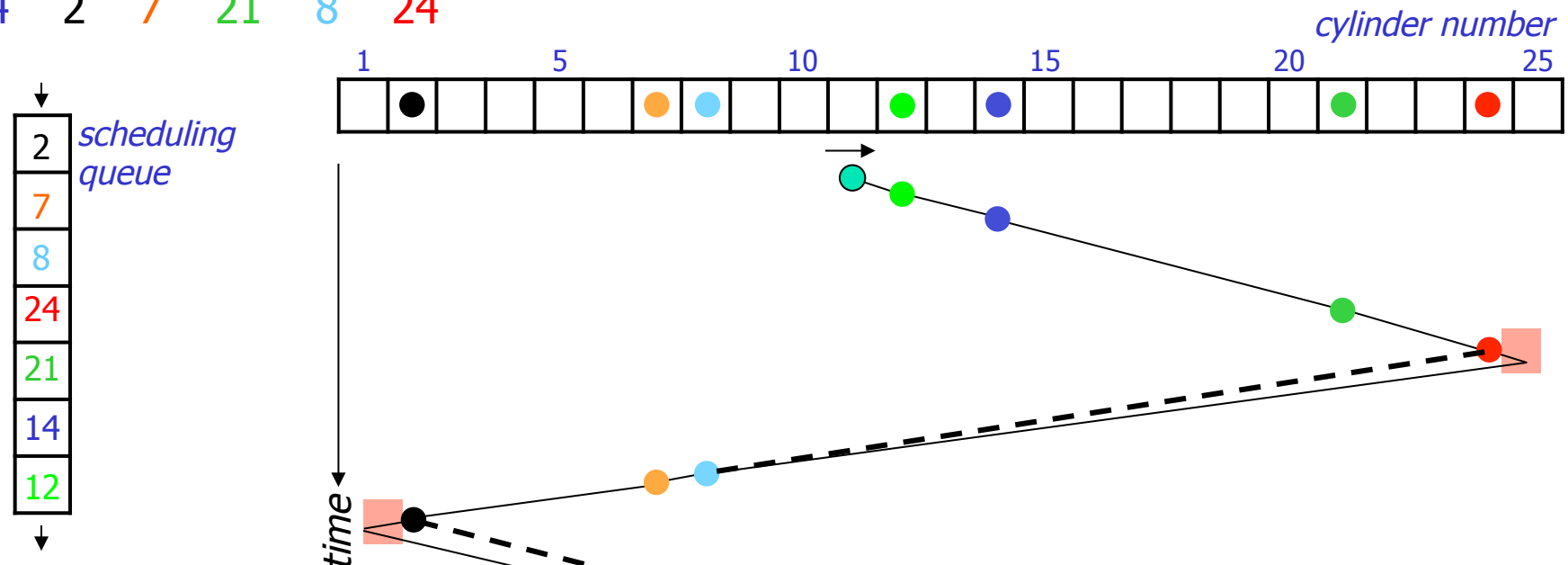
LOOK and C-LOOK

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 request
- increased efficiency
- SCAN vs. LOOK example:

incoming requests (in order of arrival):

12 14 2 7 21 8 24



V-SCAN(R)

- **V-SCAN(R)** combines SCAN (or LOOK) and SSTF
 - define an **R**-sized unidirectional SCAN window, i.e., C-SCAN, and use SSTF outside the window
 - Example: V-SCAN(0.6)
 - makes a C-SCAN window over 60 % of the cylinders
 - uses SSTF for requests outside the window

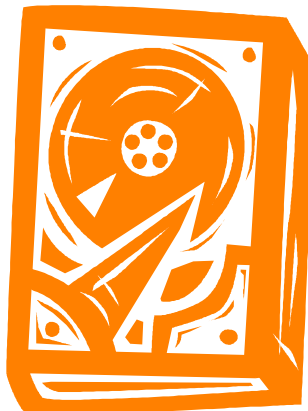


- V-SCAN(0.0) equivalent with SSTF
- V-SCAN(1.0) equivalent with C-SCAN
- V-SCAN(0.2) is supposed to be an appropriate configuration

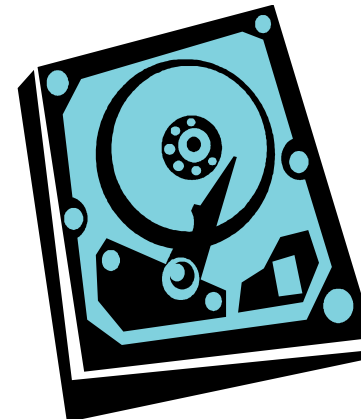
Modern Disk Scheduling

- 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

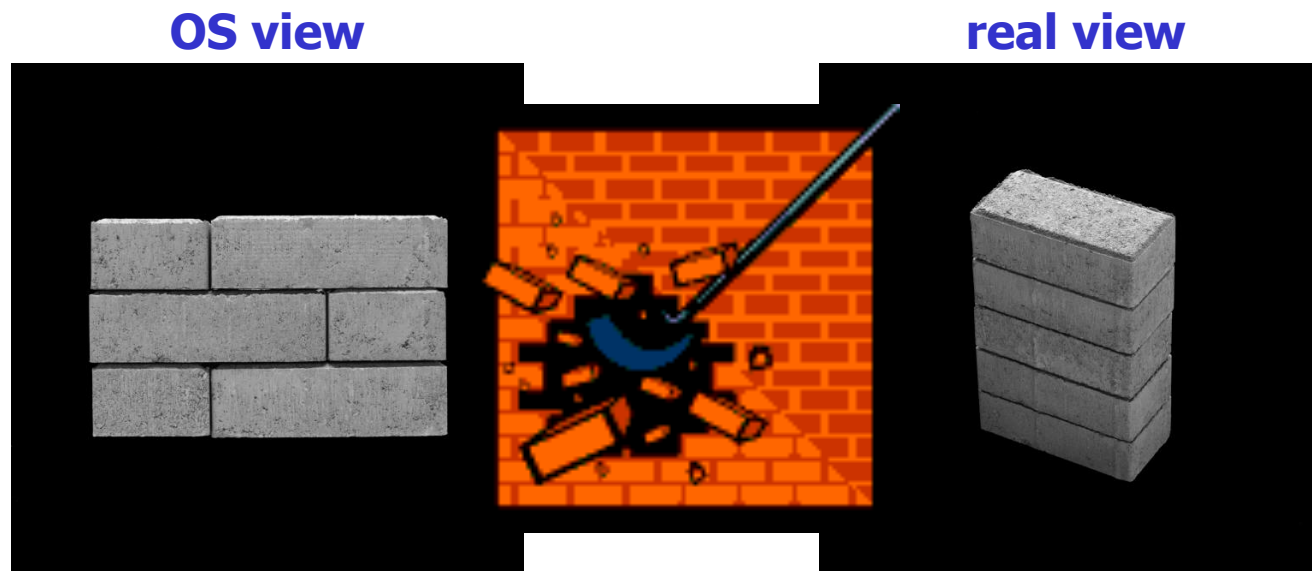


real view



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

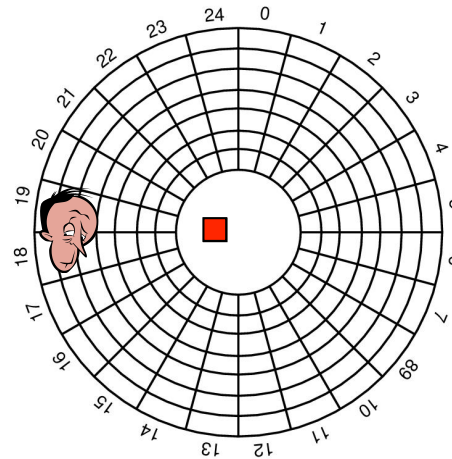
Seagate X15.3:

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

- Disk used to be simple devices & performed by OS (file system or disk controller)
- ... but, new disks are more complex
 - hide their true layout
 - transparently move blocks to spare
 - have different zones

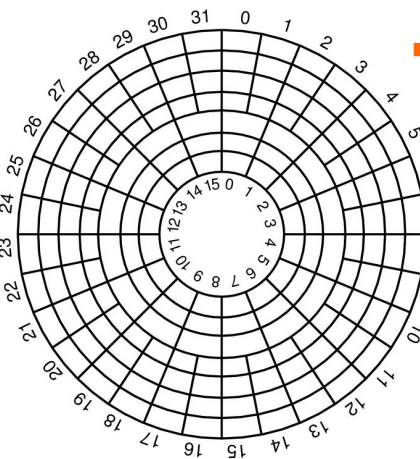
OS view

- **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

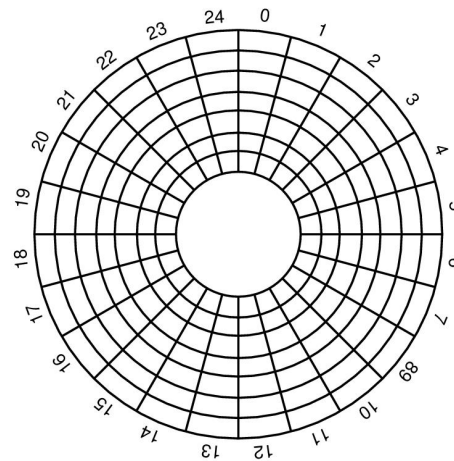
For handouts

- 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

OS view

■ Constant angular velocity (CAV) disks

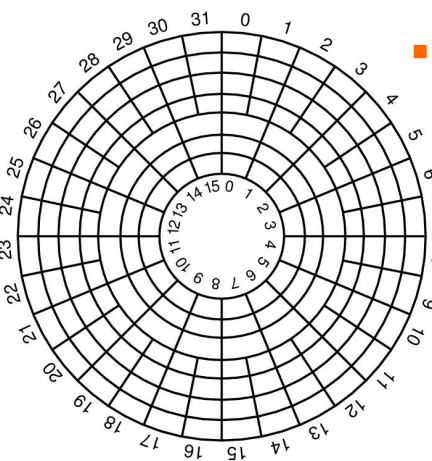
- constant rotation speed
- equal amount of data in each track
- ⇒ thus, **constant transfer time**



real view

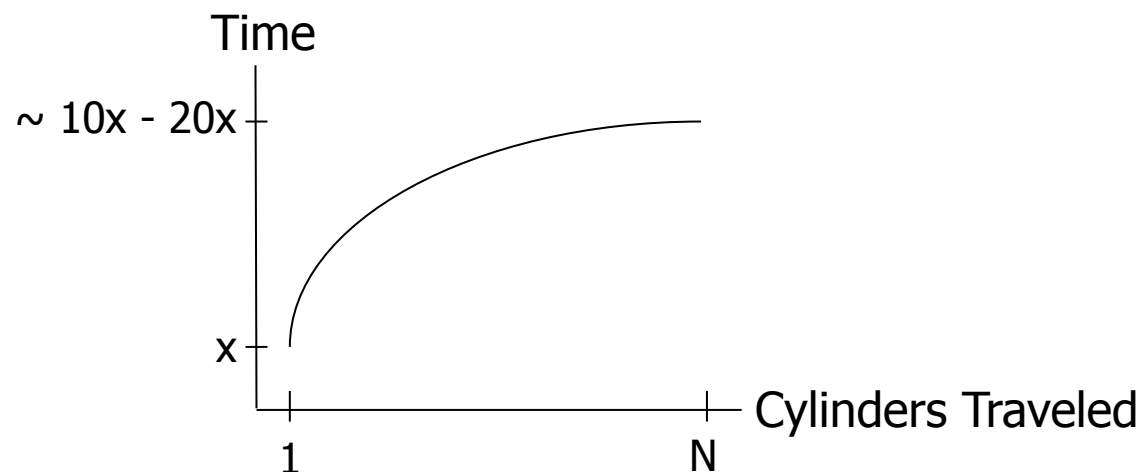
■ Zoned CAV disks

- 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**



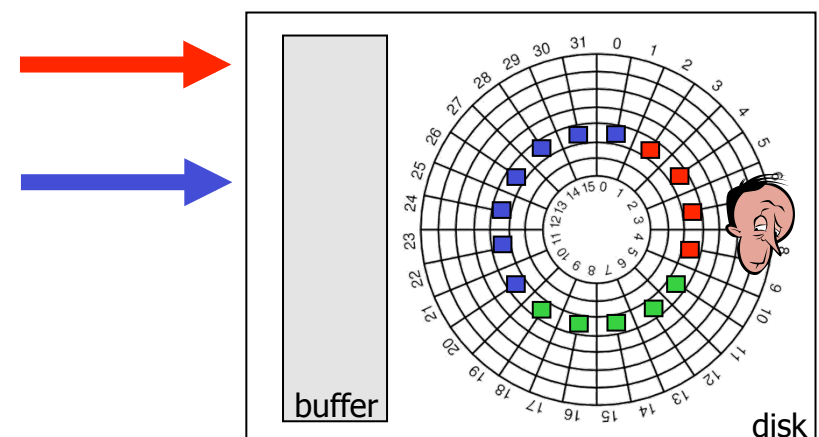
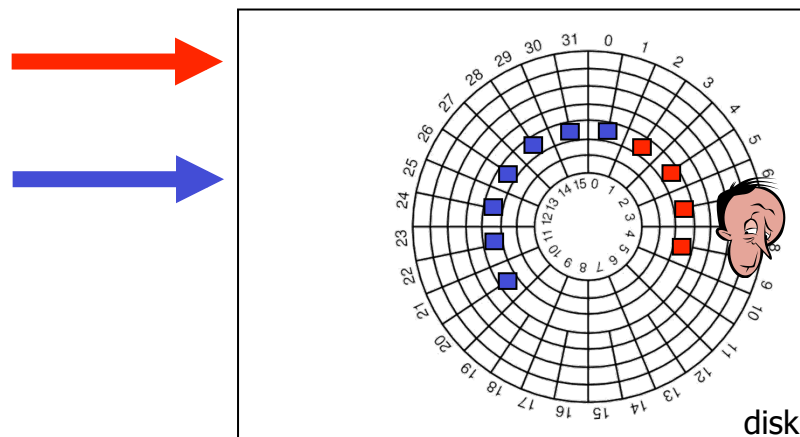
Modern Disk Scheduling

- 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 (e.g., a streaming scenario)
- 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)
 - work-conserving

```
[diamant] ~ > more /sys/block/sda/queue/scheduler  
noop anticipatory deadline [cfq]
```

Cooperative user-kernel space scheduling



⇒ **GNU/BSD Tar vs. QTAR**



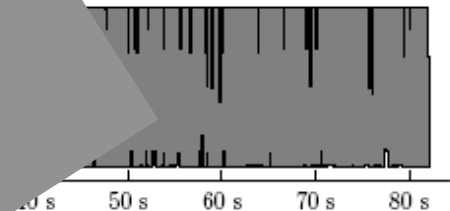
Cooperative user-kernel space scheduling

- 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`)
- ...

976768065

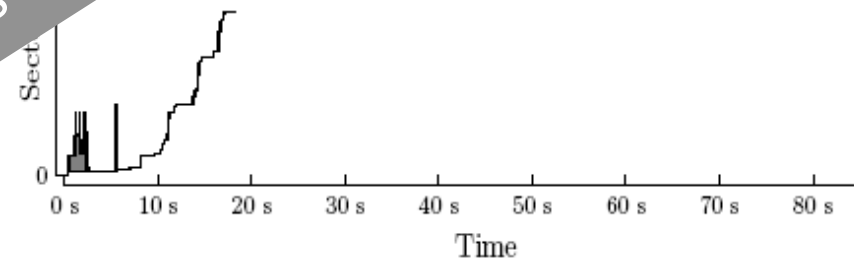


Time

(a) GNU tar (ext4)

- Only application access pattern

- use io to retrieve
- sort in use
- send I/O requests according to sorted list



(b) Qtar (ext4)

⇒ GNU/BSD Tar vs. QTAR



Cooperative user-kernel space scheduling *For handouts*

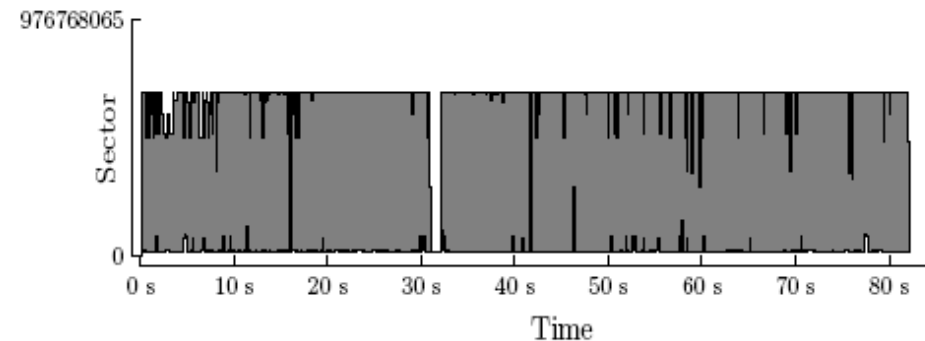
- Some times the kernel does not have enough information to make an efficient schedule

File tree traversals

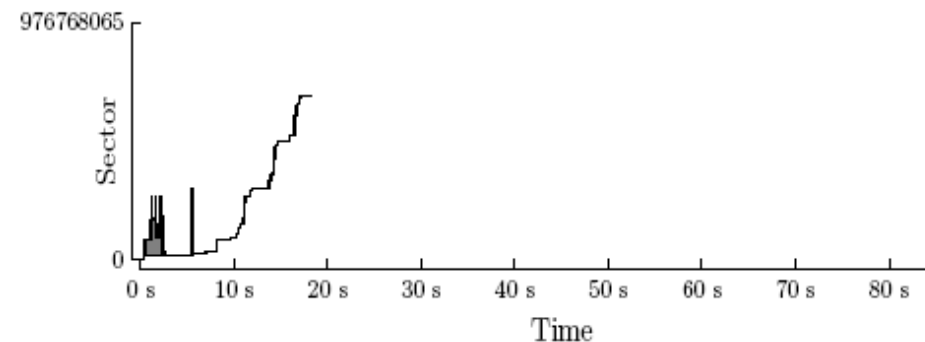
- *processing one file after another*
- tar, zip, ...
- recursive copy (`cp -r`)
- search (`find`)
- ...

- Only application knows access pattern

- use `ioctl FIEMAP (FIBMAP)` to retrieve extent locations
- sort in user space
- send I/O request according to sorted list



(a) GNU tar (ext4)



(b) Qtar (ext4)

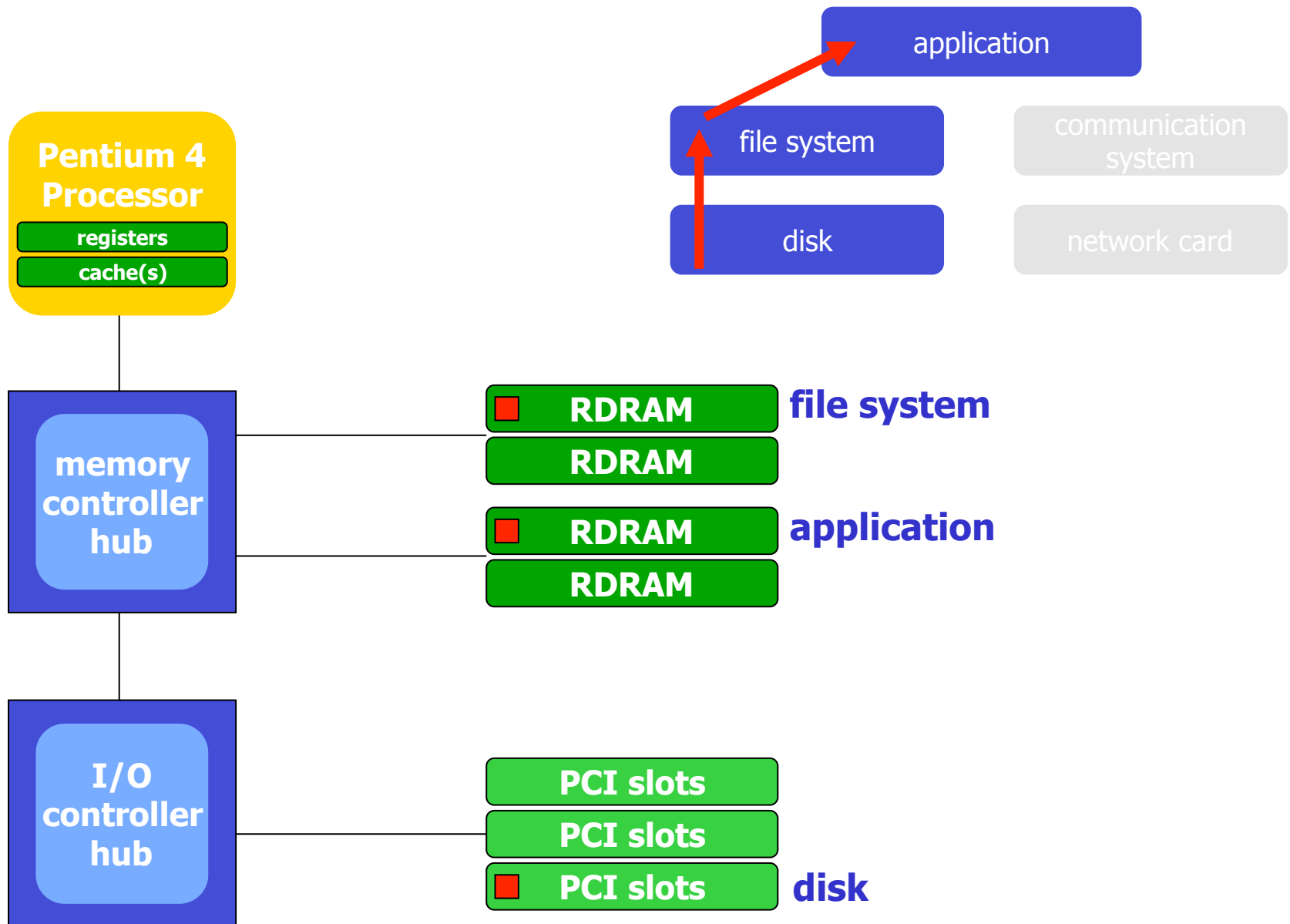
⇒ GNU/BSD Tar vs. QTAR



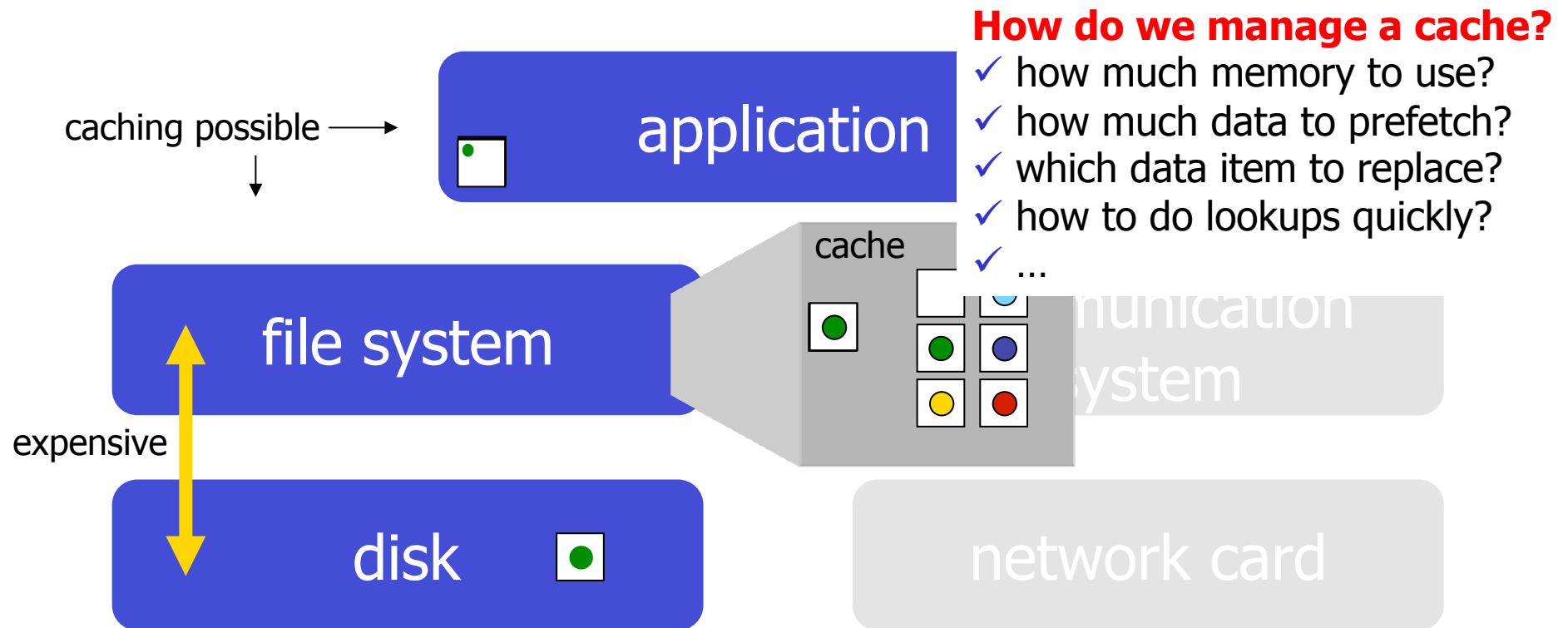


Memory Caching

Data Path (Intel Hub Architecture)



Buffer Caching



Buffer Caching: Windows XP

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

- I/O requests processing:

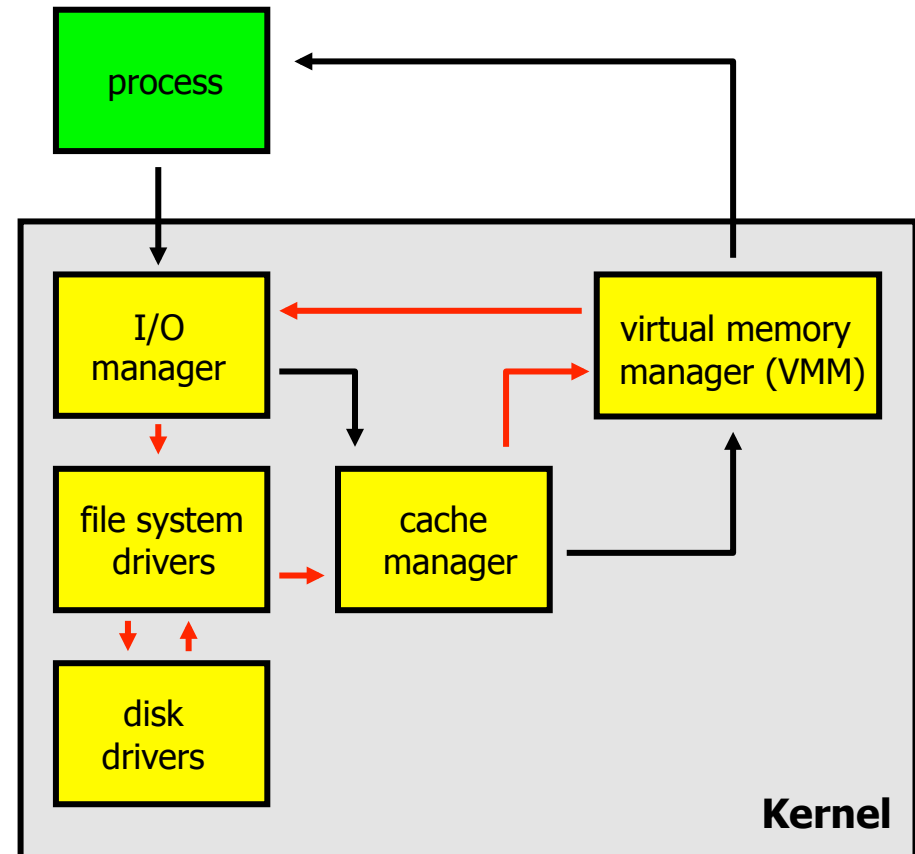
1. I/O request from process
2. I/O manager forwards to cache manager

- **in cache:**

3. cache manager locates and copies data to process buffer via VMM
4. VMM notifies process

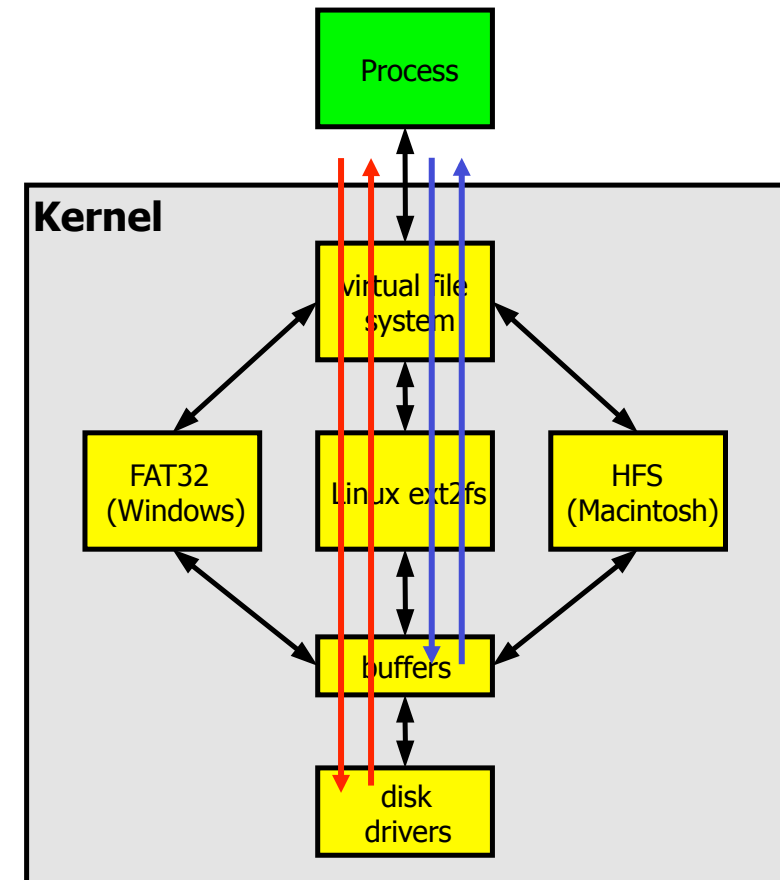
- **on disk:**

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

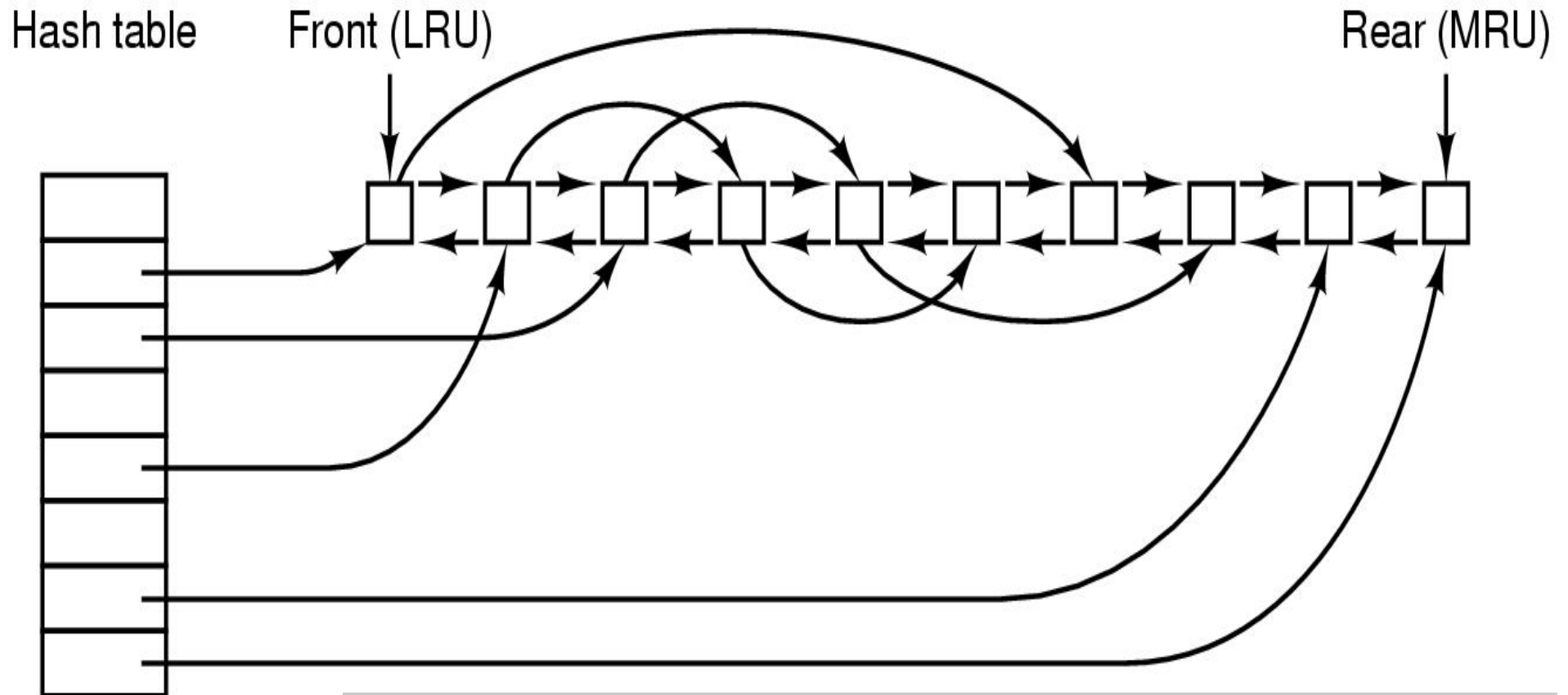


Buffer Caching: Linux / Unix

- 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

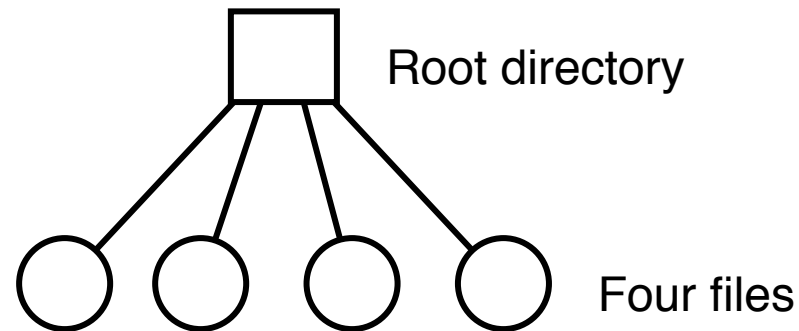
File Systems

- File systems organize data in files and manage access regardless of device type, e.g.:
 - **storage management** – allocating space for files on secondary storage
 - **file management** – providing 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

Organizing Files - Directories

- A system usually has a large number of different files
- To organize and quickly locate files, file systems use **directories**
 - contain no data itself
 - file containing name and locations of other files
 - several types
 - single-level (flat) directory structure
 - hierarchical directory structure

Single-level Directory Systems



- CP/M
 - Microcomputers
 - Single user system
- VM
 - Host computers
 - “Minidisks”: one partition per user

Hierarchical Directory Systems

■ Tree structure

- nodes = directories
- root node = root directory
- leaves = files

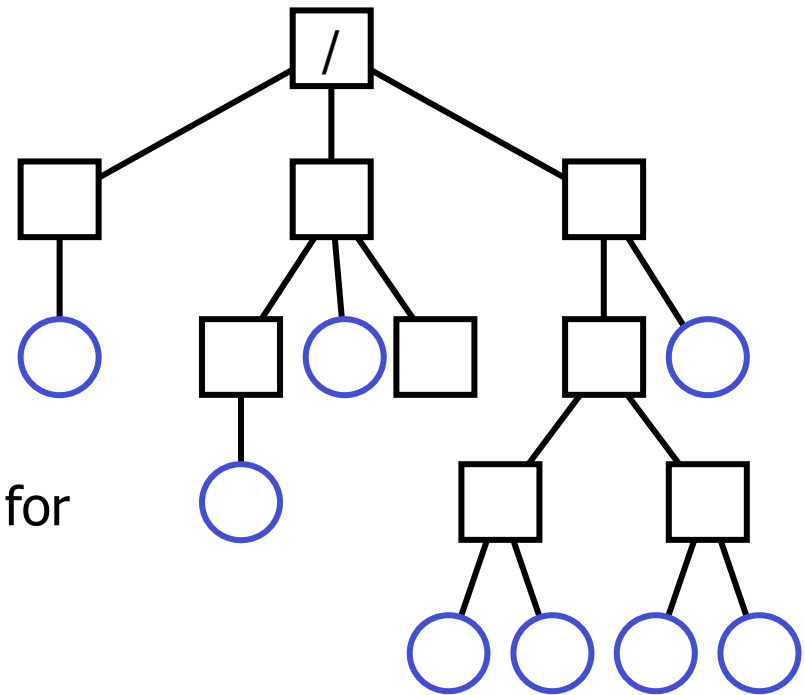


■ Directories

- stored on disk
- attributes just like files

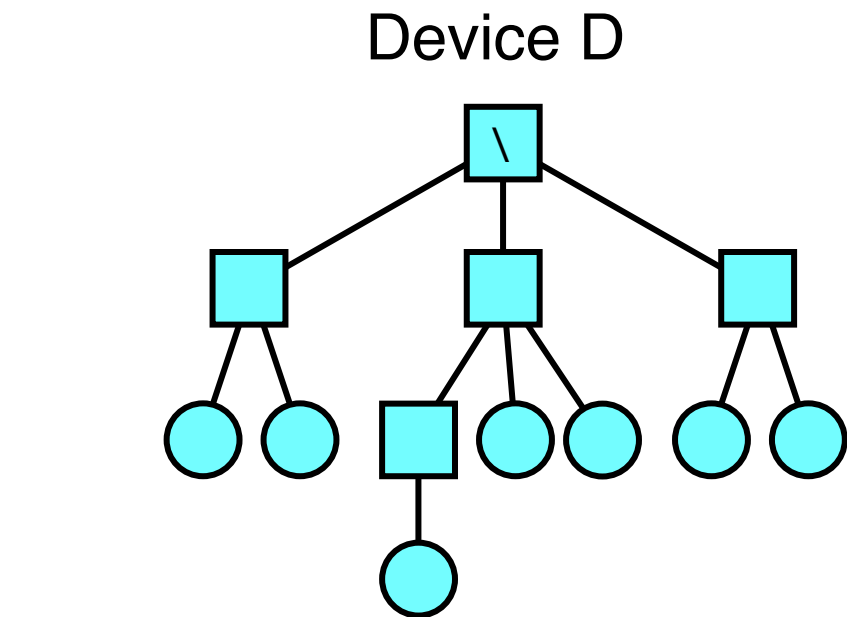
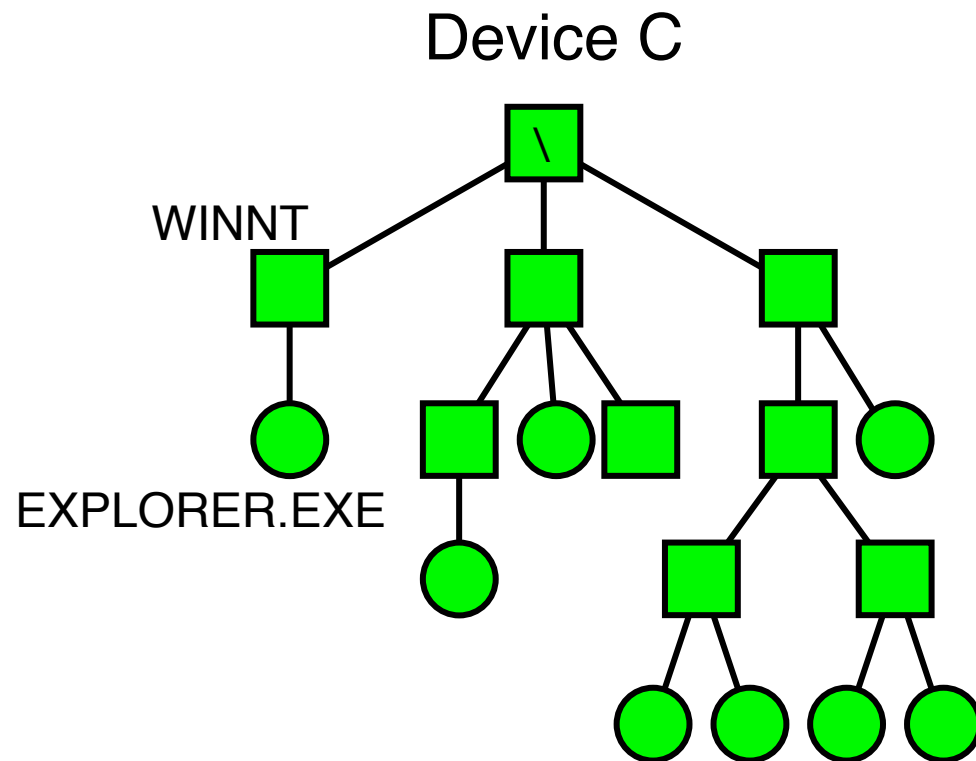
■ To access a file

- must (often) test all directories in path for
 - existence
 - being a directory
 - permissions
- similar tests on the file itself



Hierarchical Directory Systems

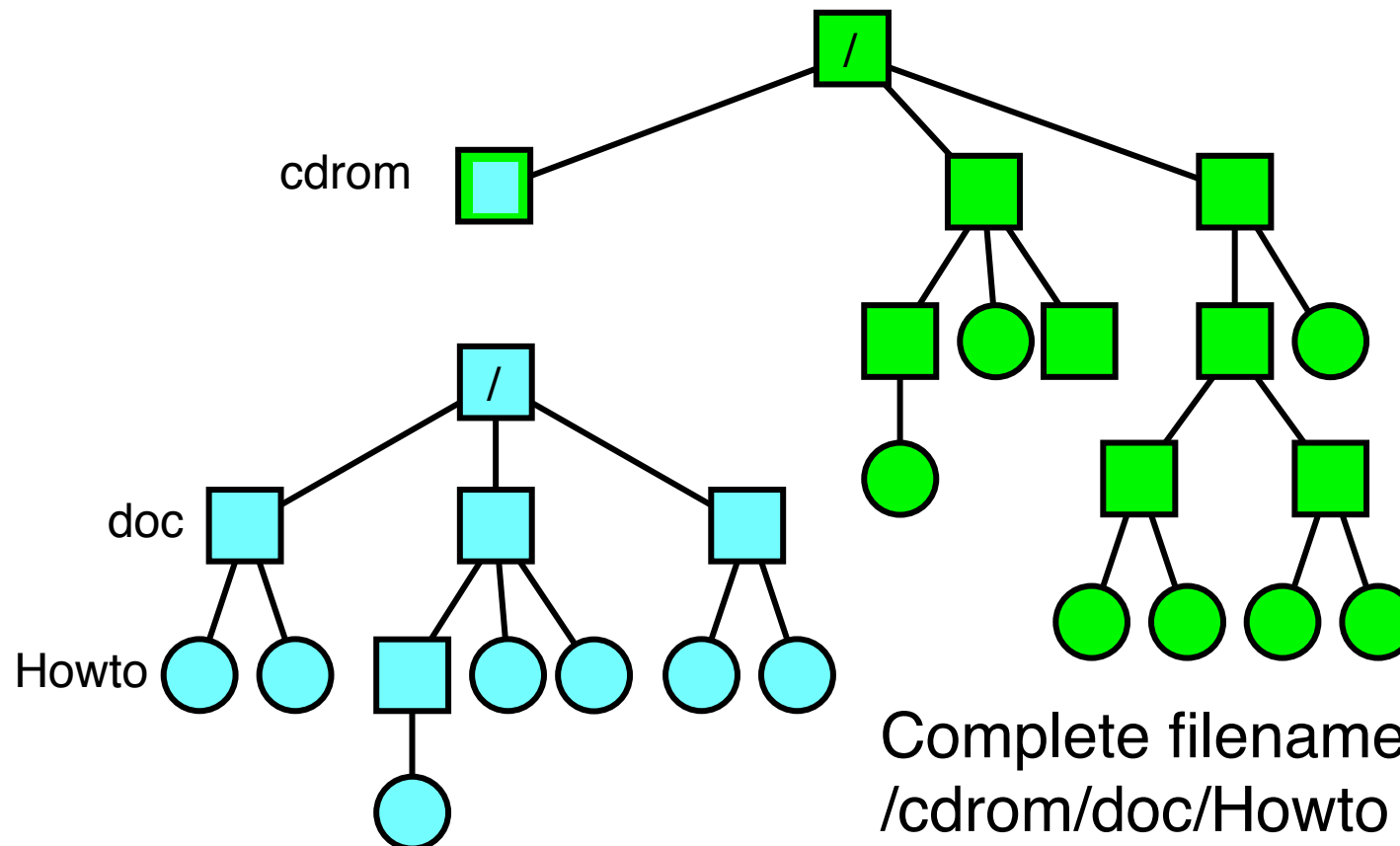
- **Windows**: one tree per partition or device



Complete filename example:
C:\WinNT\EXPLORER.EXE

Hierarchical Directory Systems

- **Unix:** single acyclic graph spanning several devices



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 */
    }

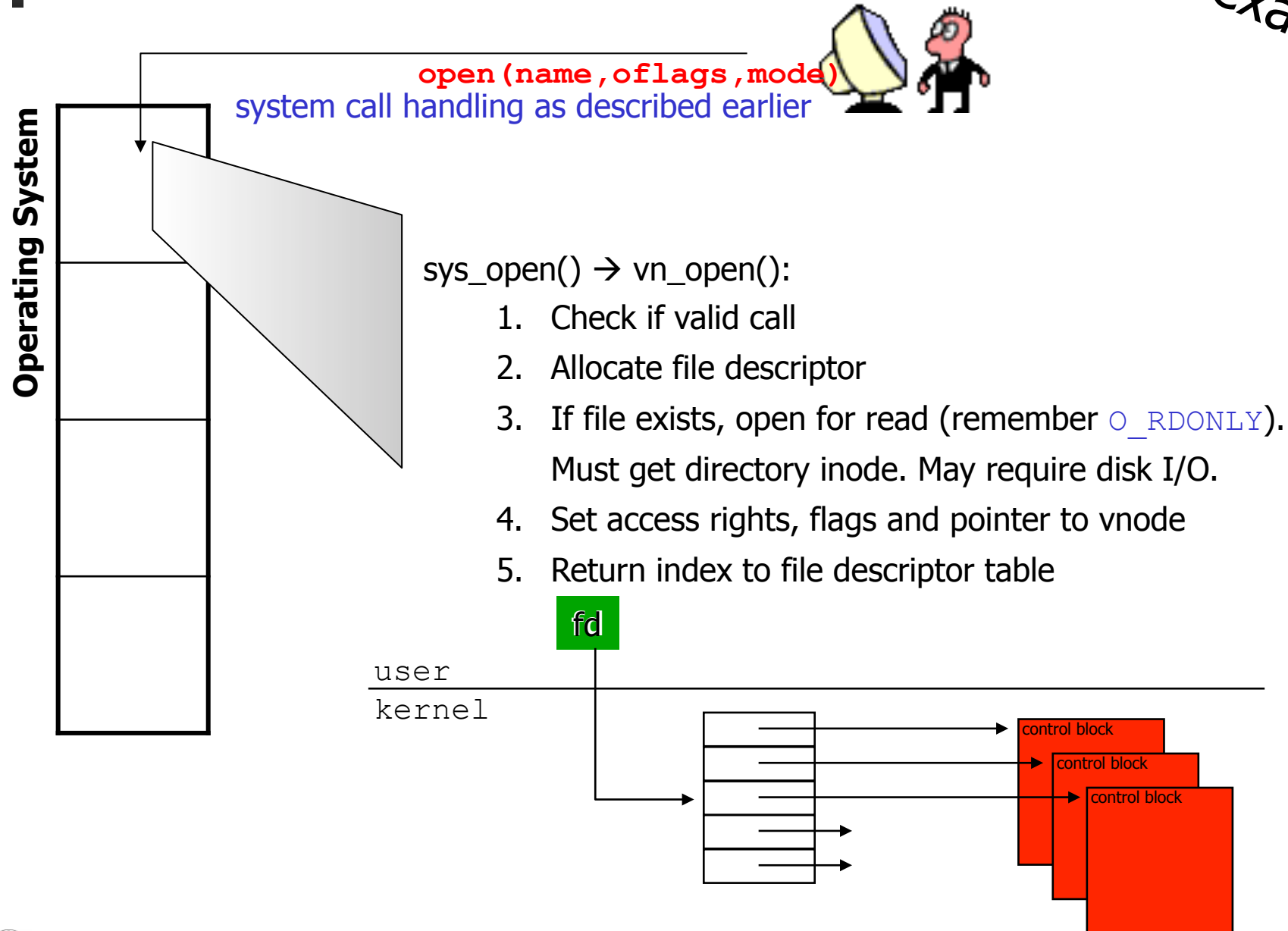
    while ((n = read(fd, buf, BUFSIZE) > 0) {
        <<USE DATA IN BUFFER>>
    }

    close(fd);

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

Open

BSD example



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 */
    }

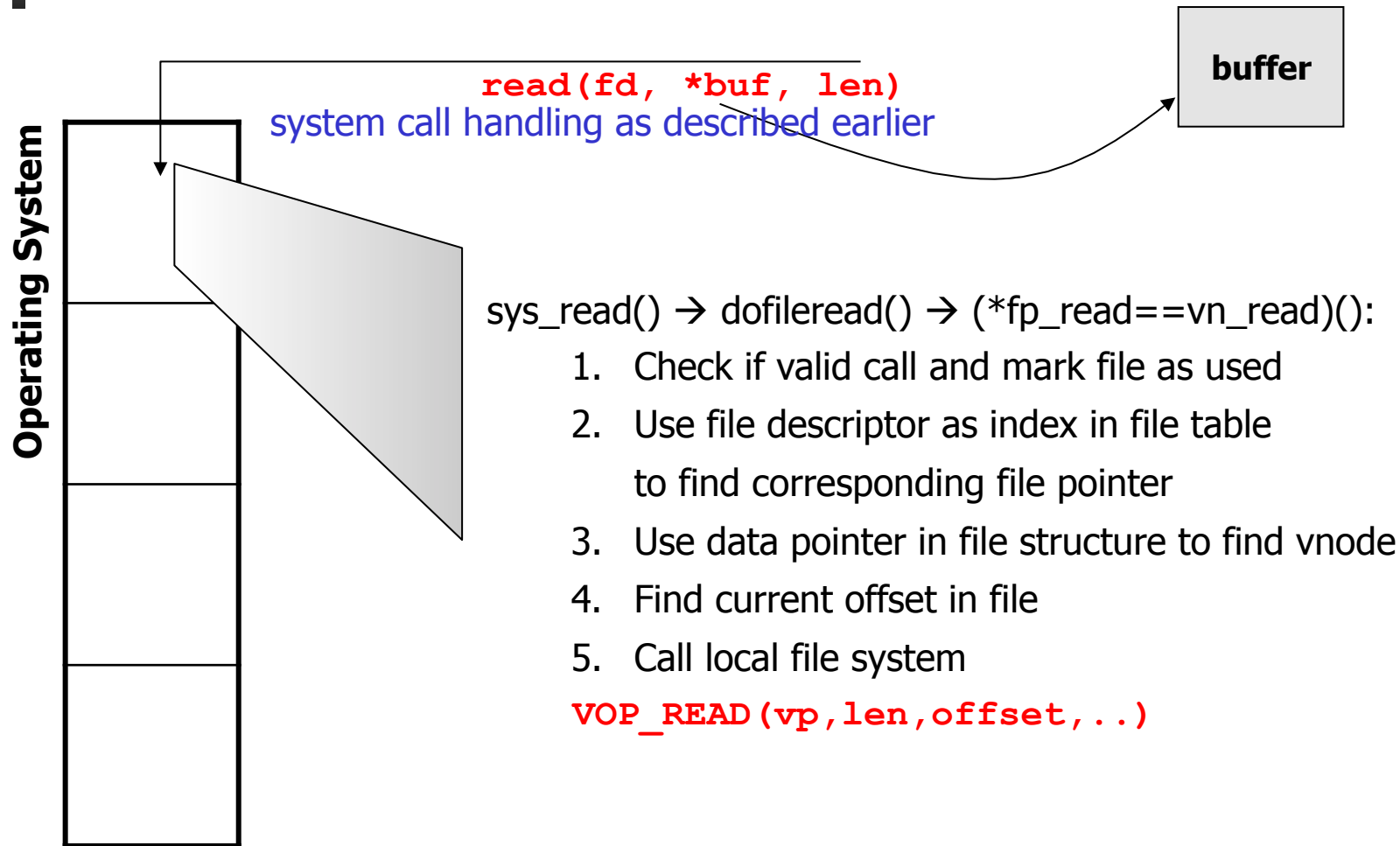
    while ((n = read(fd, buf, BUFSIZE) > 0) {
        <<USE DATA IN BUFFER>>
    }

    close(fd);

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

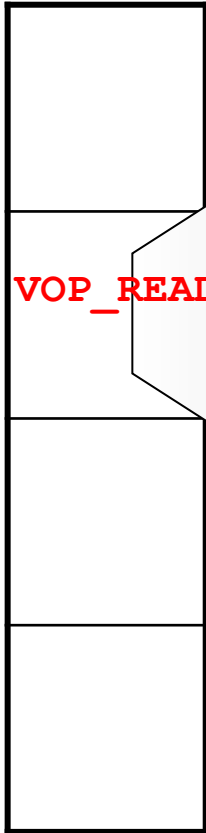

Read

BSD example



Read

Operating System



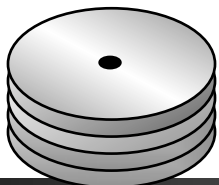
`VOP_READ(vp, len, offset, ...)`

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

READ():

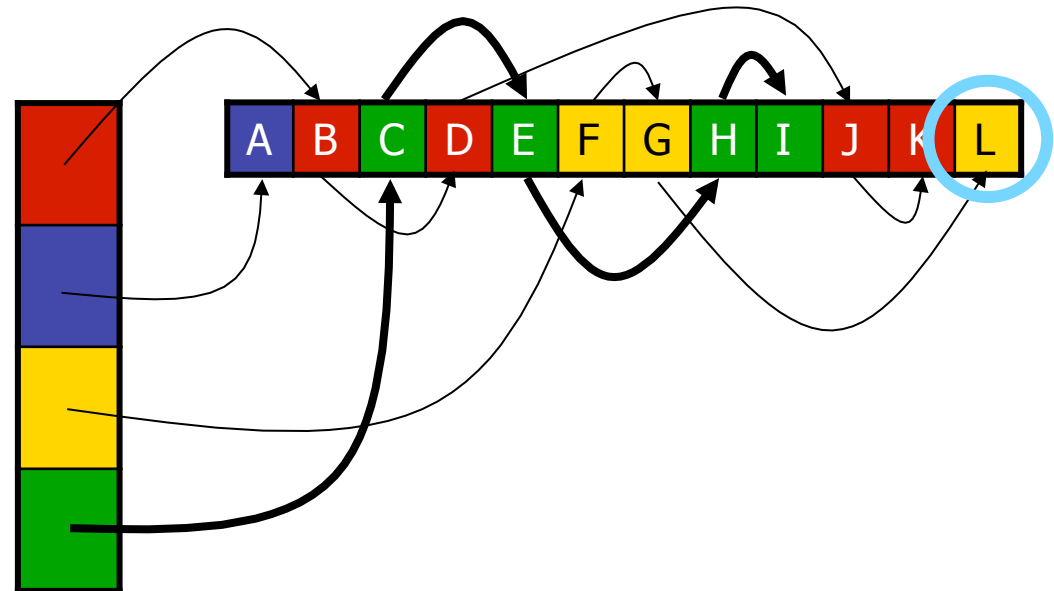
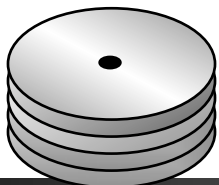
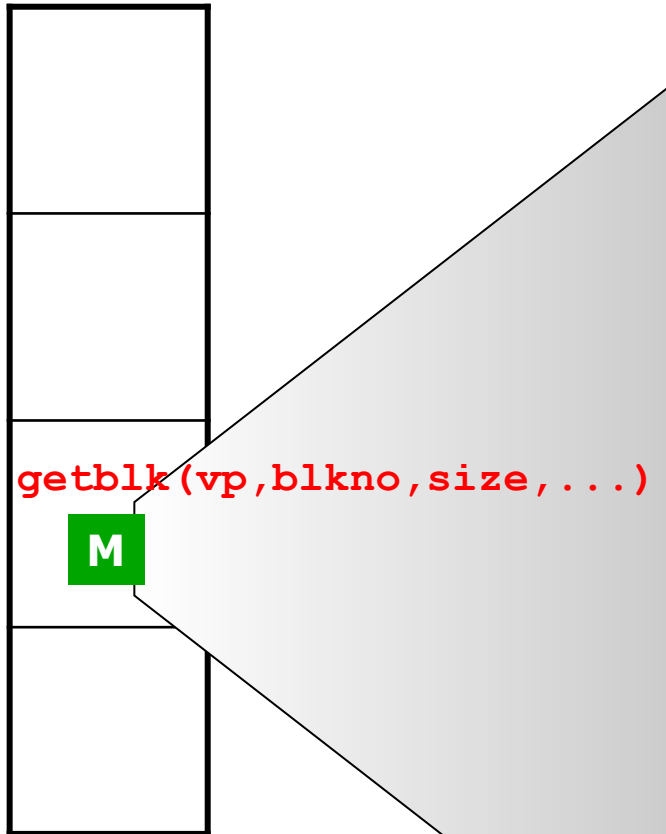
1. Find corresponding inode
2. Check if valid call: $\text{len} + \text{offset} \leq \text{file size}$
3. Loop and find corresponding blocks
 - find logical blocks from inode, offset, length
 - do block I/O, fill buffer structure
e.g., `bread(...)` \rightarrow `bio_doread(...)` \rightarrow `getblk()`

`getblk(vp, blkno, size, ...)`
 - return and copy block to user



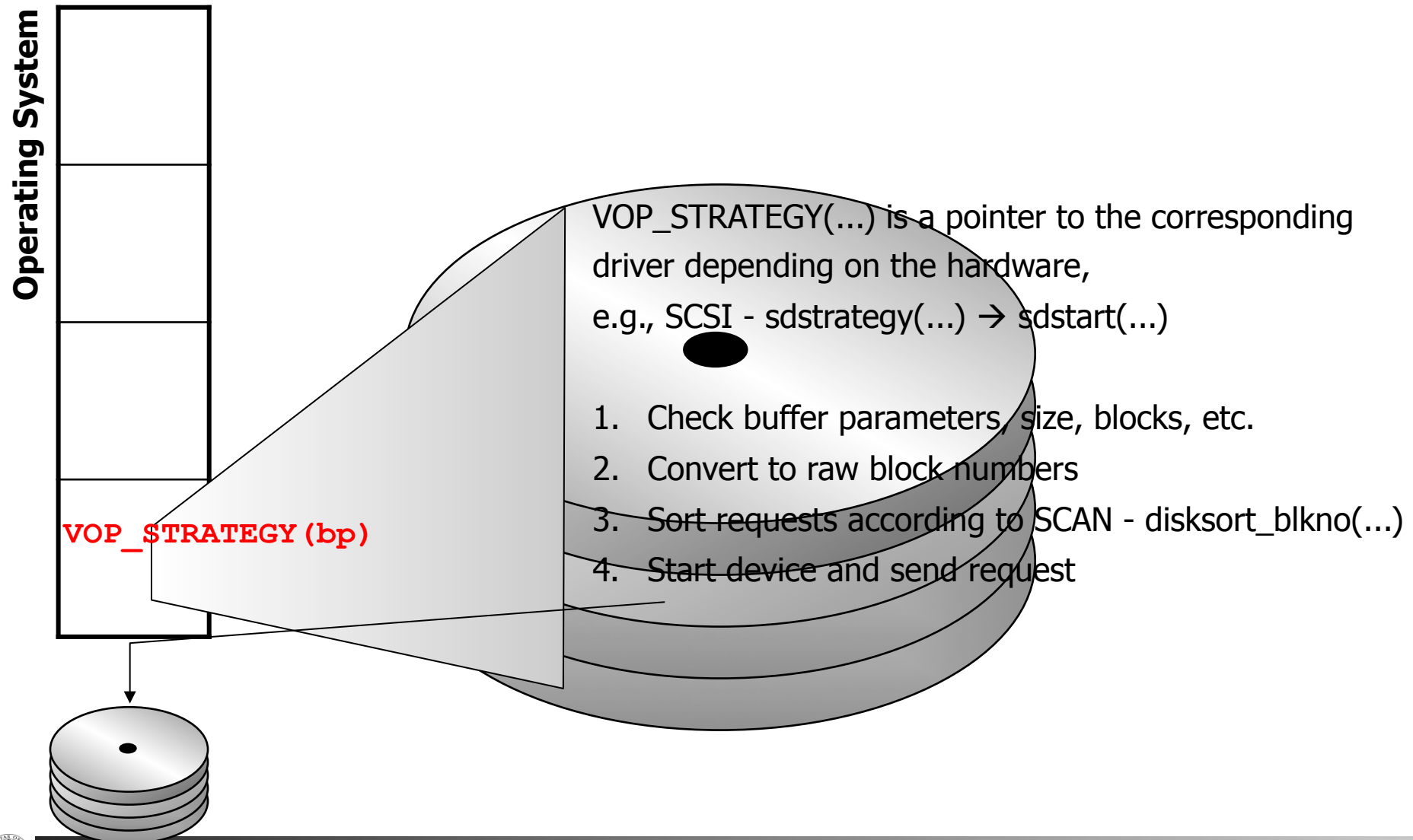
Read

Operating System

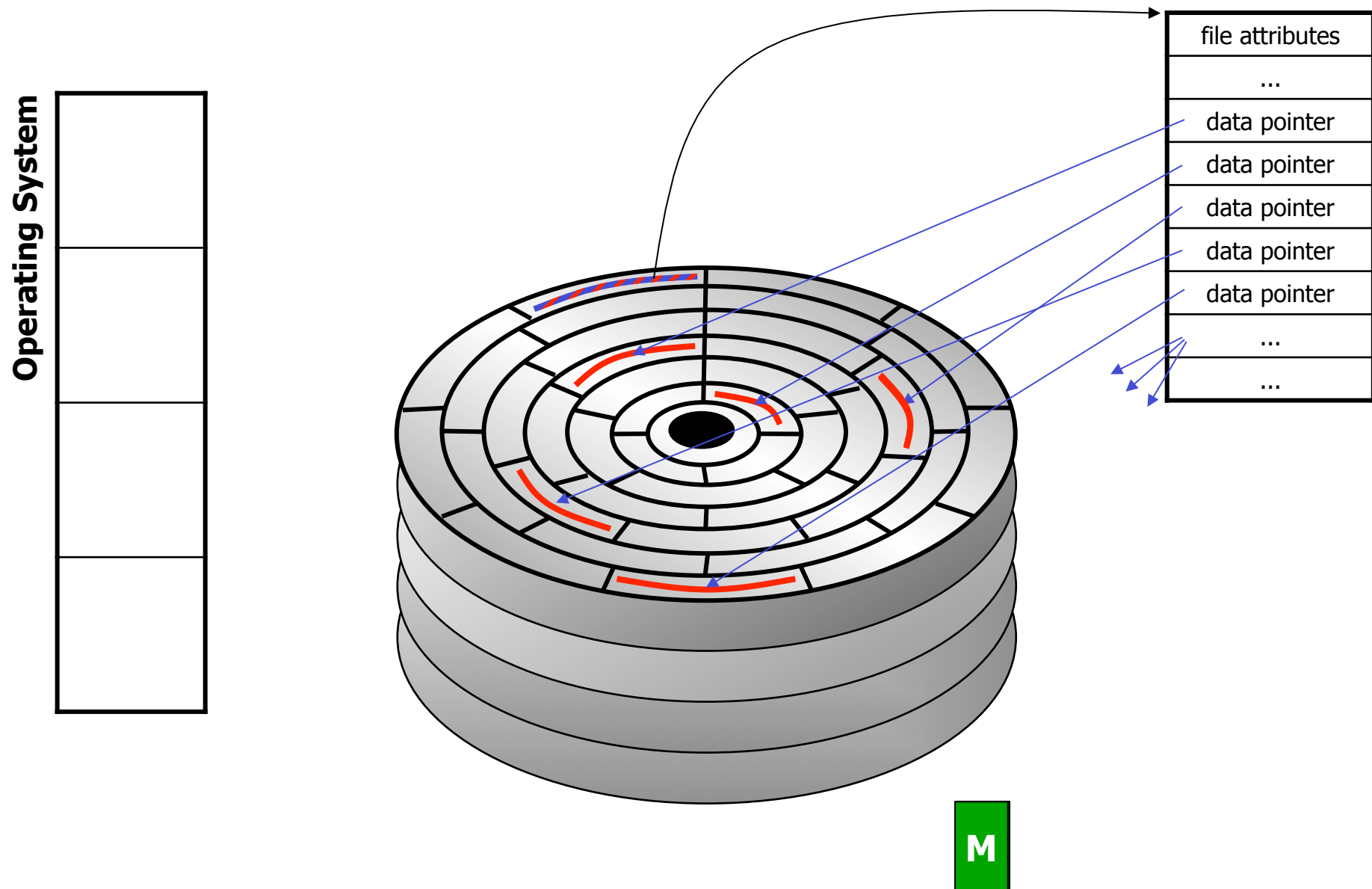


1. Search for block in buffer cache, return if found (hash vp and blkno and follow linked hash list)
2. Get a new buffer (LRU, age)
3. Call disk driver - sleep or do something else
VOP_STRATEGY(bp)
4. Reorganize LRU chain and return buffer

Read

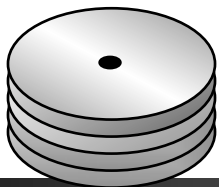
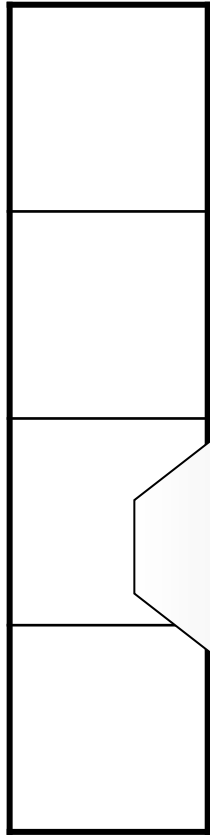


Read

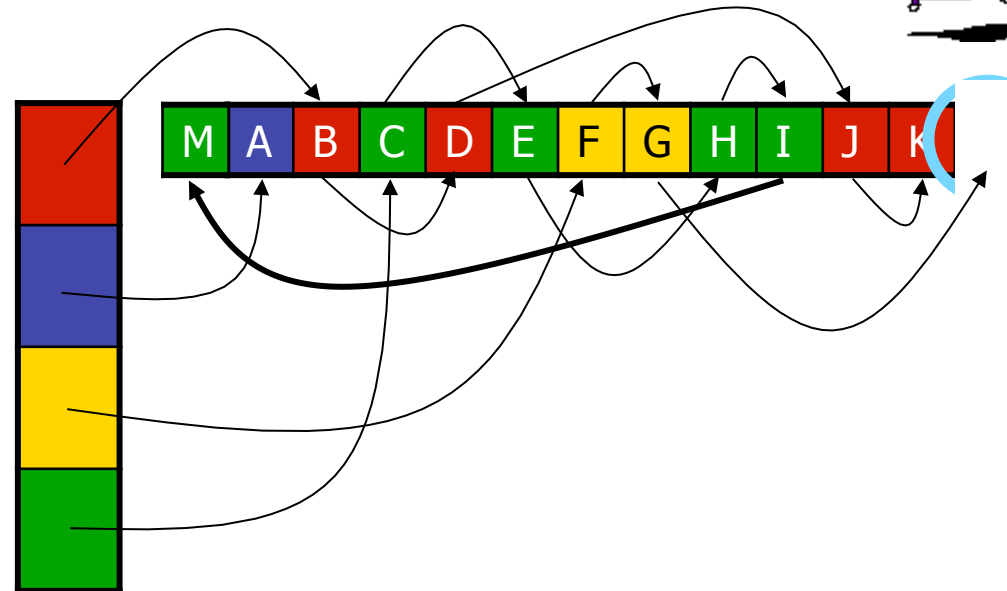


Read

Operating System

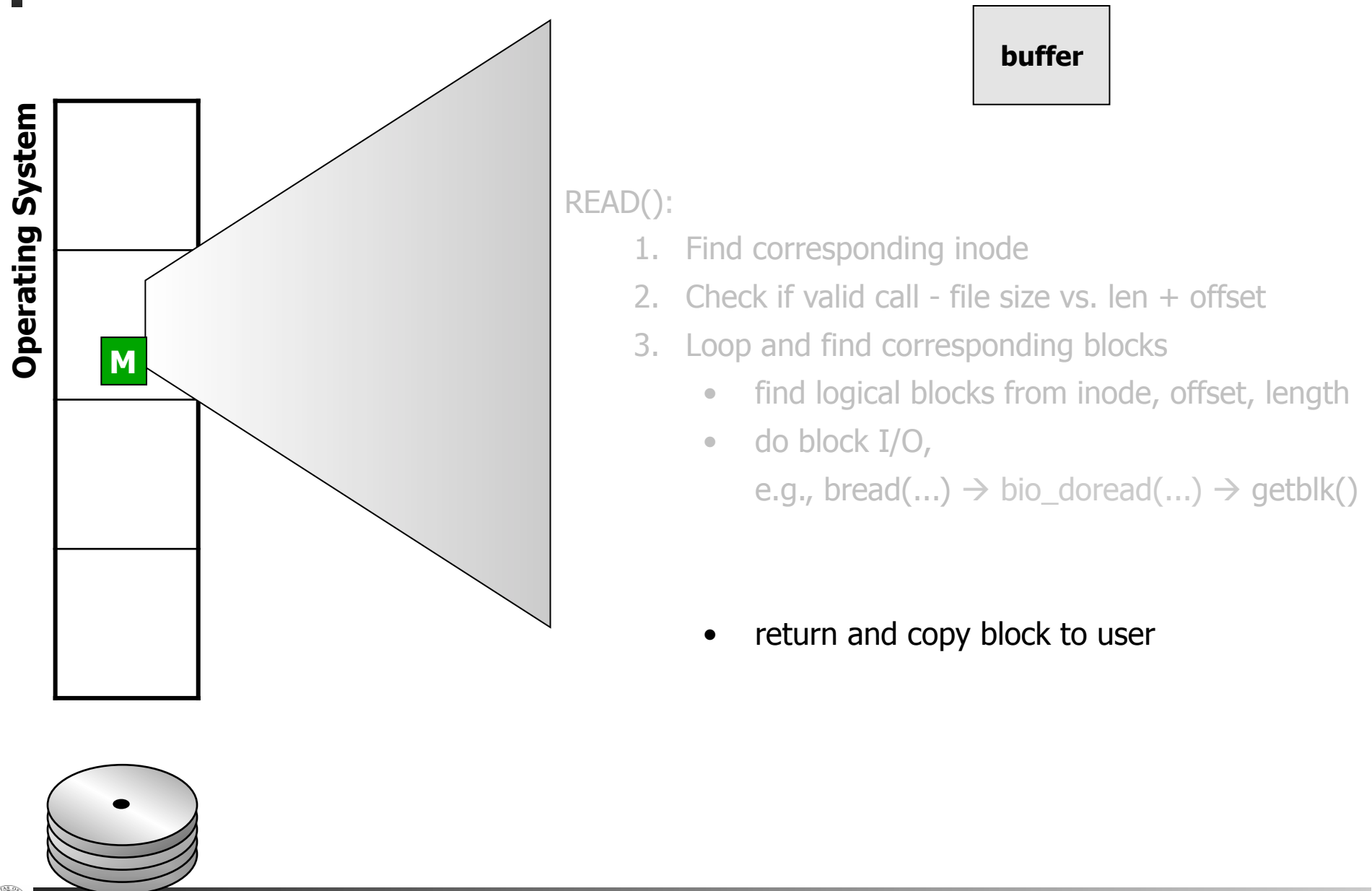


Interrupt to notify end of disk IO
Kernel may awaken sleeping process



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



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

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

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

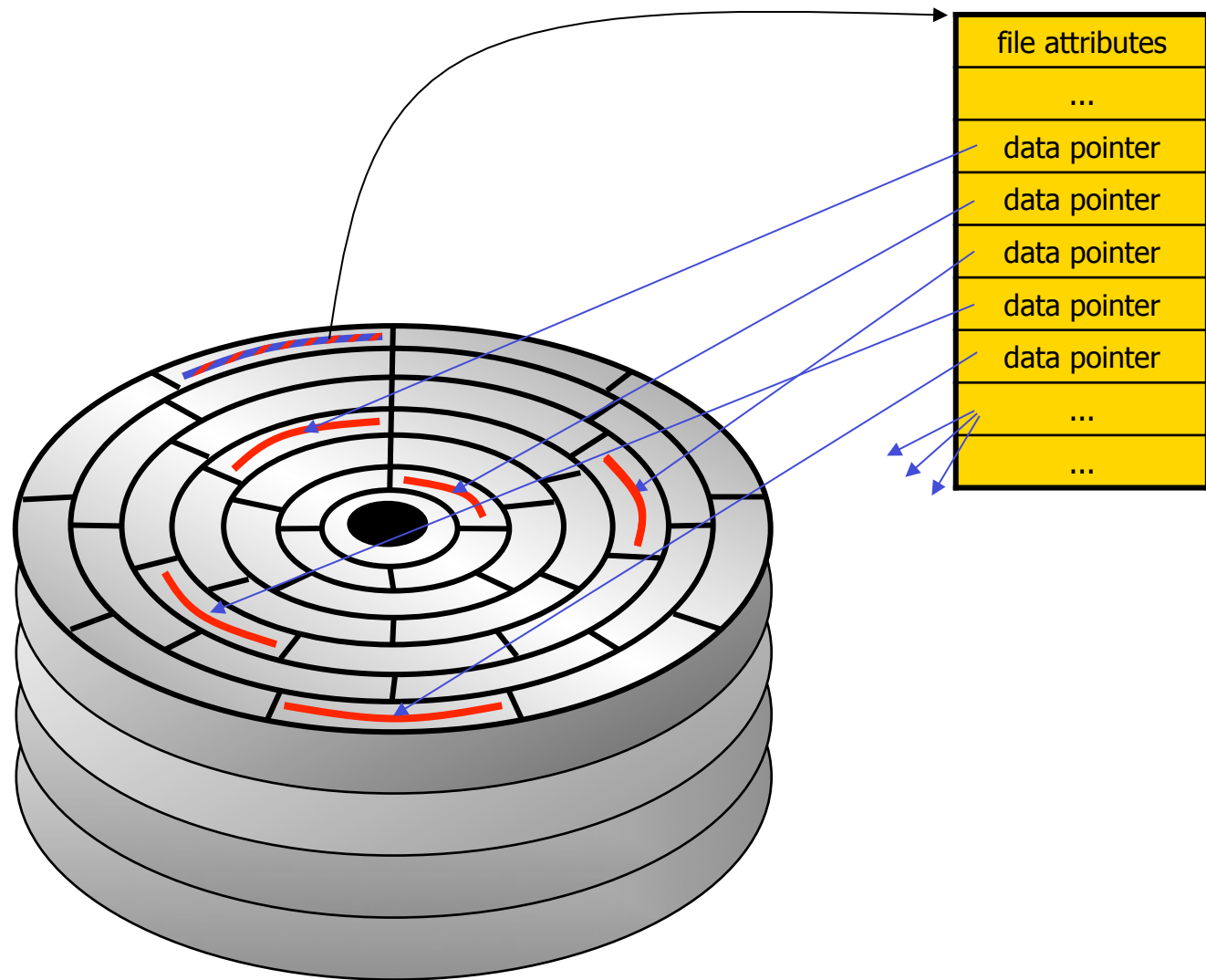
    if ((fd = open( "my.file" , O_RDONLY , 0 )) == -1) {
        printf("Cannot open my.file!\n");
        exit(1); /* EXIT_FAILURE */
    }

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

    close(fd);

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


Management of File Blocks

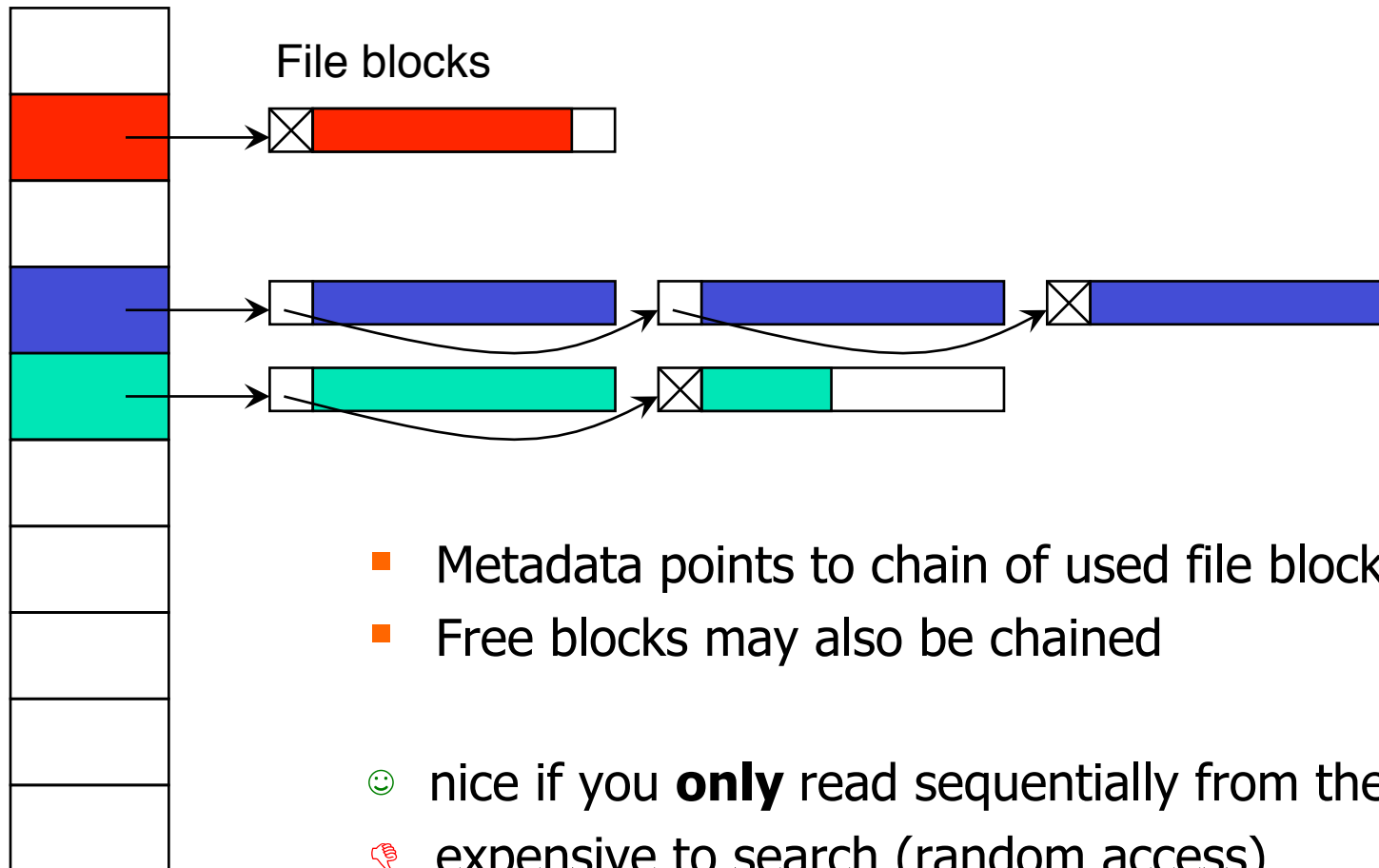


Management of File Blocks

- Many files consist of several blocks
 - relate blocks to files
 - how to locate a given block
 - maintain order of blocks
- Approaches
 - chaining in the media
 - chaining in a map
 - table of pointers
 - extent-based allocation

Chaining in the Media

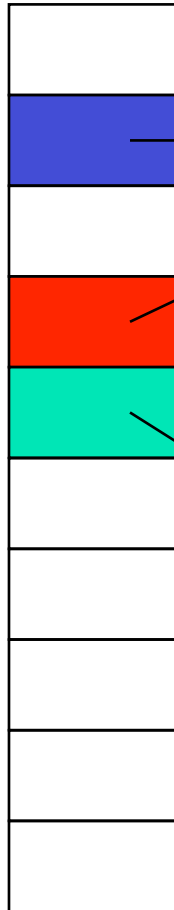
Metadata



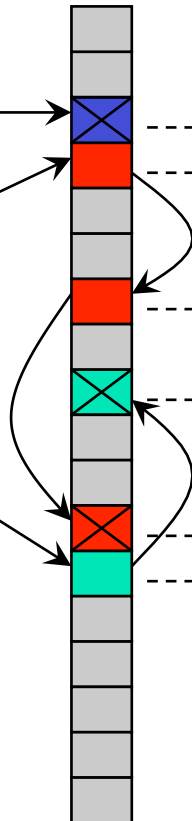
- Metadata points to chain of used file blocks
- Free blocks may also be chained
- 😊 nice if you **only** read sequentially from the start
- 👎 expensive to search (random access)
- 👎 must read block by block

Chaining in a Map

Metadata



Map



File blocks



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

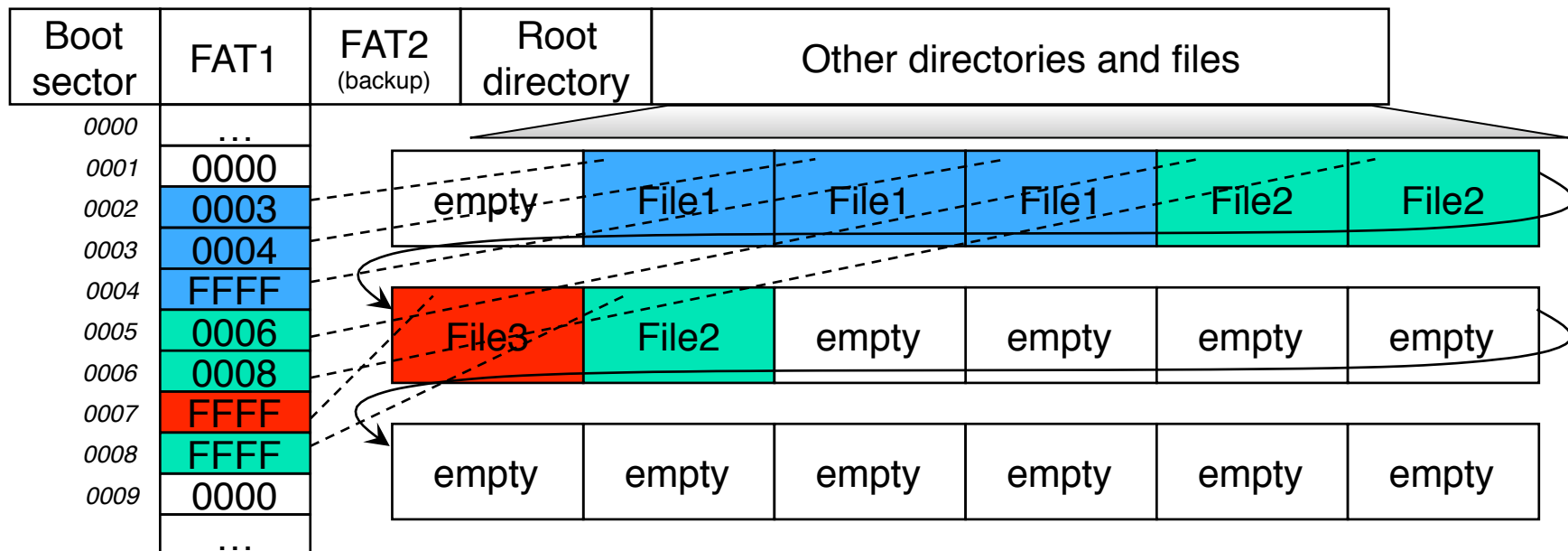


Table of Pointers

Metadata

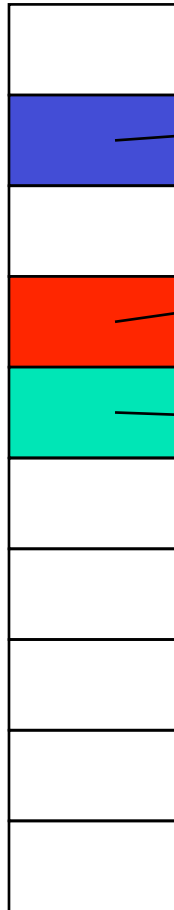
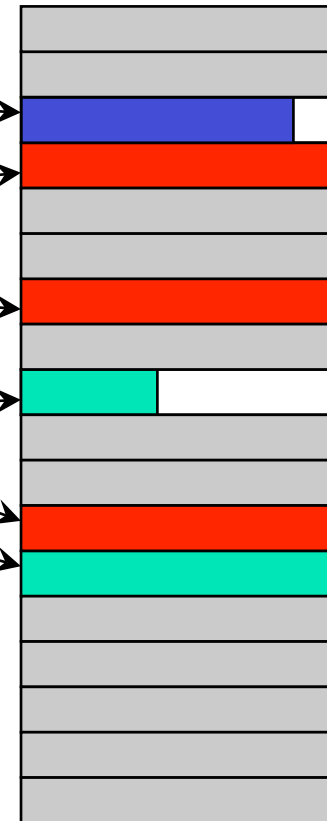


Table of pointers

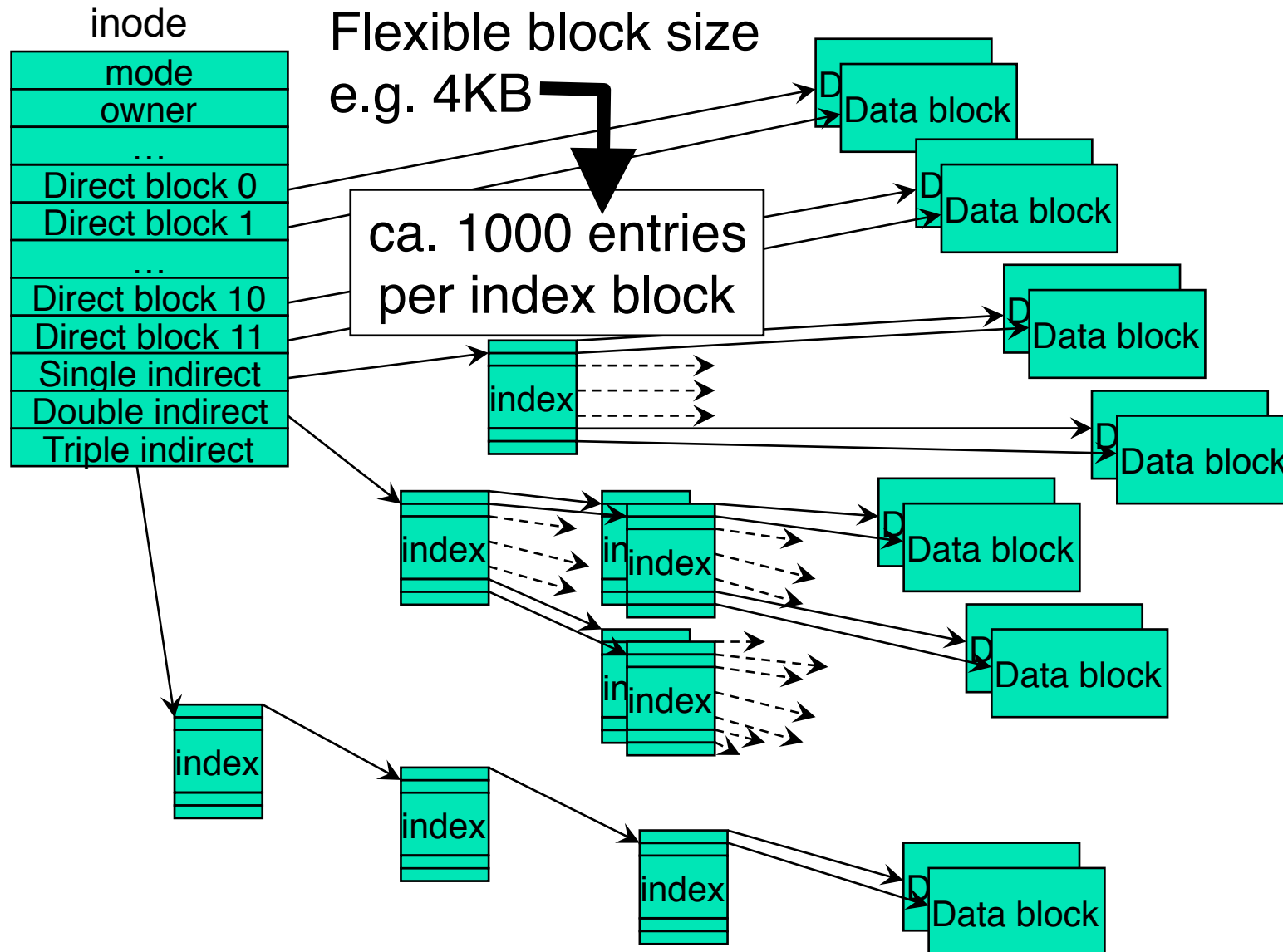


File blocks



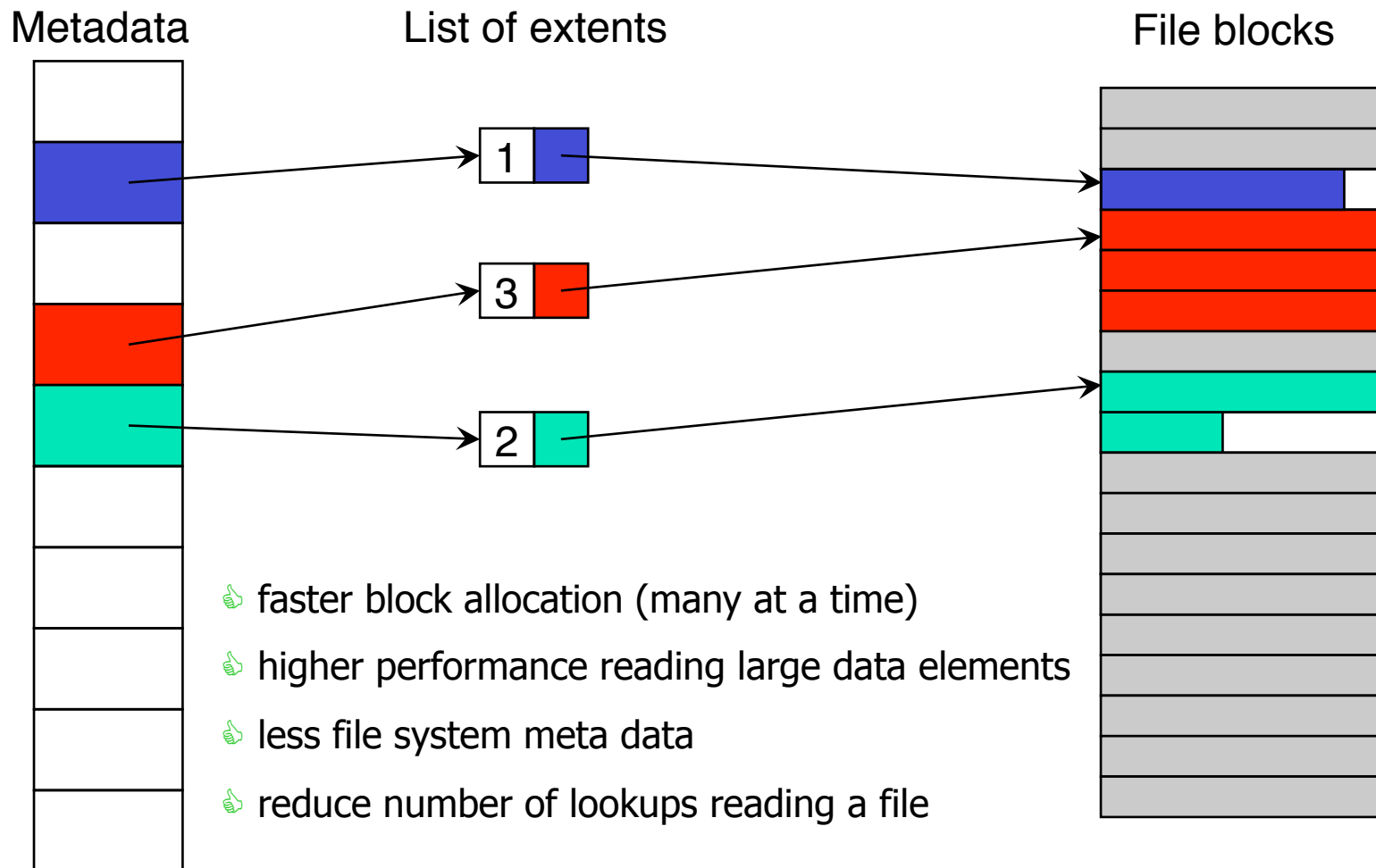
- 👍 good random and sequential access
- 👍 main structure small, extra blocks if needed
- 👎 uses one indirect block regardless of size
- 👎 can be too small

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



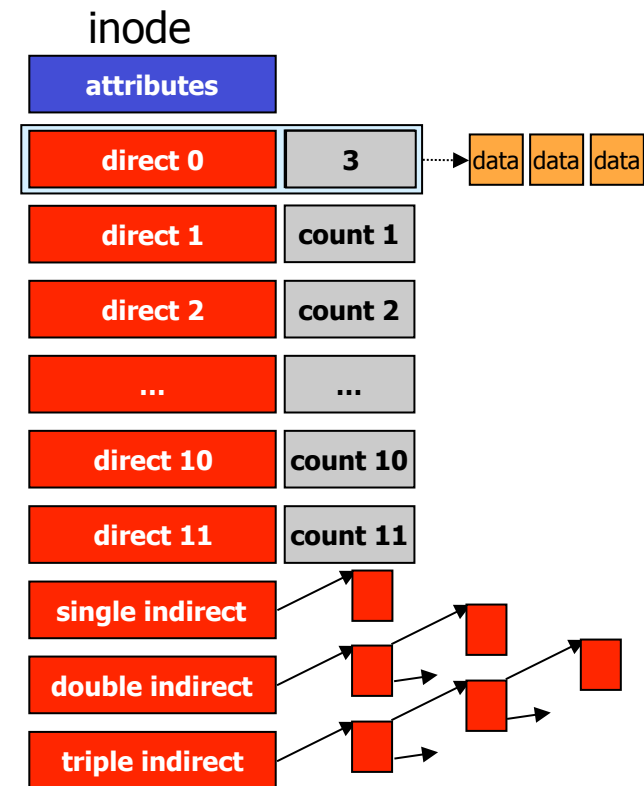
Extent-based Allocation

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



Linux Example: XFS, JFS, EXT4...

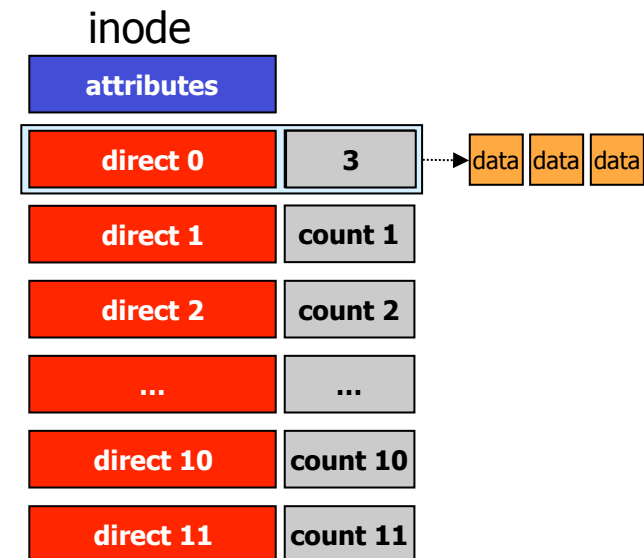
- Count-augmented address indexing in the extent sections
- Introduce a new inode structure
 - add counter field to original direct entries
 - direct** points to a disk block
 - count** indicated how many other blocks is following the first block (contiguously)



ext4 inode

```
struct ext4_inode {
    __le16 i_mode;           /* File mode */
    __le16 i_uid;            /* Low 16 bits of Owner Uid */
    __le32 i_block [NUM];    /* 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 */
};
```



ext4 inode

i_block [NUM]

ext4_extent_header
ext4_extent
ext4_extent
ext4_extent
ext4_extent

...
__le16 eh_depth;
...

Tree of extents organized using an HTREE

```
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_lo; /* low 32 bits of physical block */  
};
```

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 $4 \times 128 = 512$ MB files?
What about **fragmented disks**??



ext4 inode

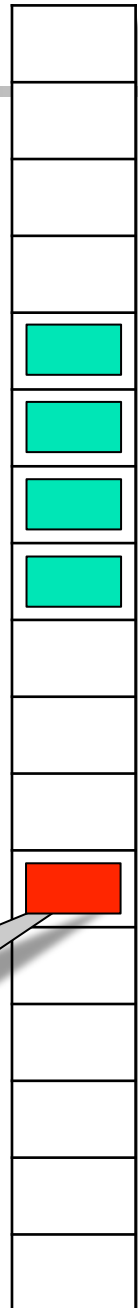
i_block [NUM]

ext4_extent_header
ext4_extent_idx
ext4_extent_idx
ext4_extent_idx
ext4_extent_idx

```
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;  
};
```

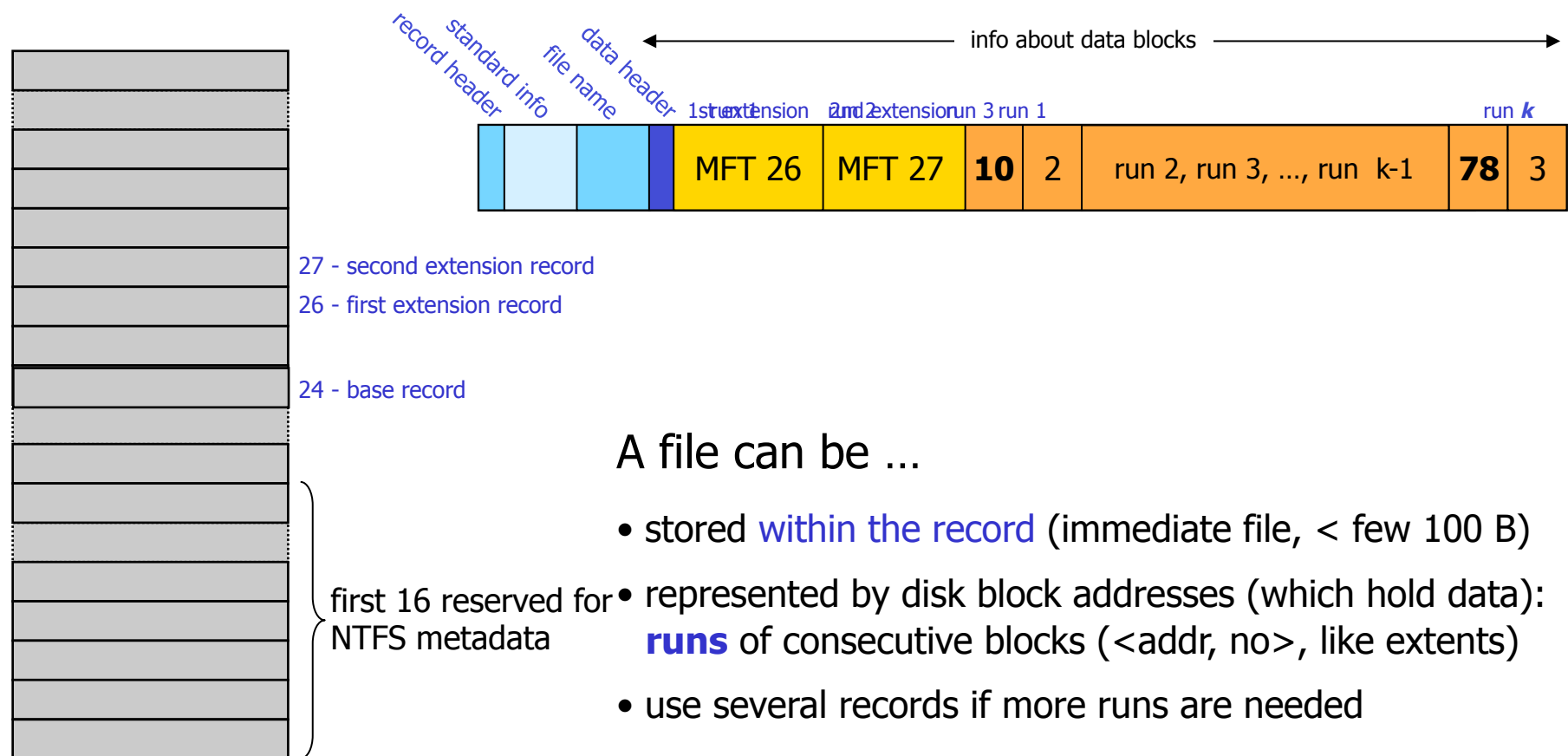
```
...  
    __le16 ee_len; 4  
    __le16 ee_start_hi;  
    __le32 ee_start;
```

- one 4 KB can hold 340 `ext4_extents(_idx)`
- first level **can** hold 170 GB
- second level can hold 56 TB (limited 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)



Recovery & Journaling

- When data is **written** to a file, both metadata and data must be updated
 - metadata is written asynchronously, data may be written earlier
 - if a system crashes, the file system may be corrupted and data is lost
- Journaling file systems provide improved consistency and recoverability
 - makes a log to keep track of changes
 - the log can be used to undo partially completed operations
 - e.g., ReiserFS, JFS, XFS and Ext3 (all Linux)
 - NTFS (Windows) provide journaling properties where all changes to MFT and file system structure are logged

DAS vs. NAS vs. SAN??

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

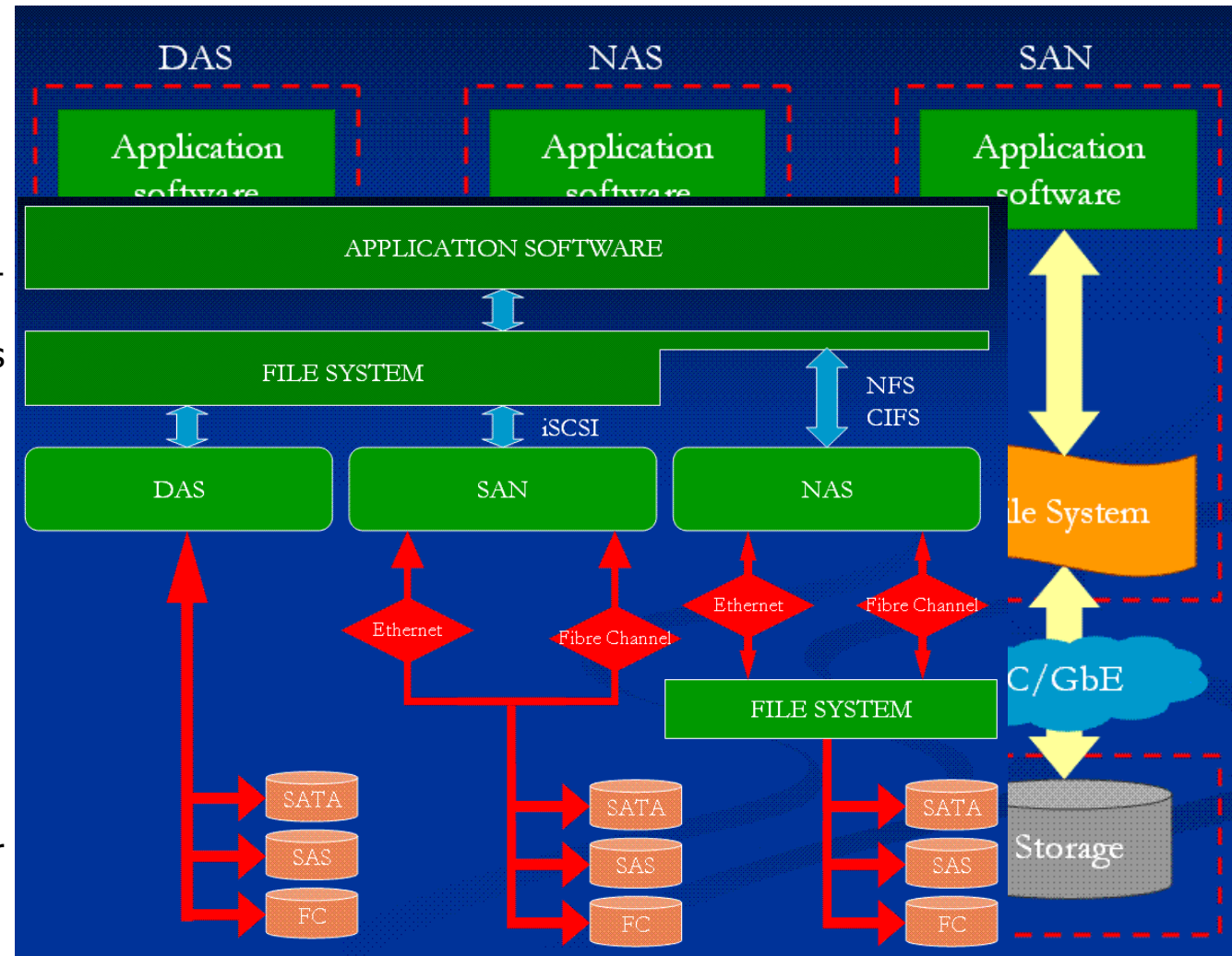
- Direct attached storage

- 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



Mechanical Disks vs. Solid State Disks???

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

	Storage capacity (GB)	Average (seek) time / latency (ms)	Sustained transfer rate (MBps)	Interface (Gbps)
Seagate Cheetah X15.6 (3.5 inch)	450	3.4 (track to track 0.2)	110 - 171	SAS (3) FC (4)
Seagate Savvio 15K (2.5 inch)	73	2.9 (track to track 0.2)	29 - 112	SAS (3)
OCM Flash Media Core Series V2	250	< .2 - .3	up to 170	SATA (3)
Intel X25-E (extreme)	64	0.075	250	SATA (3)
Intel X25-M (mainstream)	160	0.085	250	SATA (3)
Mtron SSD Pro 7500 series	128	0.100	130	SATA (1.5)
Intel DC S3700 Series 	2000	0.020	2000	PCIe, v3 (7.8/lane)
Intel Drive 910 Series	800	< 0.065	2000	SAS (6)

Why spend a lecture talking about HDD?

- SSDs are
 - “almost like memory”
 - much faster
 - more expensive
 - smaller

- 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 County, USA

- **Google Early 2013**

- ✓ 46,400 servers - Profile Park, Dublin, Ireland
 - ✓ 200,000 servers - Jurong West, Singapore (projected estimate)
 - ✓ 200,000 servers - Kowloon, Hong Kong (projected estimate)
 - ✓ 139,200 additional servers - Mayes County, USA

- Estimated grand total: 2,376,640
(early 2013)

- Insert one 0.5 TB SSD in each
 - cheapest komplett.no: 4.4 billion NOK
 - Intel P3700: 17.7 billion NOK

Google data center locations:

Americas

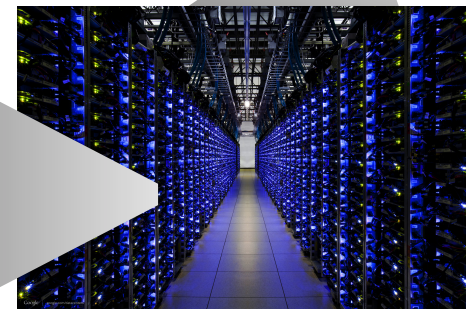
Berkeley County, South Carolina
Council Bluffs, Iowa
Douglas County, Georgia
Quilicura, Chile
Mayes County, Oklahoma
Lenoir, North Carolina
The Dalles, Oregon

Asia

Singapore
Chang

Europe

Hamina
St. Ghislain
Dublin





The End: Summary

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 derivatives)
 - 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, ...
 - ...