# Secondary Storage

# Secondary Storage: Q & A

- What is Secondary Storage?
  - *It is a non-volatile repository for (both user and system) data and programs*
- Who manages Secondary Storage?
  - *File System – part of the OS*
- What kind of data can be stored in Secondary Storage?
  - Programs (source, object), temporary storage, virtual memory

# Secondary Storage: Q & A

- What are the main requirements on file systems?
  - Should provide persistent storage
  - Handle very large information (large chunks, large numbers)
  - Concurrent access
- What are main user requirements?
  - How the storage appears to them
  - File naming and protection, operations allowed on files

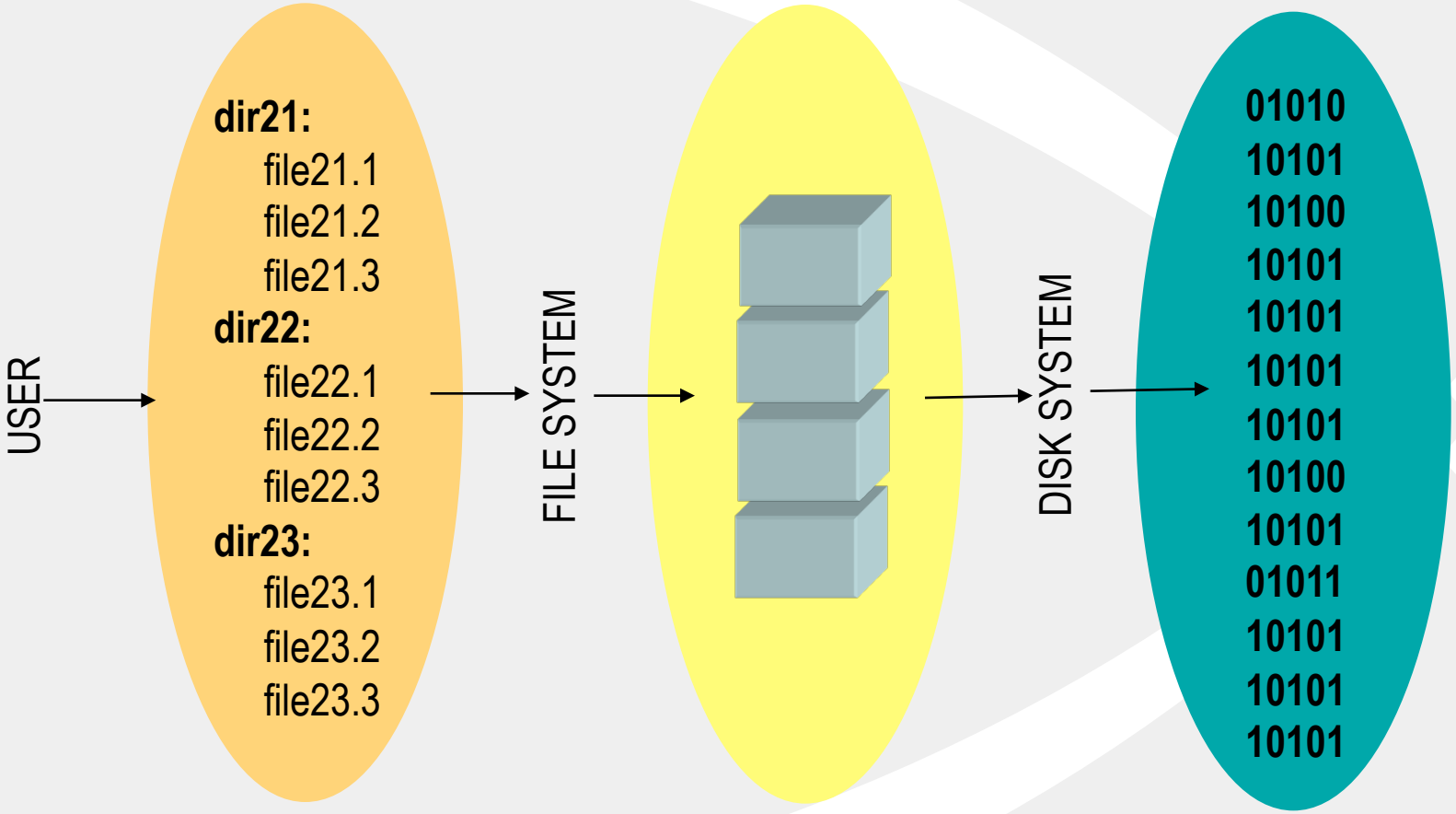
# File concept

- A file is a *named* collection of related information, usually as a sequence of bytes, with two views:
  - *Logical (programmer's ) view*, as the users see it.
  - *Physical (operating system) view*, as it actually resides on secondary storage.

# File concept…

- What is the difference between a file and a data structure in memory?
  - files are intended to be non-volatile; hence in principle, they are long lasting,
  - files are intended to be moved around (i.e., copied from one place to another), accessed by different programs and users

# A view of storage management

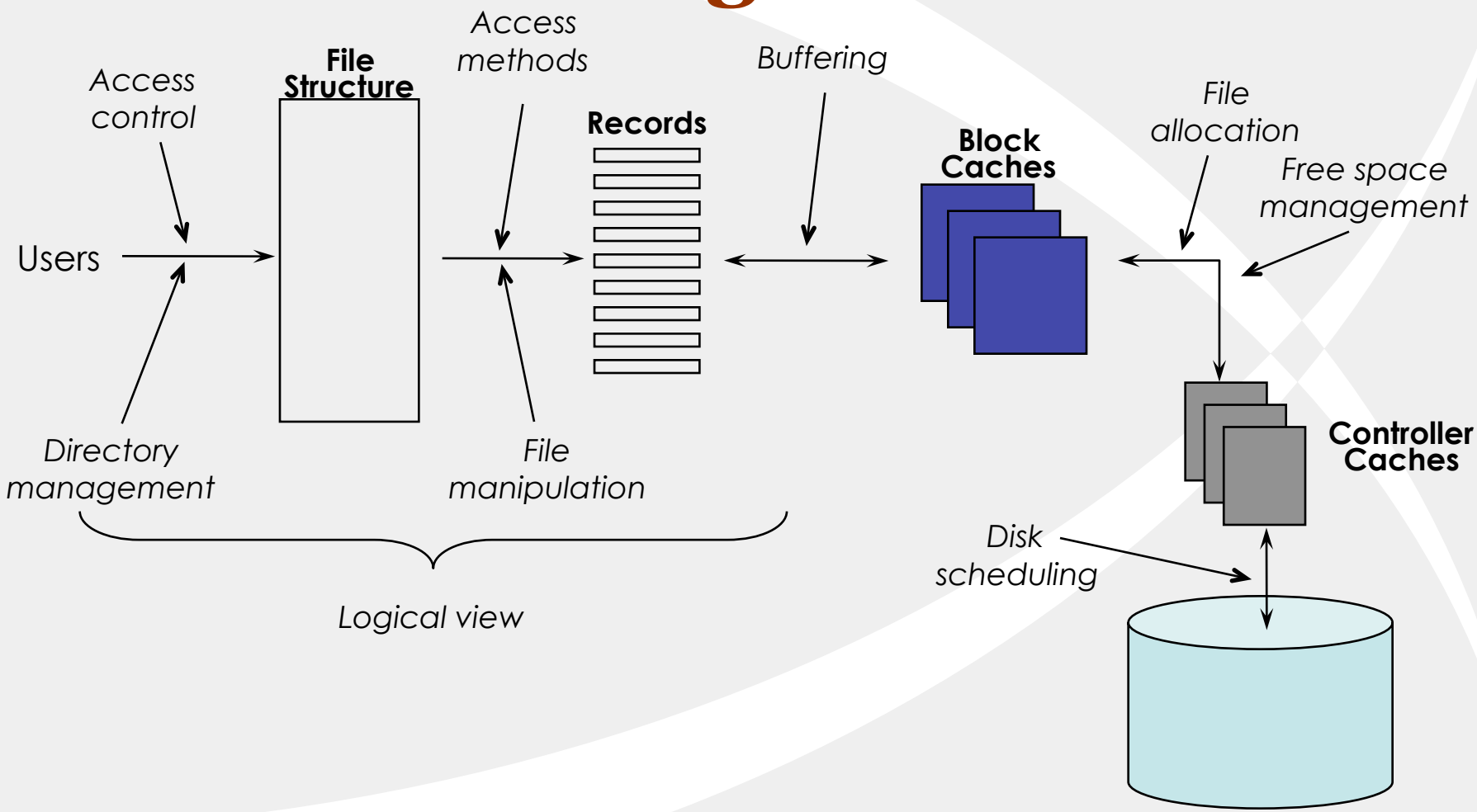



File system based view – content visible as files and directories

Block based view content considered as blocks of data

Physical view – all data just a stream of bits

# Elements of storage management



# File attributes

- Each file is associated with a collection of information, known as *attributes*:
  - NAME, owner, creator
  - type (e.g., source, data, binary)
  - location (e.g., I-node or disk address)
  - organization (e.g., sequential, indexed, random)
  - access permissions
  - time and date (creation, modification, and last accessed)
  - size
  - variety of other (e.g., maintenance) information.

# File names

- Files provide a way to store and retrieve data from backing storage using **file names**
- Many OSs support two or more parts separated by a period in names
- In UNIX, files extensions are just conventions and not enforced by OS
- Applications like C compiler might enforce their own conventions
- MS-Windows associates programs with files using file extensions.

# File types

- Commonly used extensions:

| File type | Extension | Function |
|---|---|---|
| Executable | exe, com, bin | ready-to-run code |
| Text | txt, doc | textual data, documents |
| Source | c, f77, asm | source in various languages |
| Object | obj, o | object code |
| Library | lib, a | library routines |
| Archive | tar, zip, arc | grouped files |
| Compressed | Z, gz | compressed |
| Print/view | ps, eps | printing or viewing |
| Word processor | ppt, wp, tex | various word |

# Directories

- What is a directory?
  - Is a *symbol table* that can be searched for information about the files. Implemented as a file.
- Directories give a name space for the objects (files) to be placed.
- *Directory entry* contains information (attributes) about a file
  - entries are added in a directory as files are created, and are removed when files are deleted.

# Name spaces

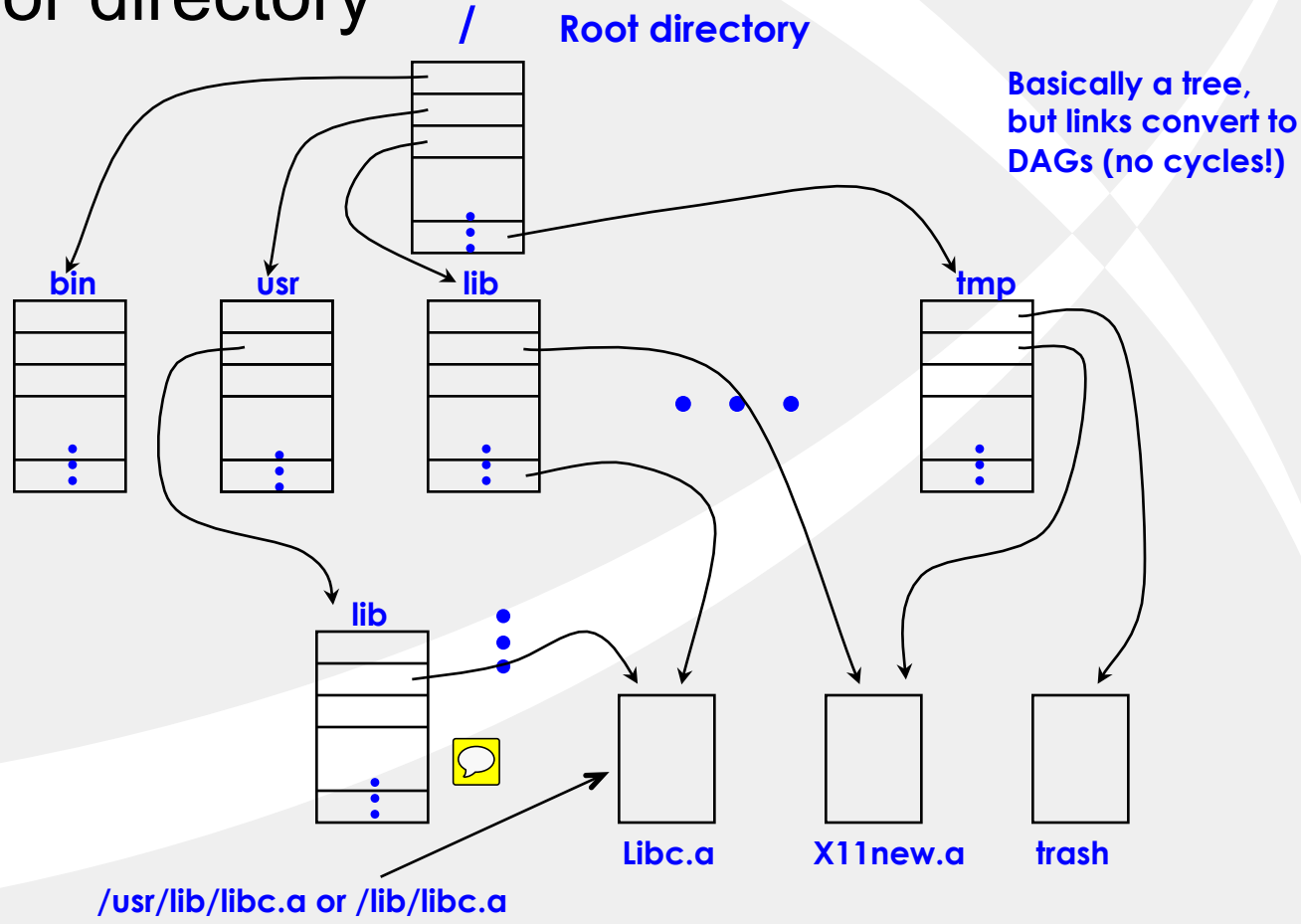
- Set of symbols that are used to organize objects of various kinds, so that these objects may be referred to by name
- File systems are namespaces that assign names to files

# Directories

- Common directory structures are:
  - **Single-level (flat):** shared by *all* users – worlds' first supercomputer CDC 6600 also had
  - **Two-level:** one level for *each* user – should have some form of "user" login to identify the user and locate the user's files.
  - **Tree:** arbitrary (sub)-tree for *each* user – login required.

# An example: UNIX directories

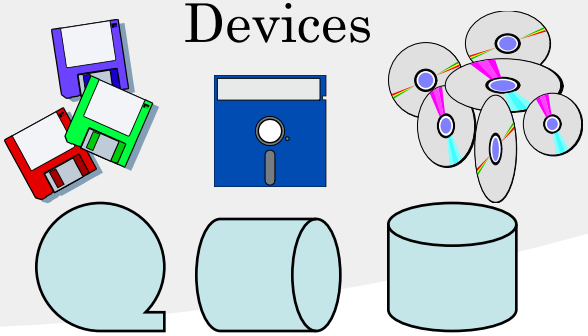
- UNIX uses *directed acyclic-graph (DAG)* structure for directory



# File systems

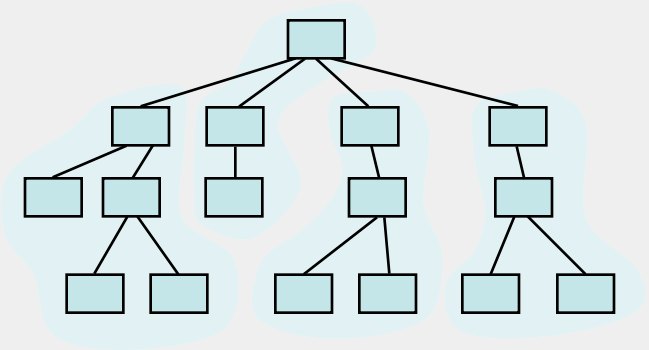
- The basic services of a file system include:
  - keeping track of files (knowing location),
  - I/O support, especially the transmission mechanism to and from main memory,
  - management of secondary storage,
  - sharing of I/O devices,
  - providing protection mechanisms for information held on the system.

# File system abstraction

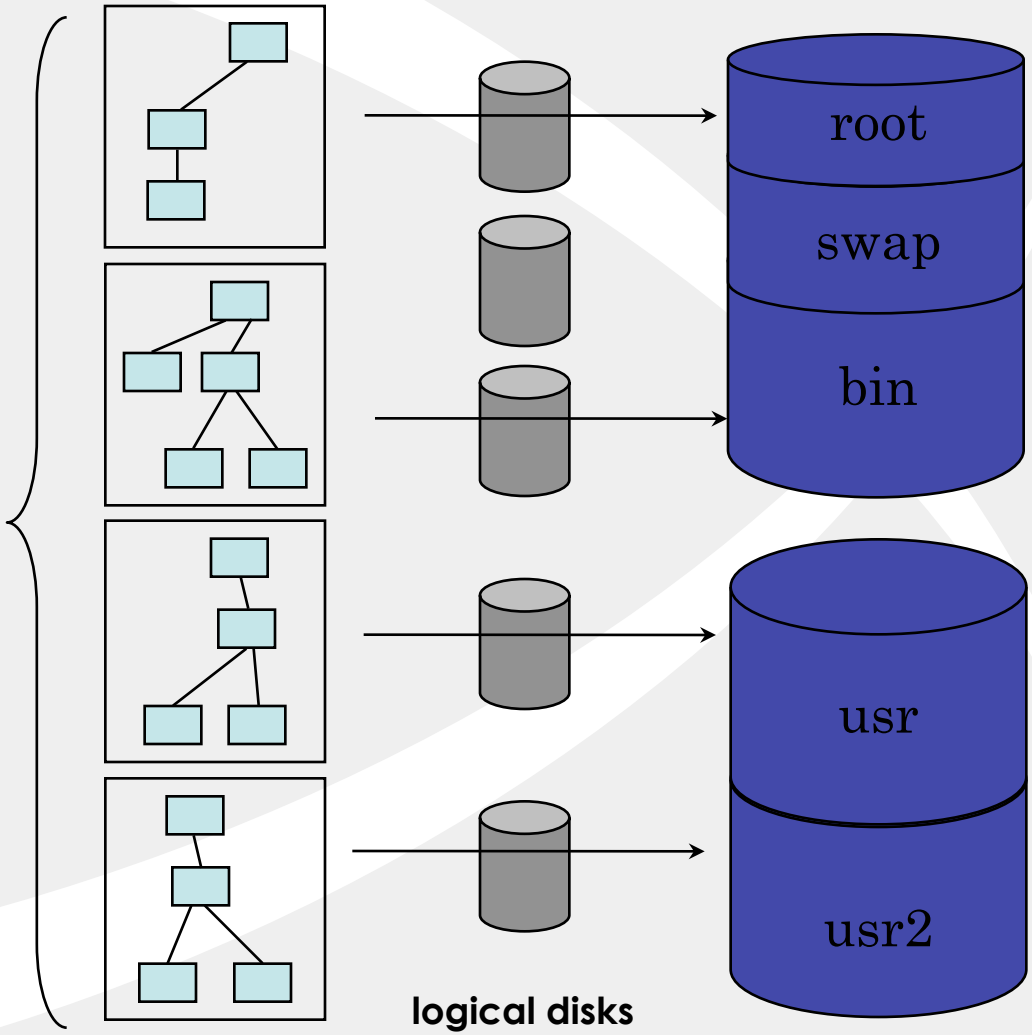
| | *Objects* | *Typical operations* |
|---|---|---|
| Interactive (Shells) | files | copy, delete, rename |
| Applications and system programs | logical elements (records) | open/close, buffering seek (logical) |
| **File System** | | file system check soft repair partitioning |
| Devices  | physical elements (head, cylinder, …) | raw read/write, seek (physical) low-level format |

# Example —UNIX file system

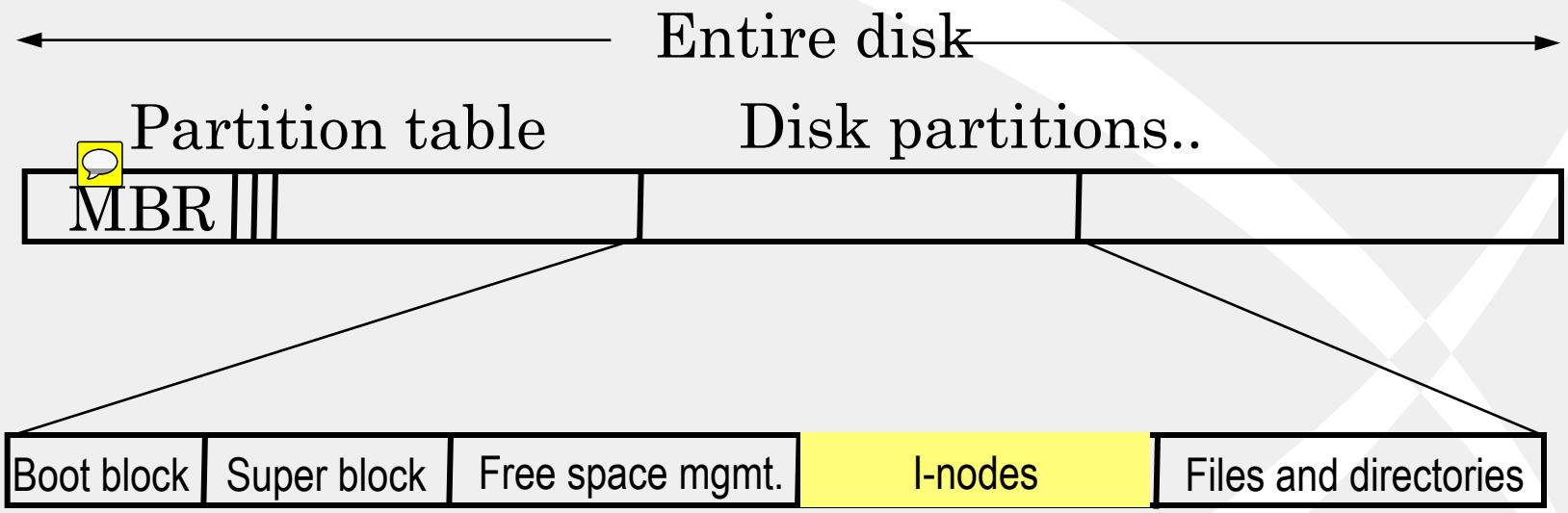
*Mapping file systems to disks*



logical file system



# File system layout

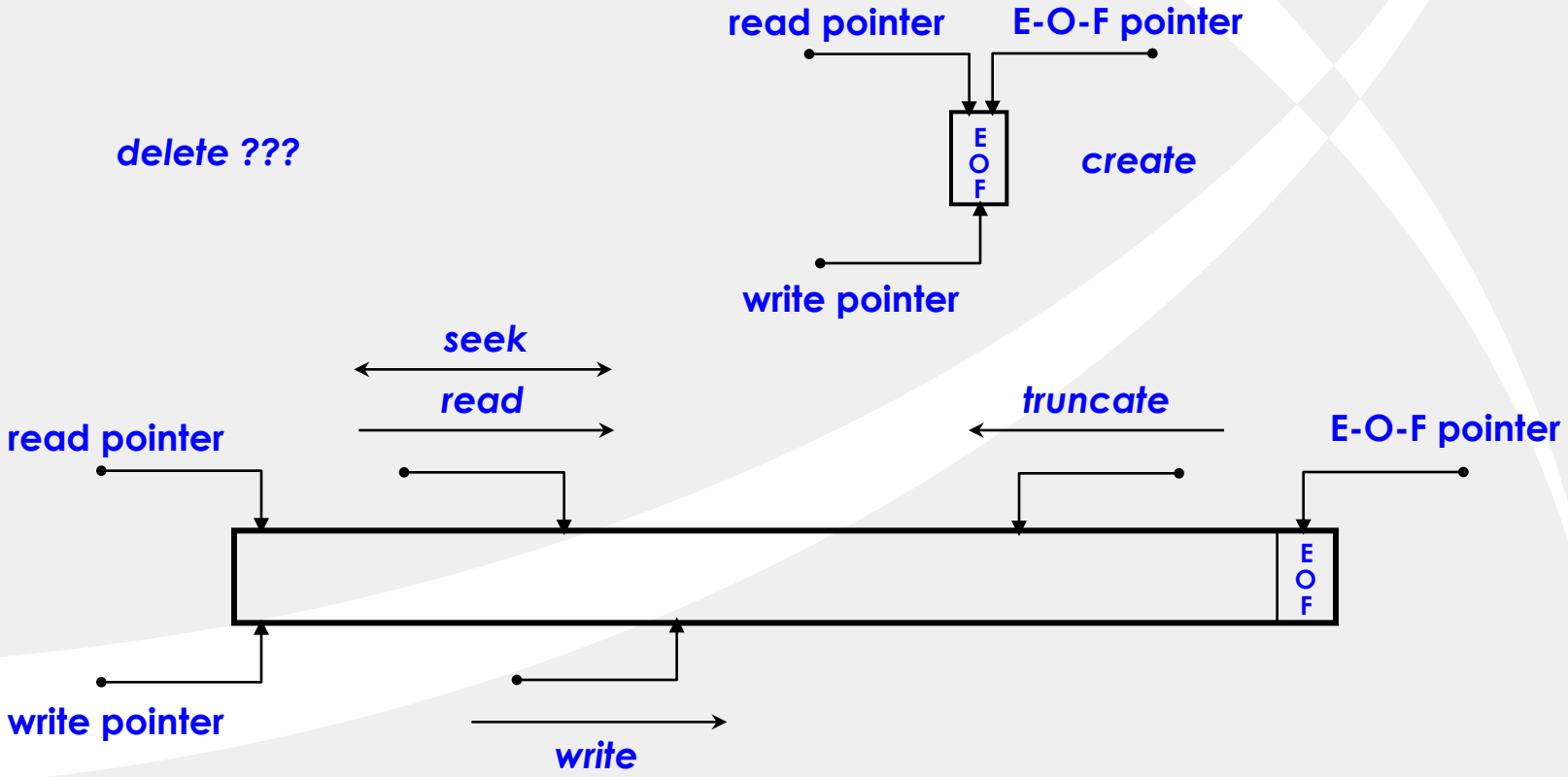


# File sharing

- File sharing raises the issue of **protection**
- One approach is to provide *controlled access* to files through a set of operations such as read, write, delete, list, and append
- One popular protection mechanism is a condensed version of access list containing: user, group, and other

# File operations

- Six basic operations for file manipulation: *create, write, read, delete, reposition r/w pointer* (a.k.a. *seek*), and *truncate* (rare)



# File operations

- Create
  - file is created with no data
  - sets some file attributes
- Delete
  - file is no longer needed
  - storage released to free pool
- Open
  - process first opens a file -- fetch the attributes and list of disk addresses into main memory
- Close
  - frees the internal table space
  - OSs enforce by limiting number of open files

# File operations

- Read
  - comes from current position in file
  - bytes to read and where to put specified
- Seek
  - repositions the file pointer for random access
  - after seeking data can be read or written
- Write
  - data written to file at current position
  - If current position is at end-of-file, file's size increases

# Example C program

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
      int in_fd, out_fd, rd_count, wt_count;
      char buffer[BUF_SIZE];

      if (argc != 3) exit(1);           /* syntax error if argc is not 3 */
```

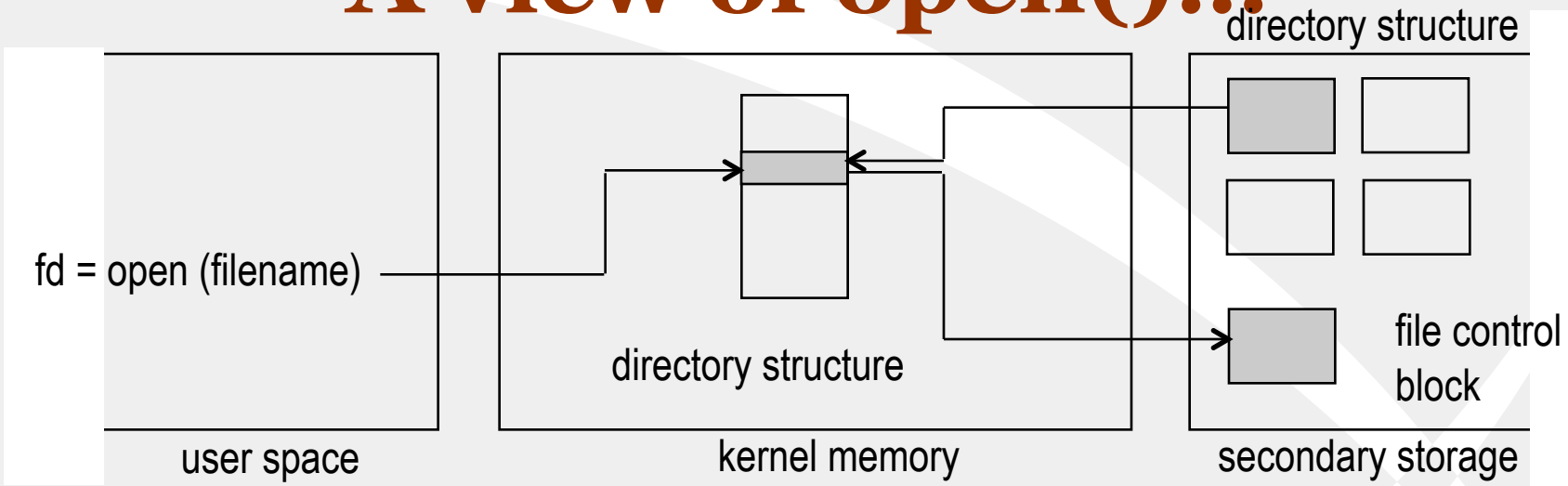
# Example C program...

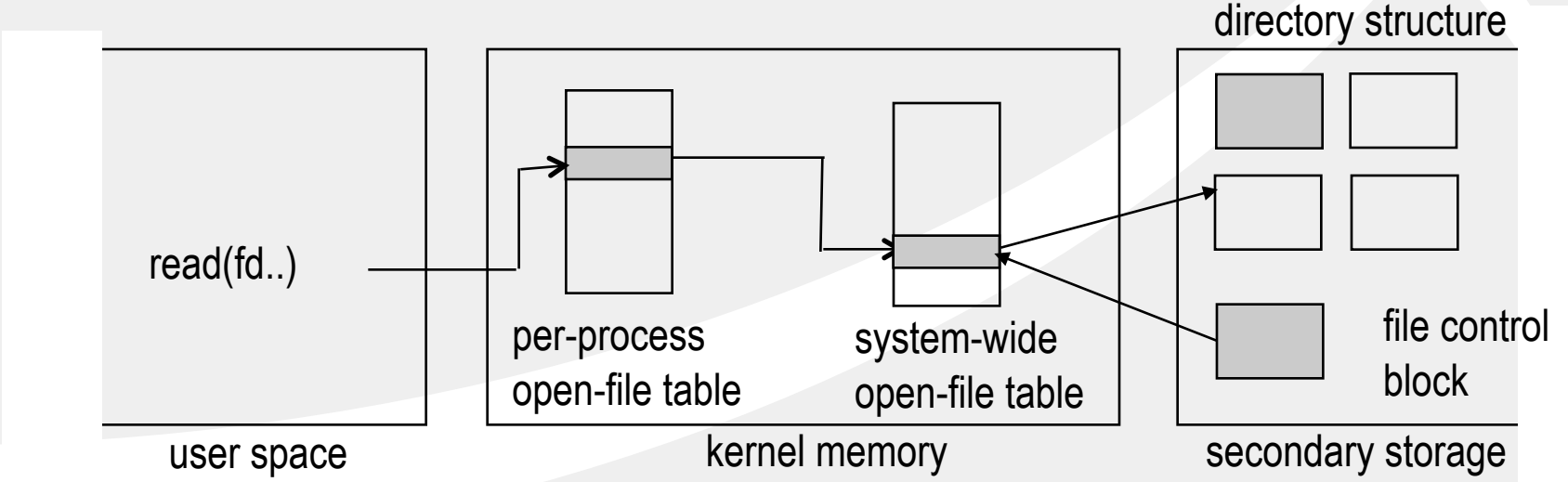
```
    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

# A view of open()...



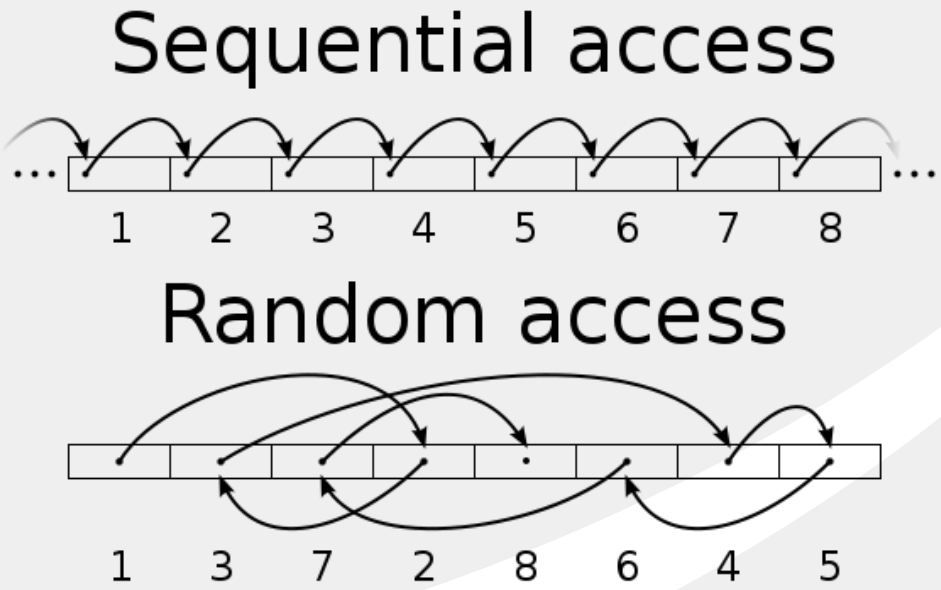



# File access methods

- The information stored in a file can be accessed in a variety of methods:
  - *Sequential*: in order, one record after another. Convenient with sequential access devices (e.g., magnetic tape).
  - *Direct (random)*: in any order, skipping the previous records.
  - *Indexed*: in any order, but accessed using particular key(s); e.g., hash table or dictionary. *Database access (e.g., patient database in a hospital).*

# File access methods

- (wikipedia)

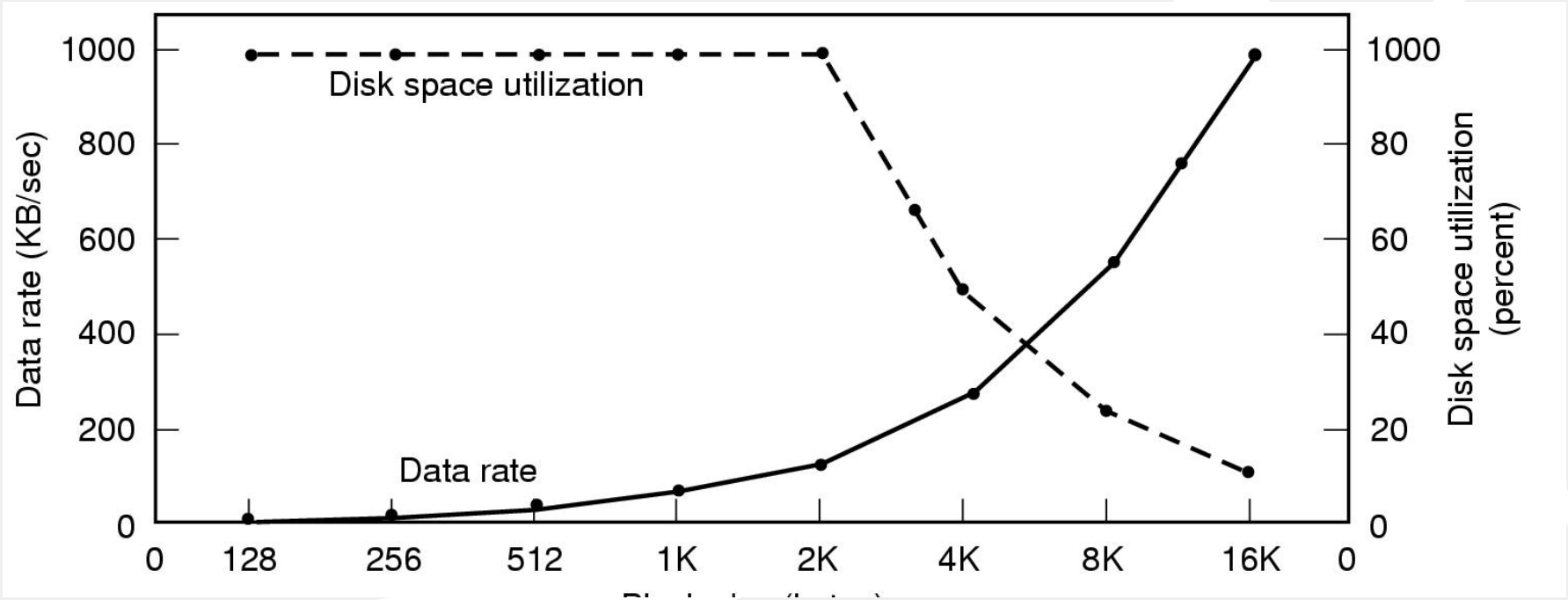


# What is the file allocation problem?

- Allocating disk space for files
- File allocation impacts space and time
  - Space usage on disk (wastage due to fragmentation)
  - Access speed (latency of access)
- Common file allocation techniques are:
  - Contiguous
  - Chained (linked)
  - Indexed

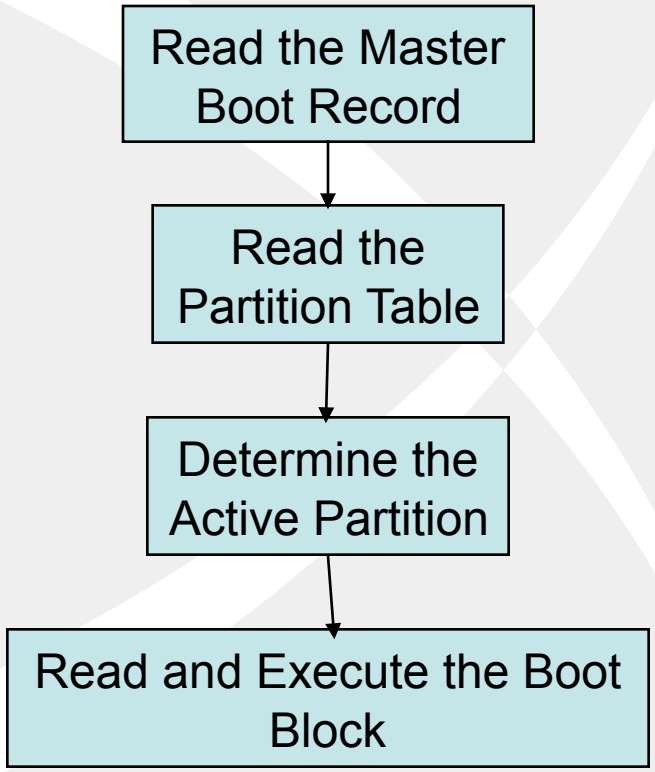
# What is the file allocation problem?

- Disk space is allocated on a block basis

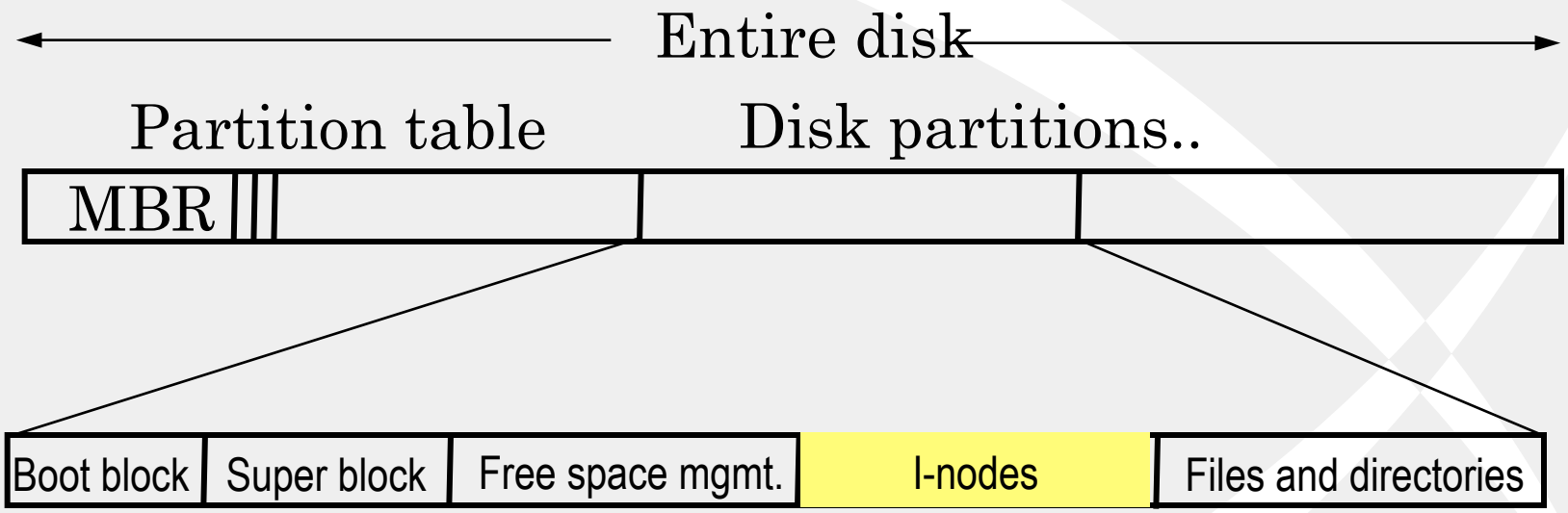


# File system layout

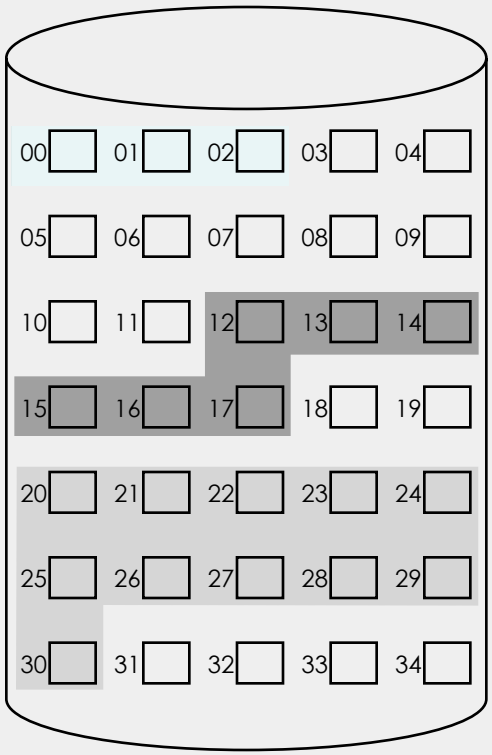
- File systems are stored on disk partitions
- Sector 0 of the disk is called the **MBR (master boot record)**



# File system layout



# Contiguous allocation

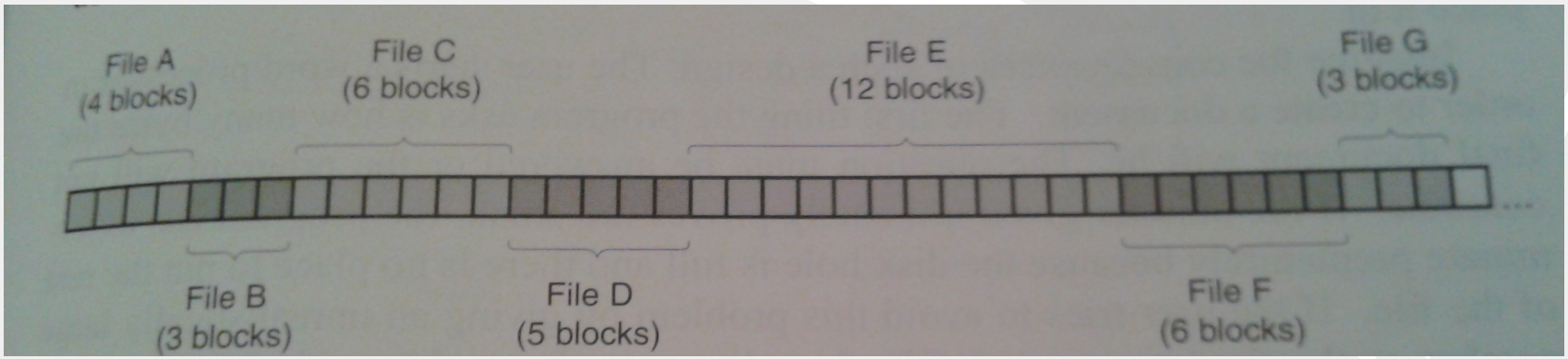



Directory

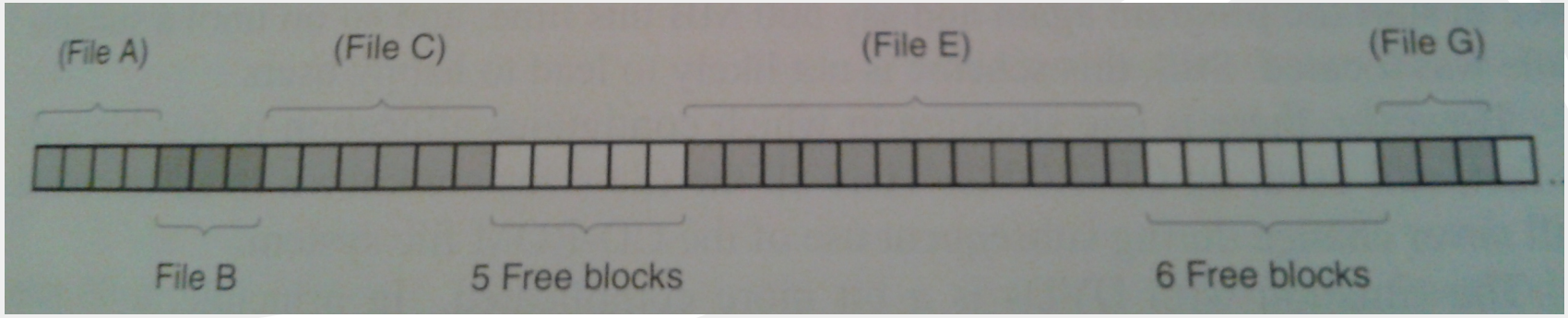
| *name* | *start* | *len.* |
|---|---|---|
| a.out | 00 | 3 |
| hw1.c | 12 | 6 |
| report.tex | 20 | 11 |

- Keep a free list of unused disk space.
- *Advantages*:
  - easy access, both sequential and random
  - simple
  - few seeks
- *Disadvantages*:
  - external fragmentation
  - may not know the file size in advance

# What is the file allocation problem?

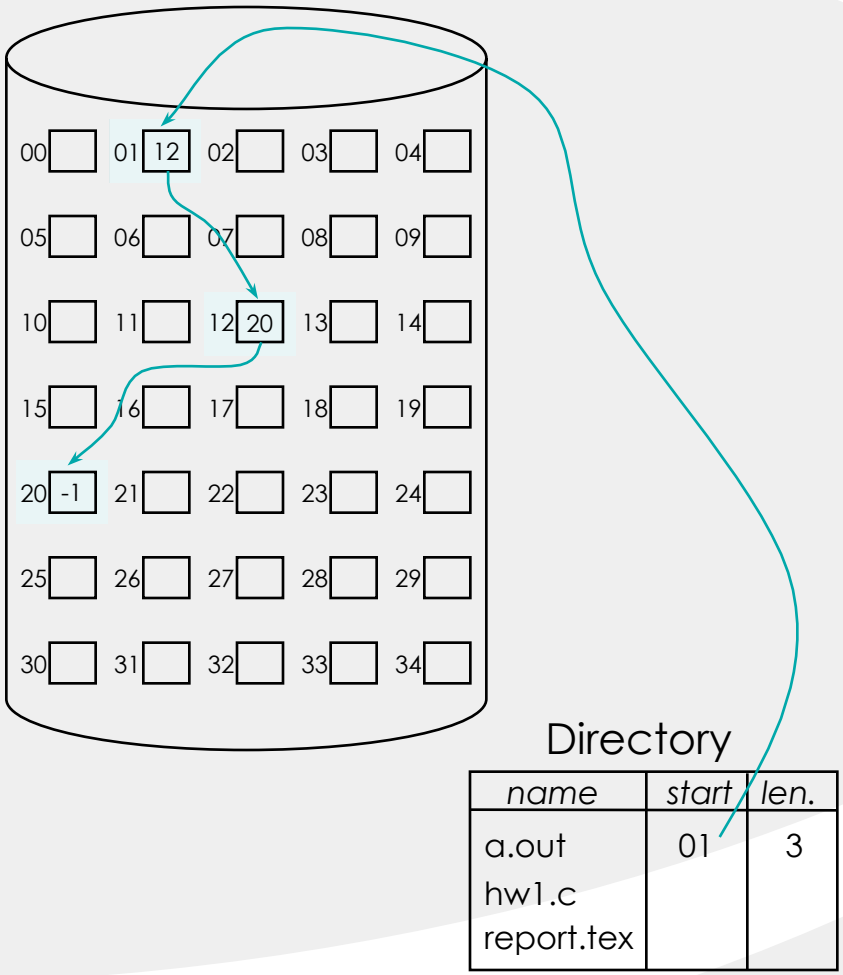



Contiguous allocation of files.



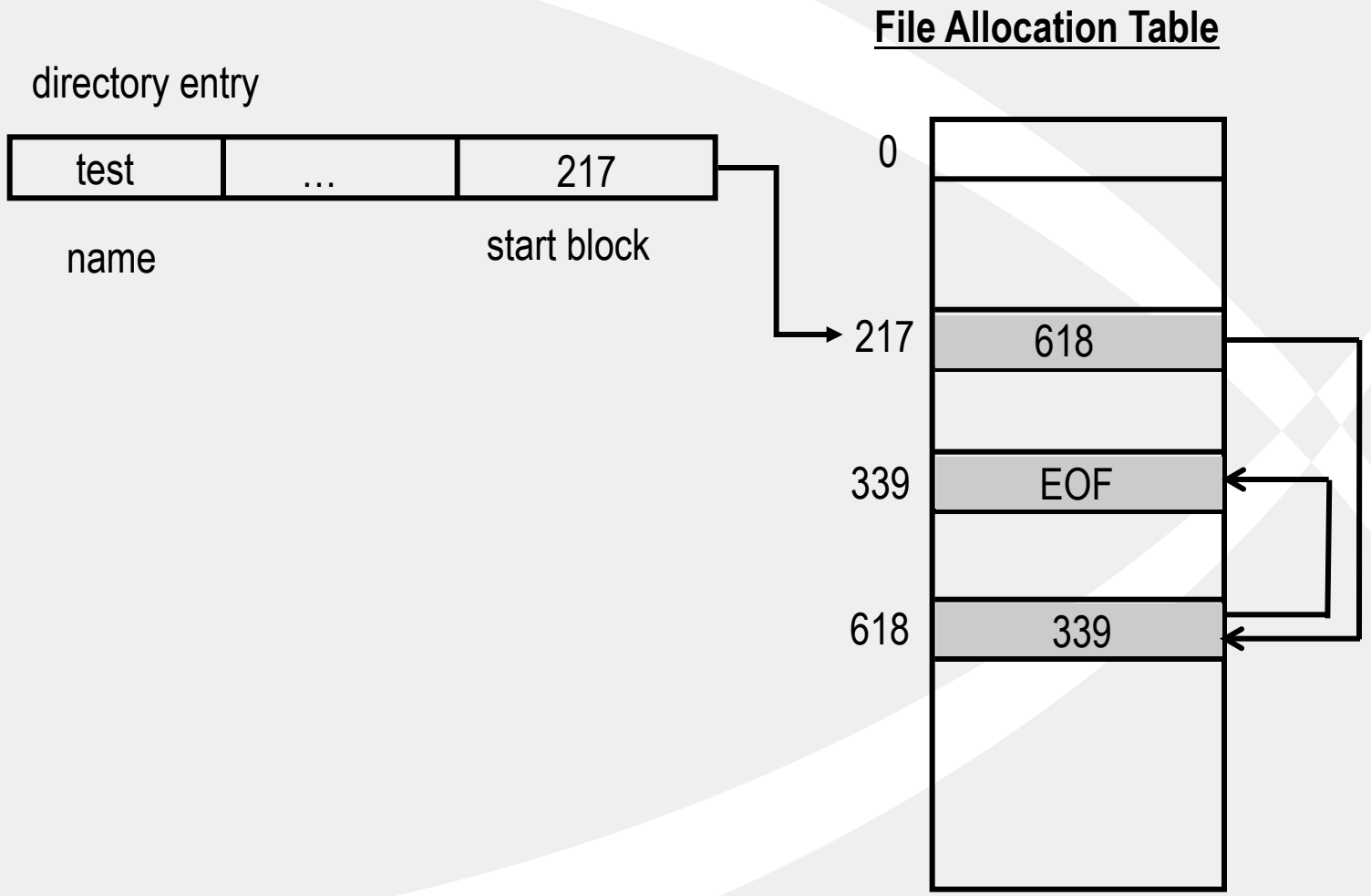

Contiguous allocation after files D and F removed

# Chained (linked) allocation

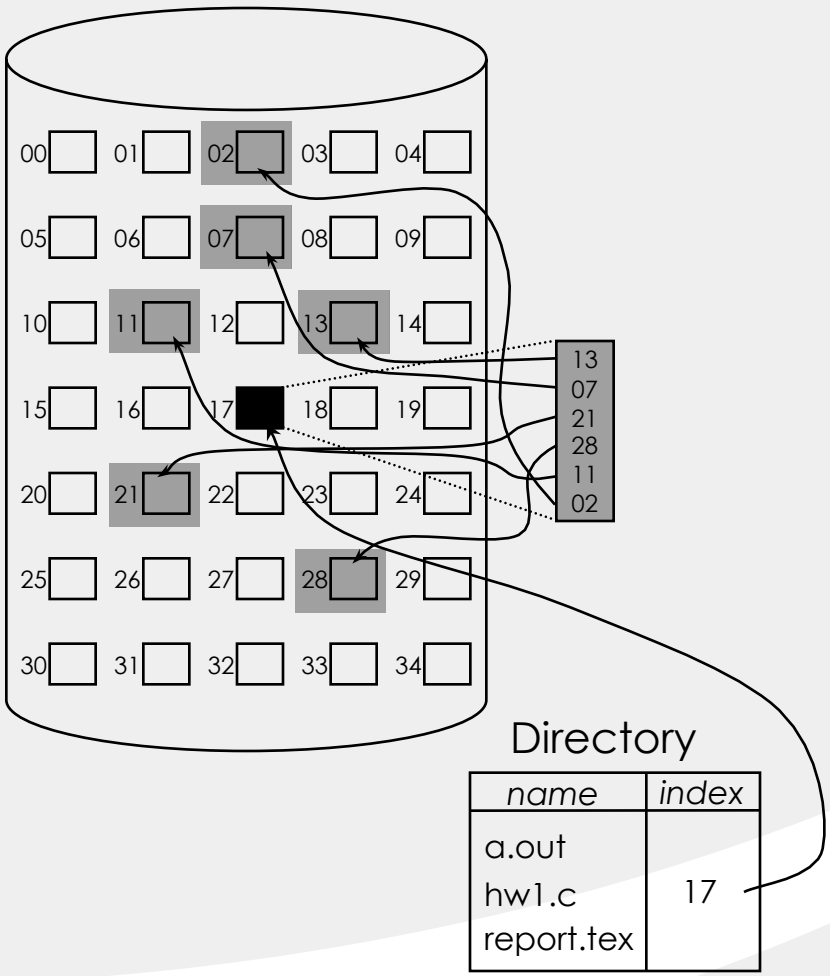



- Mark allocated blocks as in-use.
- *Advantages*:
  - no external fragmentation
  - files can grow easily
- *Disadvantages*:
  - lots of seeking
  - random access difficult
- *Enhancements:*
  - Can be enhanced with a in-memory table – called File allocation table (FAT)

# File allocation table



# Indexed allocation




Allocate an array of pointers during file creation. Fill the array as new disk blocks are assigned.

*Advantages*:

- small internal fragmentation
- easy sequential and random access

*Disadvantages*:

- lots of seeks if the file is big
- maximum file size is limited by the size of a block

*Example*:

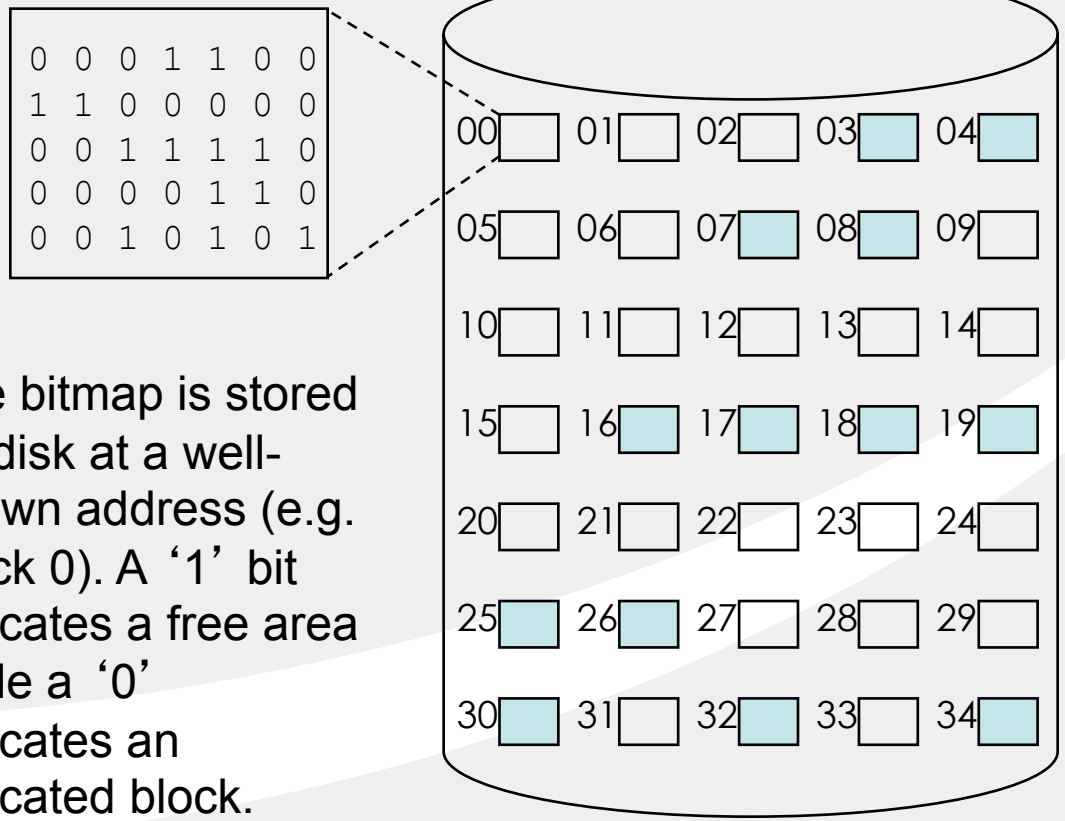
UNIX file system

# Free space management

- Disk space is fixed
  - Need to reuse space freed by deleted files
  - Need to keep track of free space as in main memory management
- Can use following techniques for free space management:
  - *Bit vectors*
  - *Linked lists or chains*
    - single list of a set of free block lists
  - *Indexing*
    - single level, multiple levels

# Free space management—bit vectors

- Use 1 bit per fixed size free area (e.g. block) on disk.

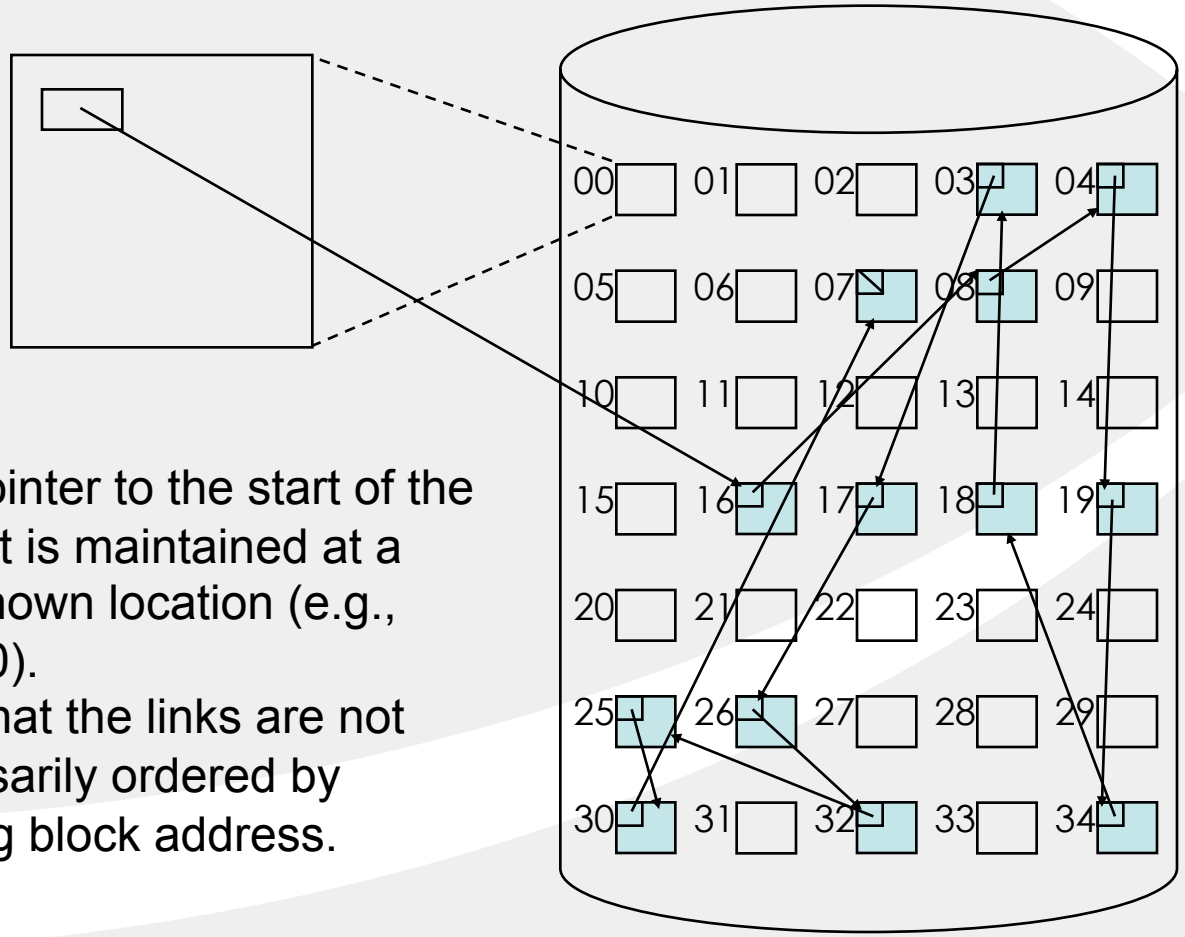



The bitmap is stored on disk at a well-known address (e.g. block 0). A '1' bit indicates a free area while a '0' indicates an allocated block.

*Free blocks on disk are shown shaded*

# Free space management—chains

- Chain/link pointers are stored *in* each disk block.

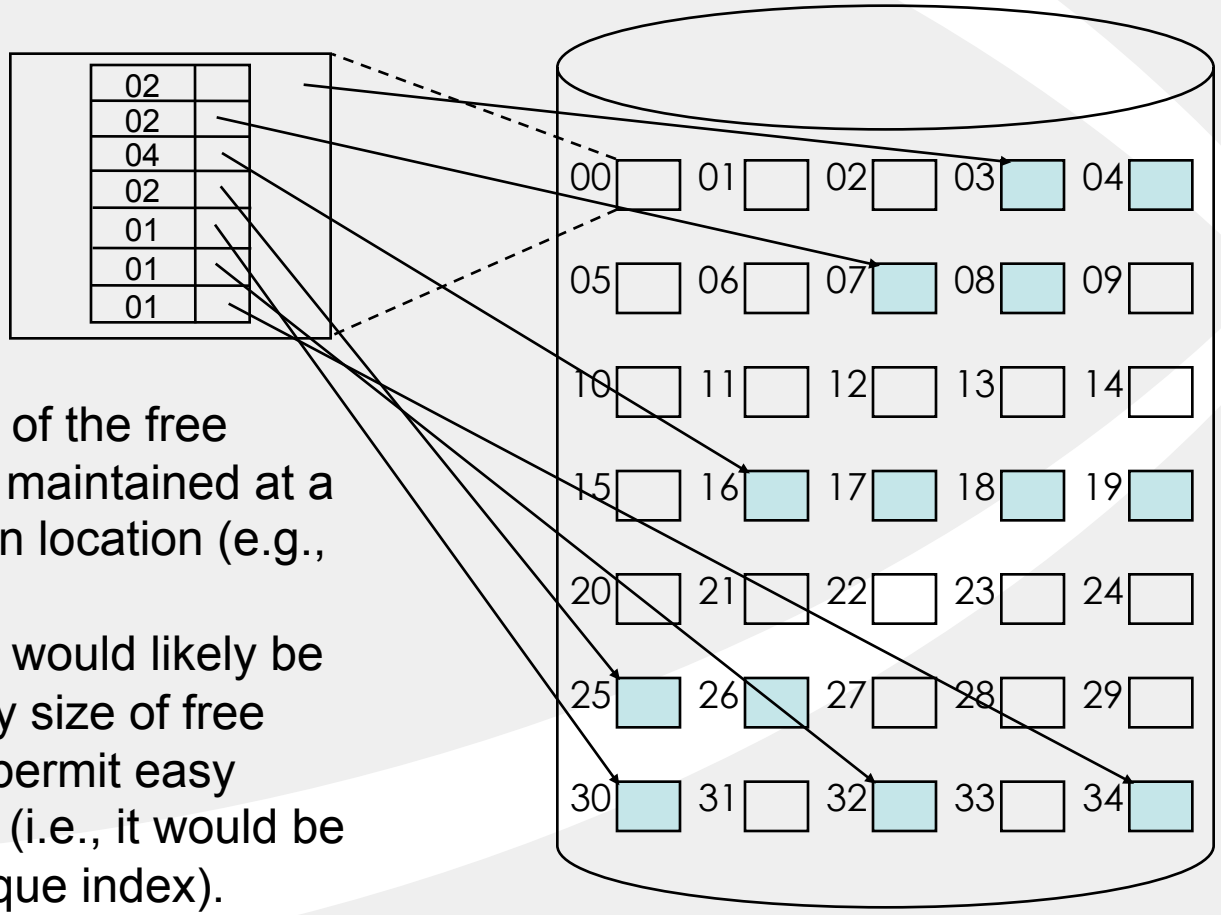



*Free blocks on disk are shown shaded*

The pointer to the start of the free list is maintained at a well-known location (e.g., block 0).
Note that the links are not necessarily ordered by starting block address.

# Free space management—indexed

- Maintain index pointing to all free blocks




*Free blocks on disk are shown shaded*

The index of the free regions is maintained at a well-known location (e.g., block 0).
The index would likely be ordered by size of free region to permit easy searching (i.e., it would be a non-unique index).

# Finding free space

- We must be able to search for free space of a specific size and how this is done depends on the free space management technique used
  - ***Bitmaps***—search bitmap for a sequence of sufficient 1 bits
    - Can use hardware instructions on some machines (e.g. find first one – `FFO`)
  - ***Index***—store the number of available blocks in each index entry and order index by size
    - e.g., Starting Block Number of Free Blocks

# Finding free space
*cont.*

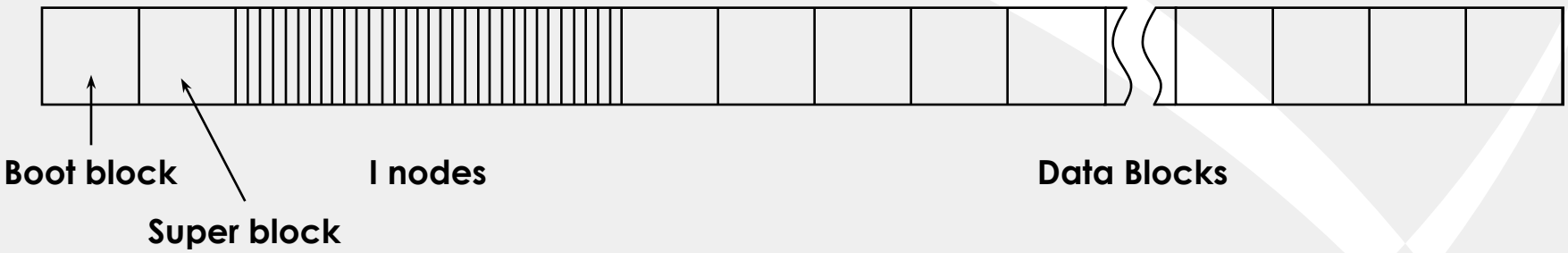
- ***Chained***
  - **First Fit**—find the first block on the chain big enough to accommodate the request
  - **Best Fit**—find the region on the chain nearest in size but at least as large as the request
  - **Worst Fit**—allocate from the largest region on the chain
  - **Next Fit**—like first fit but continue searching on the chain from where you last left off
    - Maintain chain in circular fashion
    - Avoids constant allocation in localized region of memory

# Other file system issues

- *Disk blocking*
  - multiple sectors per block for efficiency
- *Disk quotas*
- *Reliability*
  - Backup/restore (disaster scenarios)
  - File system (consistency) check (e.g., UNIX **`fsck`**)
- *Performance*
  - Block or buffer caches (a collection of blocks kept in memory)

# An example: UNIX file system

*Disk (partition) layout in traditional UNIX systems*

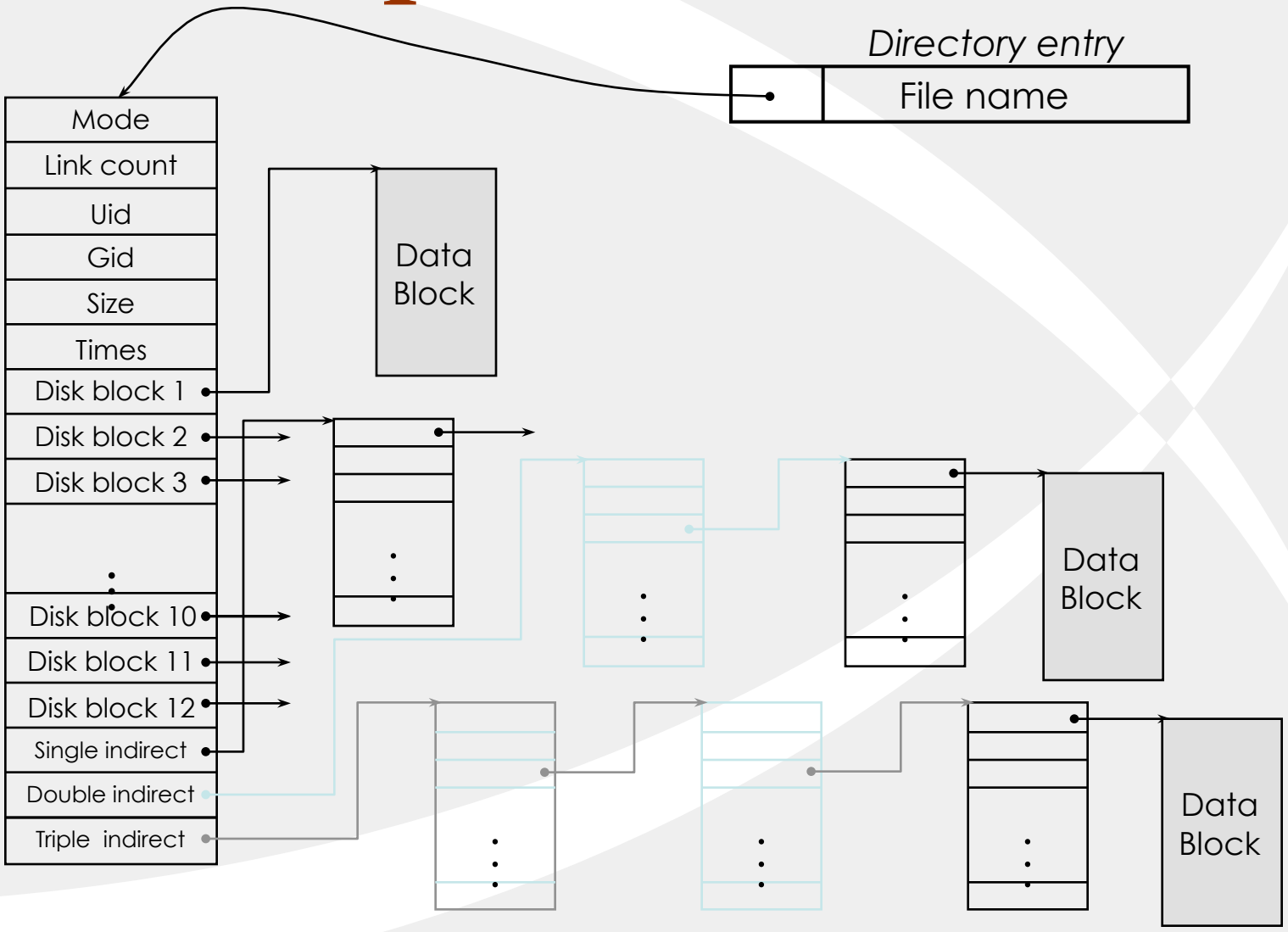



The boot block usually contains (bootstrap) code to boot the system.

The super block contains critical information about the layout of the file system, such as number of I-nodes and the number of disk blocks.

Each I-node entry contains the file attributes, except the name. The first I-node points to the block containing the root directory of the file system.

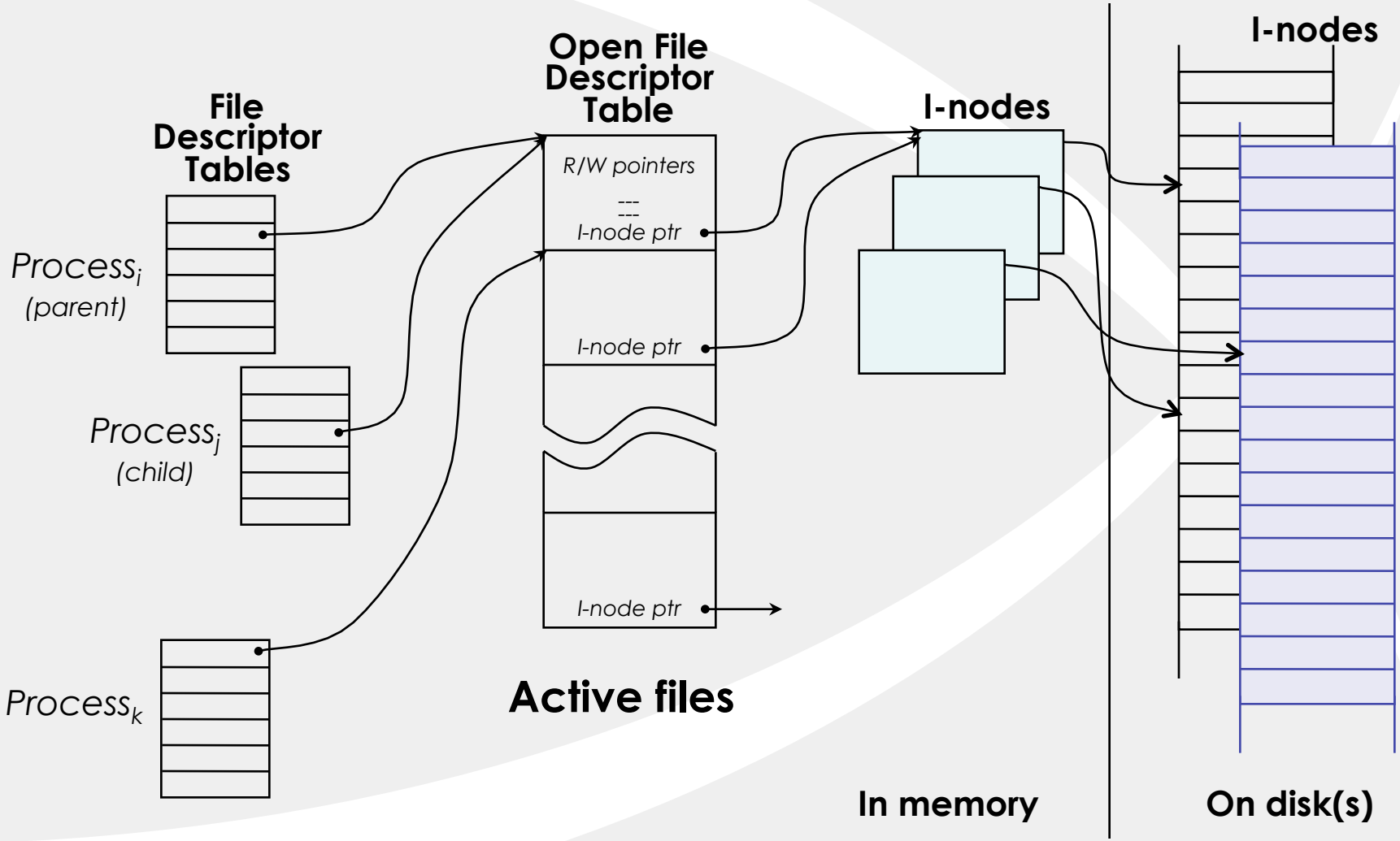
# An example: UNIX I-nodes



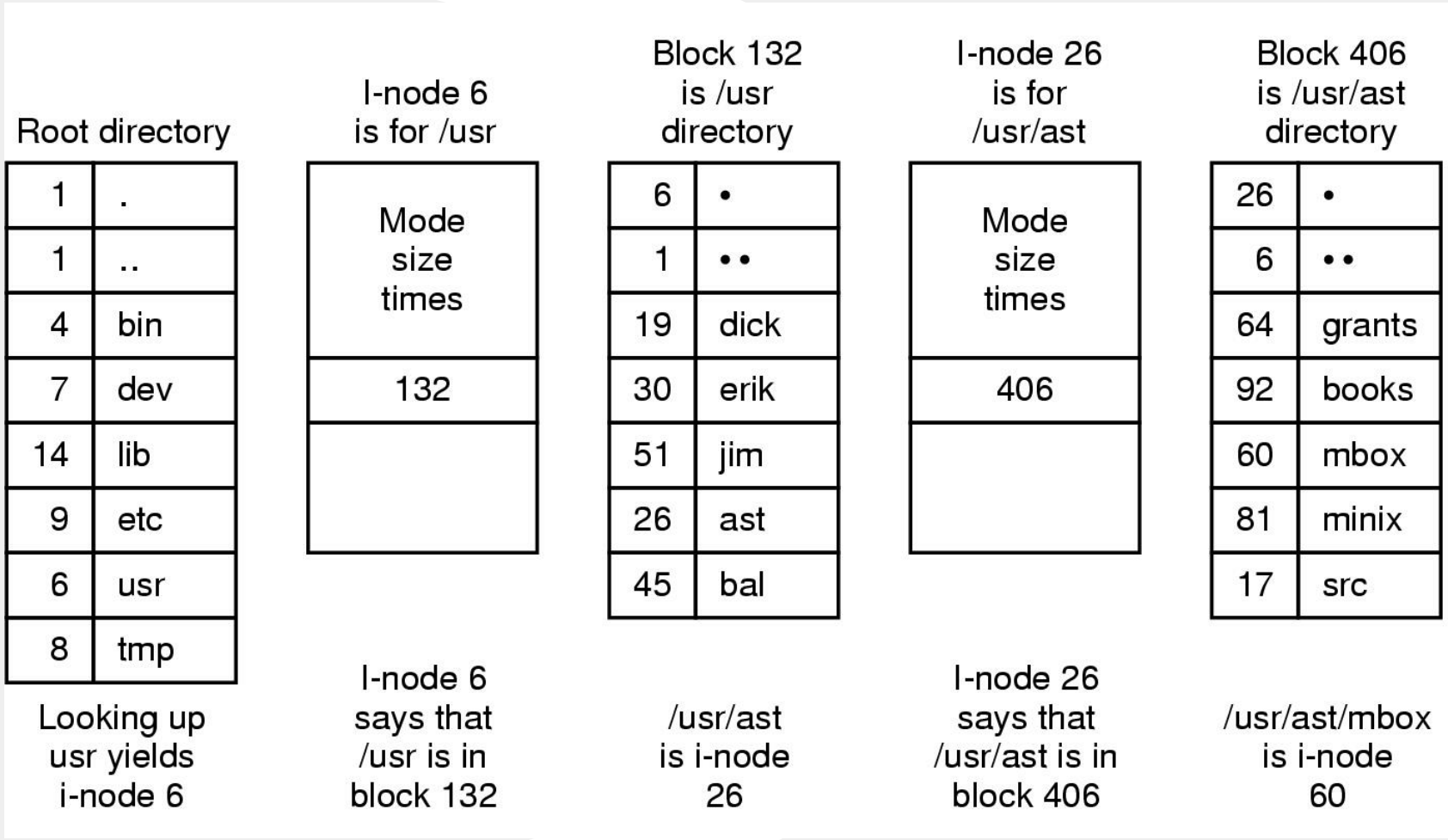
# An example: UNIX file system

- There are three different indirection to access a file:
  - *File Descriptor Table:* one per process, keeping track of open files.
  - *Open File Table*: one per system, keeping track of all the files currently open.
  - *I-node Table:* one per system (disk volume or partition) keeping track all files.

# An example: UNIX file system

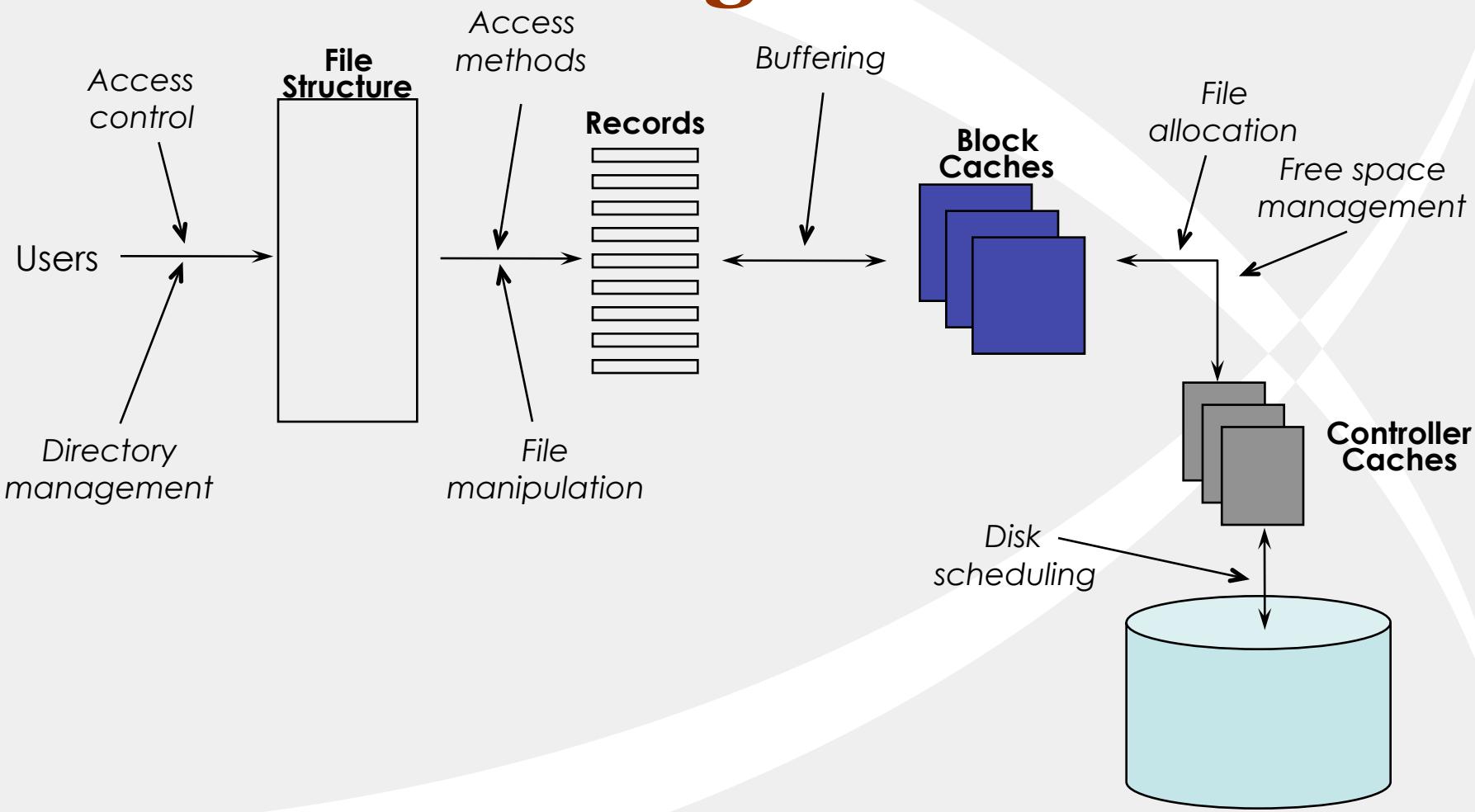


# UNIX File System Example

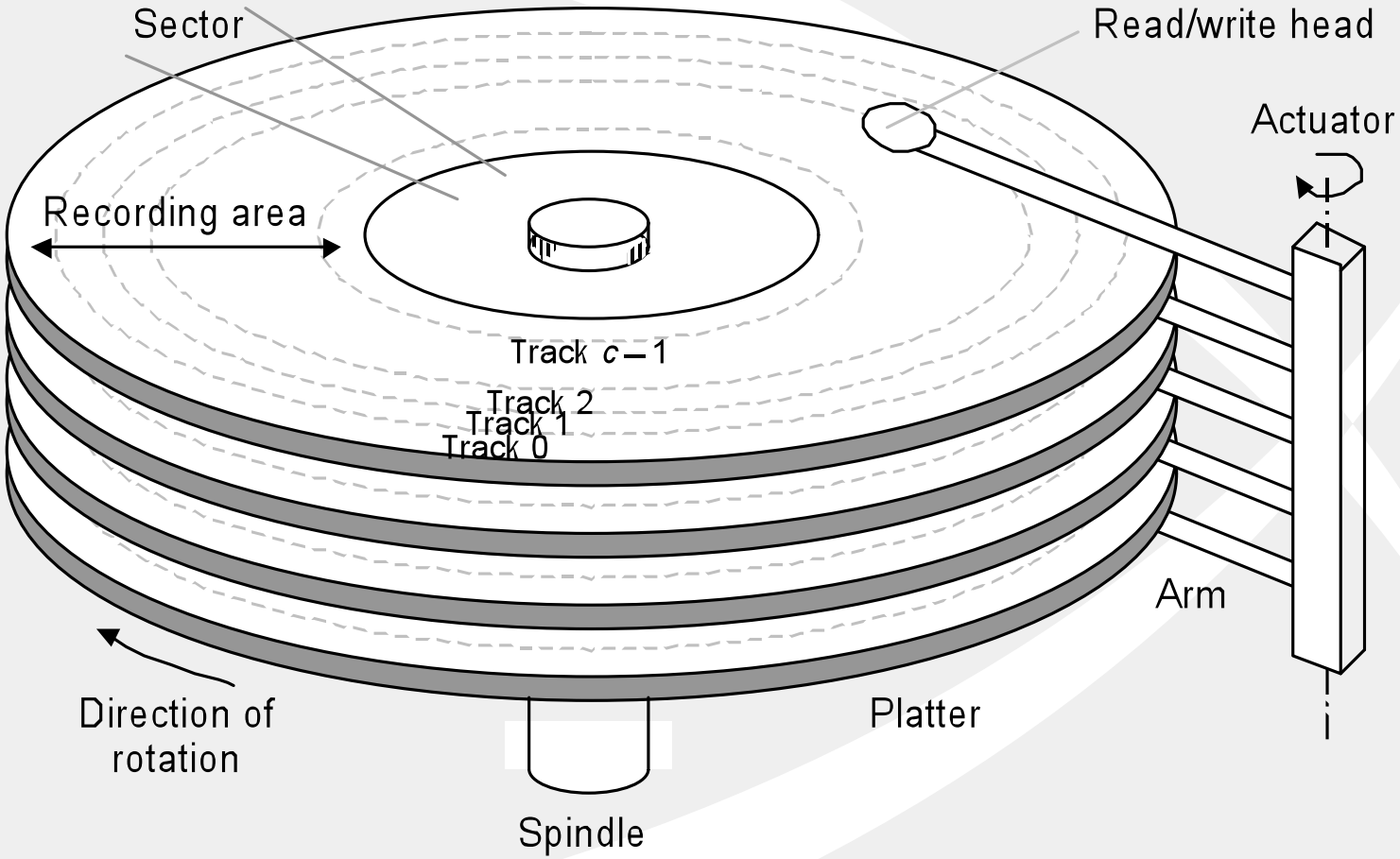



The steps in looking up */usr/ast/mbox*

# Elements of storage management

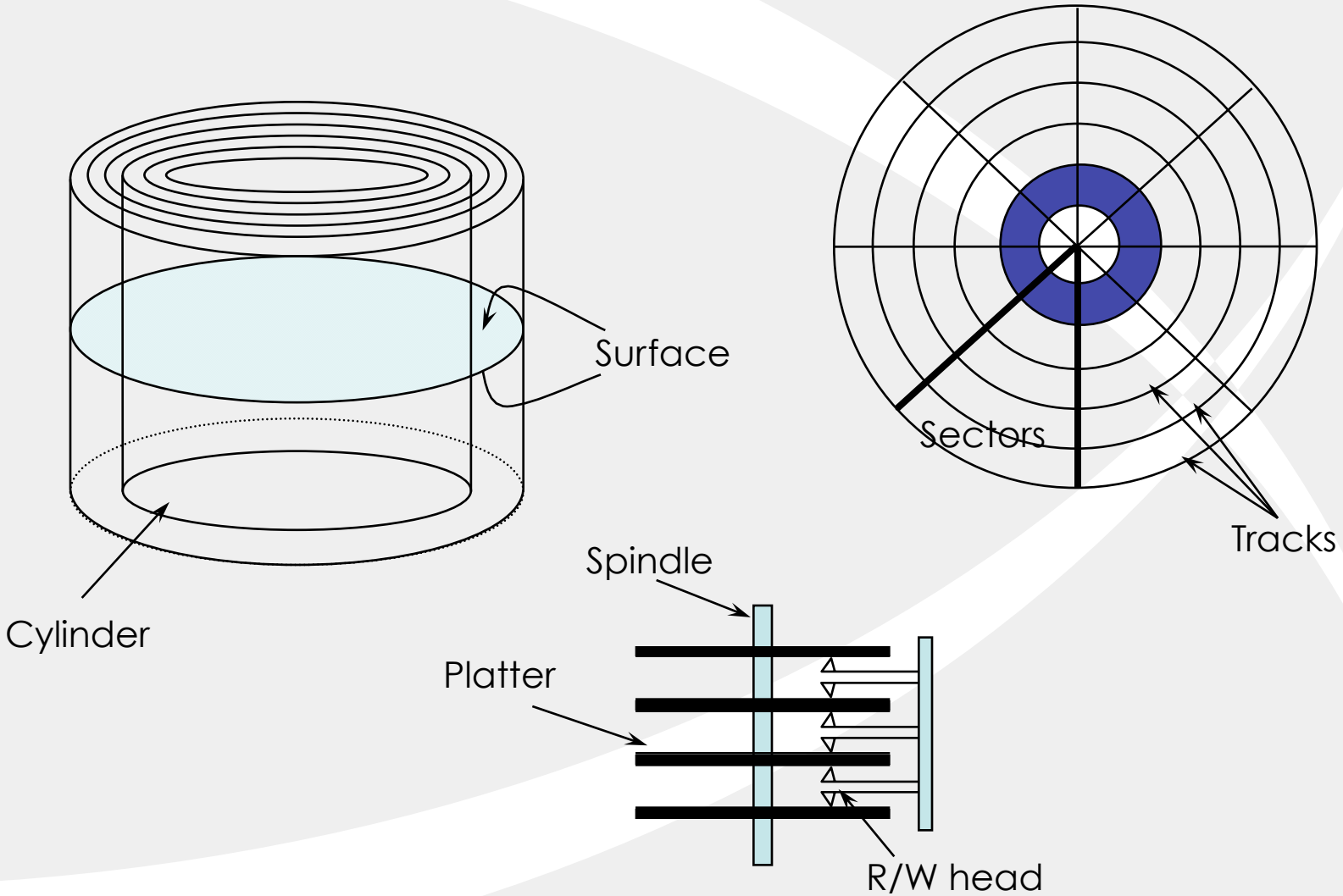


# Disk Memory Basics

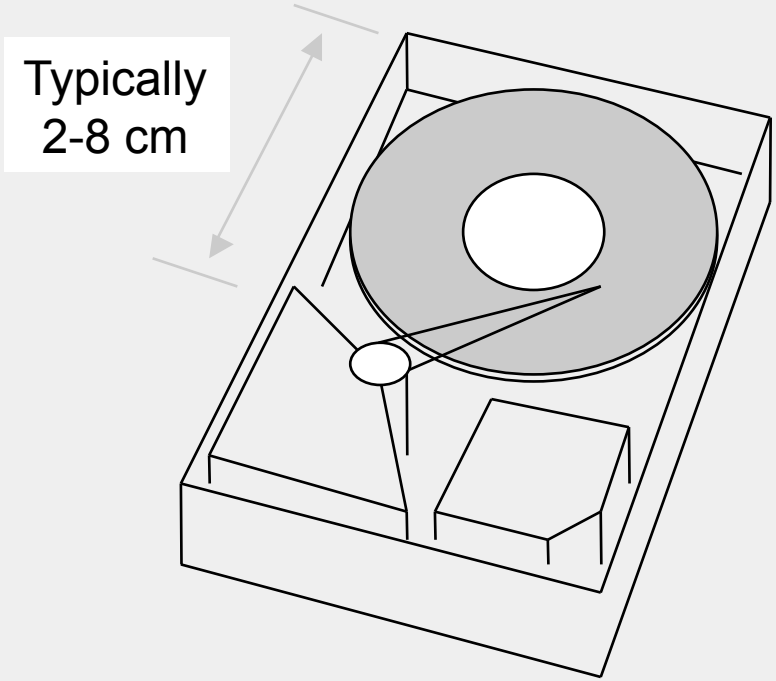



Disk memory elements and key terms.

# Disk structure (another view)

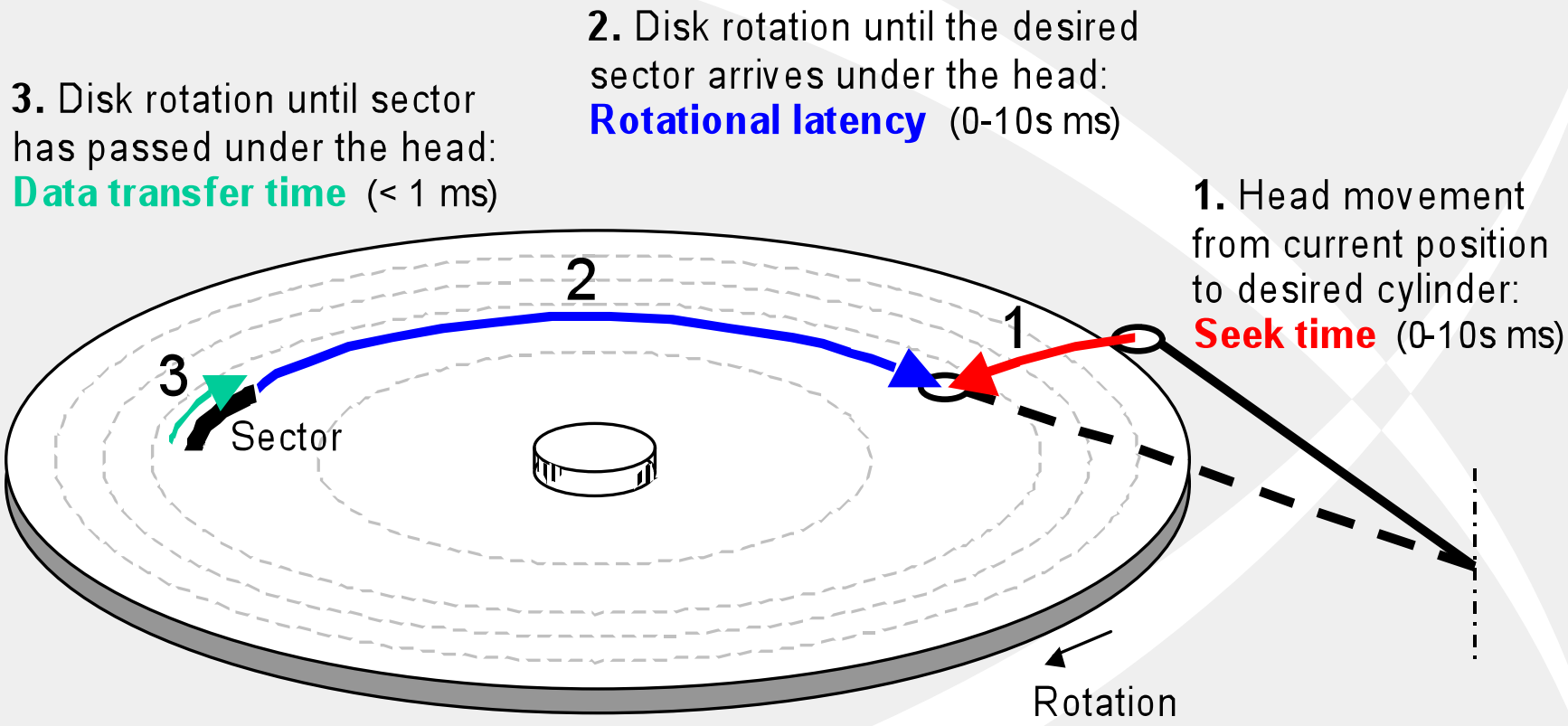


# Disk Drives




Comprehensive info about disk memory: http://www.storageview.com/guide/

# Access Time for a Disk



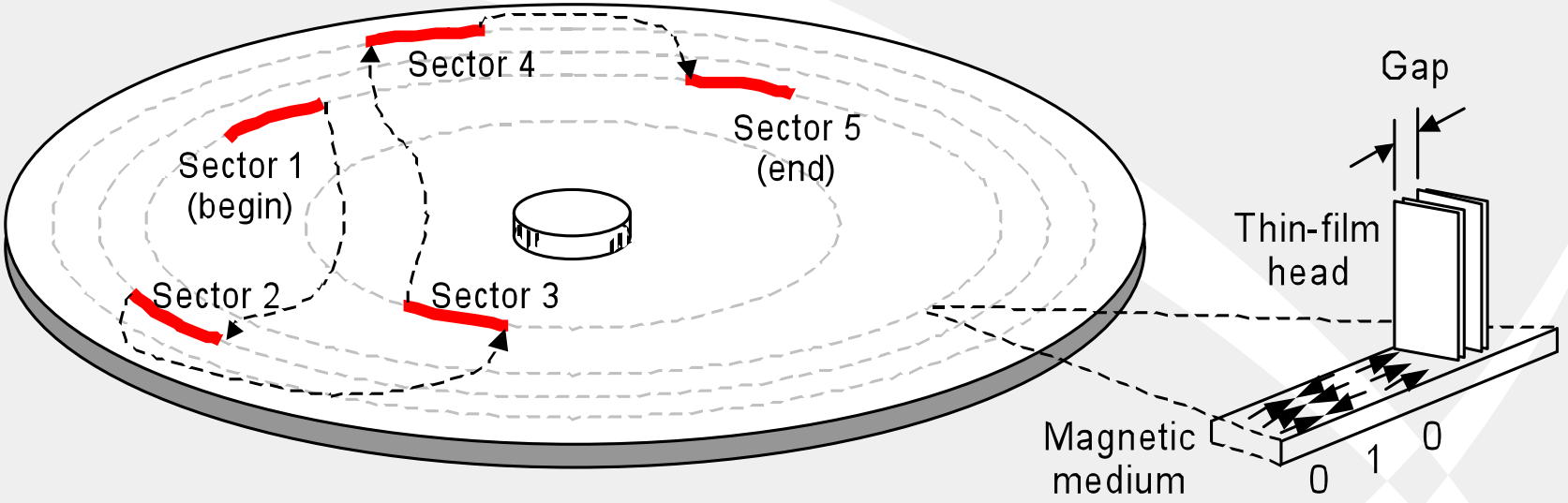

The three components of disk access time. Disks that spin faster have a shorter average and worst-case access time.

# Representative Magnetic Disks

Key attributes of three representative magnetic disks, from the highest capacity to the smallest physical size (ca. early 2003).

| Manufacturer and Model Name | Seagate Barracuda 180 | Hitachi DK23DA | IBM Microdrive |
|---|---|---|---|
| Application domain | Server | Laptop | Pocket device |
| Capacity | 180 GB | 40 GB | 1 GB |
| Platters / Surfaces | 12 / 24 | 2 / 4 | 1 / 2 |
| Cylinders | 24 247 | 33 067 | 7 167 |
| Sectors per track, avg | 604 | 591 | 140 |
| Buffer size | 16 MB | 2 MB | 1/8 MB |
| Seek time, min,avg,max | 1, 8, 17 ms | 3, 13, 25 ms | 1, 12, 19 ms |
| Diameter | 3.5″ | 2.5″ | 1.0″ |
| Rotation speed, rpm | 7 200 | 4 200 | 3 600 |
| Typical power | 14.1 W | 2.3 W | 0.8 W |

# Organizing Data on Disk



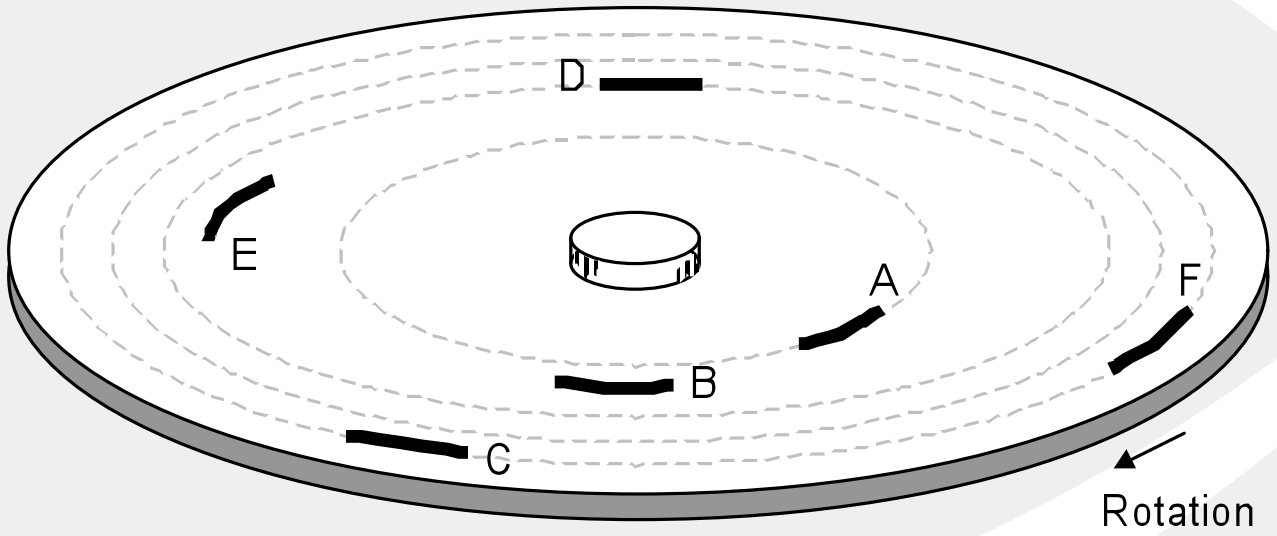

Magnetic recording along the tracks and the read/write head.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 16 | 32 | 48 | 1 | 17 | 33 | 49 | 2 | Track *i* |
| | 30 | 46 | 62 | 15 | 31 | 47 | 0 | 16 | 32 | Track *i* + 1 |
| | 60 | 13 | 29 | 45 | 61 | 14 | 30 | 46 | 62 | Track *i* + 2 |
| | 27 | 43 | 59 | 12 | 28 | 44 | 60 | 13 | 29 | Track *i* + 3 |

Logical numbering of sectors on several adjacent tracks.

# Disk Performance

Average rotational latency = 30 / rpm s = 30 000 / rpm ms




Arrival order of access requests:

A, B, C, D, E, F

Possible out-of-order reading:

C, F, D, E, B, A

Reducing average seek time and rotational latency by performing disk accesses out of order.

# Disk Caching

**Same idea as processor cache: bridge main-disk speed gap**

Read/write an entire track with each disk access:

"Access one sector, get 100s free," hit rate around 90%

Disks listed in above table have buffers from 1/8 to 16 MB

Rotational latency eliminated; can start from any sector

Need back-up power so as not to lose changes in disk cache

**Placement options for disk cache**

In the disk controller:

Suffers from bus and controller latencies even for a cache hit

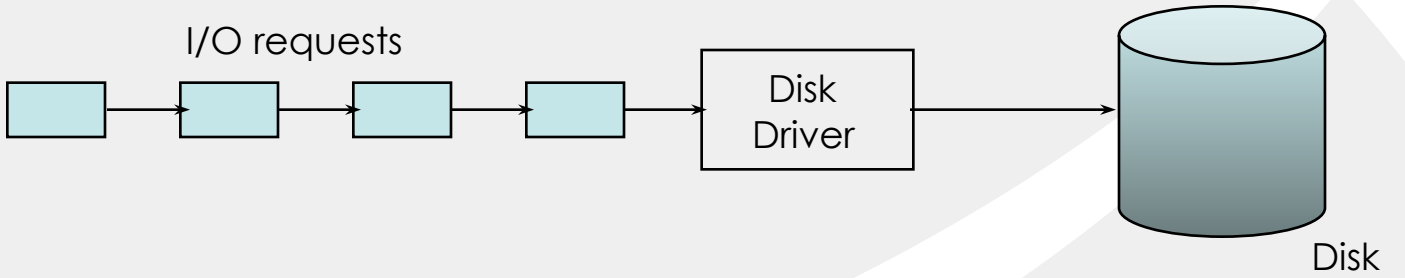
Closer to the CPU:

Avoids latencies and allows for better utilization of space

Intermediate or multilevel solutions

# Disk scheduling

- Scheduling problem: maximize the throughput with concurrent I/O requests from




- Strategy: minimize the time spent by disk seeking for data – maximizes the time spent reading/writing data

# Disk scheduling strategies

- Commonly used strategies include:
  - *Random*
    - select from pool randomly
    - worst performer
    - sometimes useful as a benchmark for analysis and simulation
  - *First Come First Served (FCFS) or FIFO*
    - fairest of them all; no starvation; requests are honored in the order they are received
    - works well for few processes (principle of locality)
    - approaches to r*andom* as number of processes competing for the given disk increases

# Disk scheduling strategies

- *Priority*
  - access to the disk is not actually controlled by the disk management software
  - based on processes' execution priority
  - designed to meet job throughput criteria and *not* to optimize the disk usage
- *Last In First Out (LIFO)*
  - service the most recently arriving request first
  - can be useful for processing sequential files
  - real danger of starvation; once a process falls behind it can be services only if the entire list ahead of it empties

# Disk scheduling strategies

- Above algorithms have based disk scheduling decision on the requestor process
- Following algorithms use requested item, i.e. the requested disk address
  - *Shortest Service Time First (SSTF)*
    - select the item requiring the shortest seek time
    - no guarantee of improved average seek time in any particular circumstance but
    - in general better average seek time than FIFO
    - random tie breaker used, if needed, to decide in which direction to move

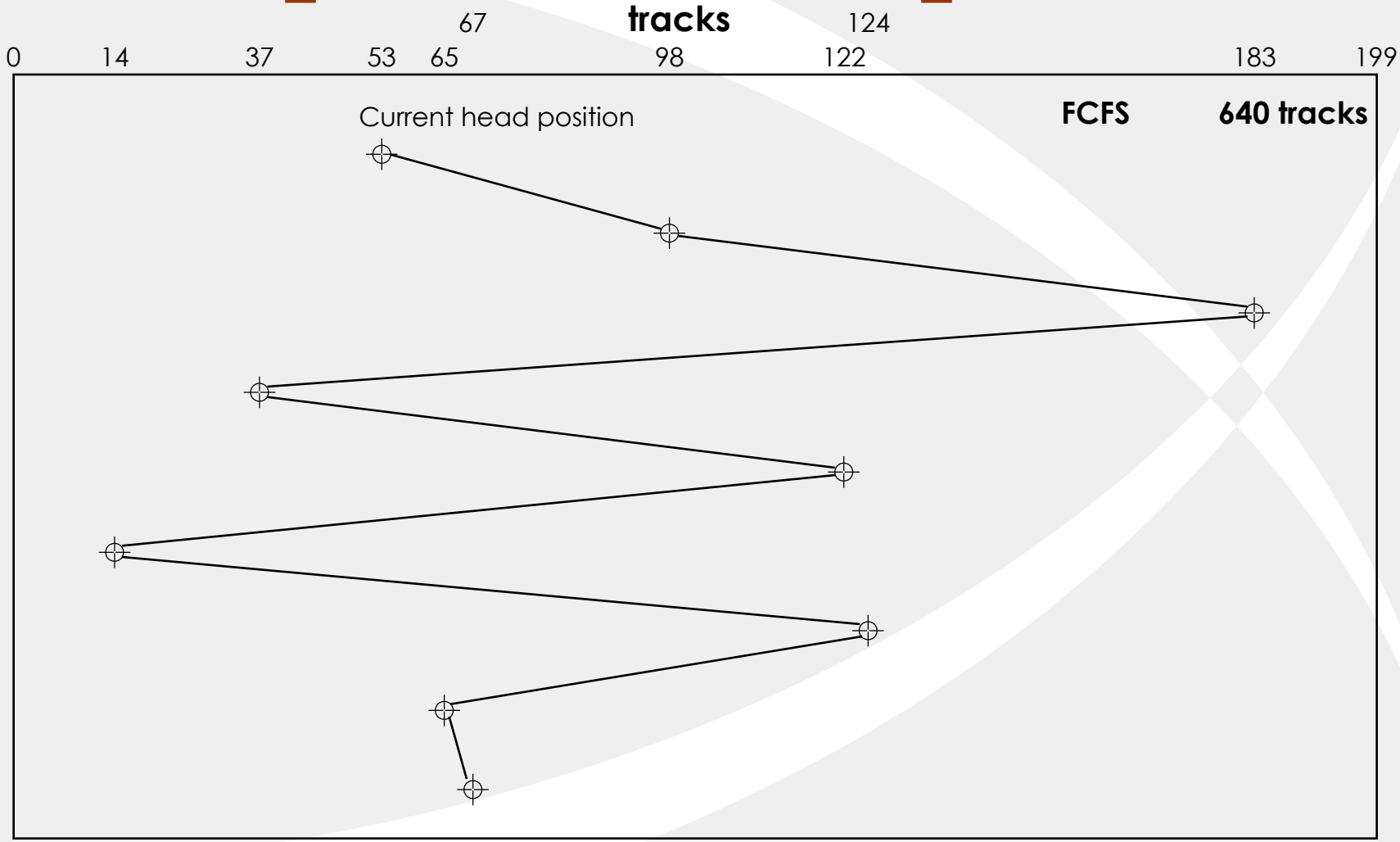
# Disk scheduling strategies

- *SCAN*—back and forth over disk
  - heads move in only one direction until the last track is reached, then the direction is reversed
  - if the direction of the heads' travel is reversed once there are no more requests in that direction this method is called *LOOK*
  - no danger of starvation, but
  - biased **against** the most recently used area on the disk
    - Doesn't exploit locality as well as SSTF though or as LIFO
  - often similar to *SSTF*
- *C-SCAN*—circular SCAN or one way SCAN and fast return
  - scans in only one direction to the last track and then returns heads quickly to the beginning
  - reduces the maximum delay for new arrivals

# Disk scheduling strategies

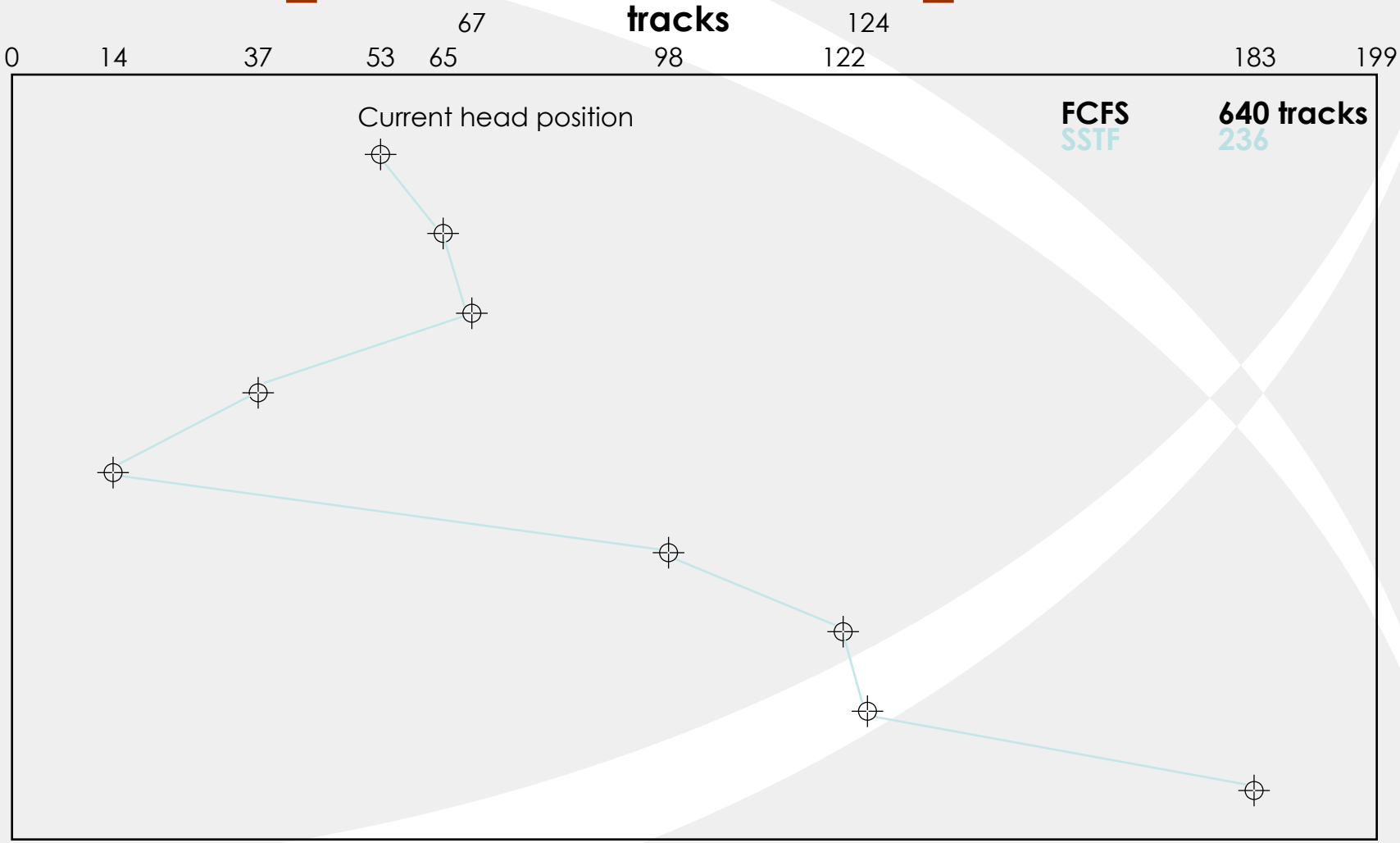
- *LOOK*—look for a request before moving in that direction
  - Heads move in only one direction until there are no more requests in that direction, then the direction is reversed.
- *C-LOOK*—circular LOOK
  - Scans in only one direction until there are no more requests in that direction and then returns heads quickly to the beginning

# A comparative example—FCFS



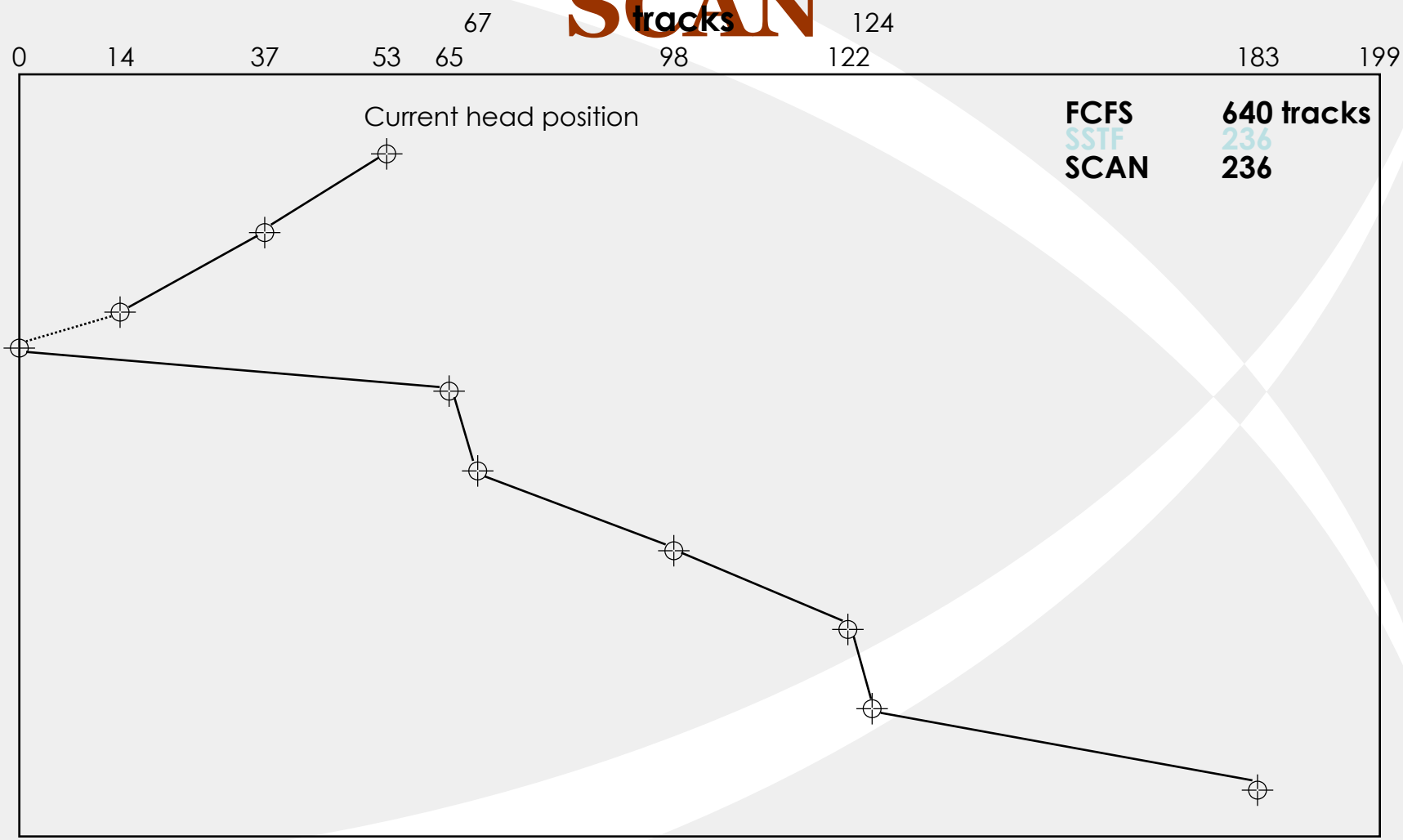

**Request queue:** 98, 183, 37, 122, 14, 124, 65, 67

# A comparative example—SSTF



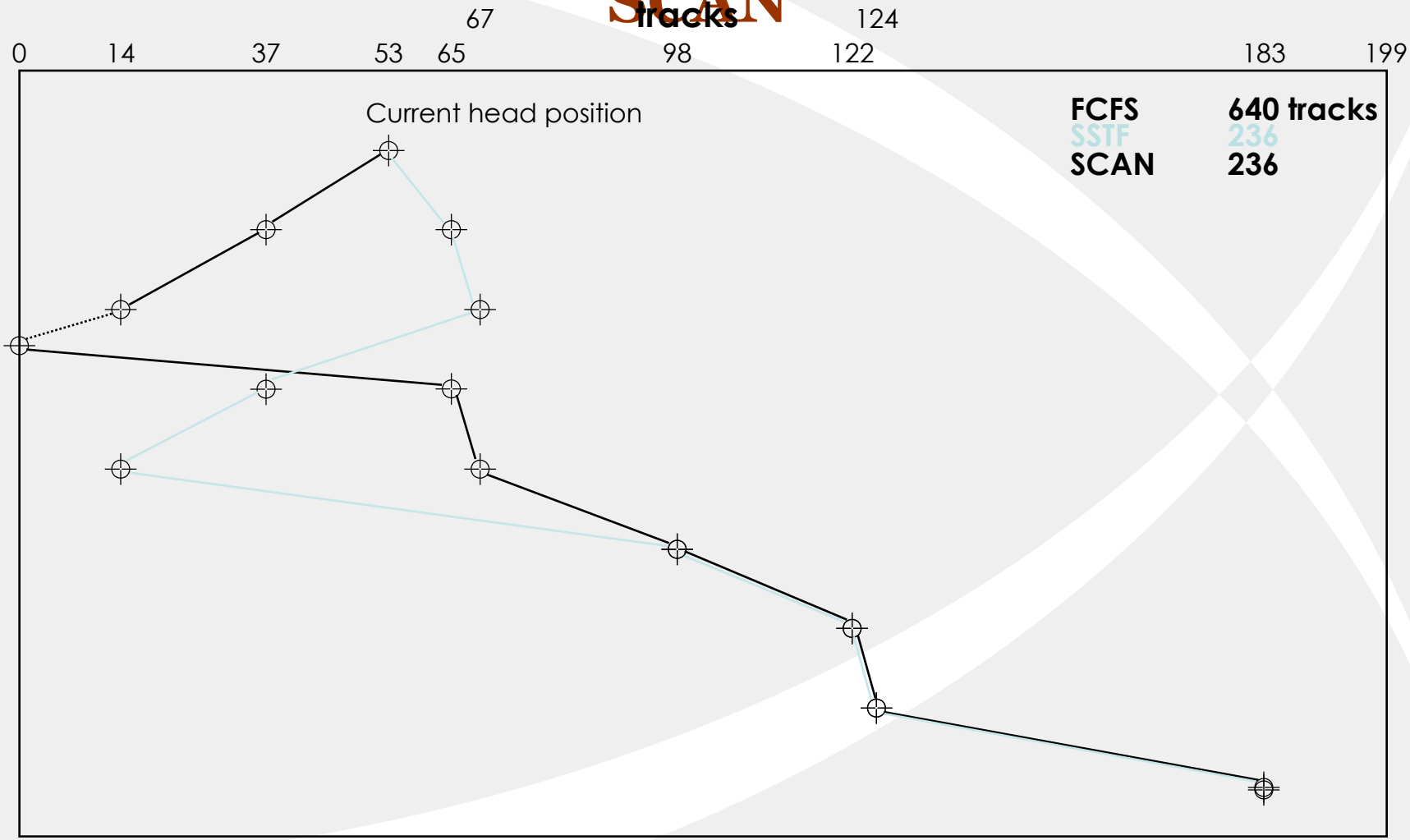

**Request queue:** 98, 183, 37, 122, 14, 124, 65, 67

# A comparative example—SCAN



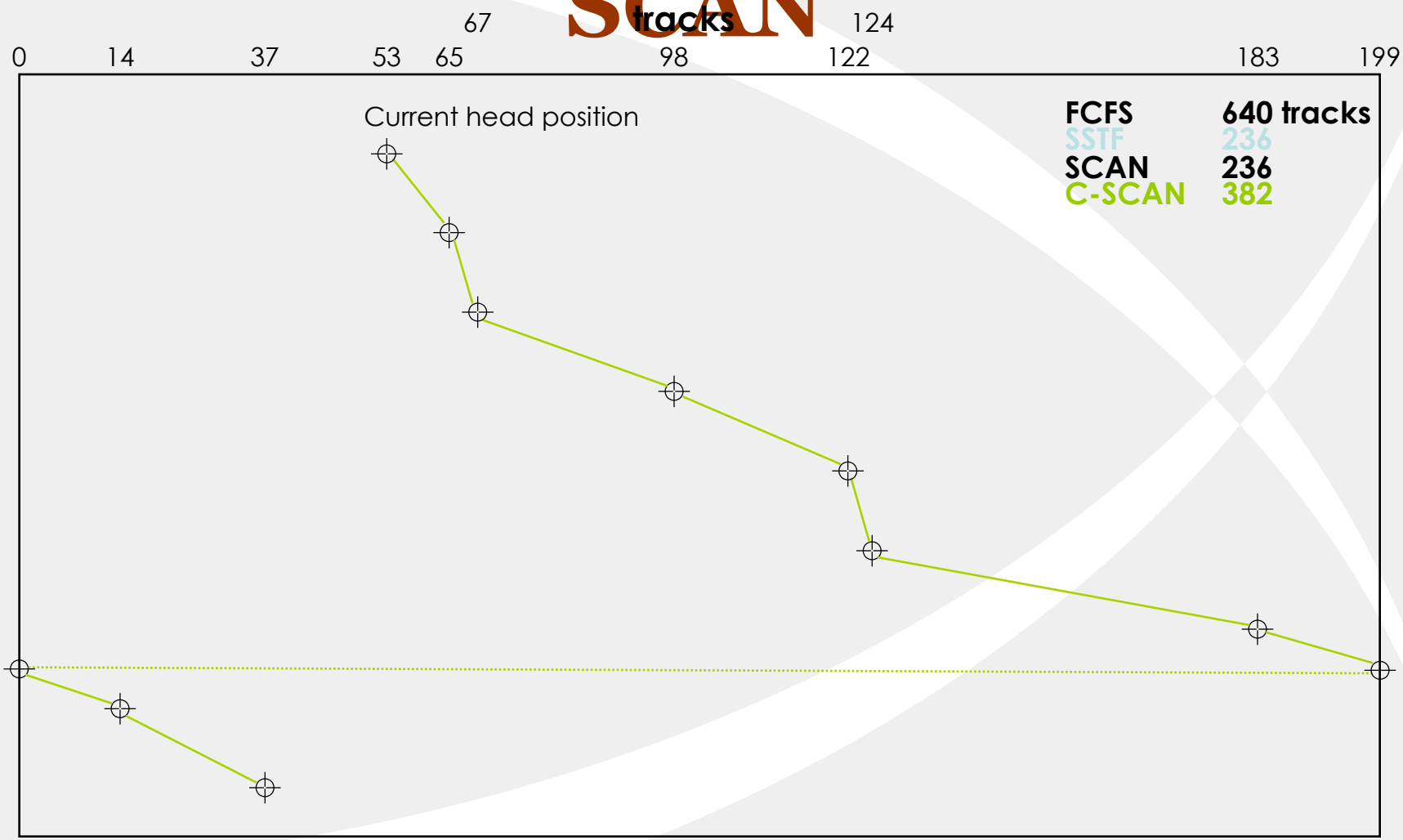

**Request queue:** 98, 183, 37, 122, 14, 124, 65, 67

# A comparative example—SSTF vs SCAN



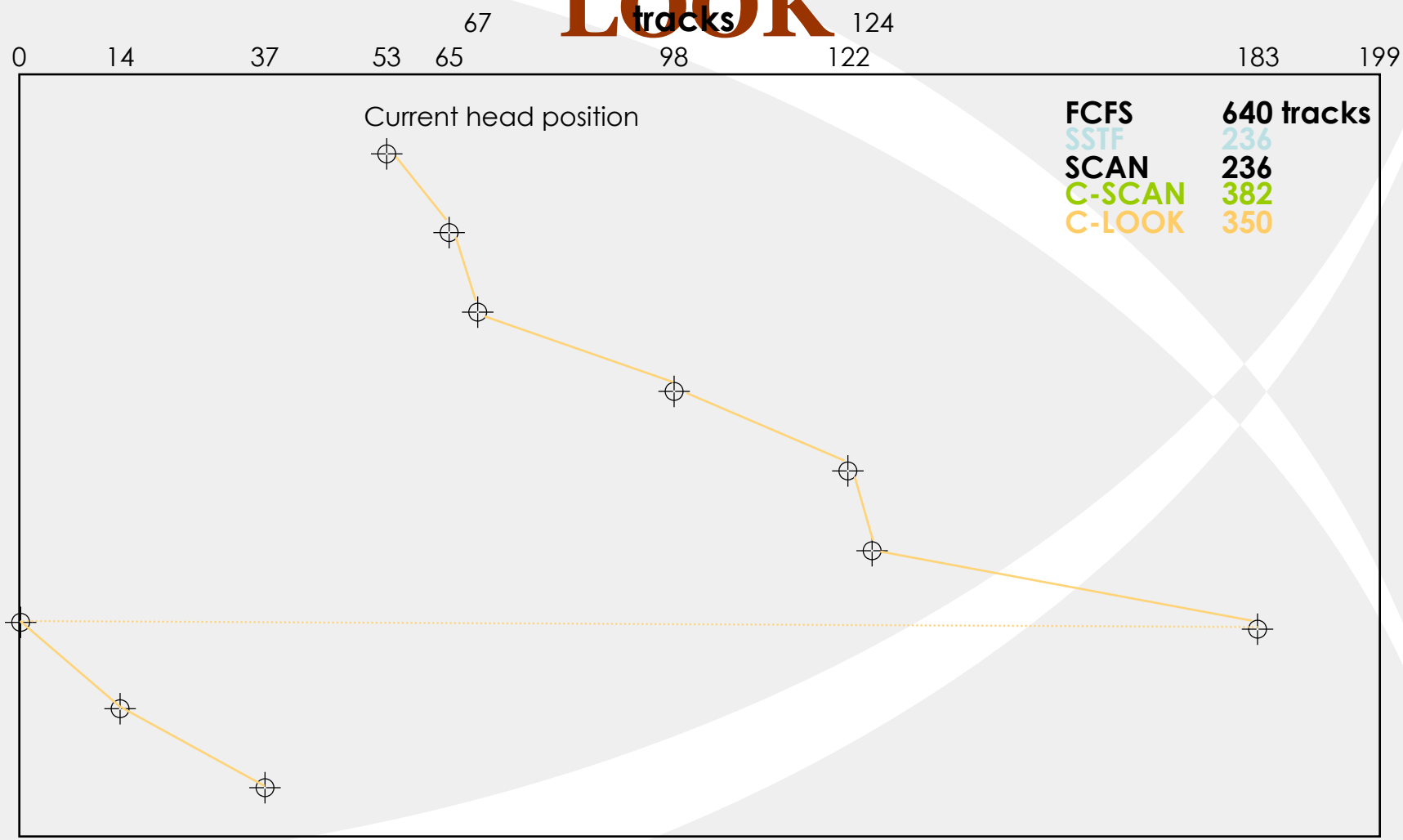

**Request queue:** 98, 183, 37, 122, 14, 124, 65, 67

# A comparative example—C-SCAN



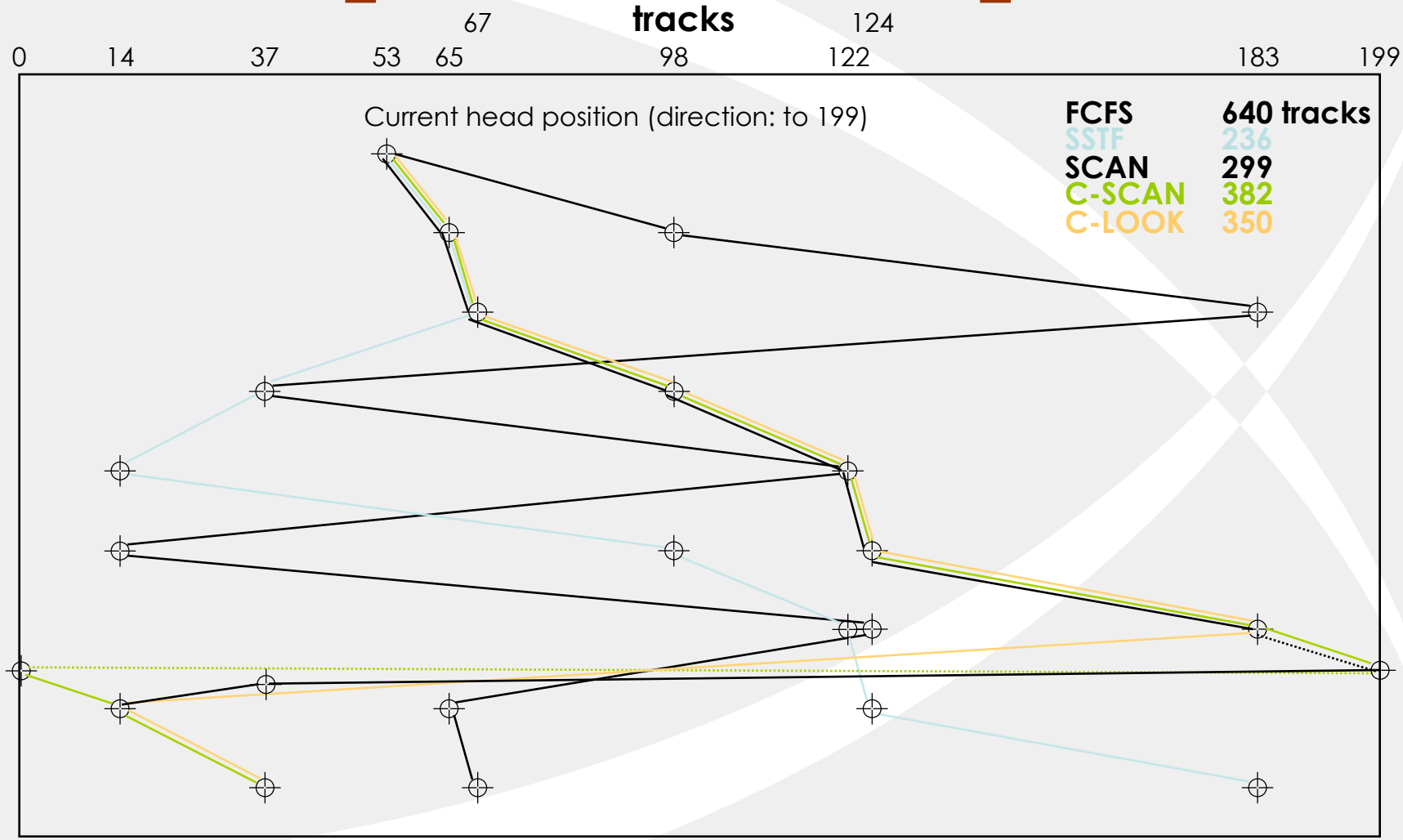

**Request queue:** 98, 183, 37, 122, 14, 124, 65, 67

# A comparative example—C-LOOK



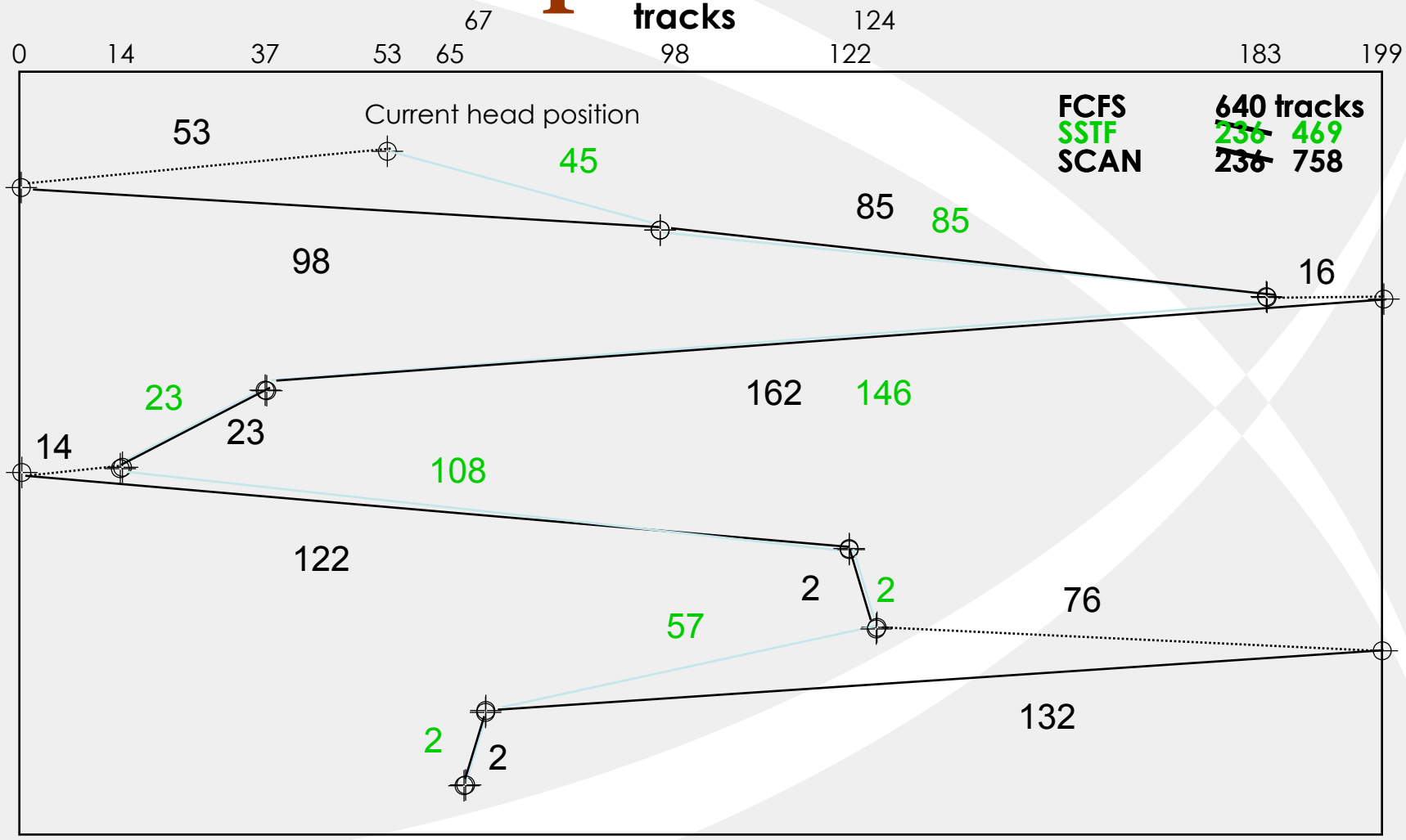

**Request queue:** 98, 183, 37, 122, 14, 124, 65, 67

# A comparative example—All




**Request queue:** 98, 183, 37, 122, 14, 124, 65, 67

# Grouped arrivals




**Request queue:** 98, 183, 37, 122, 14, 124, 65, 67
**Arrivals**: 98, 183; 37; 122, 14; 124, 65, 67