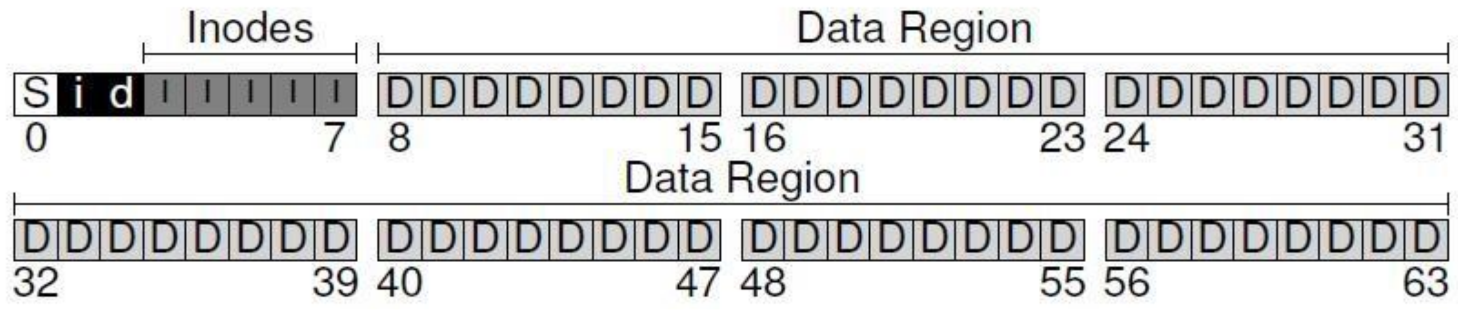
File System and Disk Scheduling

File System

- An organization of files and directories on disk
- OS has one or more file systems
- Two main aspects of file systems
  - Data structures to organize data and metadata on disk
  - Implementation of system calls like open, read, write using the data structures
- Disks expose a set of blocks (usually 512 bytes)
- File system organizes files onto blocks
  - System calls translated into reads and writes on blocks

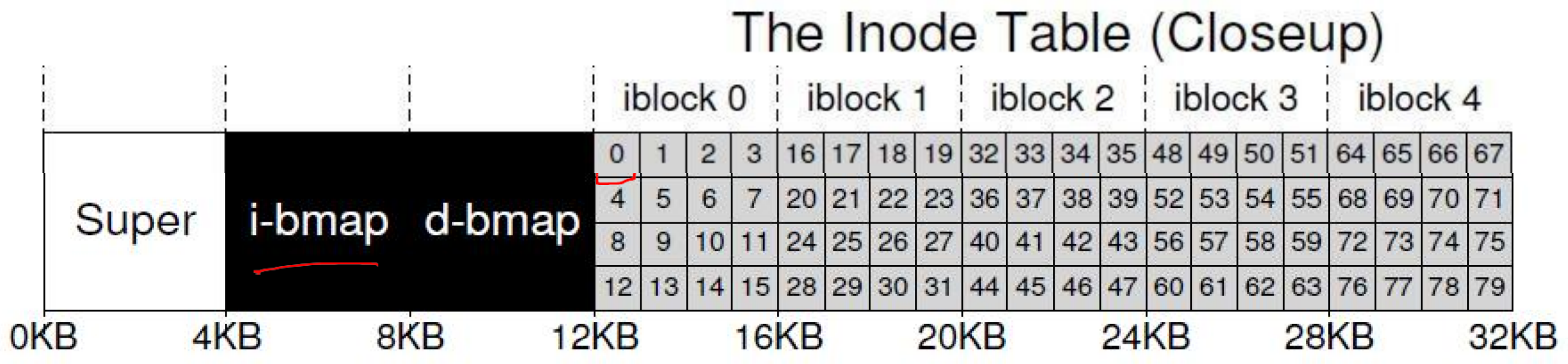
Example: a simple file system



- Data blocks: file data stored in one or more blocks
- Metadata about every file stored in inode
  - Location of data blocks of a file, permissions etc.
- Inode blocks: each block has one or more inodes
- Bitmaps: indicate which inodes/data blocks are free
- Superblock: holds master plan of all other blocks (which are inodes, which are data blocks etc.)

Inode table

- Usually, inodes (index nodes) stored in array
  - Inode number of a file is index into this array



- What does inode store?
  - File metadata: permissions, access time, etc.
  - Pointers (disk block numbers) of file data

Inode structure

- File data not stored contiguously on disk, need to track multiple block numbers of a file
- How does inode track disk block numbers?
  - Direct pointers: numbers of first few blocks are stored in inode itself (suffices for small files)
  - Indirect block: for larger files, inode stores number of indirect block, which has block numbers of file data
  - Similarly, double and triple indirect blocks (multi-level index)

File Allocation Table (FAT)

- Alternate way to track file blocks
- FAT stores next block pointer for each block
  - FAT has one entry per disk block
  - Entry has number of next file block, or null (if last block)
  - Pointer to first block stored in inode

Opening a file

- Why open? To have the inode readily available (in memory) for future operations on file
  - Open returns fd which points to in-memory inode
  - Reads and writes can access file data from inode
- What happens during open?
  - The pathname of the file is traversed, starting at root
  - Inode of root is known, to bootstrap the traversal
  - Recursively do: fetch inode of parent directory, read its data blocks, get inode number of child, fetch inode of child. Repeat until end of path
  - If new file, new inode and data blocks will have to be allocated using bitmap, and directory entry updated

Open file table

- Global open file table
  - One entry for every file opened (even sockets, pipes)
  - Entry points to in-memory copy of inode (other data structures for sockets and pipes)
- Per-process open file table
  - Array of files opened by a process
  - File descriptor number is index into this array
  - Per-process table entry points to global open file table entry
  - Every process has three files (standard in/out/err) open by default (fd 0, 1, 2)
- Open system call creates entries in both tables and returns fd number

Reading and writing a file

- For reading/writing file
  - Access in-memory inode via file descriptor
  - Find location of data block at current read/write offset
  - Fetch block from disk and perform operation
  - Writes may need to allocate new blocks from disk using bitmap of free blocks
  - Update time of access and other metadata in inode

Virtual File System

- File systems differ in implementations of data structures (e.g., organization of file records in directory)
- Linux supports virtual file system (VFS) abstraction
- VFS looks at a file system as objects (files, directories, inodes, superblock) and operations on these objects (e.g., lookup filename in directory)
- System call logic is written on VFS objects
- To develop a new file system, simply implement functions on VFS objects and provide pointers to these functions to kernel
- Syscall implementation does not have to change with file system implementation details

Disk Scheduling

- Requests to disk are not served in FIFO, they are reordered with other pending requests
- Why? In order to read blocks in sequence as far as possible, to minimize seek time and rotational delay
- Who does scheduling? OS does not know internal geometry of disk, so scheduling done mostly by disk controller

Disk Scheduling (Cont.)

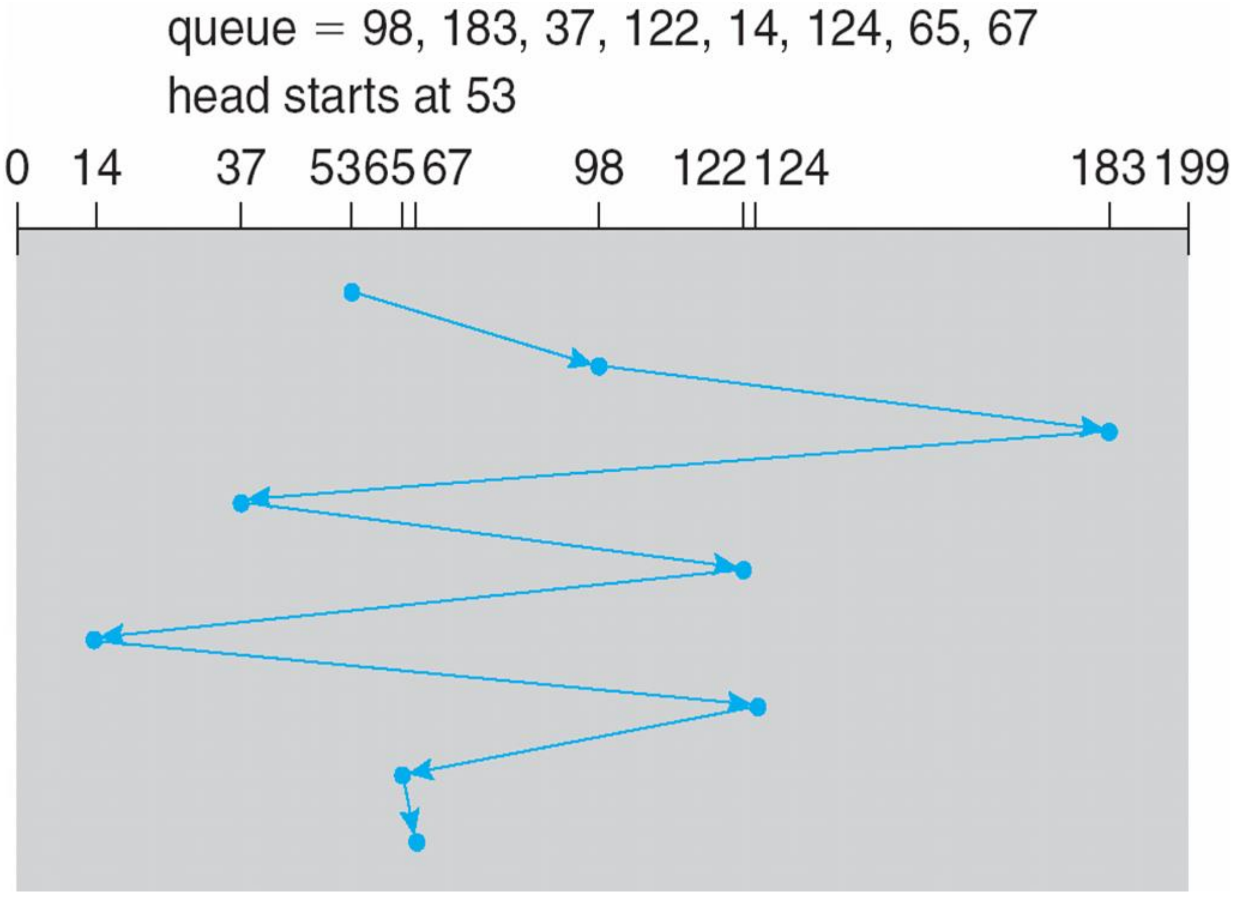
- Several algorithms exist to schedule the servicing of disk I/O requests
- We illustrate them with a request queue (0-199)

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

Head pointer 53

FCFS

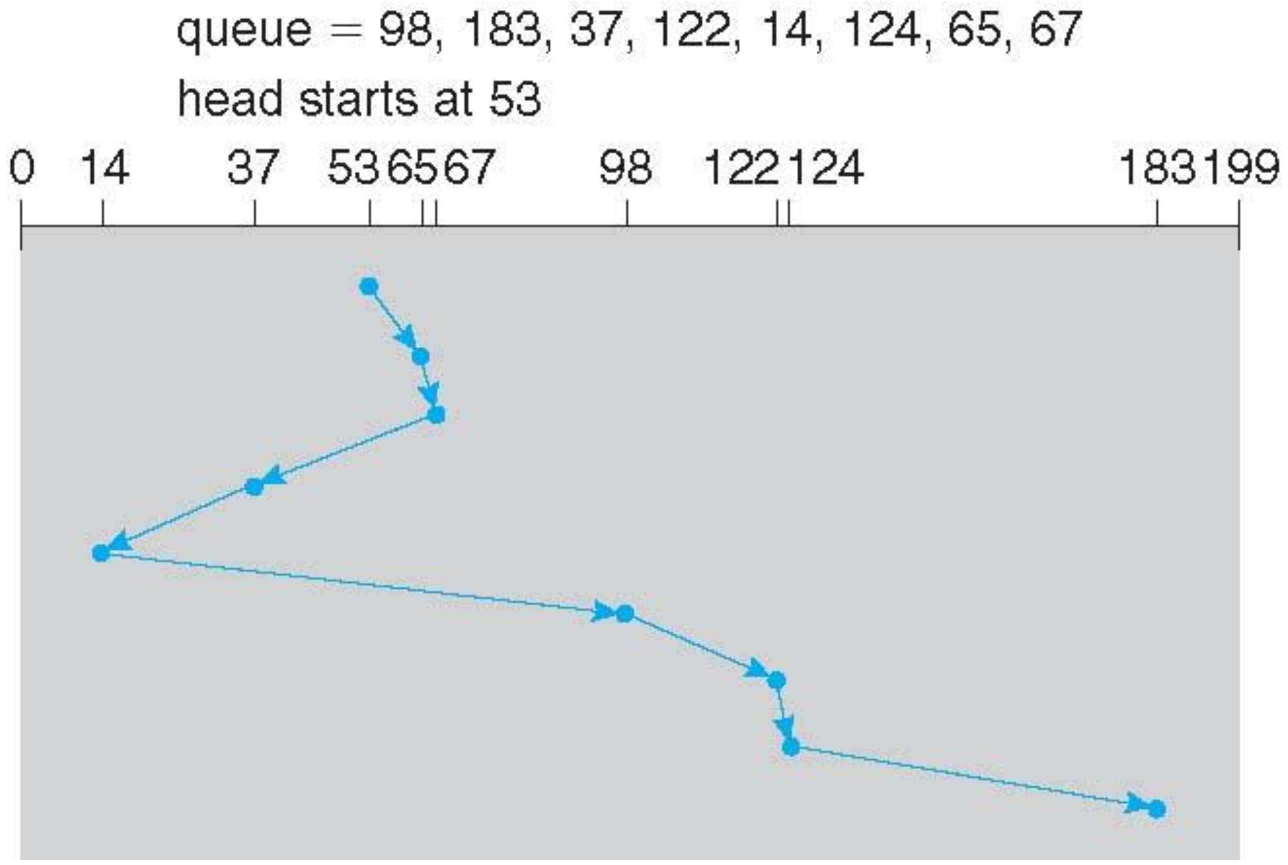
Illustration shows total head movement of 640 cylinders



SSTF

- Selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests
- Illustration shows total head movement of 236 cylinders

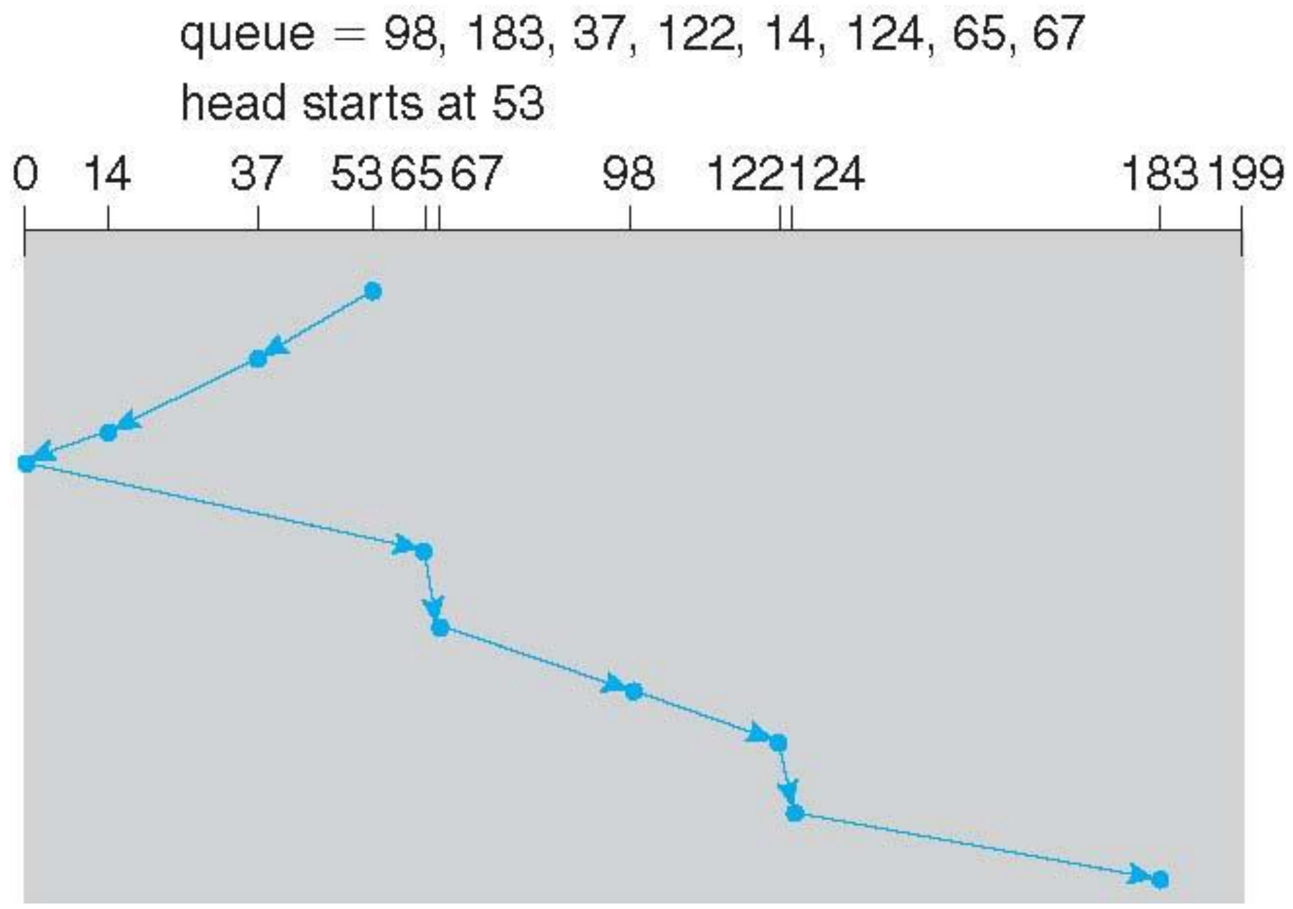
SSTF (Cont.)


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

SCAN

- The disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- **SCAN algorithm** sometimes called the **elevator algorithm**
- Illustration shows total head movement of 208 cylinders

SCAN (Cont.)


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

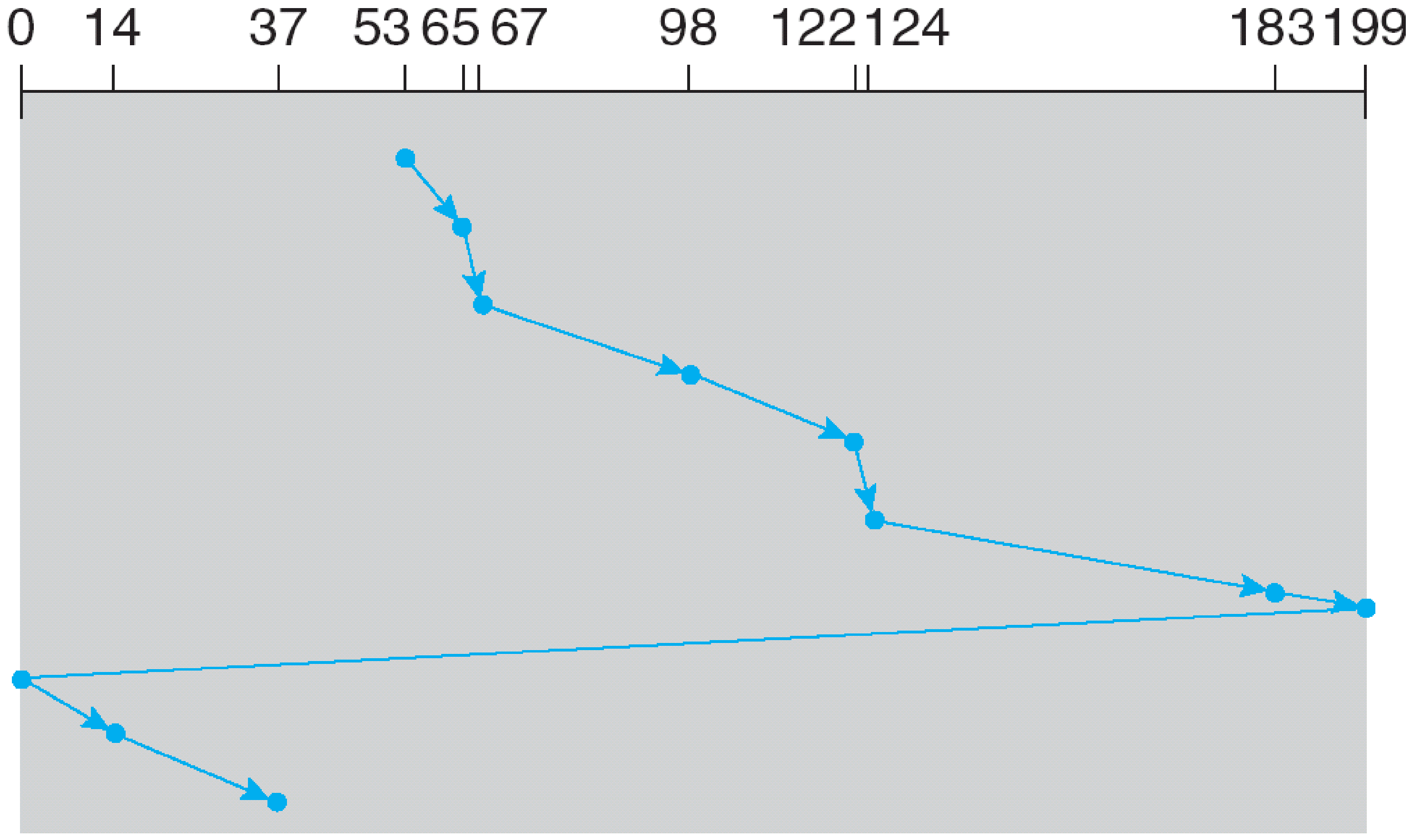
C-SCAN

- Provides a more uniform wait time than SCAN
- The head moves from one end of the disk to the other, servicing requests as it goes
  - When it reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip
- Treats the cylinders as a circular list that wraps around from the last cylinder to the first one

C-SCAN (Cont.)

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

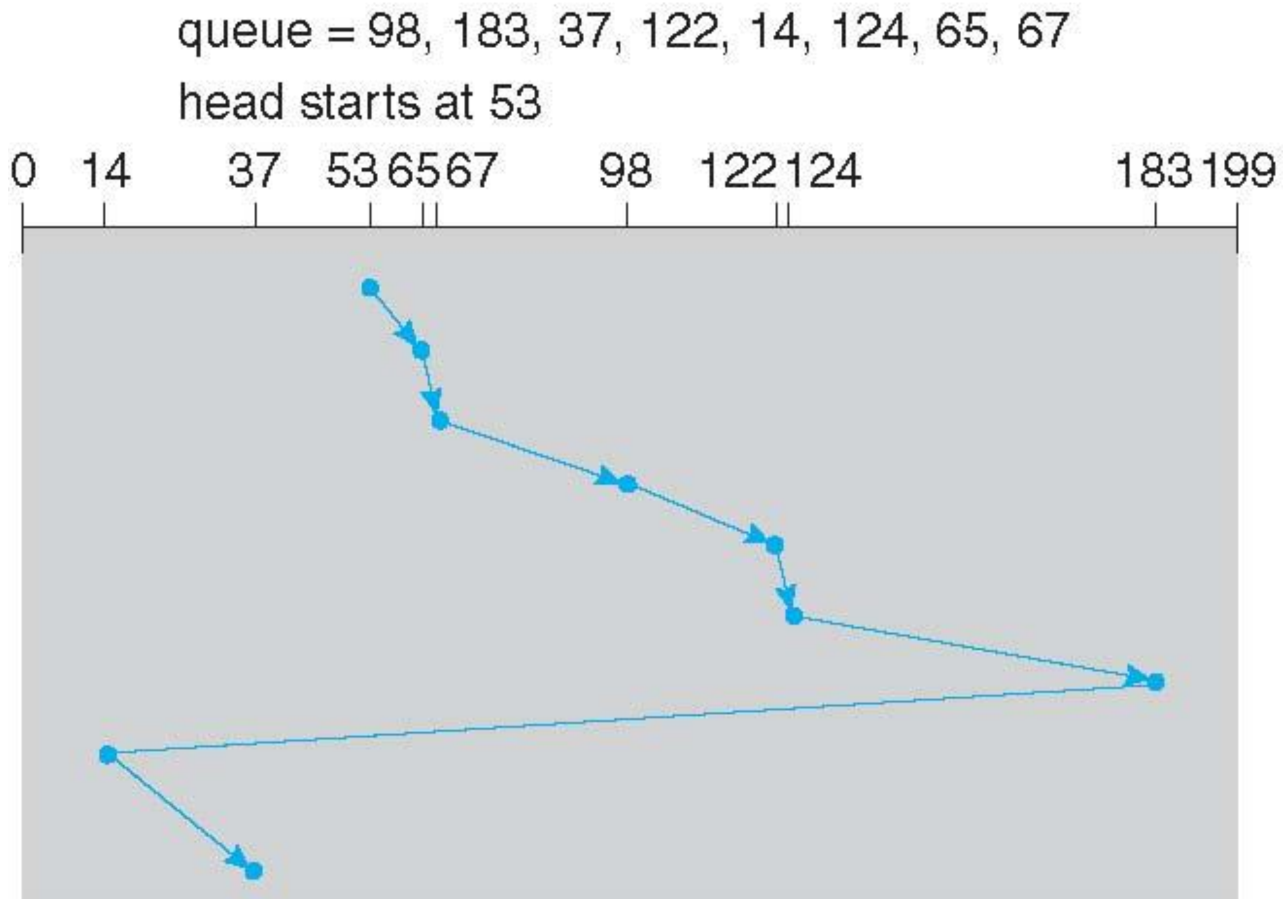
head starts at 53



C-LOOK

- Version of C-SCAN
- Arm only goes as far as the last request in each direction, then reverses direction immediately, without first going all the way to the end of the disk

C-LOOK (Cont.)


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0
14
37
53
65
67
98
122
124
183
199

Thank you!