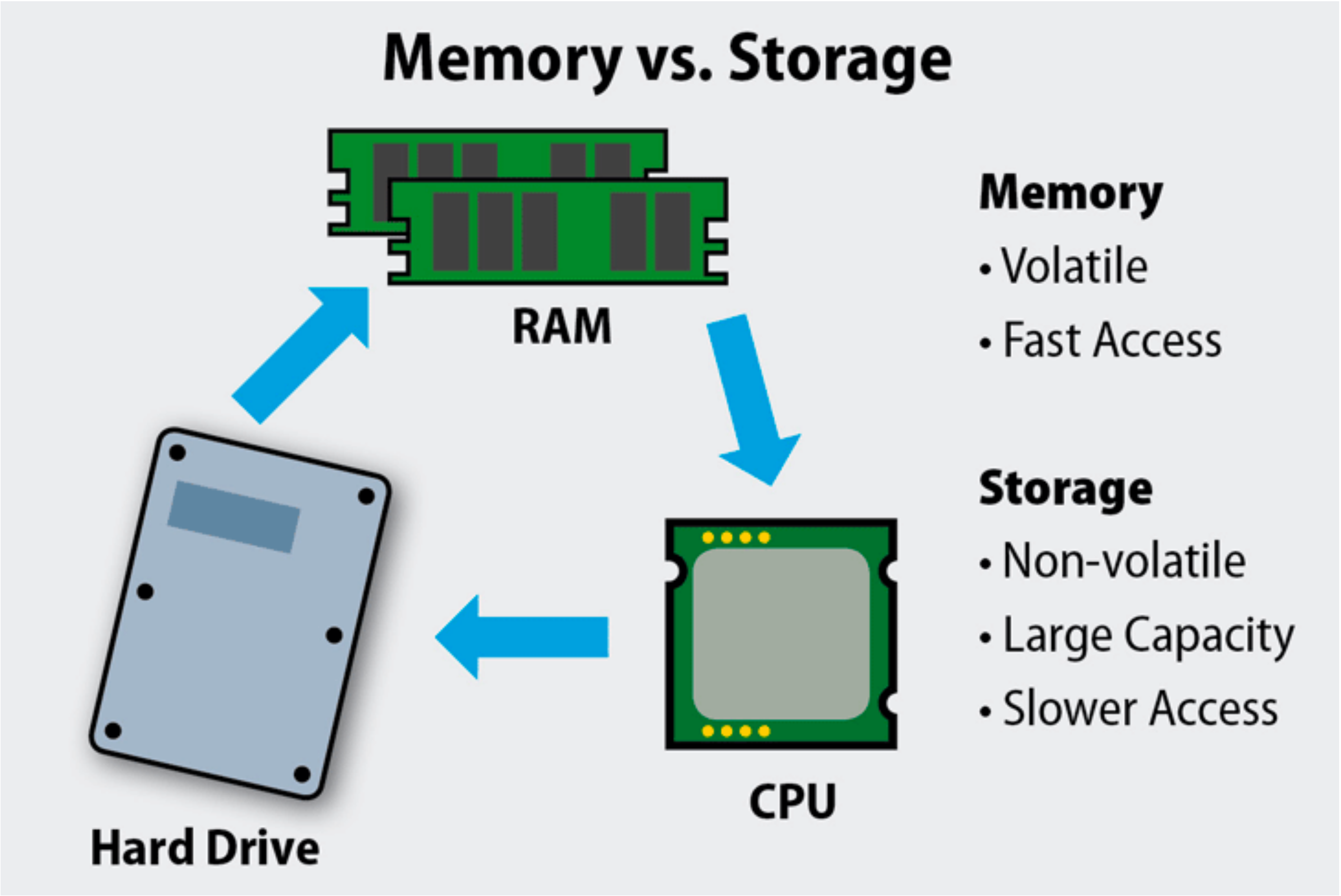
CS 423
Operating System Design: Disk Scheduling Algorithms

Tianyin Xu

Time to talk about storage



Question

- What functions should file systems provide?

Why Files?

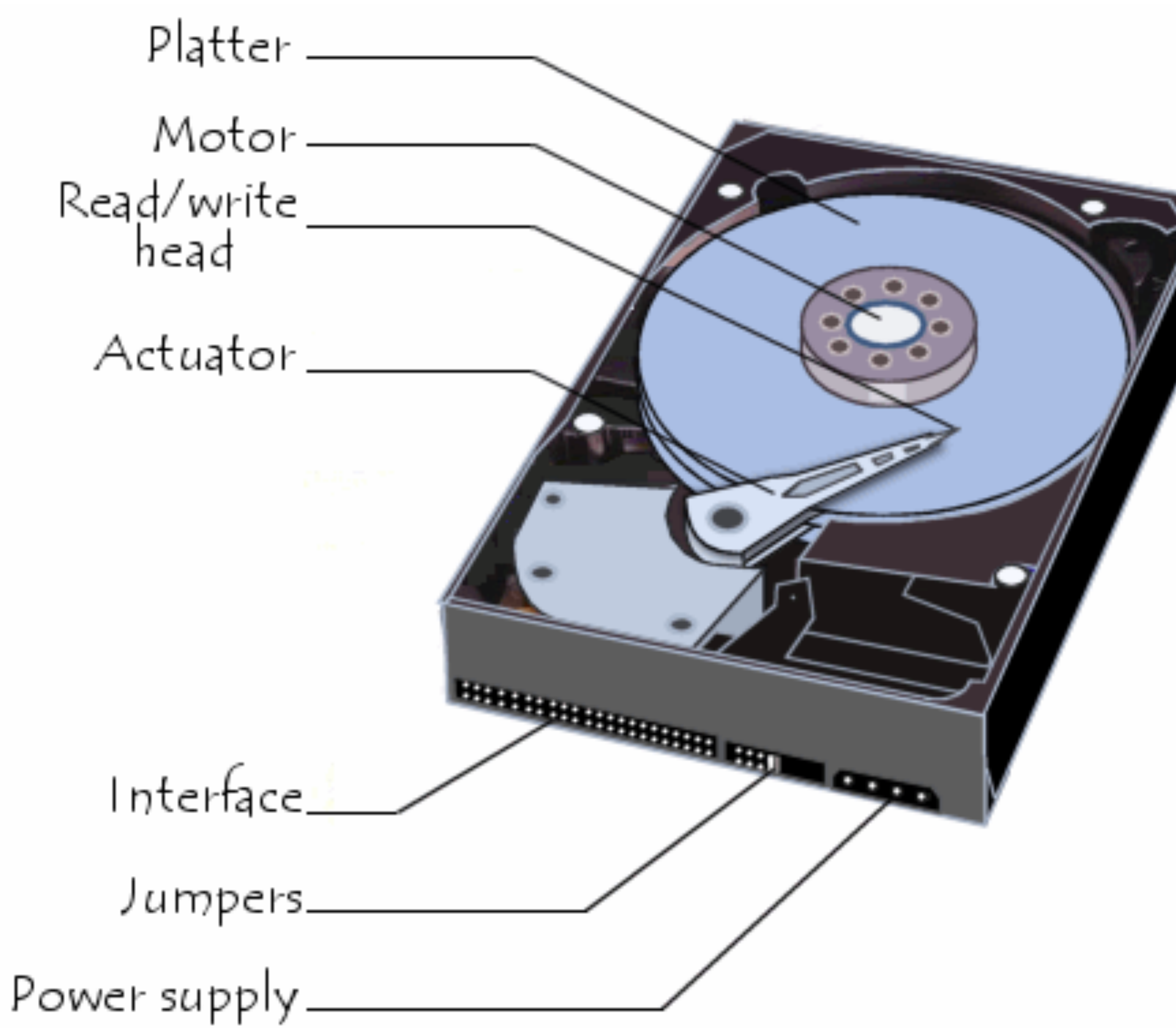
- Physical reality
  - Block oriented
  - Physical sector #s
  - No protection among users of the system
  - Data might be corrupted if machine crashes

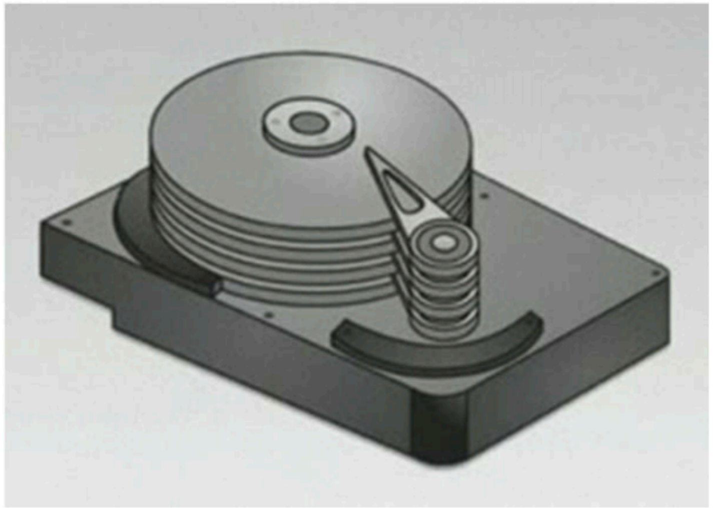
- Filesystem model
  - Byte oriented
  - Named files
  - Users protected from each other
  - Robust to machine failures

File System Requirements

- Users must be able to:
  - create and delete files at will.
  - read, write, and modify file contents with a minimum of fuss about blocking, buffering, etc.
  - share each other's files with proper authorization
  - refer to files by symbolic names.
  - see a logical view of files without concern for how they are stored.
  - retrieve backup copies of files lost through accident or malicious destruction.

Disk


Platter
Motor
Read/write head
Actuator
Interface
Jumpers
Power supply

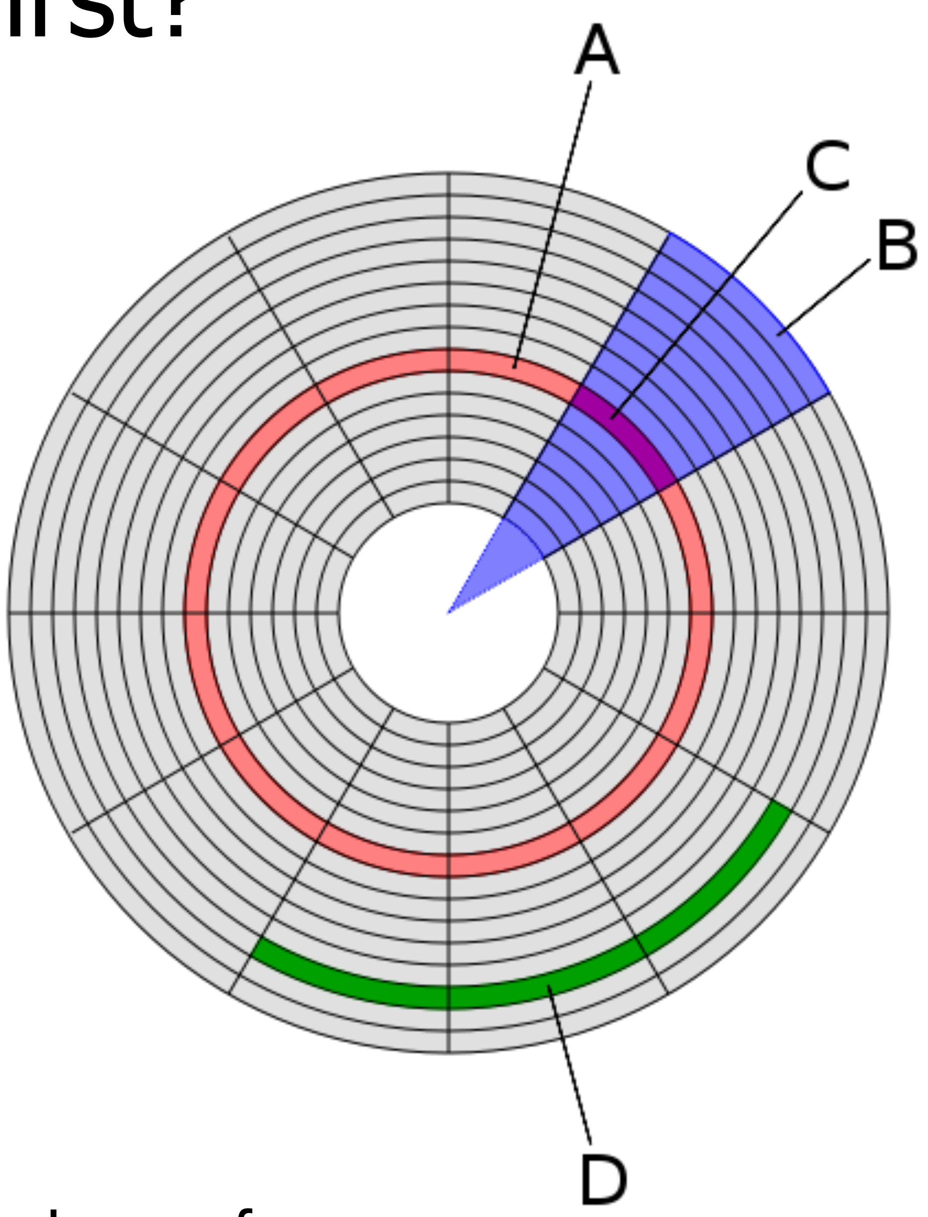


Watch

https://www.youtube.com/watch?v=NtPc0jI21i0

Disk Scheduling

- Which disk request is serviced first?
  - FCFS
  - Shortest seek time first
  - SCAN (Elevator)
  - C-SCAN (Circular SCAN)



A: Track.
B: Sector.
C: Sector of Track.
D: File

Disk Scheduling Decision — Given a series of access requests, on which track should the disk arm be placed next to maximize fairness, throughput, etc?

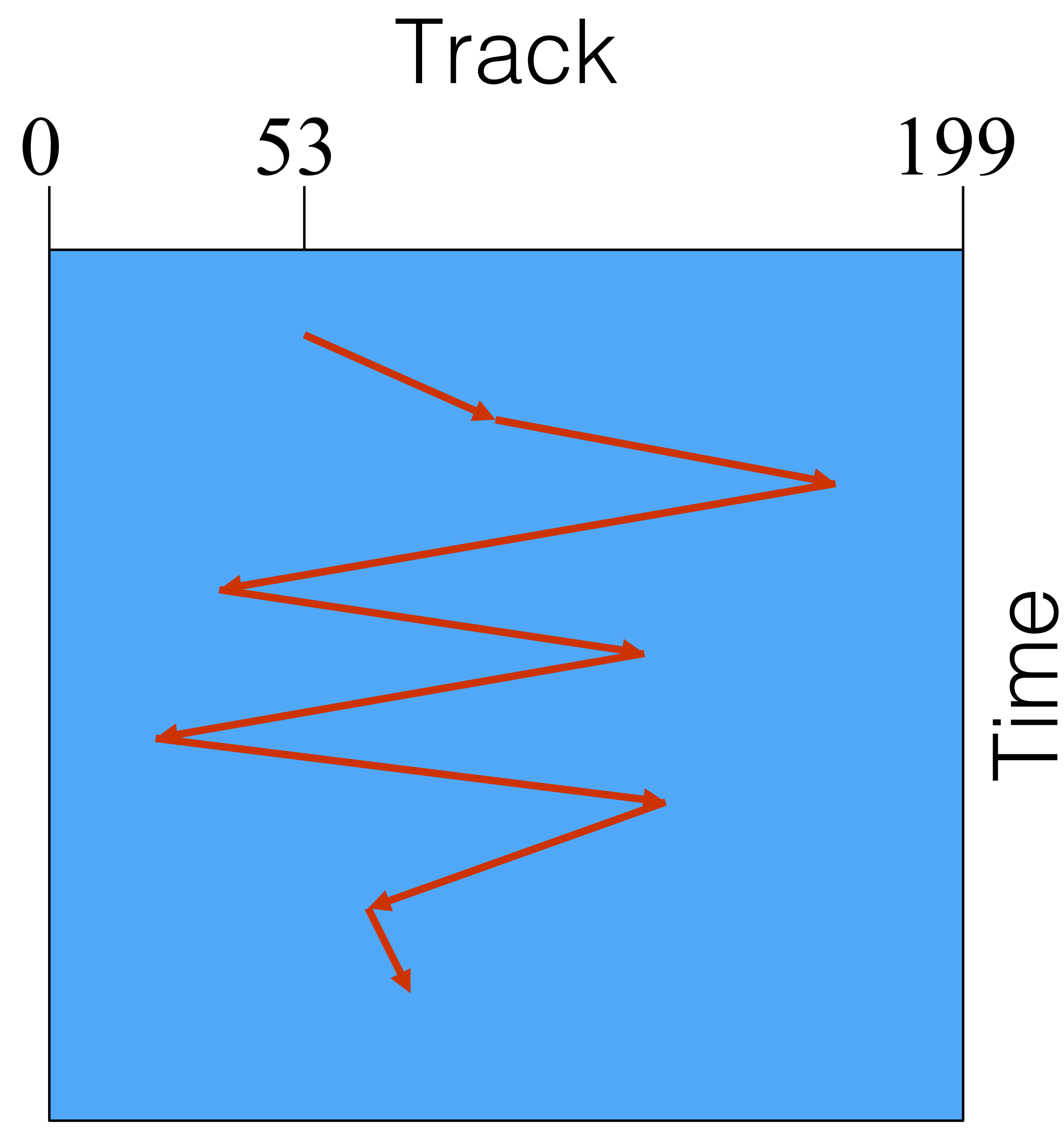
Disk Access Time Example

- Disk Parameters
  - Transfer Size is 8K bytes
  - Advertised average seek time is 12 ms
  - Disk spins at 7200 RPM
  - Transfer rate is 4 MB/sec
  - Controller Overhead is 2 ms
- Assume idle disk (i.e., no queuing delay)

```
Disk Access Time =     12 ms
                    + 0.5/(7200 RPM / 60)
                    + 8 KB / 4 MB per sec
                    + 2 ms
```

FIFO (FCFS) Order

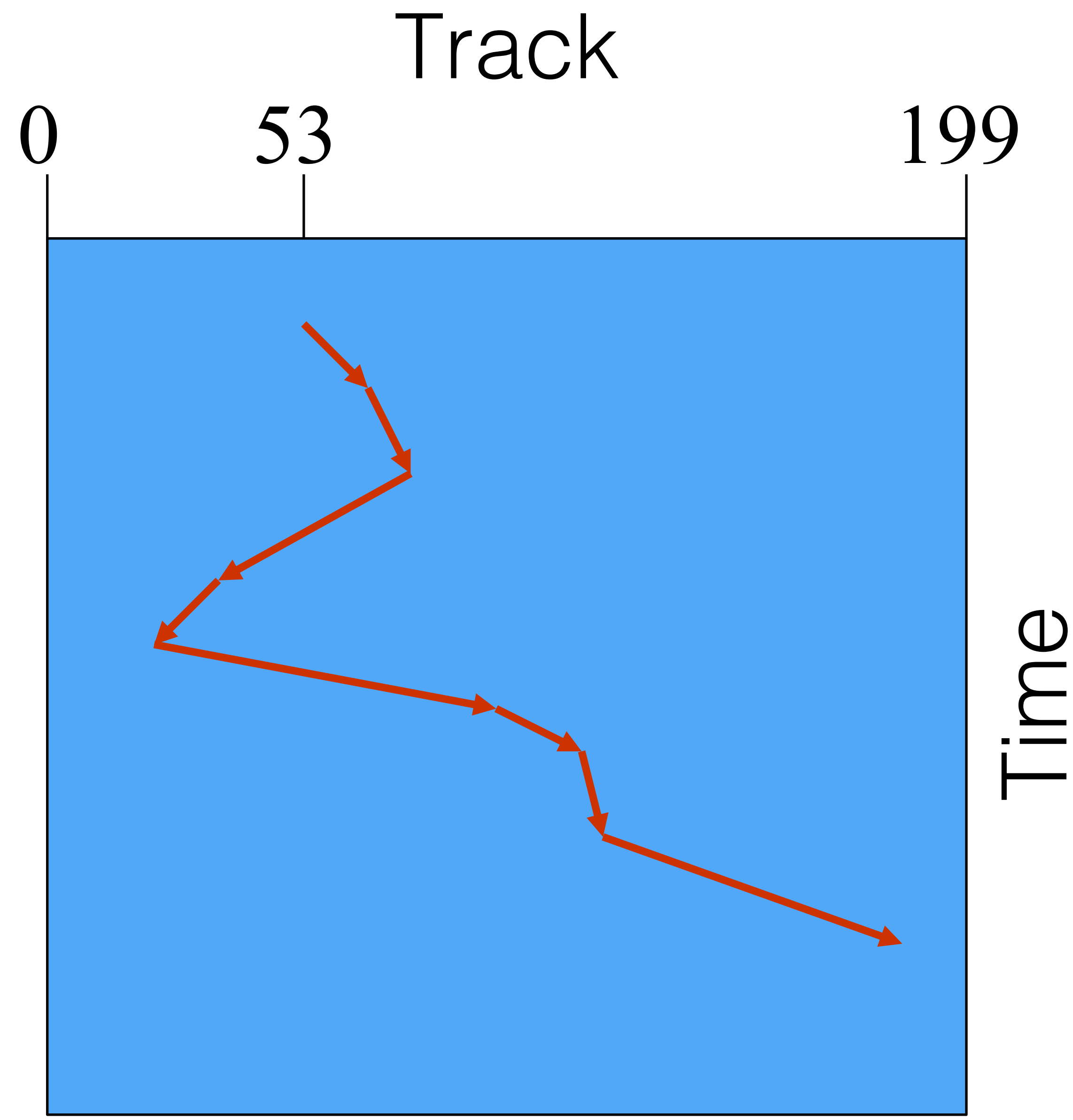
- Method
  - First come first serve
- Pros?
  - Fairness among requests
  - In the order applications expect
- Cons?
  - Arrival may be on random spots on the disk (long seeks)
  - Wild swings can happen
- Analogy:
  - FCFS elevator scheduling?



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

SSTF (Shortest Seek Time First)

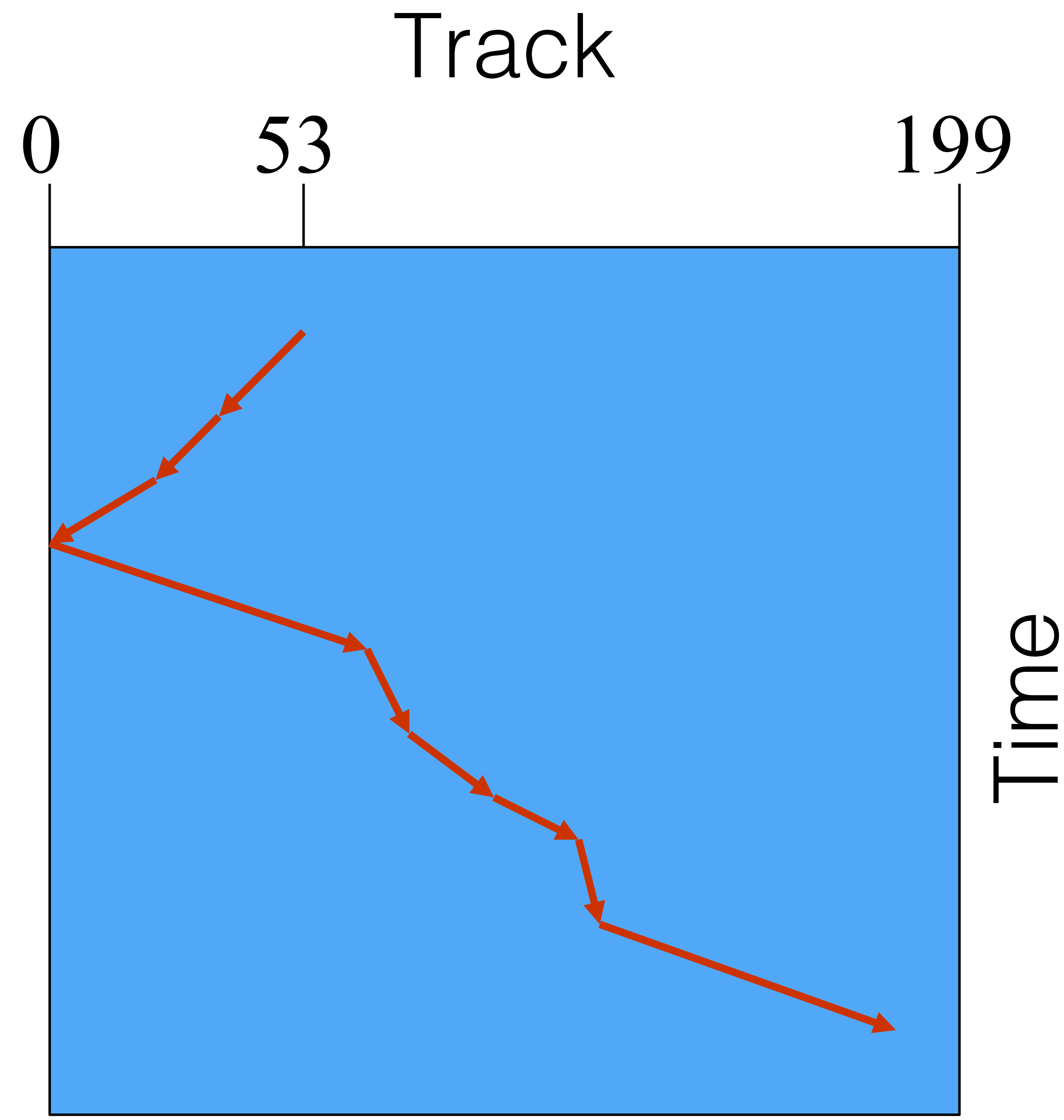
- Method
  - Pick the one closest on disk
- Pros?
  - Tries to minimize seek time
- Cons?
  - Starvation
- Questions
  - Is SSTF optimal?
  - Is this fair to all disk accesses?
  - Are we worried about sorting overhead?
  - Can we avoid starvation?



98, 183, 37, 122, 14, 124, 65, 67
(65, 67, 37, 14, 98, 122, 124, 183)

SCAN (Elevator)

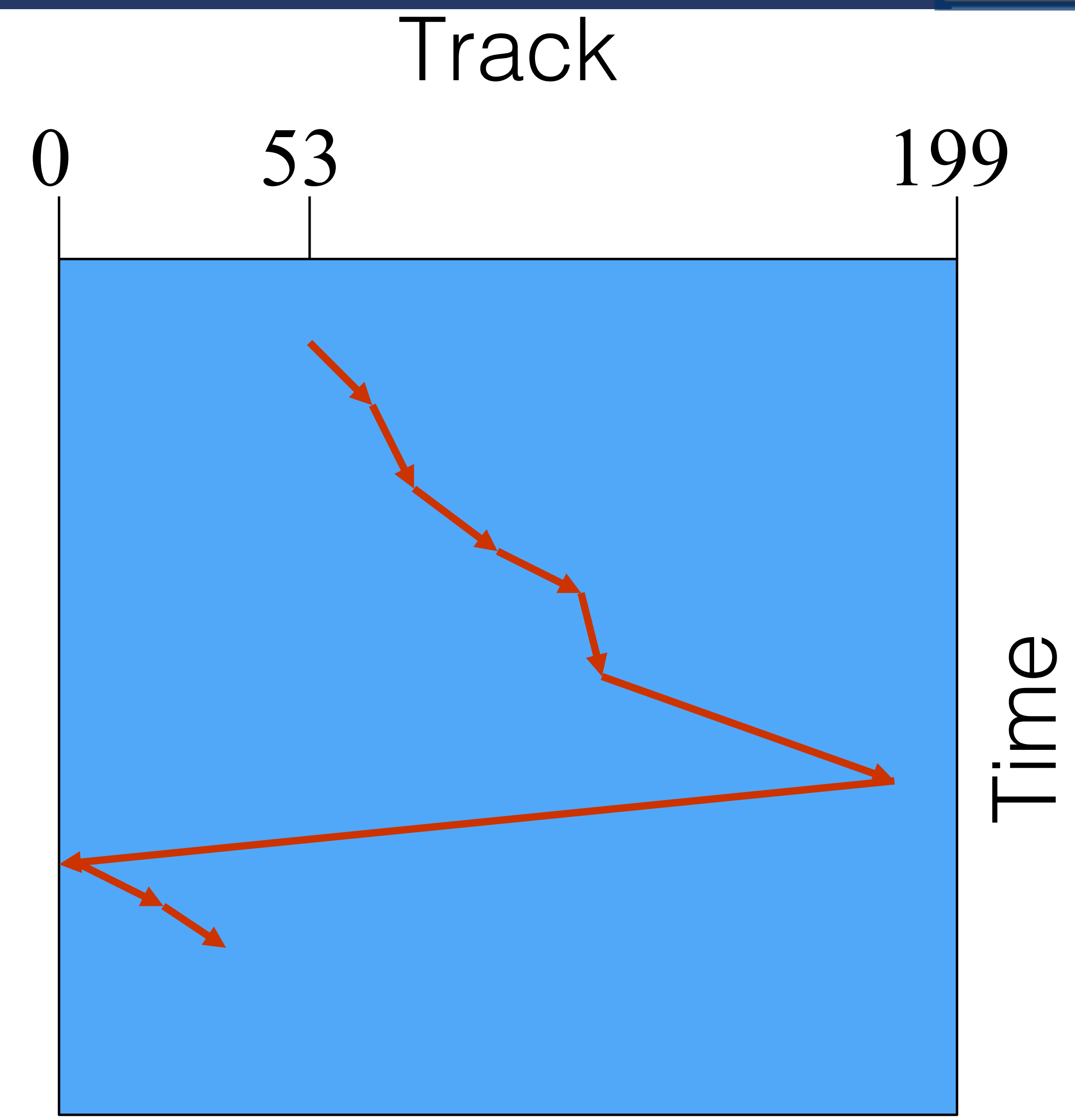
- Method
  - Take the closest request in the direction of travel
- Pros
  - Bounded time for each request
- Cons
  - Request at the other end will take a while
- Question
  - Is this fair to all disk accesses?
  - How to fix?



98, 183, 37, 122, 14, 124, 65, 67
(37, 14, 65, 67, 98, 122, 124, 183)

C-SCAN (Circular SCAN)

- Method
  - Like SCAN
  - But, wrap around
- Pros
  - Uniform service time
- Cons
  - Do nothing on the return (i.e., higher overhead)



98, 183, 37, 122, 14, 124, 65, 67
(65, 67, 98, 122, 124, 183, 14, 37)

Scheduling Algorithms

Algorithm Name	Description
FCFS	First-come first-served
SSTF	Shortest seek time first; process the request that reduces next seek time
SCAN (aka Elevator)	Move head from end to end (has a current direction)
C-SCAN	Only service requests in one direction (circular SCAN)
LOOK	Similar to SCAN, but do not go all the way to the end of the disk.
C-LOOK	Circular LOOK. Similar to C-SCAN, but do not go all the way to the end of the disk.

Disk Scheduling Performance

- What factors impact disk performance?
  - Seek Time: Time taken to move disk arm to a specified track
  - Rotational Latency: Time taken to rotate desired sector into position
  - Transfer Time: Time to read/write data. Depends on rotation speed of disk and transfer amount.

```
Disk Access Time =    Seek Time
                    + Rotational Latency
                    + Transfer Time
                   (+ Controller Latency)
```

Disk Access Time Example

- Disk Parameters
  - Transfer Size is 8K bytes
  - Advertised average seek time is 12 ms
  - Disk spins at 7200 RPM
  - Transfer rate is 4 MB/sec
  - Controller Overhead is 2 ms
- Assume idle disk (i.e., no queuing delay)

```
Disk Access Time =     12 ms
                   + 0.5/(7200 RPM / 60)
                   + 8 KB / 4 MB per sec
                   + 2 ms
```

Linux I/O Schedulers

- What disk (I/O) schedulers are available in Linux?

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

^ scheduler enabled on our VMs

- As of Linux 2.6.10, it is possible to change the IO scheduler for a given block device on the fly!
- How to enable a specific scheduler?

```
$ echo SCHEDNAME > /sys/block/DEV/queue/scheduler
```

  - SCHEDNAME = Desired I/O scheduler
  - DEV = device name (e.g., sda)

Linux NOOP Scheduler

- Insert all incoming I/O requests into a simple FIFO
- Merges duplicate requests (results can be cached)
- When would this be useful?

Linux NOOP Scheduler

- Insert all incoming I/O requests into a simple FIFO
- Merges duplicate requests (results can be cached)
- When would this be useful?
  - Solid State Drives! Avoids scheduling overhead
  - Scheduling is handled at a lower layer of the I/O stack (e.g., RAID Controller, Network-Attached)
  - Host doesn’t actually know details of sector positions (e.g., RAID controller)

Linux Deadline Scheduler

- Imposes a deadline on all I/O operations to prevent starvation of requests
- Maintains 4 queues:
  - 2 Sorted Queues (R, W), order by Sector
  - 2 Deadline Queues (R, W), order by Exp Time
- Scheduling Decision:
  - Check if 1st request in deadline queue has expired.
  - Otherwise, serve request(s) from Sorted Queue.
  - Prioritizes reads (DL=500ms) over writes (DL=5s) .Why?

Linux CFQ Scheduler

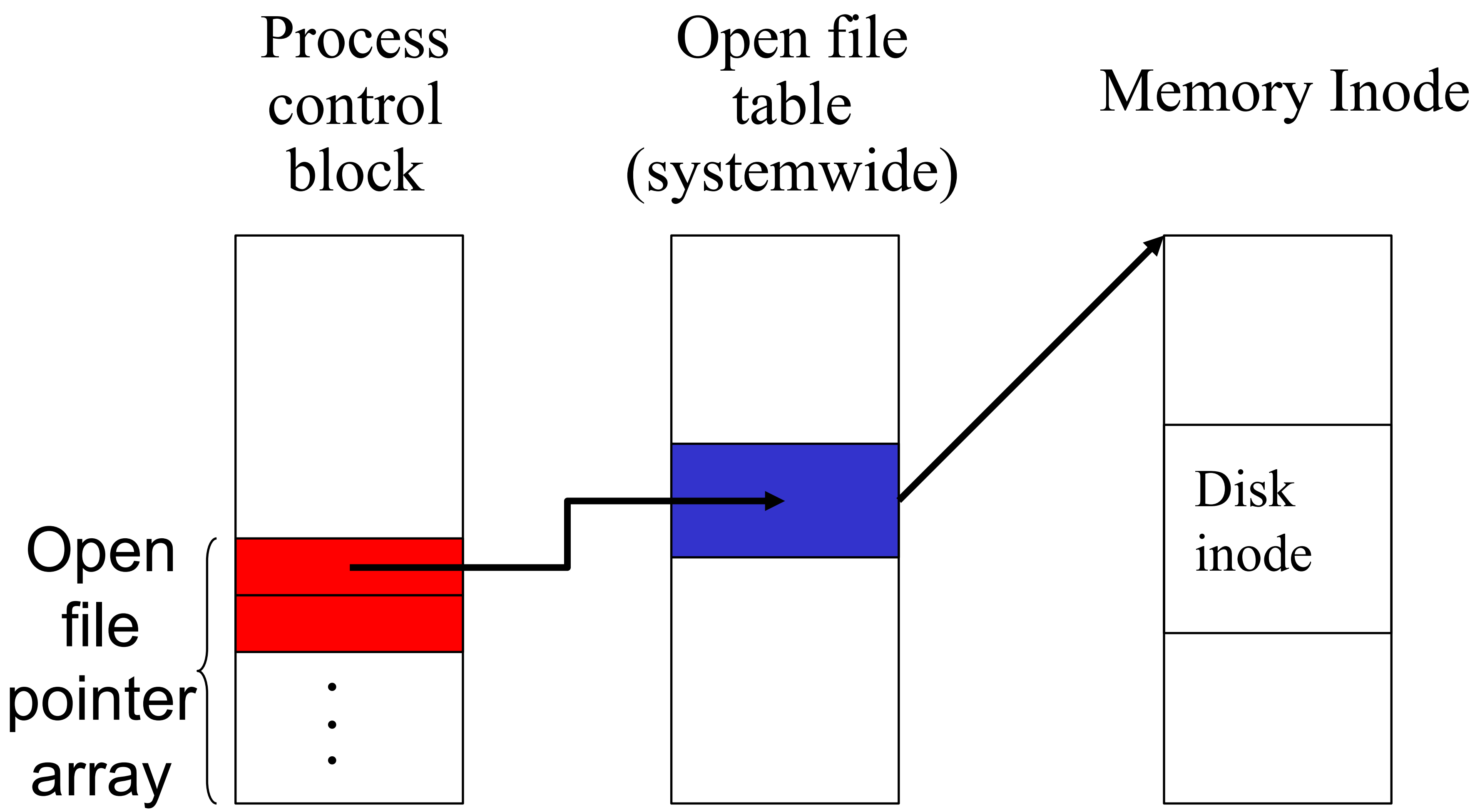
- CFQ = Completely Fair Queueing!
- Maintain per-process queues.
- Allocate time slices for each queue to access the disk
- <u>I/O Priority</u> dictates time slice, # requests per queue
- Asynchronous requests handled separately — batched together in priority queues
- CFQ is often the default scheduler

Linux Anticipatory Scheduler

- <u>Deceptive Idleness</u>: A process appears to be finished reading from disk, but is actually processing data. Another (nearby) request is coming soon!
- Bad for synchronous read workloads because seek time is increased.
- <u>Anticipatory Scheduling</u>: Idle for a few milliseconds after a read operation in *anticipation* of another close-by read request.
- Deprecated — CFQ can approximate.

Data Structures for a FS

Data structures in a typical file system:



Disk Layout for a FS

Disk layout in a typical file system:

Boot block	Super block	File metadata (i-node in Unix)	File data blocks

- Data Structures:
  - File data blocks: File contents
  - File metadata: How to find file data blocks
  - Directories: File names pointing to file metadata
  - Free map: List of free disk blocks

Disk Layout for a FS

Disk layout in a typical file system:

Boot block	Super block	File metadata (i-node in Unix)	File data blocks

- Superblock defines a file system
  - size of the file system
  - size of the file descriptor area
  - free list pointer, or pointer to bitmap
  - location of the file descriptor of the root directory
  - other meta-data such as permission and various times
- For reliability, replicate the superblock

Design Constraints

- How can we allocate files efficiently?
  - For small files:
    - Small blocks for storage efficiency
    - Files used together should be stored together
  - For large files:
    - Contiguous allocation for sequential access
    - Efficient lookup for random access
  - Challenge: May not know at file creation where our file will be small or large!!

Design Challenges

- Index structure
  - *How do we locate the blocks of a file?*
- Index granularity
  - *How much data per each index (i.e., block size)?*
- Free space
  - *How do we find unused blocks on disk?*
- Locality
  - *How do we preserve spatial locality?*
- Reliability
  - *What if machine crashes in middle of a file system op?*

File Allocation

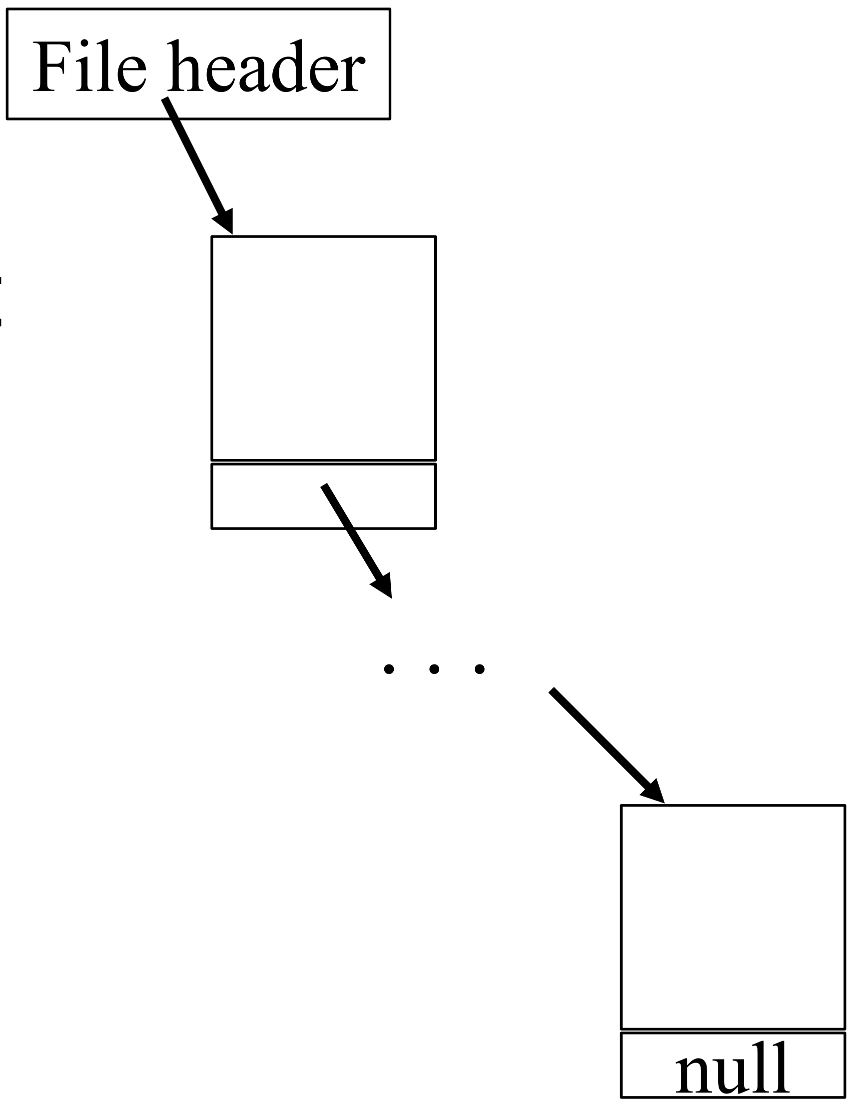
- Contiguous
- Non-contiguous (linked)
- Tradeoffs?

Contiguous Allocation

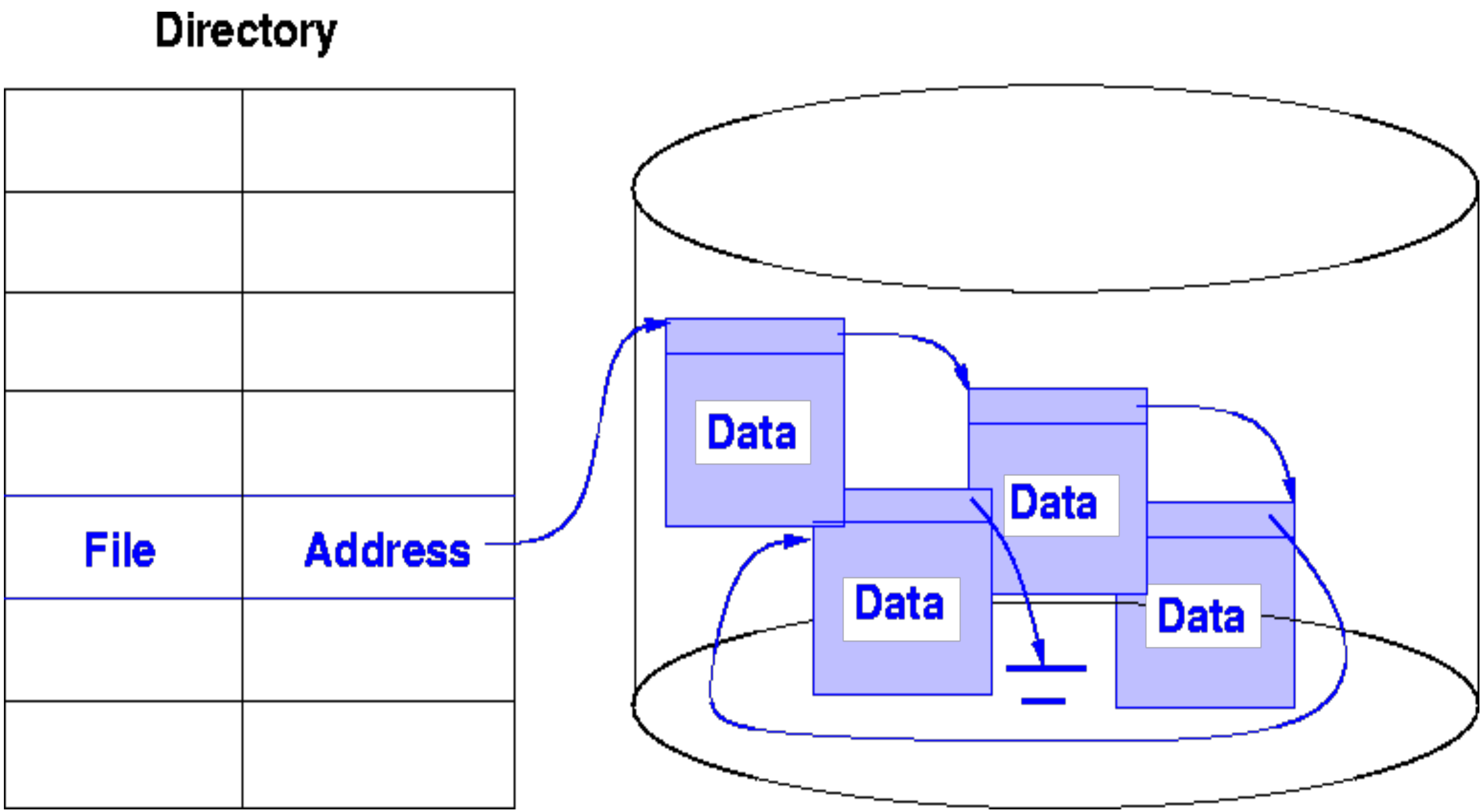
- Request in advance for the size of the file
- Search bit map or linked list to locate a space
- File header
  - first sector in file
  - number of sectors
- Pros
  - Fast sequential access
  - Easy random access
- Cons
  - External fragmentation
  - Hard to grow files

Linked Files

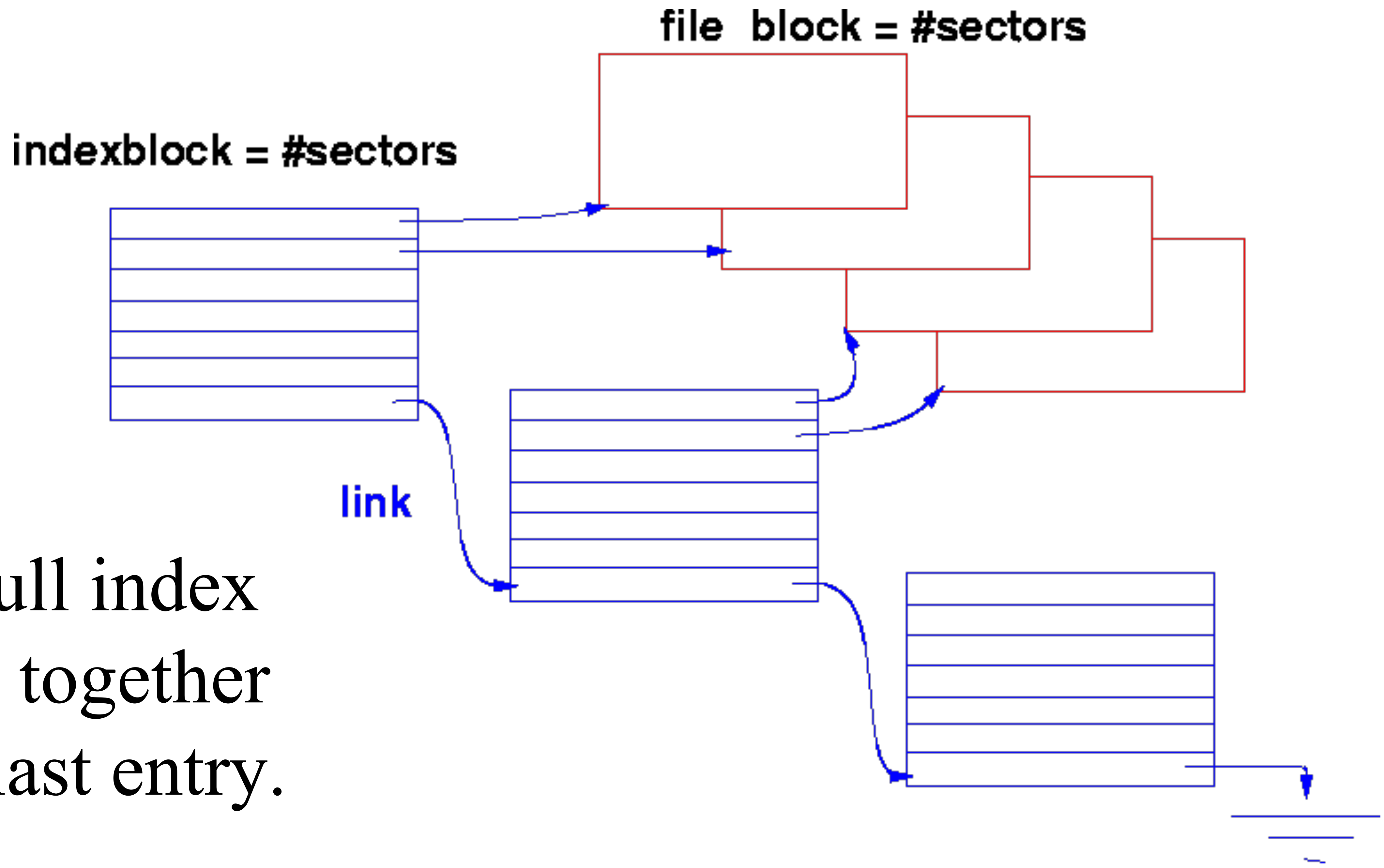
- File header points to 1st block on disk
- Each block points to next
- Pros
  - Can grow files dynamically
  - Free list is similar to a file
- Cons
  - random access: horrible
  - unreliable: losing a block means losing the rest



Linked Allocation

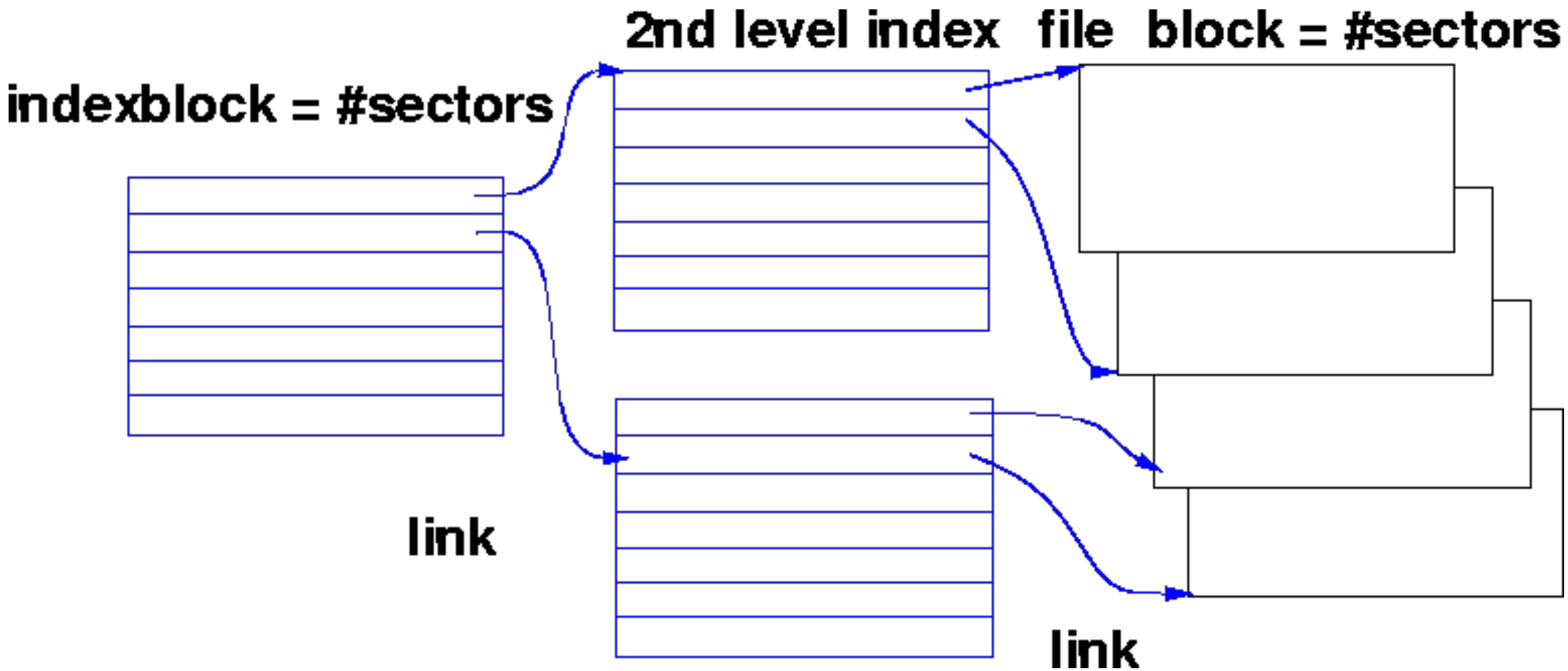


Indexed File Allocation



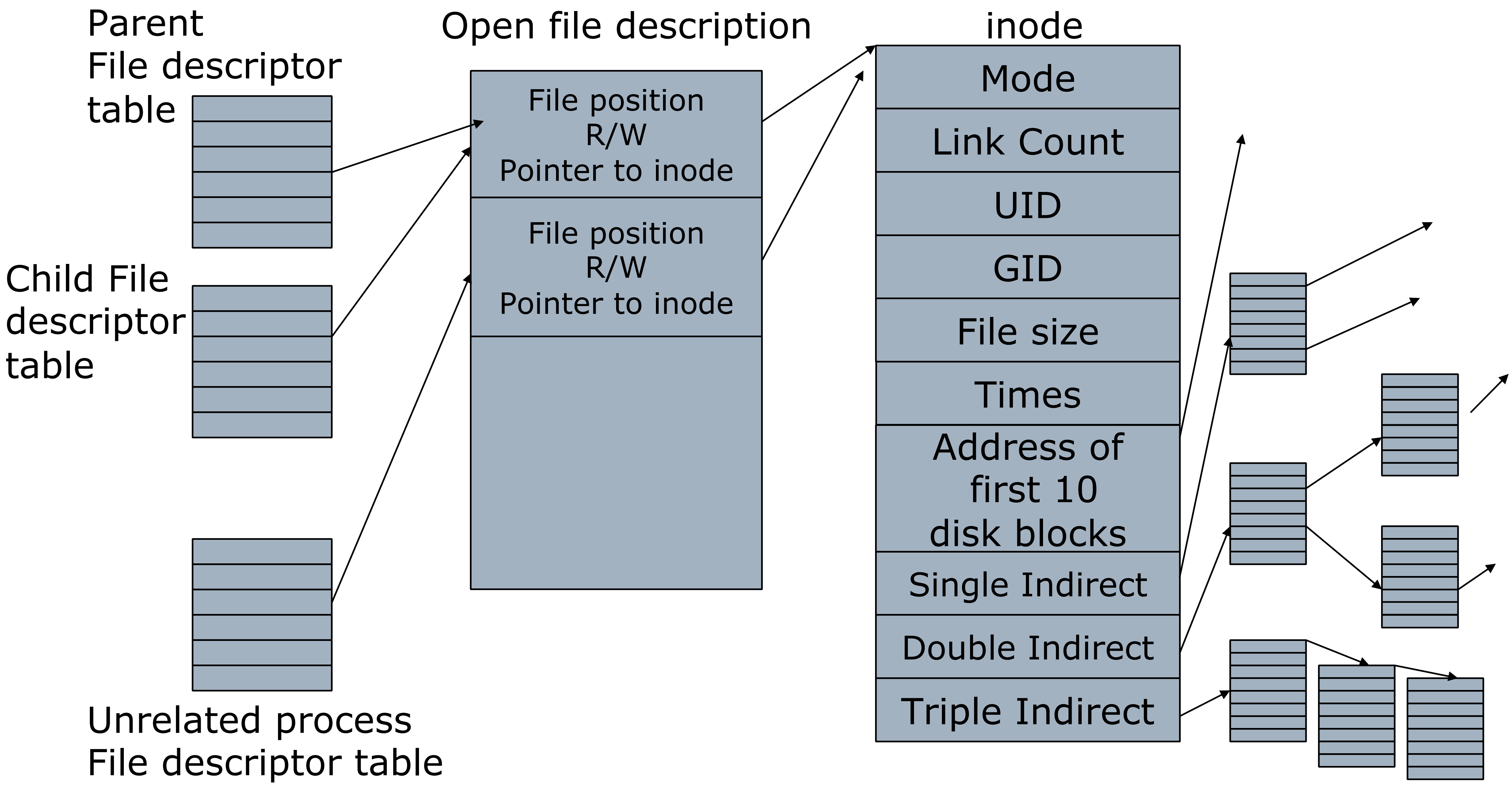
Link full index blocks together using last entry.

Multilevel Indexed Files



Multiple levels of index blocks

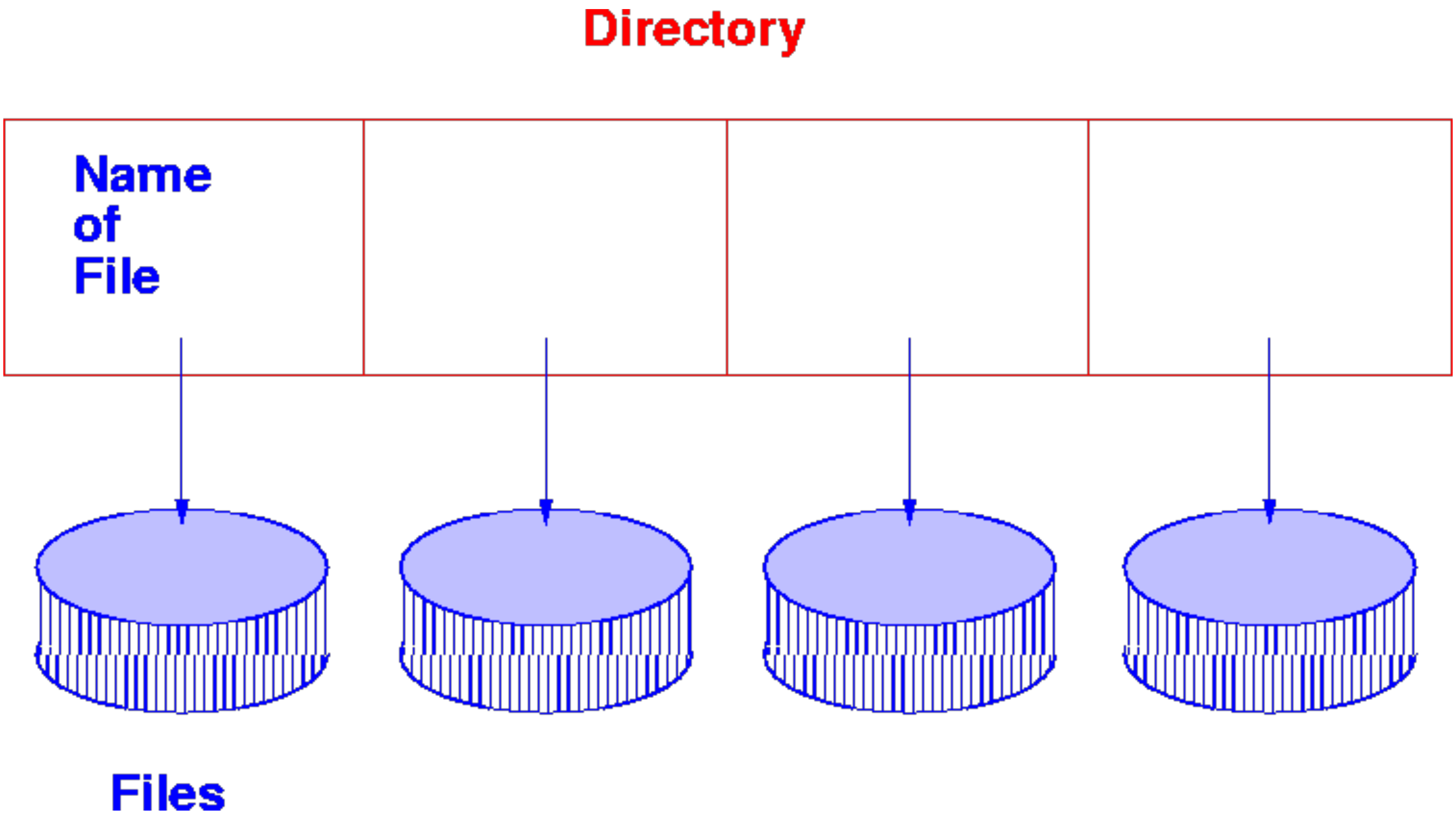
UNIX FS Implementation



Directory Structure Org.

- maps symbolic names into logical file names
  - search
  - create file
  - list directory
  - backup, archival, file migration

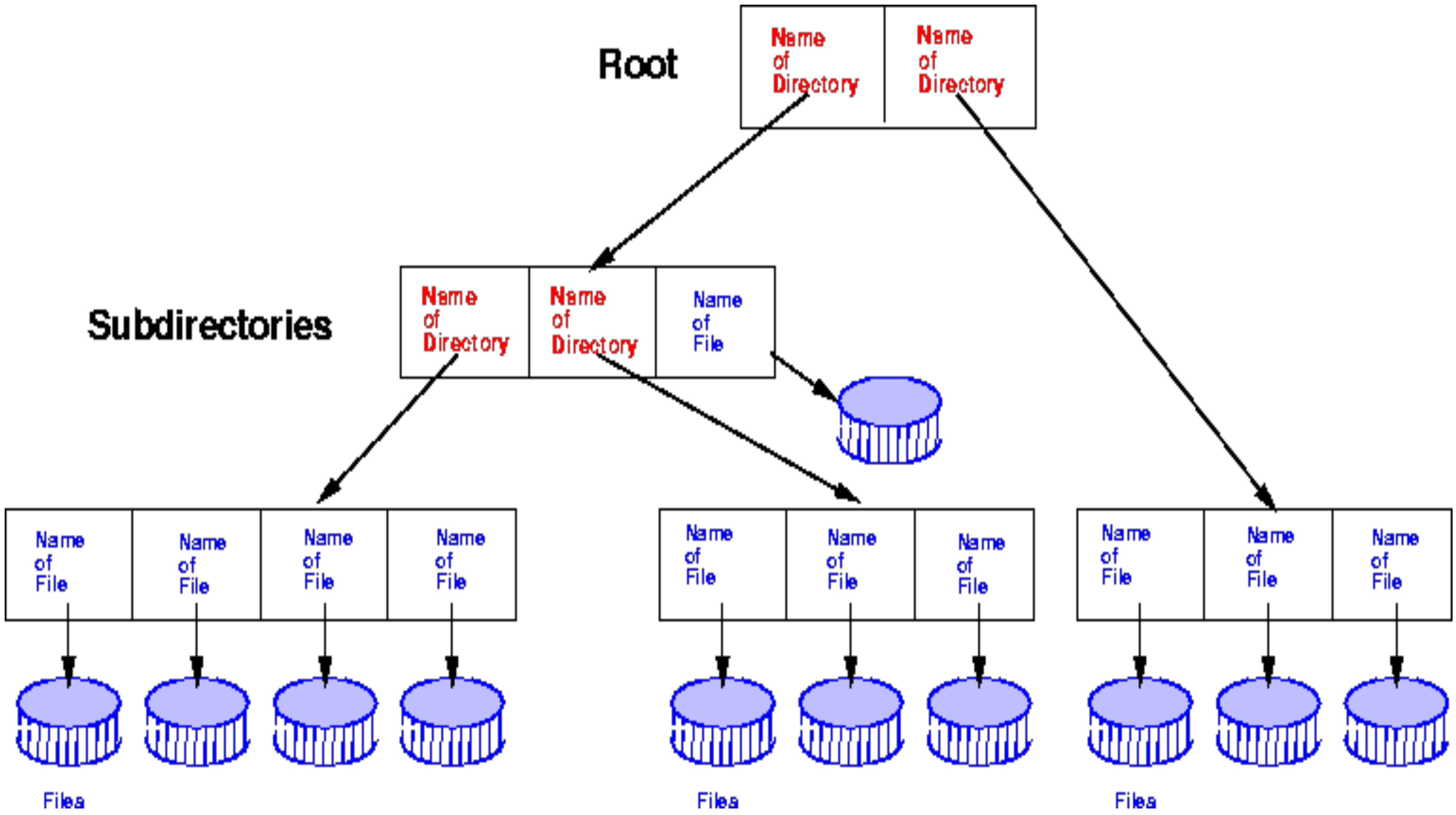
Single-level Directory



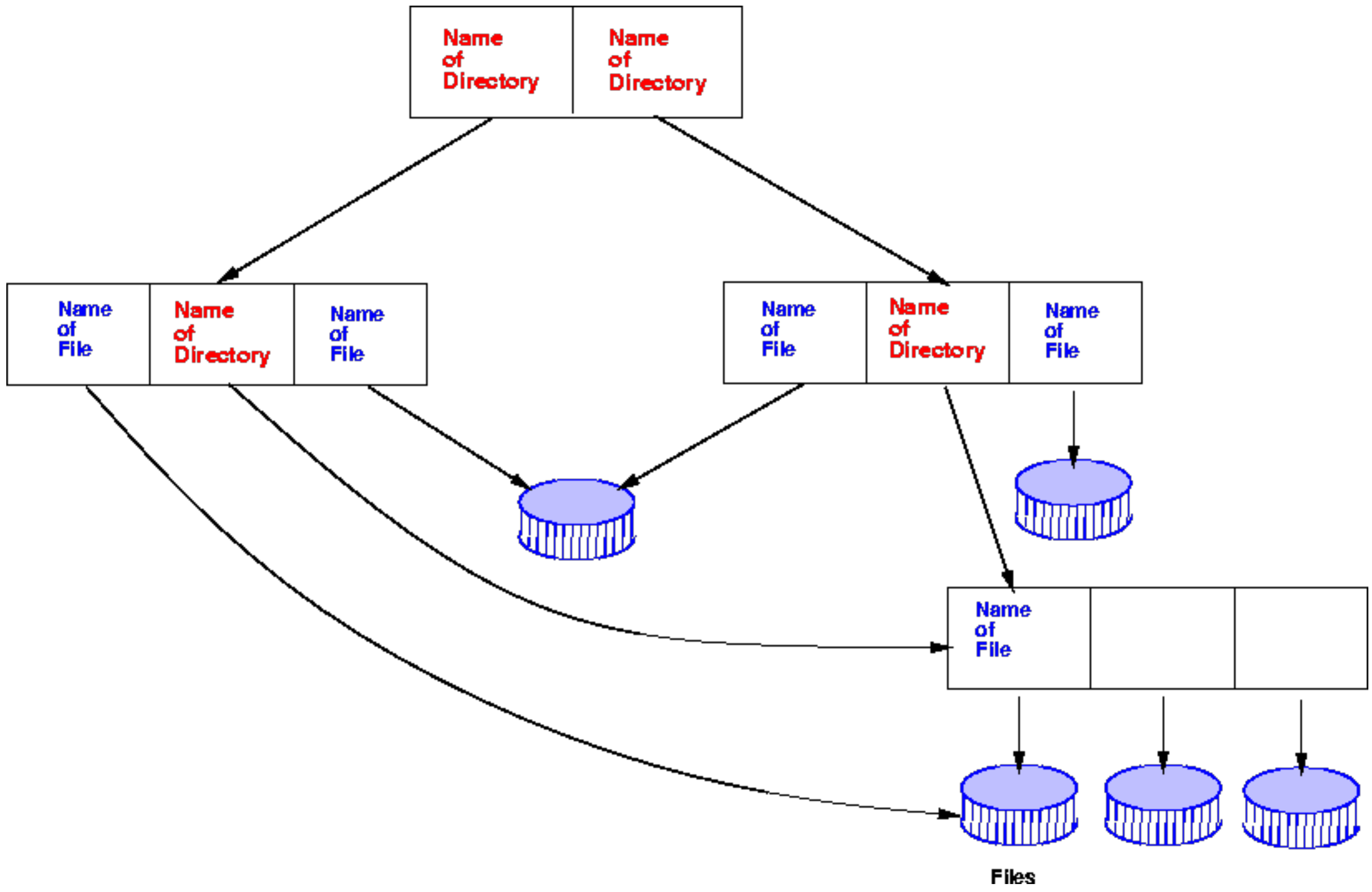
Tree-Structured Directories

- arbitrary depth of directories
- leaf nodes are files
- interior nodes are directories
- path name lists nodes to traverse to find node
- use absolute paths from root
- use relative paths from current working directory pointer

Tree-Structured Directories



Acyclic Graph Structured Dir.'s



Symbolic Links

- **Symbolic** links are different than regular links (often called **hard links**). Created with **ln -s**
- Can be thought of as a directory entry that points to the name of another file.
- Does not change link count for file
  - When original deleted, symbolic link remains
- They exist because:
  - Hard links don't work across file systems
  - Hard links only work for regular files, not directories