File Systems, Continued

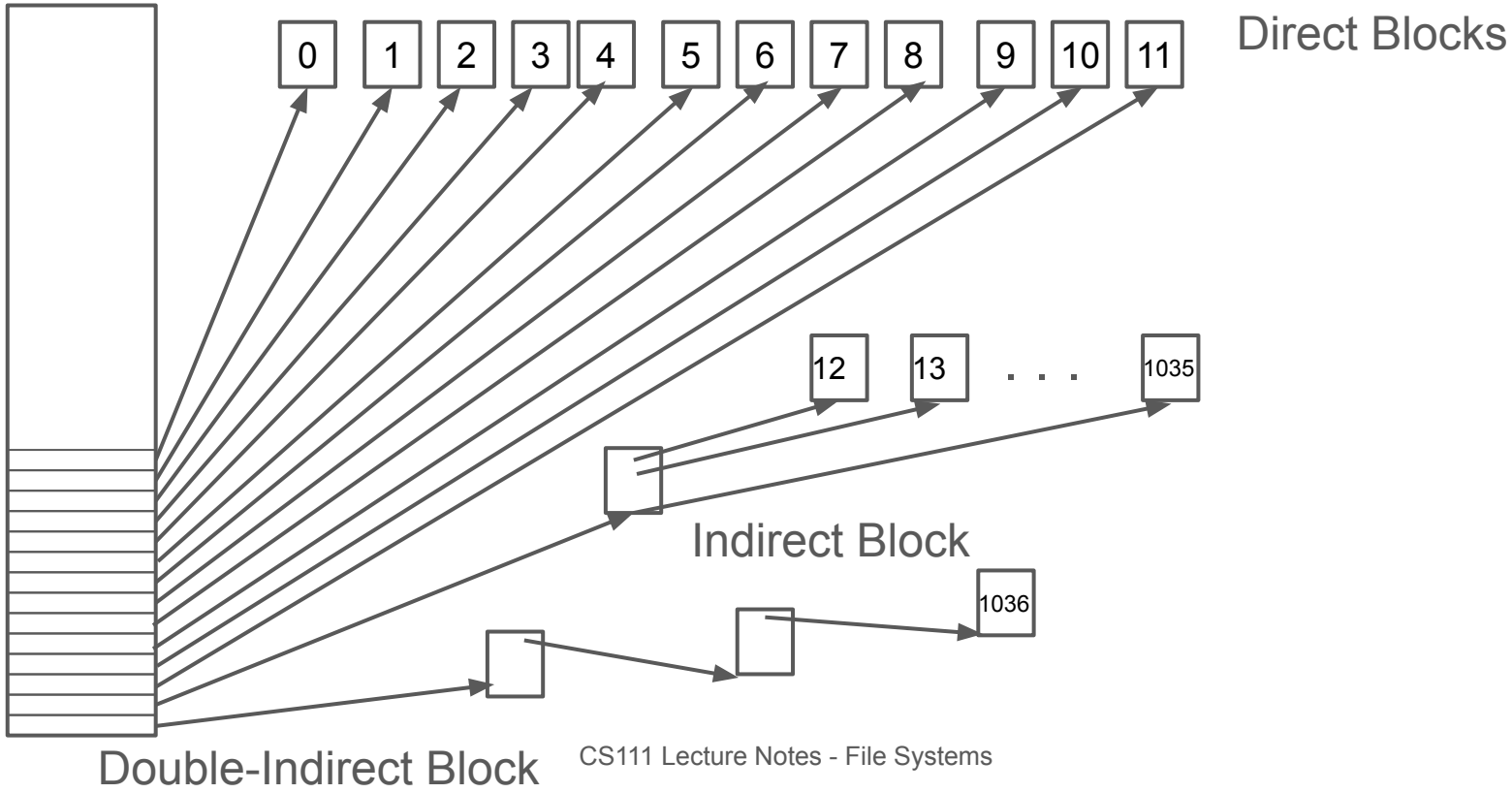
Mendel Rosenblum

Multi-level indexes (4.3 BSD Unix)

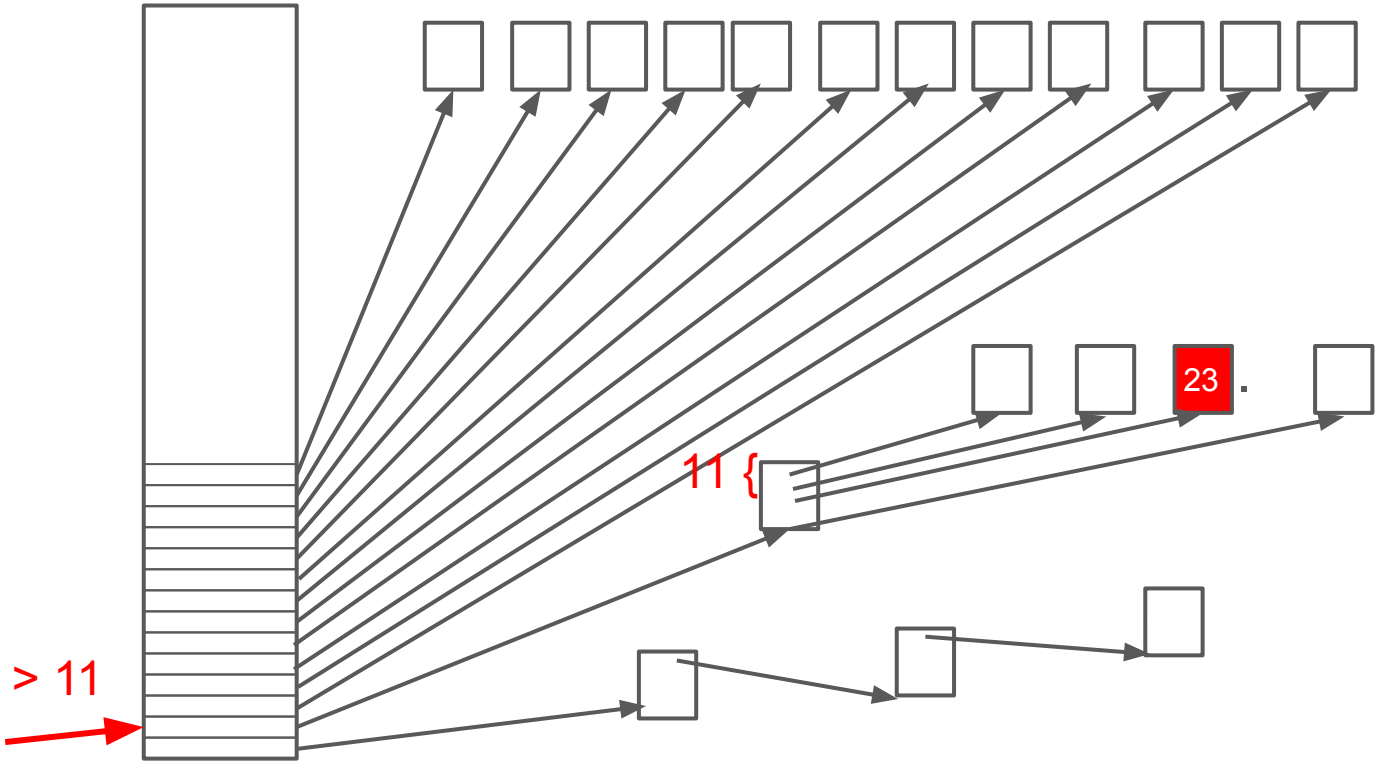
- Disk divided into 4 Kbytes blocks; Files divided into 4 Kbytes blocks
- Inode is the root node of a tree block pointers (similar to page tables)
  - Inode has 14 block pointers (0 value means no block)
- First 12 block pointers are **direct blocks**: point to first 12 blocks of the file
- For files more than 12 blocks, 13th block pointer points to an **indirect block**
  - Indirect blocks contain 1024 4-byte block pointers: point to the next 1024 blocks of the file
- For files more than 1036 blocks, 14th block points is a **doubly-indirect block**
  - Doubly-indirect blocks point to 1024 indirect block
- Maximum file length is fixed, but large. Indirect blocks only allocated if needed.

4GB (4TB with triple)

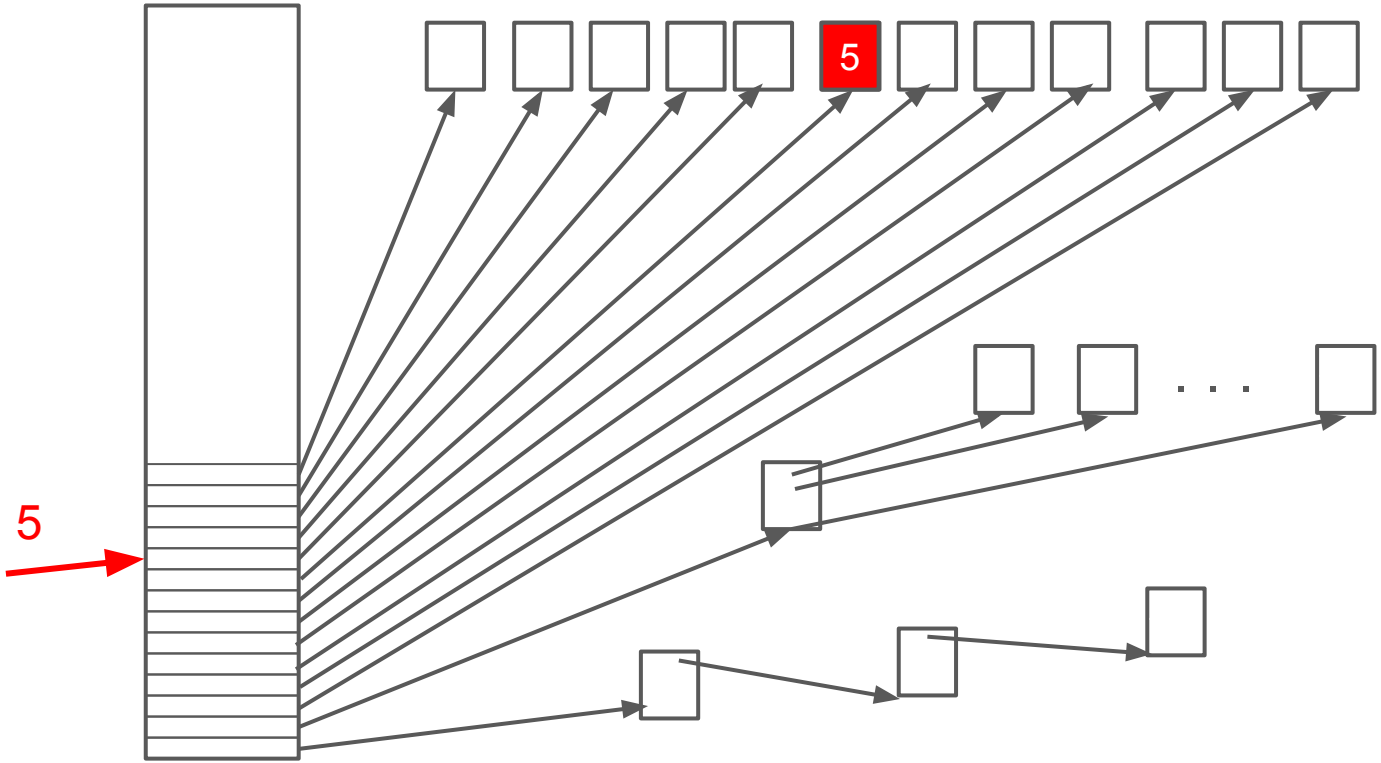
BSD Inode Example: Growing a file



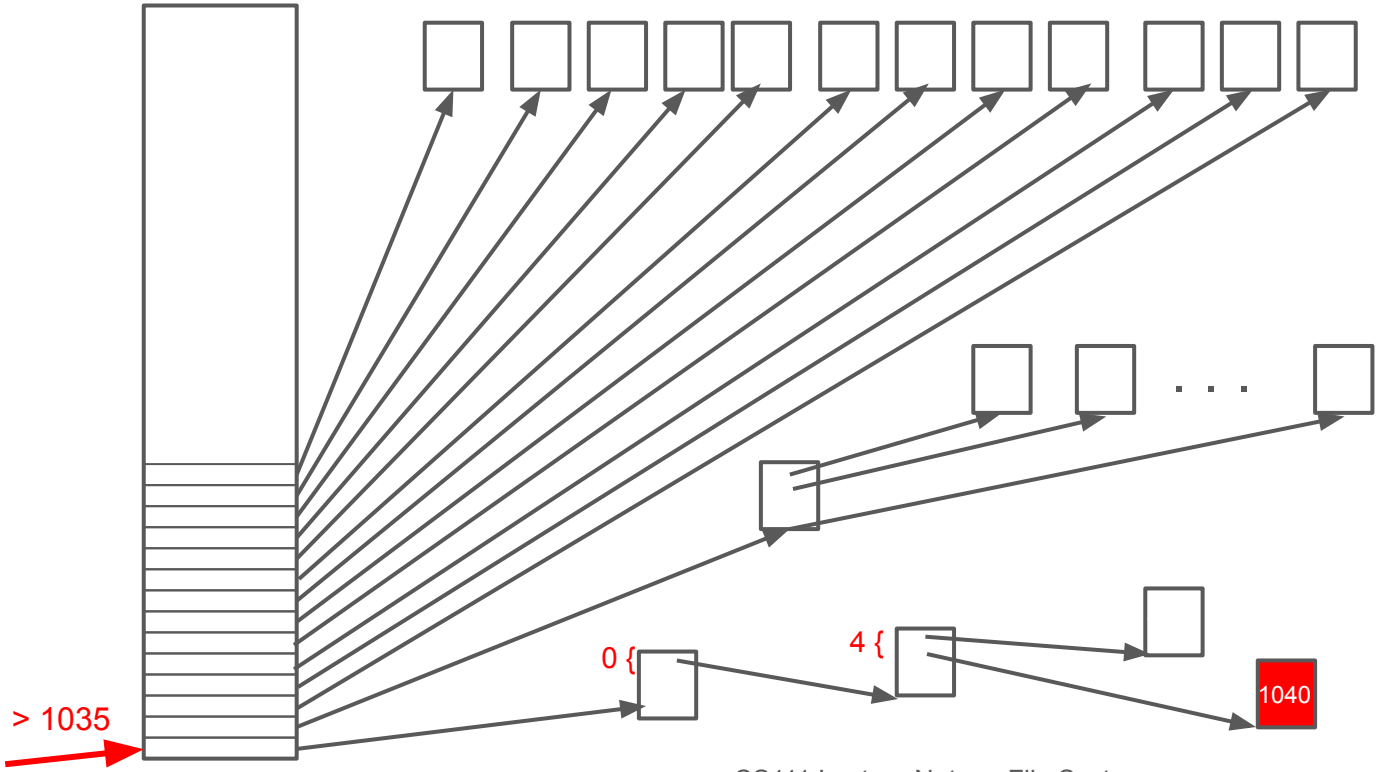
Example: Read Block 23



Example: Read Block 5



Example: Read Block 1040



BSD Inode evaluation

- Advantages
  - Simple, easy to implement (e.g Assignment 7)
  - No need to pre-declare file sizes
  - Indirect blocks and double-indirect blocks allocated only needed
  - Fast access for small files
  - Less memory than FAT
- Drawbacks
  - Indirect mechanism doesn't provide very efficient access to large files
    - Up to 2 index reads for each real operation (double-fault problem)
  - Block-by-block organization of free list like FAT: poor locality

Block Cache

- Use part of main memory to retain recently-accessed disk blocks
  - Blocks that are referenced frequently are usually in the cache
    - e.g indirect blocks for large files
      - This solves the problem of slow access to large files
    - Blocks that have inodes
  - LRU replacement
- Modern systems have huge block caches
  - OS controls memory given to both processes (virtual memory) and block cache

What happens when a block in the cache is modified?

- **Synchronous writes** - immediately write through to disk
  - Safe: data won't be lost if the machine crashes
  - Slow: process can't continue until disk I/O completes

- **Delayed writes** don't immediately write to disk
  - Wait a while (30 seconds?) in case there are more writes to a block or the block is deleted
  - Fast: writes return immediately
  - Eliminates disk I/Os in many cases:
    - Many small writes to the same block
      - An indirect block during file writes
      - Some files are deleted quickly (e.g., temporary files)
  - Dangerous: may lose data after a system crash

Free Space Management

- Early Unix systems just used a linked list of free blocks
  - At the beginning, free list is sorted, so blocks in a file are allocated contiguously
  - Free list quickly becomes scrambled, so files are spread all over disk.
- Most common approach: **free map** (bitmap):
  - Keep an array of bits, one per block.
  - 1 means block is free, 0 means block in use
  - 1 TB capacity => 2**28 4 KByte blocks => 32 MByte free map
- Allocation: Can look for a "good" block
  - Search free map: a block that's close to the file's previous block is good
  - If disk isn't full, this usually works pretty well.

Issues with using bitmaps for free map

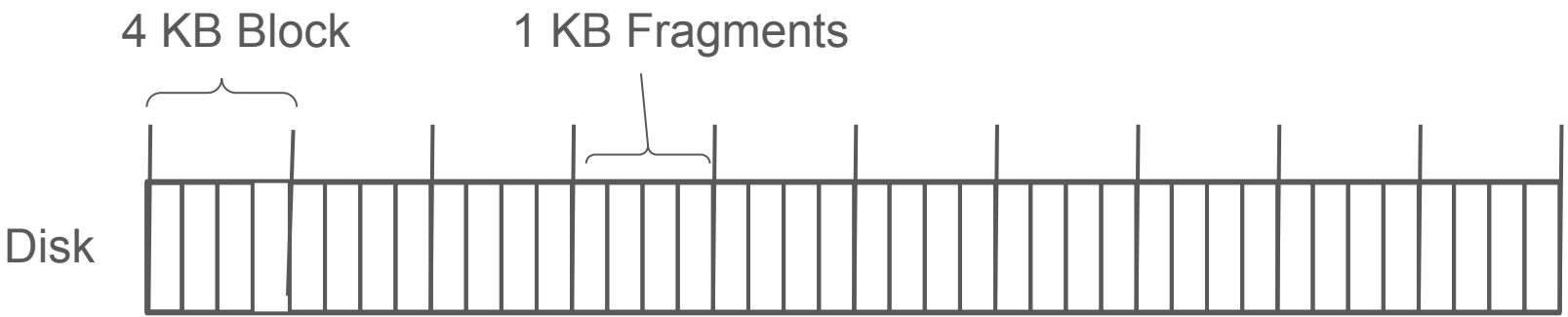
- If disk isn't full, bitmaps usually works pretty well
  - Almost always can find a "good" free block
- If disk is nearly full this becomes a problem:
  - Expensive to find free blocks
  - Doesn't produce much locality
- Solution: don't let the disk fill up!
  - Pretend disk has 10% less capacity than it really has
  - If disk is 90% full, tell users it's full and don't allow any more data to be written.

Block Sizes

- Many early file systems (e.g. Unix) used a block size of 512 bytes
  - The size of a disk sector for many years
  - Inefficient I/O: more distinct transfers, hence more seeks.
  - Bulkier metadata: only 128 pointers in an indirect block
  - Pointers will occupy 1% of disk space
- Increase block size (e.g. 4 KB)?
  - Improves efficiency of I/O and inodes
  - Worsens internal fragmentation: most files are small,
  - Can't share blocks between files. 4KB blocks would waste almost half of the disk space

4.3BSD solution: multiple block sizes

- Large blocks are 4 KBytes; most blocks are large
- **Fragments** are multiples of 512 bytes, fitting within a single large block
  - Only the last block in a file can be a fragment
  - One large block can hold fragments from multiple files
- Bit map for free blocks is based on fragments



File System Challenges

- Don't know the size of files
- File grow incrementally - one block at a time
- Want to allocate big chunks of block (extents)
  - Small files will fragment the free space.
  - Hard for large and small files to coexist!

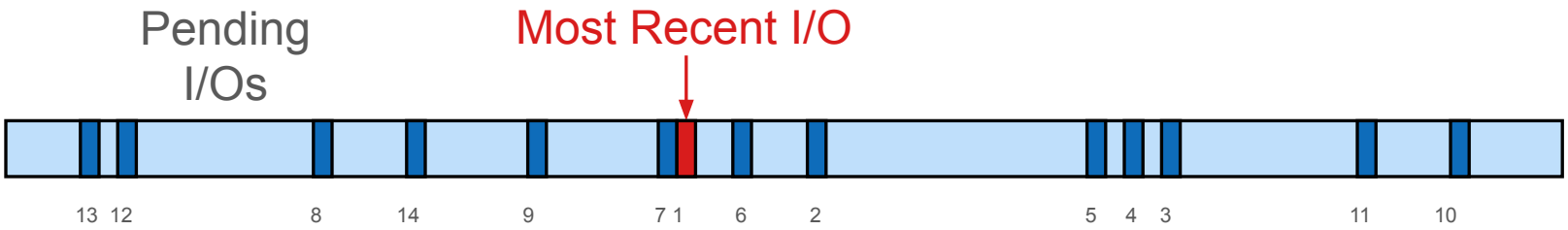
Some newer techniques

- Use even larger block sizes: 16 KB large blocks, 2K fragments (or extents)
- Initially, allocate blocks one at a time (but probably won't be consecutive)
  - When a file reaches a certain size, reallocate blocks looking for large contiguous clusters
- Delay space allocation until flushing blocks from cache:
  - By then, many more blocks will have been written
  - Allocate cluster(s) for all known blocks
- Improves efficiency of I/O and inodes.
- Disk space is so cheap that the internal fragmentation doesn't matter much today.

Disk Scheduling

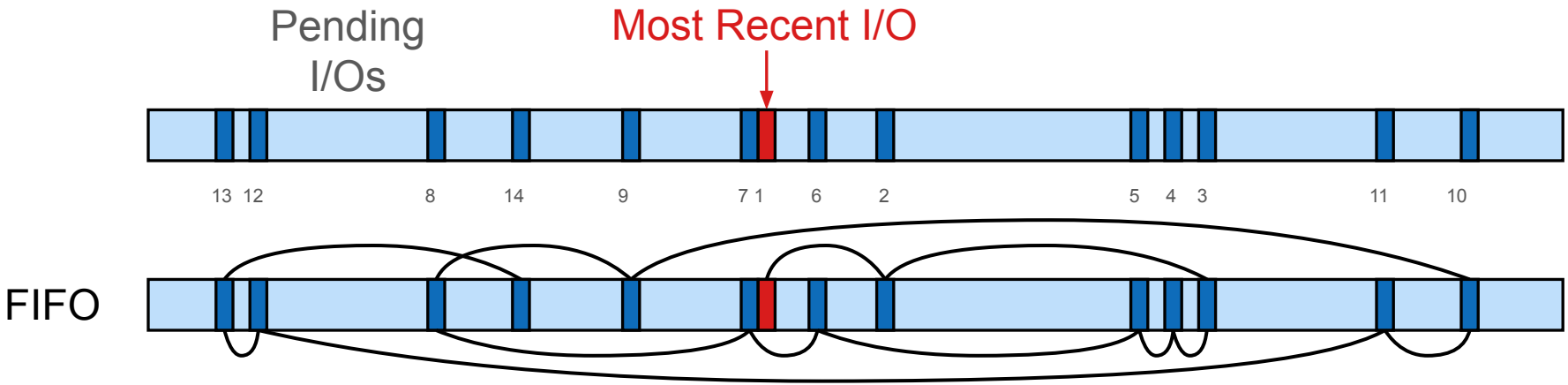
- If there are several disk I/O's waiting to be executed, what is the best order in which to execute them?
- Goal is to minimize seek time
  - First in first out (FIFO)
    - Simple, but does nothing to optimize seeks
  - Shortest positioning time first (SPTF)
    - Minimize seeks but can have starvation
  - Back-and-forth elevator like (SCAN)
    - Gets smallest seek but only in one direction
  - Circular scan (CSCAN)
    - Like scan but only process requests in one direction

Disk Scheduling



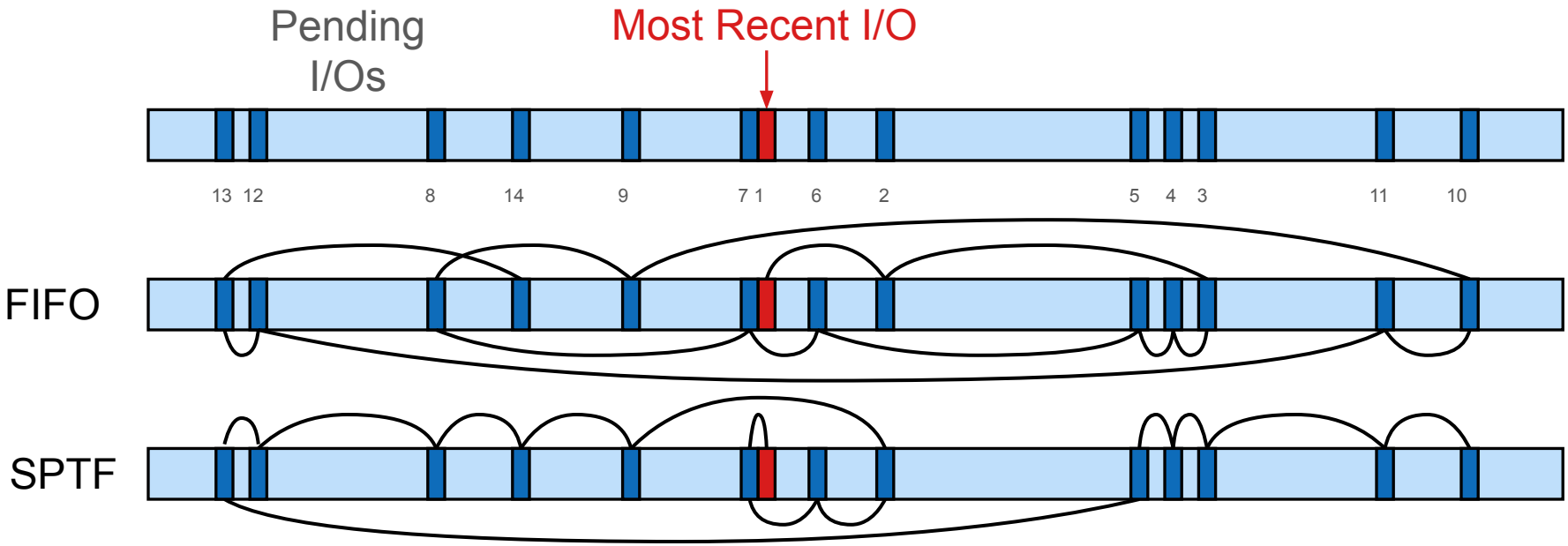
Disk scheduling - What order should we send these request to the disk?

Disk Scheduling FIFO



Lots of long seeks

Disk Scheduling SPTF



Minimal seeks

Disk Scheduling CSCAN