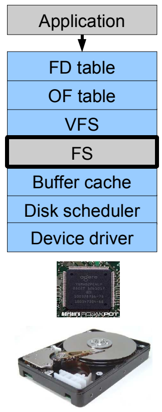
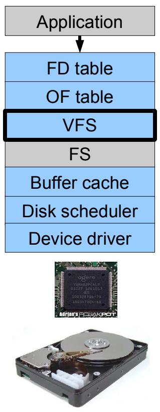
UNIX File Management (continued)

OS storage stack (recap)



Virtual File System (VFS)



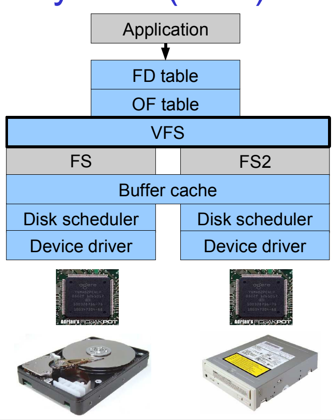
Older Systems only had a single file system

- They had file system specific open, close, read, write, … calls.
- However, modern systems need to support many file system types
  - ISO9660 (CDROM), MSDOS (floppy), ext2fs, tmpfs

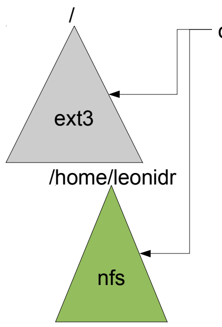
Supporting Multiple File Systems

- Alternatives
  - Change the file system code to understand different file system types
    - Prone to code bloat, complex, non-solution
  - Provide a framework that separates file system independent and file system dependent code.
    - Allows different file systems to be “plugged in”

Virtual File System (VFS)



Virtual file system (VFS)



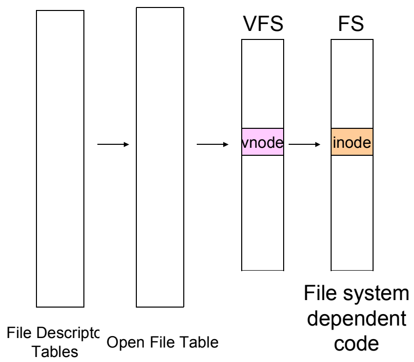
open("/home/leonidr/file", …);

Traversing the directory hierarchy may require VFS to issue requests to several underlying file systems

Virtual File System (VFS)

- Provides single system call interface for many file systems
  - E.g., UFS, Ext2, XFS, DOS, ISO9660,…
- Transparent handling of network file systems
  - E.g., NFS, AFS, CODA
- File-based interface to arbitrary device drivers (`/dev`)
- File-based interface to kernel data structures (`/proc`)
- Provides an indirection layer for system calls
  - File operation table set up at file open time
  - Points to actual handling code for particular type
  - Further file operations redirected to those functions

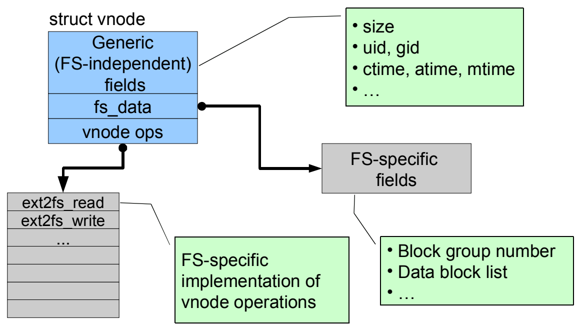
The file system independent code deals with vfs and vnodes



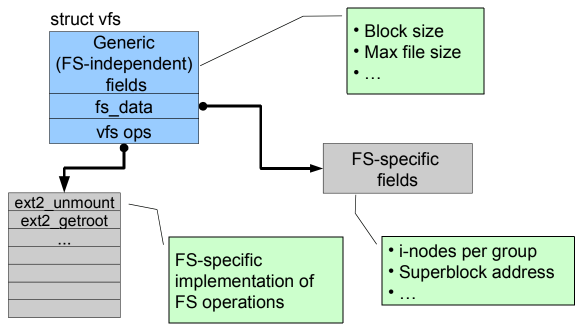
VFS Interface

- Reference
  - S.R. Kleiman., "Vnodes: An Architecture for Multiple File System Types in Sun Unix," USENIX Association: Summer Conference Proceedings, Atlanta, 1986
  - Linux and OS/161 differ slightly, but the principles are the same
- Two major data types
  - vfs
    - Represents all file system types
    - Contains pointers to functions to manipulate each file system as a whole (e.g. mount, unmount)
      - Form a standard interface to the file system
  - vnode
    - Represents a file (inode) in the underlying filesystem
    - Points to the real inode
    - Contains pointers to functions to manipulate files/inodes (e.g. open, close, read, write,…)

Vfs and Vnode Structures



Vfs and Vnode Structures



A look at OS/161's VFS

The OS161's file system type
Represents interface to a mounted filesystem

```
struct fs {
   int            (*fs_sync)(struct fs *);
   const char    *(*fs_getvolname)(struct fs *);
   struct vnode *(*fs_getroot)(struct fs *);
   int            (*fs_unmount)(struct fs *);

   void *fs_data;
};
```

- Force the filesystem to flush its content to disk
- Retrieve the volume name
- Retrieve the vnode associated with the root of the filesystem
- Unmount the filesystem Note: mount called via function ptr passed to **vfs_mount**
- Private file system specific data

Vnode

```
struct vnode {
  int vn_refcount;
  int vn_opencount;
  struct lock *vn_countlock;
  struct fs *vn_fs;
  void *vn_data;

  const struct vnode_ops *vn_ops;
};
```

- Count the number of "references" to this vnode
- Number of times vnode is currently open
- Lock for mutual exclusive access to counts
- Pointer to FS containing the vnode
- Pointer to FS specific vnode data (e.g. inode)
- Array of pointers to functions operating on vnodes

Vnode Ops

```
struct vnode_ops {
   unsigned long vop_magic;       /* should always be VOP_MAGIC */

   int (*vop_open)(struct vnode *object, int flags_from_open);
   int (*vop_close)(struct vnode *object);
   int (*vop_reclaim)(struct vnode *vnode);


   int (*vop_read)(struct vnode *file, struct uio *uio);
   int (*vop_readlink)(struct vnode *link, struct uio *uio);
   int (*vop_getdirentry)(struct vnode *dir, struct uio *uio);
   int (*vop_write)(struct vnode *file, struct uio *uio);
   int (*vop_ioctl)(struct vnode *object, int op, userptr_t data);
   int (*vop_stat)(struct vnode *object, struct stat *statbuf);
   int (*vop_gettype)(struct vnode *object, int *result);
   int (*vop_tryseek)(struct vnode *object, off_t pos);
   int (*vop_fsync)(struct vnode *object);
   int (*vop_mmap)(struct vnode *file /* add stuff */);
   int (*vop_truncate)(struct vnode *file, off_t len);
   int (*vop_namefile)(struct vnode *file, struct uio *uio);
```


Vnode Ops

```
   int (*vop_creat)(struct vnode *dir,
    const char *name, int excl,
    struct vnode **result);
   int (*vop_symlink)(struct vnode *dir,
      const char *contents, const char *name);
   int (*vop_mkdir)(struct vnode *parentdir,
    const char *name);
   int (*vop_link)(struct vnode *dir,
   const char *name, struct vnode *file);
   int (*vop_remove)(struct vnode *dir,
     const char *name);
   int (*vop_rmdir)(struct vnode *dir,
    const char *name);

   int (*vop_rename)(struct vnode *vn1, const char *name1,
     struct vnode *vn2, const char *name2);


   int (*vop_lookup)(struct vnode *dir,
     char *pathname, struct vnode **result);
   int (*vop_lookparent)(struct vnode *dir,
         char *pathname, struct vnode **result,
         char *buf, size_t len);
};
```


Vnode Ops

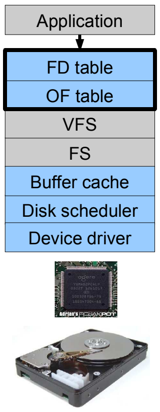
- Note that most operations are on vnodes. How do we operate on file names?
  - Higher level API on names that uses the internal VOP_* functions

```
int vfs_open(char *path, int openflags, struct vnode **ret);
void vfs_close(struct vnode *vn);
int vfs_readlink(char *path, struct uio *data);
int vfs_symlink(const char *contents, char *path);
int vfs_mkdir(char *path);
int vfs_link(char *oldpath, char *newpath);
int vfs_remove(char *path);
int vfs_rmdir(char *path);
int vfs_rename(char *oldpath, char *newpath);

int vfs_chdir(char *path);
int vfs_getcwd(struct uio *buf);
```

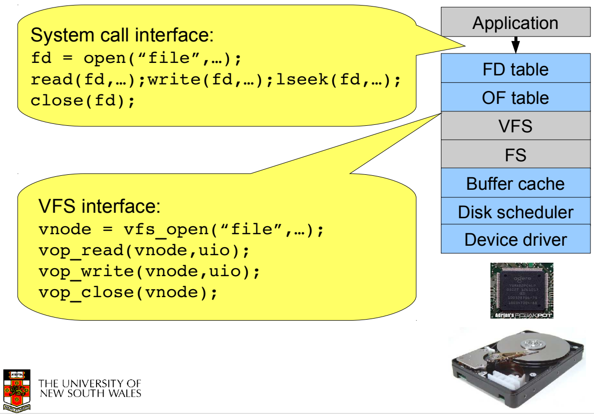

Example: OS/161 emufs vnode ops

```
/*
 * Function table for emufs files.
 */
static const struct vnode_ops emufs_fileops = {
   VOP_MAGIC,   /* mark this a valid vnode ops table */

   emufs_open,
   emufs_close,
   emufs_reclaim,

   emufs_read,
   NOTDIR,   /* readlink */
   NOTDIR,   /* getdirentry */
   emufs_write,
   emufs_ioctl,
   emufs_stat,
   emufs_file_gettype,
   emufs_tryseek,
   emufs_fsync,
   UNIMP,    /* mmap */
   emufs_truncate,
   NOTDIR,   /* namefile */

   NOTDIR,   /* creat */
   NOTDIR,   /* symlink */
   NOTDIR,   /* mkdir */
   NOTDIR,   /* link */
   NOTDIR,   /* remove */
   NOTDIR,   /* rmdir */
   NOTDIR,   /* rename */

   NOTDIR,   /* lookup */
   NOTDIR,   /* lookparent */
};
```

File Descriptor & Open File Tables



Motivation



File Descriptors

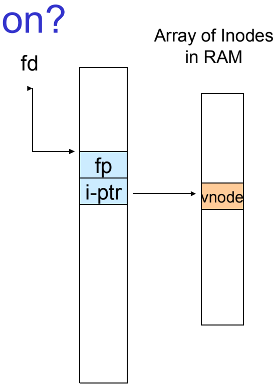
- File descriptors
  - Each open file has a file descriptor
  - Read/Write/lseek/…. use them to specify which file to operate on.
- State associated with a file fescriptor
  - File pointer
    - Determines where in the file the next read or write is performed
  - Mode
    - Was the file opened read-only, etc.…

An Option?

- Use vnode numbers as file descriptors and add a file pointer to the vnode
- Problems
  - What happens when we concurrently open the same file twice?
    - We should get two separate file descriptors and file pointers….

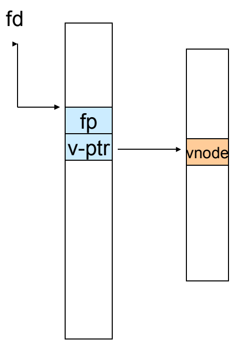
An Option?

- Single global open file array
  - *fd* is an index into the array
  - Entries contain file pointer and pointer to a vnode



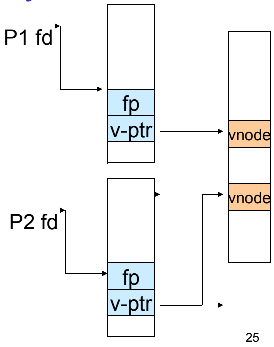
Issues

- File descriptor 1 is stdout
  - Stdout is
    - console for some processes
    - A file for others
- Entry 1 needs to be different per process!



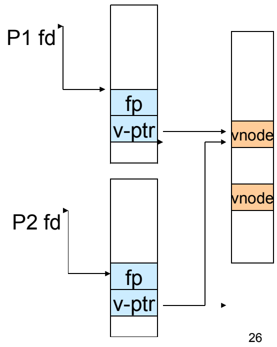
Per-process File Descriptor Array

- Each process has its own open file array
  - Contains fp, v-ptr etc.
  - *Fd* 1 can be any inode for each process (console, log file).



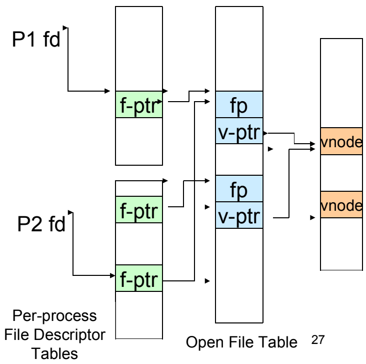
Issue

- Fork
  - Fork defines that the child shares the file pointer with the parent
- Dup2
  - Also defines the file descriptors share the file pointer
- With per-process table, we can only have independent file pointers
  - Even when accessing the same file



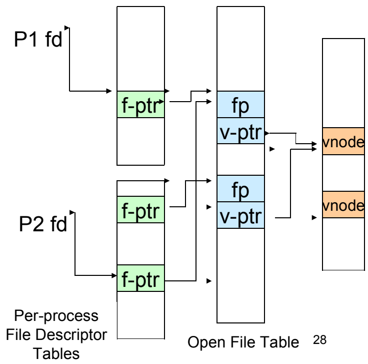
Per-Process *fd* table with global open file table

- Per-process file descriptor array
  - Contains pointers to *open file table entry*
- Open file table array
  - Contain entries with a fp and pointer to an vnode.
- Provides
  - Shared file pointers if required
  - Independent file pointers if required
- Example:
  - All three *fds* refer to the same file, two share a file pointer, one has an independent file pointer

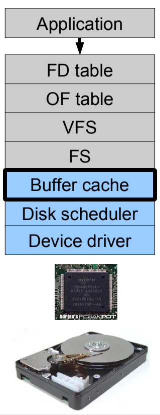


Per-Process *fd* table with global open file table

- Used by Linux and most other Unix operating systems



Buffer Cache

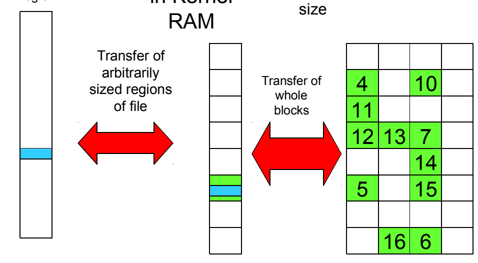


Buffer

- Buffer:
  - Temporary storage used when transferring data between two entities
    - Especially when the entities work at different rates
    - Or when the unit of transfer is incompatible
    - Example: between application program and disk

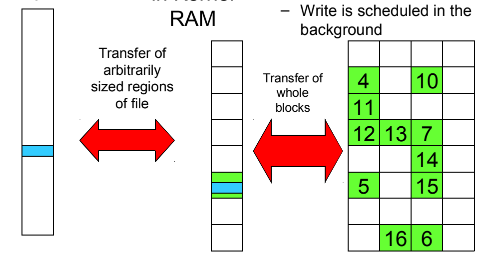
Buffering Disk Blocks

- Allow applications to work with arbitrarily sized region of a file
  - However, apps can still optimise for a particular block size



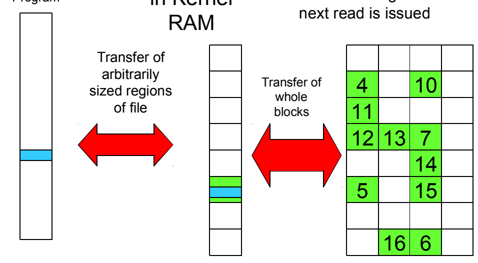
Buffering Disk Blocks

- Writes can return immediately after copying to kernel buffer
  - Avoids waiting until write to disk is complete
  - Write is scheduled in the background



Buffering Disk Blocks

- Can implement read-ahead by pre-loading next block on disk into kernel buffer
  - Avoids having to wait until next read is issued

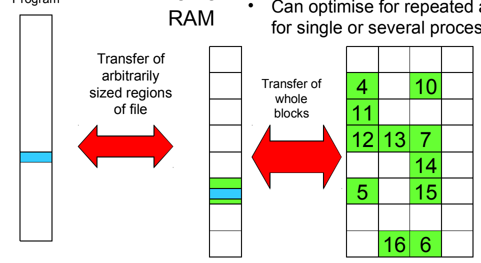


Cache

- Cache:
  - Fast storage used to temporarily hold data to speed up repeated access to the data
    - Example: Main memory can cache disk blocks

Caching Disk Blocks

- On access
  - Before loading block from disk, check if it is in cache first
    - Avoids disk accesses
- Can optimise for repeated access for single or several processes

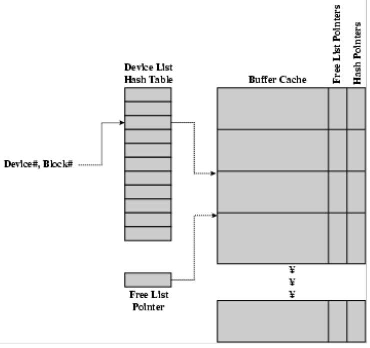


Buffering and caching are related

- Data is read into buffer; extra cache copy would be wasteful
- After use, block should be put in a cache
- Future access may hit cached copy
- Cache utilises unused kernel memory space; may have to shrink

Unix Buffer Cache

On read
- Hash the device#, block#
- Check if match in buffer cache
- Yes, simply use in-memory copy
- No, follow the collision chain
- If not found, we load block from disk into cache



Replacement

- What happens when the buffer cache is full and we need to read another block into memory?
  - We must choose an existing entry to replace
    - Need a policy to choose a victim
      - Can use First-in First-out
      - Least Recently Used, or others.
    - Timestamps required for LRU implementation
    - However, is strict LRU what we want?

File System Consistency

- File data is expected to survive
- Strict LRU could keep critical data in memory forever if it is frequently used.

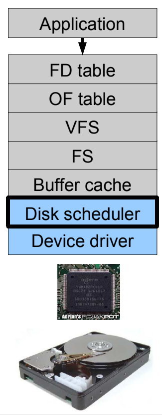
File System Consistency

- Generally, cached disk blocks are prioritised in terms of how critical they are to file system consistency
  - Directory blocks, inode blocks if lost can corrupt entire filesystem
    - E.g. imagine losing the root directory
    - These blocks are usually scheduled for immediate write to disk
  - Data blocks if lost corrupt only the file that they are associated with
    - These blocks are only scheduled for write back to disk periodically
    - In UNIX, flushd (*flush daemon*) flushes all modified blocks to disk every 30 seconds

File System Consistency

- Alternatively, use a write-through cache
  - All modified blocks are written immediately to disk
  - Generates much more disk traffic
    - Temporary files written back
    - Multiple updates not combined
  - Used by DOS
    - Gave okay consistency when
      - Floppies were removed from drives
      - Users were constantly resetting (or crashing) their machines
  - Still used, e.g. USB storage devices

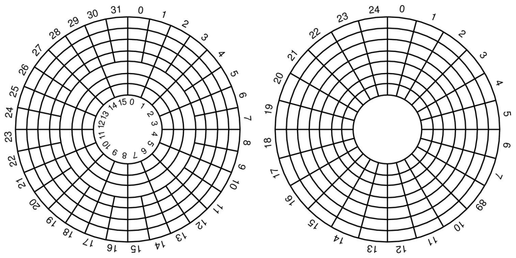
Disk scheduler



Disk Management

- Management and ordering of disk access requests is important:
  - Huge speed gap between memory and disk
  - Disk throughput is extremely sensitive to
    - Request order ⇒ Disk Scheduling
    - Placement of data on the disk ⇒ file system design
  - Disk scheduler must be aware of *disk geometry*

Disk Geometry



- Physical geometry of a disk with two zones
  - Outer tracks can store more sectors than inner without exceed max information density
- A possible virtual geometry for this disk

Evolution of Disk Hardware

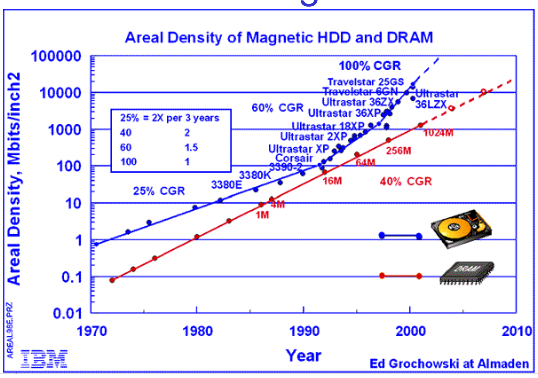
Parameter	IBM 360-KB floppy disk	WD 18300 hard disk
Number of cylinders	40	10601
Tracks per cylinder	2	12
Sectors per track	9	281 (avg)
Sectors per disk	720	35742000
Bytes per sector	512	512
Disk capacity	360 KB	18.3 GB
Seek time (adjacent cylinders)	6 msec	0.8 msec
Seek time (average case)	77 msec	6.9 msec
Rotation time	200 msec	8.33 msec
Motor stop/start time	250 msec	20 sec
Time to transfer 1 sector	22 msec	17 μsec

Disk parameters for the original IBM PC floppy disk and a Western Digital WD 18300 hard disk

Things to Note

- Average seek time is approx 12 times better
- Rotation time is 24 times faster
- Transfer time is 1300 times faster
  - Most of this gain is due to increase in density
- Represents a gradual engineering improvement

Storage Capacity is 50000 times greater



Estimating Access Time

- *Seek time* T_s: Moving the head to the required track
  - not linear in the number of tracks to traverse:
    - ➜ startup time
    - ➜ settling time
  - Typical average seek time: a few milliseconds
- *Rotational delay:*
  - rotational speed, r, of 5,000 to 10,000rpm
  - At 10,000rpm, one revolution per 6ms ⇒ average delay 3ms
- *Transfer time:*
  to transfer b bytes, with N bytes per track: $T = \frac{b}{rN}$

Total average access time: $T_a = T_s + \frac{1}{2r} + \frac{b}{rN}$

A Timing Comparison

- $T_s = 2$ ms, $r = 10,000$ rpm, 512B sect, 320 sect/track
- Read a file with 2560 sectors (= 1.3MB)
- File stored compactly (8 adjacent tracks):

Read first track	
Average seek	2ms
Rot. delay	3ms
Read 320 sectors	6ms
	11ms

$\Rightarrow$ All sectors: $11 + 7 * 8 = 67\ ms$

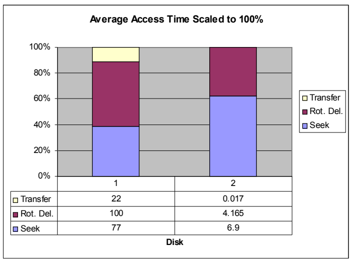
- Sectors distributed randomly over the disk:

Read any sector	
Average seek	2ms
Rot. delay	3ms
Read 1 sector	0.01875ms
	5.01875ms

$\Rightarrow$ All: $2560 * 5.01875 = 20,328ms$

Disk Performance is Entirely Dominated by Seek and Rotational Delays

- Will only get worse as capacity increases much faster than increase in seek time and rotation speed
  - Note it has been easier to spin the disk faster than improve seek time
- Operating System should minimise mechanical delays as much as possible



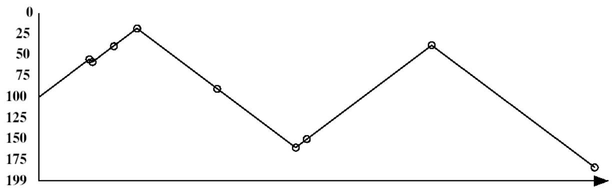
Disk Arm Scheduling Algorithms

- Time required to read or write a disk block determined by 3 factors
  1. Seek time
  2. Rotational delay
  3. Actual transfer time
- Seek time dominates
- For a single disk, there will be a number of I/O requests
  - Processing them in random order leads to worst possible performance

First-in, First-out (FIFO)

- Process requests as they come
- Fair (no starvation)
- Good for a few processes with clustered requests
- Deteriorates to random if there are many processes

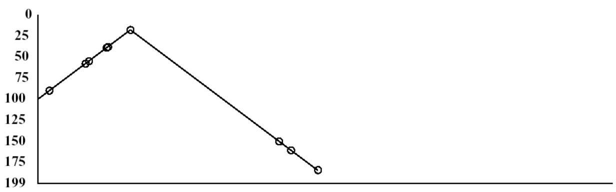
Request tracks: 55, 58, 39, 18, 90, 160, 150, 38, 184



Shortest Seek Time First

- Select request that minimises the seek time
- Generally performs much better than FIFO
- May lead to starvation

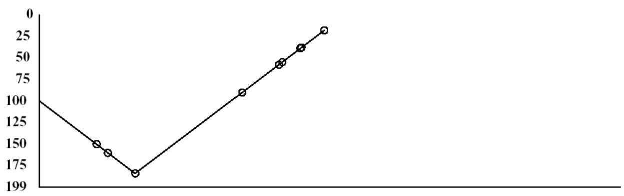
Request tracks: 55, 58, 39, 18, 90, 160, 150, 38, 184



Elevator Algorithm (SCAN)

- **Move head in one direction**
  - Services requests in track order until it reaches the last track, then reverses direction
- **Better than FIFO, usually worse than SSTF**
- **Avoids starvation**
- **Makes poor use of sequential reads (on down-scan)**
- **Less Locality**

Request tracks: 55, 58, 39, 18, 90, 160, 150, 38, 184



Modified Elevator (Circular SCAN, C-SCAN)

- Like elevator, but reads sectors in only one direction
  - When reaching last track, go back to first track non-stop
- Better locality on sequential reads
- Better use of read ahead cache on controller
- Reduces max delay to read a particular sector

Request tracks: 55, 58, 39, 18, 90, 160, 150, 38, 184