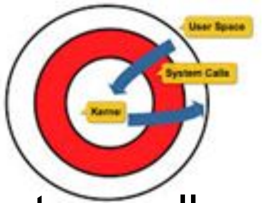
Storage and File Systems

Chester Rebeiro
IIT Madras

Two views of a file system

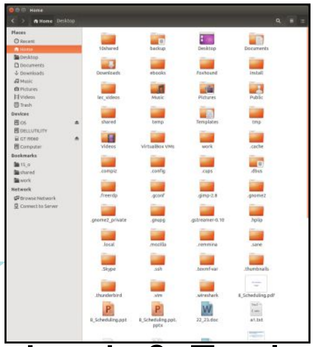


system calls

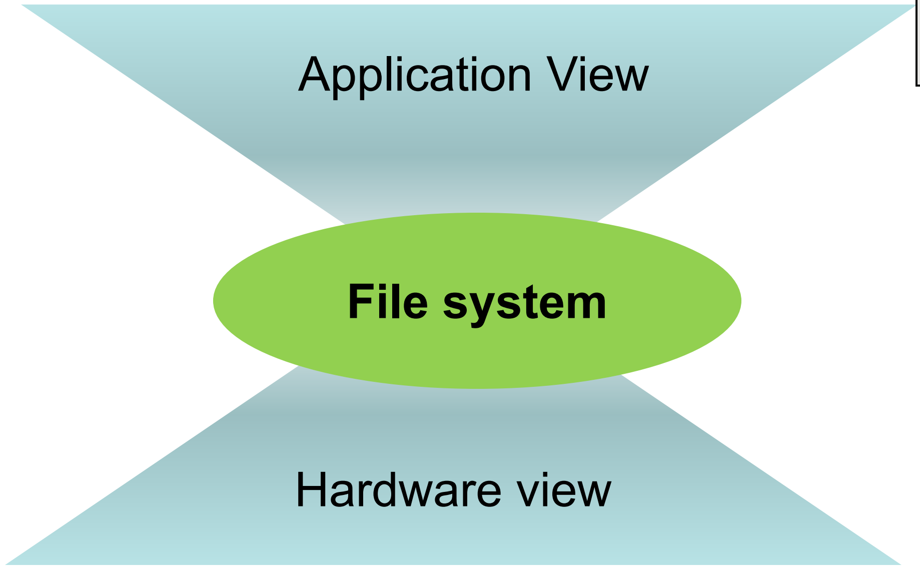
protection

rwx

attributes



Look & Feel

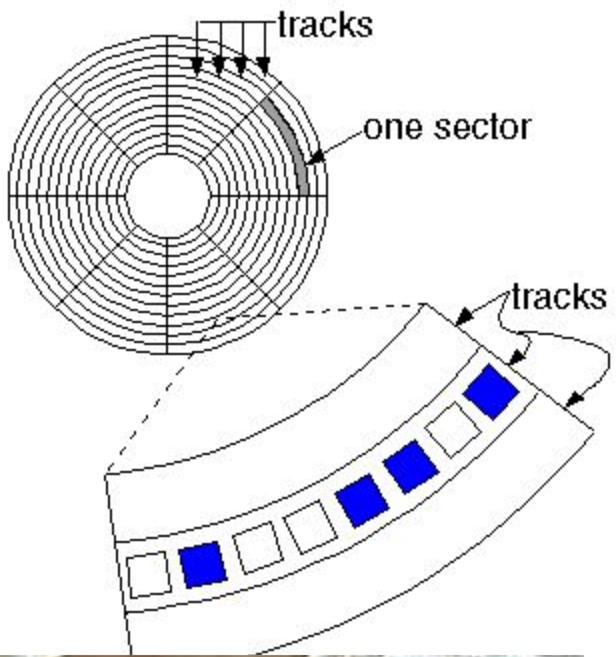


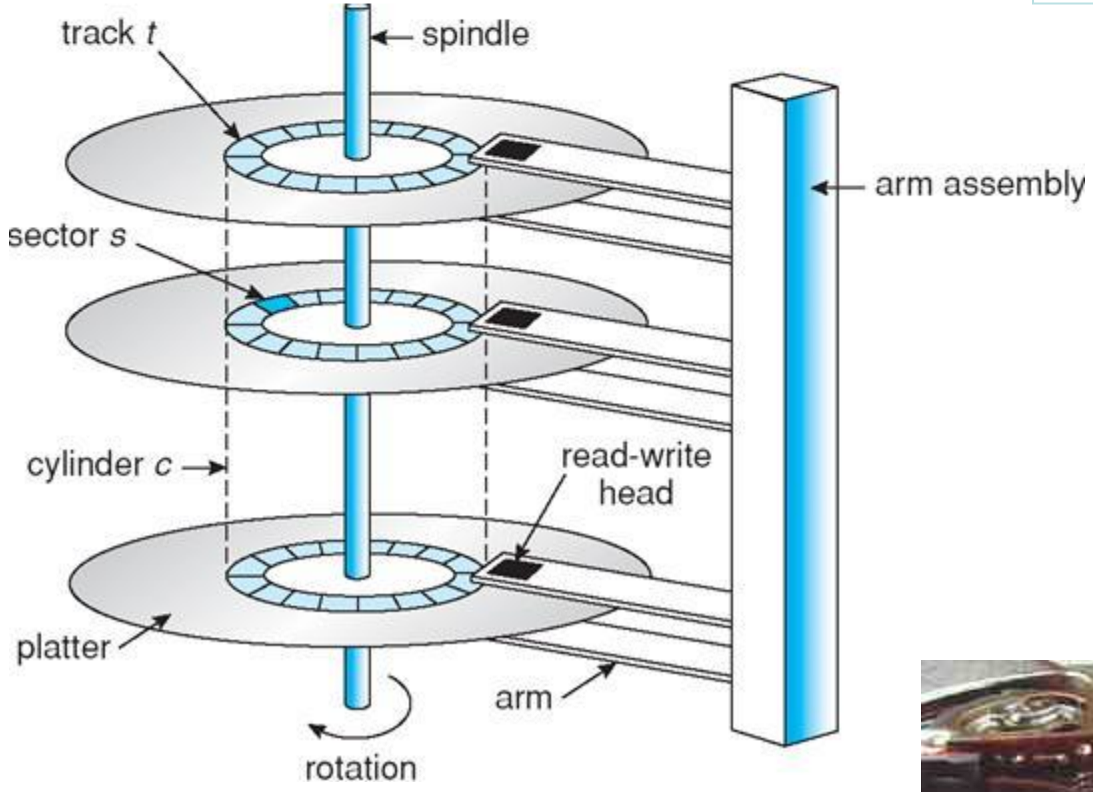
Magnetic Disks

Chester Rebeiro
IIT Madras

Magnetic Disks

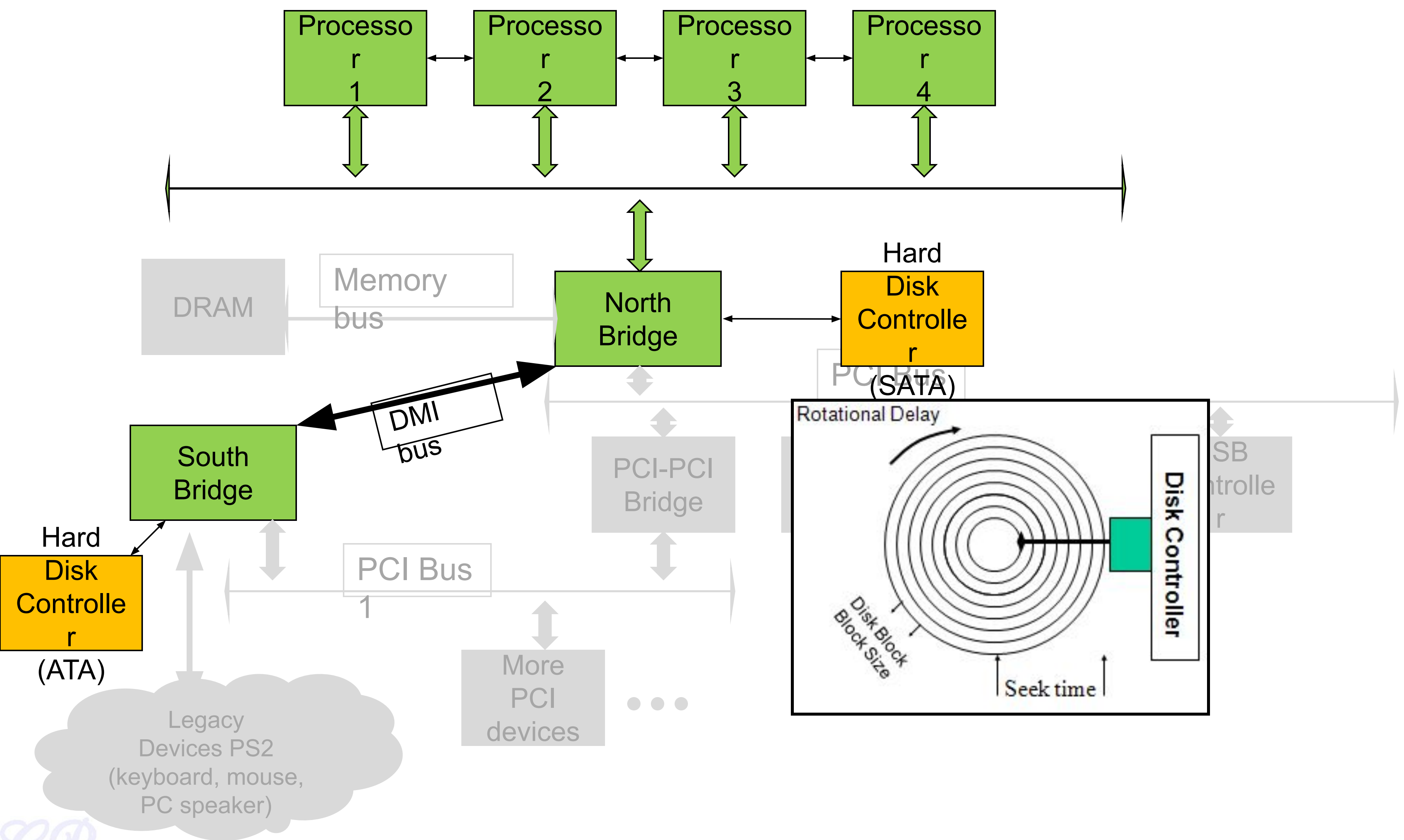
Tracks and Sectors on a platter



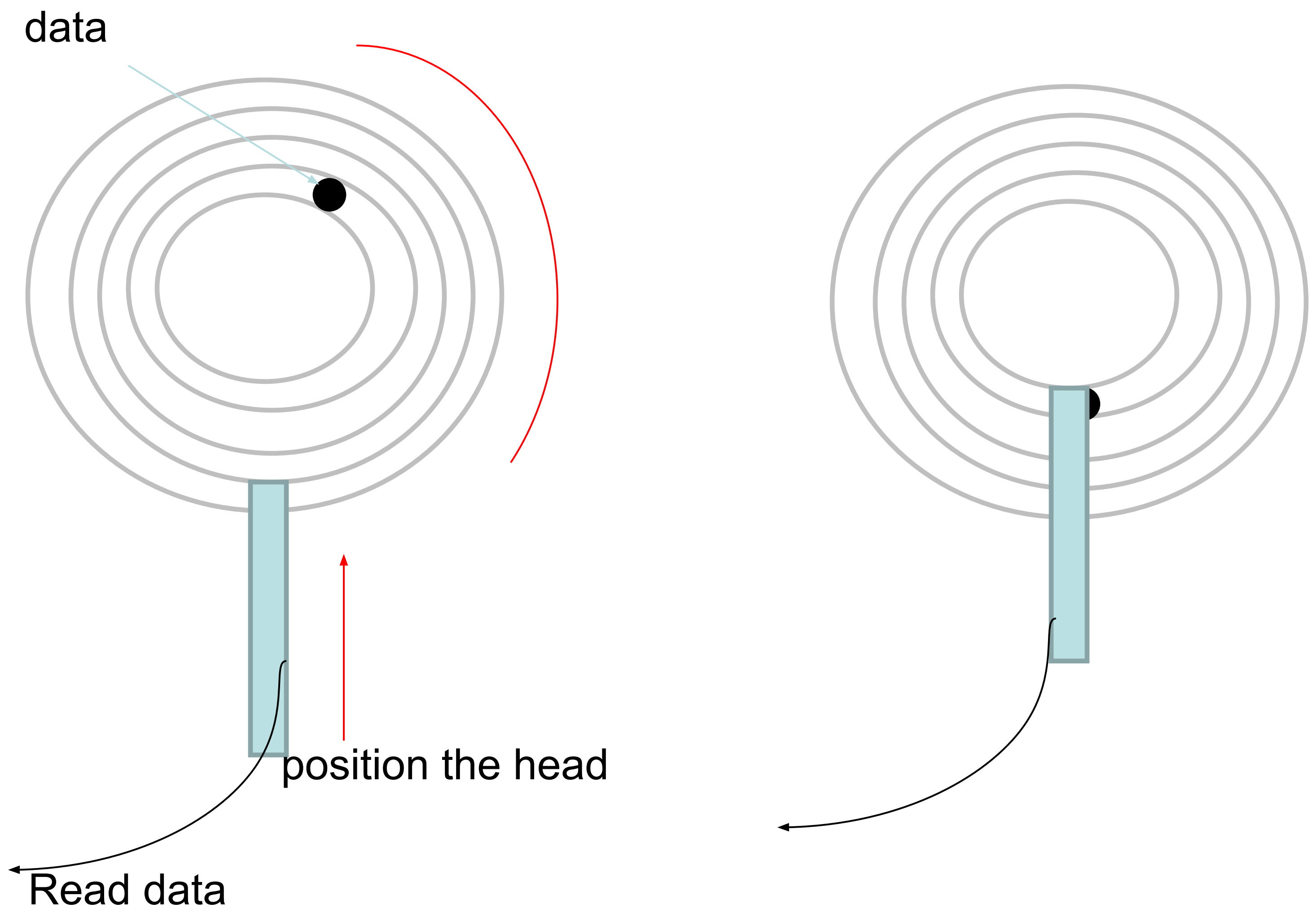


Structure of a magnetic disk

Disk Controllers



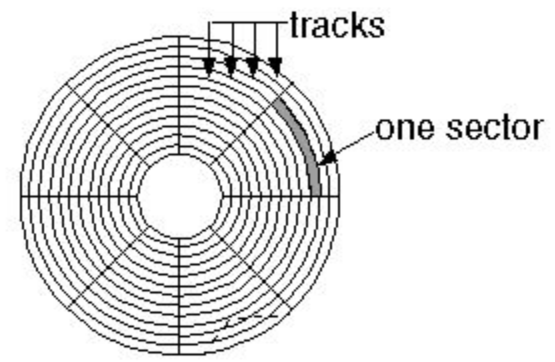
Access Time



Access Time

- **Seek Time**
  - Time it takes the head assembly to travel to the desired track
  - Seek time may vary depending on the current head location
  - Average seek time generally considered.(Typically 4ms high end servers to 15ms in external drives)
- **Rotational Latency**
  - Delay waiting for the rotation of the disk to bring the required disk sector under the head
  - Depends on the speed of the spindle motor
    - CLV vs CAV
- **Data Rate**
  - Time to get data off the disk

CLV and CAV



Outer sectors can typically store more data than inner sectors

- CLV (Constant linear Velocity) – spindle speed (rpm) varies depending on position of the head. So as to maintain constant read (or write) speeds.
  - Used in audio CDs to ensure constant read rate at which data is read from disk
- CAV (Constant angular velocity) -- spindle velocity is always a constant. Used in hard disks. Easy to engineer.
  - Allows higher read rates because there are no momentum issues

Disk Addressing

- Older schemes
  - CHS (cylinder, head, sector) tuple
  - Well suited for disks, but not for any other medium
  - Need an abstraction
- Logical block addressing (LBA)
  - Large 1-D array of logical blocks
    - Logical block, is the smallest unit of transfer. Typically of size 512 bytes (but can be changed by low level format – don't try this at home!!)
  - Addressing with 48 bits
  - Mapping from CHS to LBA

$$\mathbf{LBA = (C \times HPC + H) \times SPT + (S - 1)}$$

C : cylinder, H : head, S : sector, HPC: heads / cylinder, SPT: sectors / track

Disk Scheduling

- Objectives
  - Access time minimize
    - Two components
      - Minimize Seek time by minimizing head movement
      - Minimize Rotational latency
  - Bandwidth
    - Bytes transferred / (total time taken)

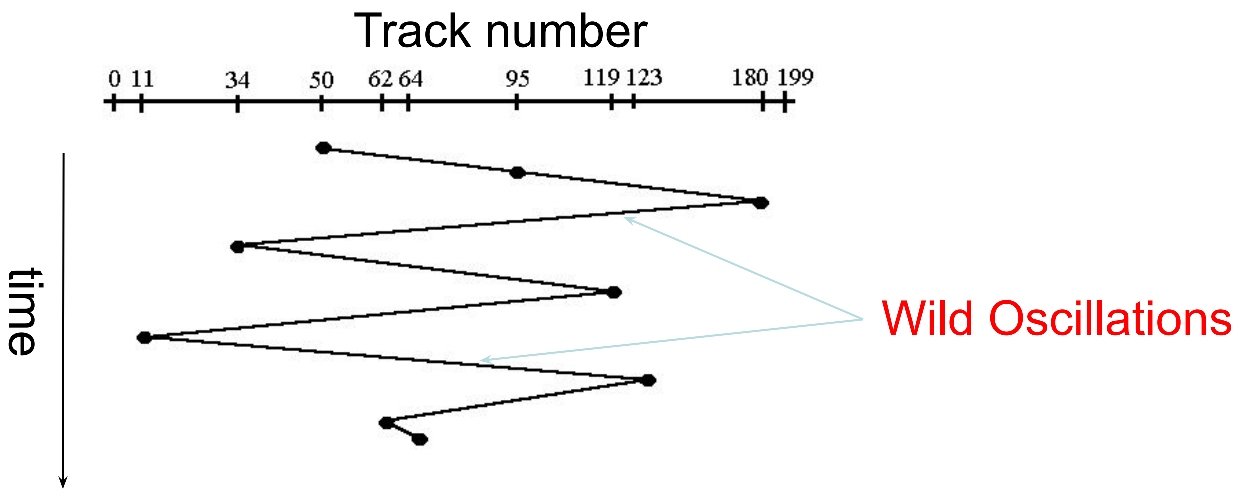
Access time and bandwidth can be managed by the order in which Disk I/O requests are serviced

Disk Scheduling

- Read/write accesses have the following cylinder queue :

    95, 180, 34, 119, 11, 123, 62, 64

- The current position of the head is 50
- FCFS

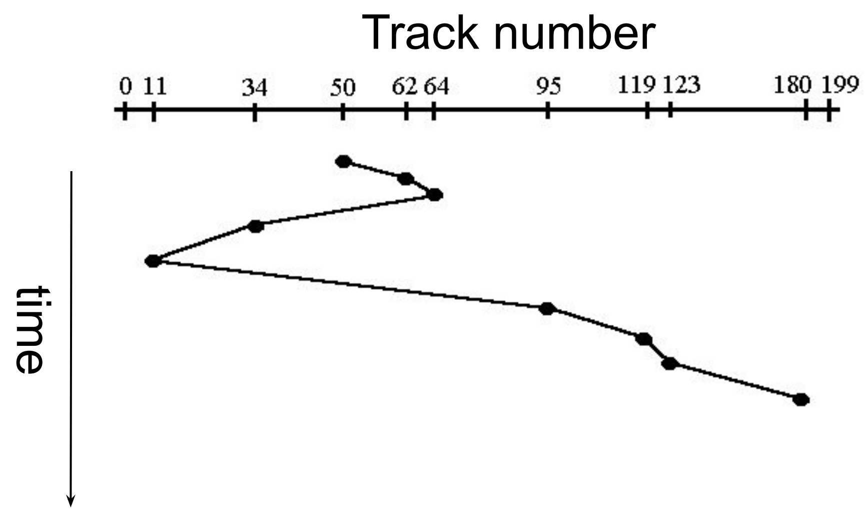


Total head movments = |(95 – 50)| + |(180 – 95)| + |(34 – 180)| + …
= 644

Shortest Seek Time First (SSTF)

95, 180, 34, 119, 11, 123, 62, 64
Starting at 50

Total head movments = 236



- Counterpart of SJF
- Could lead to starvation

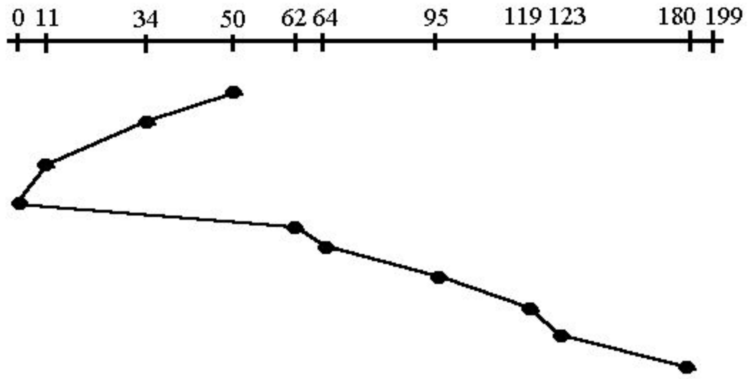
Elevators

95, 180, 34, 119, 11, 123, 62, 64

SCAN

- Start scanning toward the nearest end and goes till **0**
- Then goes all the way till the other end

Total head movements = 230

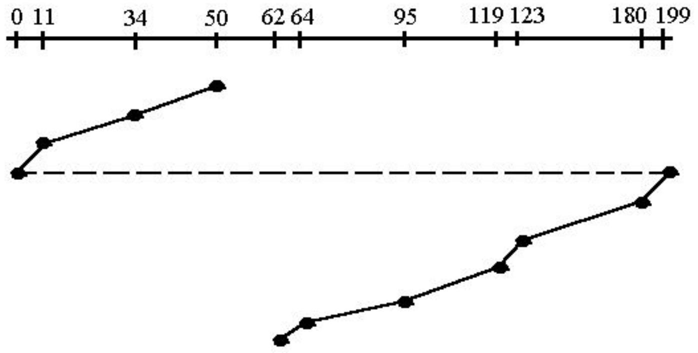


C-SCAN

- Start scanning toward the nearest end and go till the **0**
- Then go all the way to the other end

Total head movements = 187

- Useful if tracks accessed with uniform distribution
- Shifting one extreme not included in head movement count

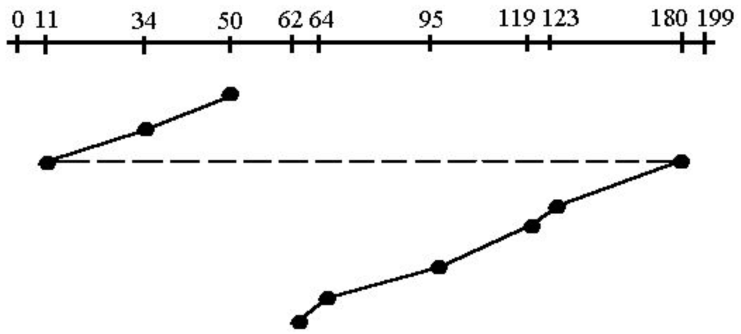


C-LOOK

95, 180, 34, 119, 11, 123, 62, 64
Starting at 50

- Like C-SCAN, but don’t go to the extreme.
- Stop at the minimum (or maximum)

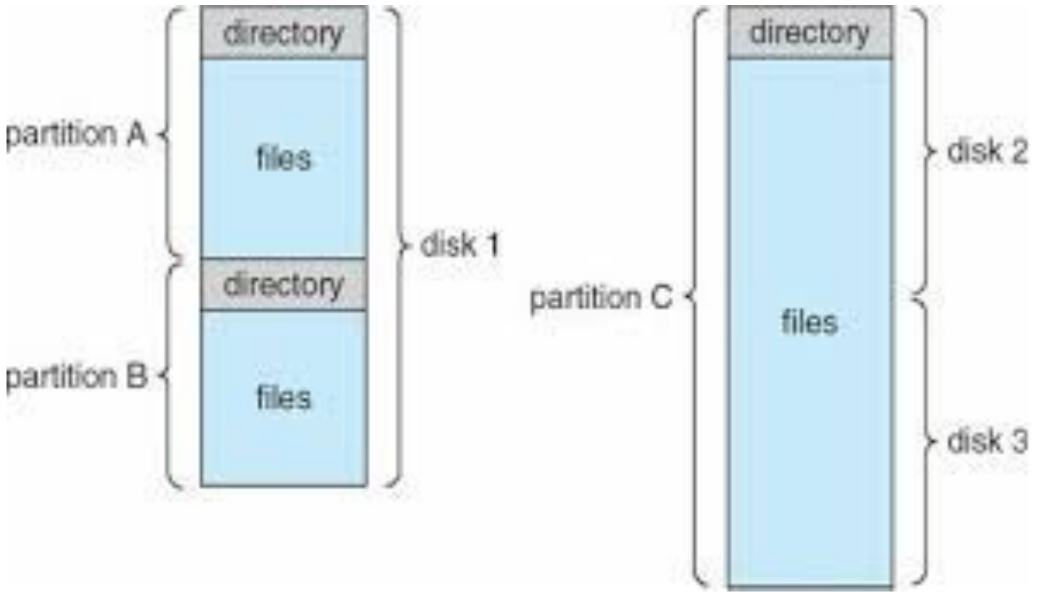
Total head movements = 157



Application View

Chester Rebeiro
IIT Madras

A File System Organization



- Volume used to store a file system
- A volume could be present in partitions, disks, or across disks
- Volume contains directories which record information about name, location, size, and type of all files on that volume

Mounting a File System

- Just like a file needs to be opened, a file system needs to be mounted
- OS needs to know
  - The location of the device (partition) to be mounted
  - The location in the current file structure where the file system is to be attached

```
$ mount /dev/sda3 /media/xyz -t ext3
```

- OS does,
  - Verify that the device has a valid file system
  - Add new file system to the mount point (/media/xyz)

Files

- From a user's perspective,
  - A byte array
  - Persistent across reboots and power failures

- From OS perspective,
  - Secondary (non-volatile) storage device
    - Hard disks, USB, CD, etc.
  - Map bytes as collection of blocks on storage device

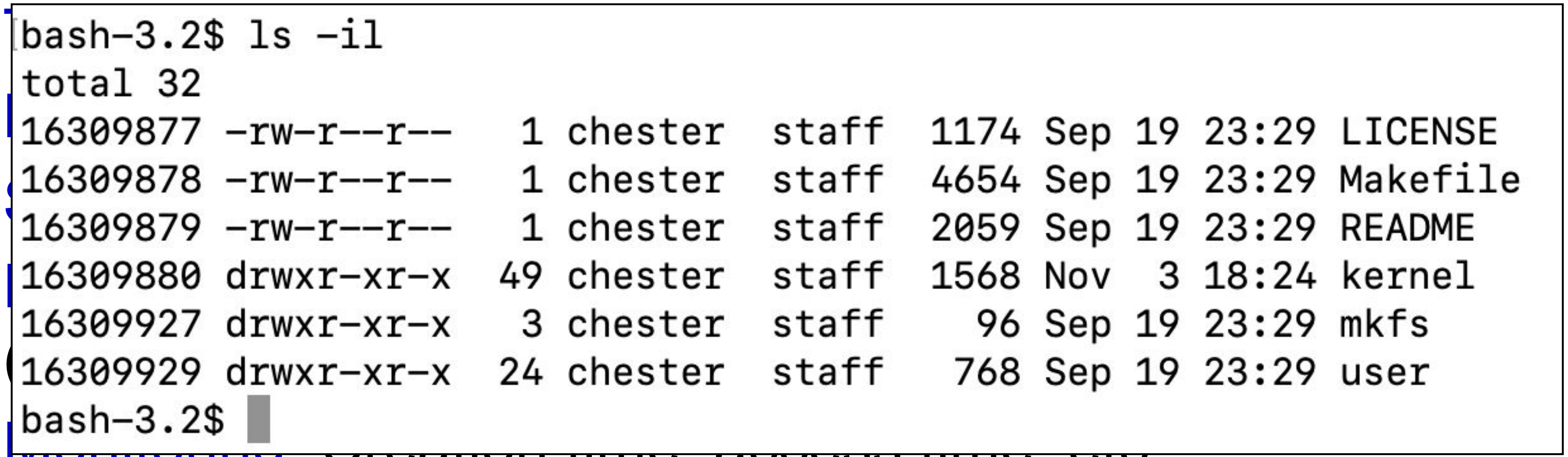
A File’s Metadata (inodes)

- Name. the only information kept in human readable form.
- Identifier. A number that uniquely identifies the file within the file system. Also called the inode number
- Type. File type (inode based file, pipe, etc.)
- Location. Pointer to location of file on device.
- Size.
- Protection. Access control information. Owner, group (r,w,x) permissions, etc. a
- Monitoring. Creation time, access time, etc.

Try ls –i on Linux to see the inode number for a file

A File's Metadata (inodes)

- Name. the only information kept in human readable form.
- Identifier. A number that uniquely identifies the file within the file system. Also called the inode number



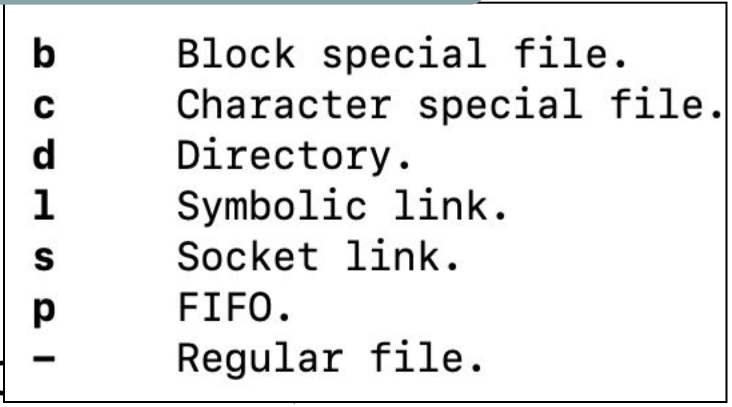
```
bash-3.2$ ls -il
total 32
16309877 -rw-r--r--   1 chester  staff  1174 Sep 19 23:29 LICENSE
16309878 -rw-r--r--   1 chester  staff  4654 Sep 19 23:29 Makefile
16309879 -rw-r--r--   1 chester  staff  2059 Sep 19 23:29 README
16309880 drwxr-xr-x  49 chester  staff  1568 Nov  3 18:24 kernel
16309927 drwxr-xr-x   3 chester  staff    96 Sep 19 23:29 mkfs
16309929 drwxr-xr-x  24 chester  staff   768 Sep 19 23:29 user
bash-3.2$ 
```

- Monitoring: Creation time, access time, etc.

Try ls –i on Linux to see the inode number for a file

A File’s Metadata (inodes)

- Name. the only information kept in human readable form.
- Identifier. A number that uniquely identifies the file within the file system. Also called the inode number
- Type. File type (inode based file, pipe, etc.)
- Location. Pointer to location
- Size.
- Protection. Access control in
  (r,w,x) permissions, etc. a
- Monitoring. Creation time, ac



b	Block special file.
c	Character special file.
d	Directory.
l	Symbolic link.
s	Socket link.
p	FIFO.
-	Regular file.

Try ls –i on Linux to see the inode number for a file

A File’s Metadata (inodes)

- Name. the only information kept in human readable form.
- Identifier. A number that uniquely identifies the file within the file system. Also called the inode number
- Type. File type (inode based file, pipe, etc.)
- Location. Pointer to location of file on device.
- Size.
- Protection. Access control information. Owner, group (r,w,x) permissions, etc. a
- Monitoring. Creation time, access time, etc.

Try ls –i on Linux to see the inode number for a file

Files vs Memory

- Every memory location has an address that can be directly accessed
- In files, everything is relative
  - A location of a file depends on the directory it is stored in
  - A pointer must be used to store the current read or write position within the file
  - Eg. To read a byte in a specific file.
    - First search for the file in the directory path and resolve the identifier

      expensive for each access !!!
    - Use the read pointer to seek the byte position
  - Solution : Use open system call to open the file before any access

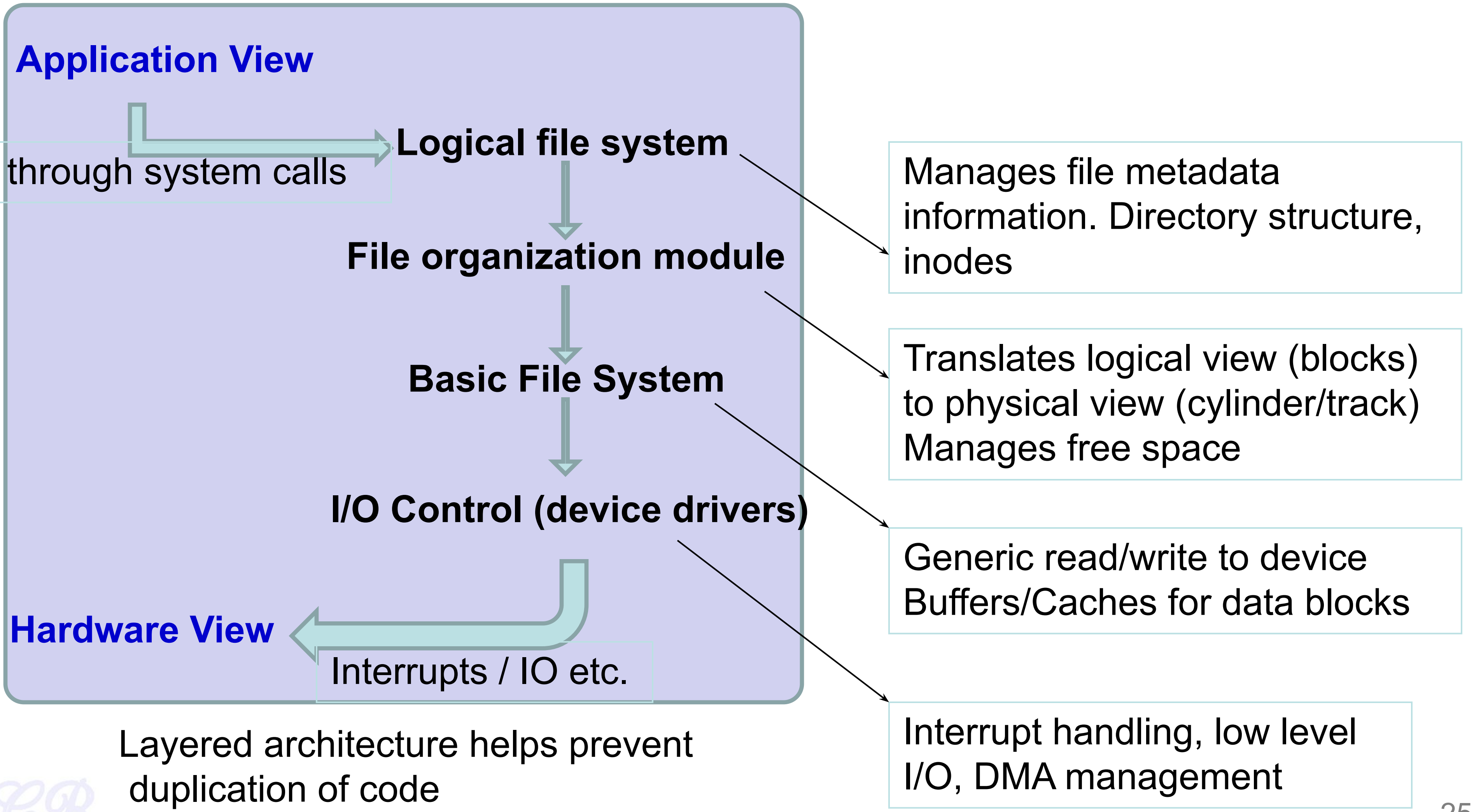
    (and close system call to close the file after all accesses are complete)

Implementing a File System

Chester Rebeiro

IIT Madras

FS Layers



Manages file metadata information. Directory structure, inodes

Translates logical view (blocks) to physical view (cylinder/track)
Manages free space

Generic read/write to device
Buffers/Caches for data blocks

Interrupt handling, low level I/O, DMA management

Layered architecture helps prevent duplication of code

File System : disk contents

- Boot control block (per volume)
  - If no OS, then boot control block is empty
- Volume control block (per volume)
  - Volume(or partition details) such as number of blocks in the partition, size of blocks, free blocks, etc.
  - Sometimes called the superblock
- Directory structure
  - To organize the files. In Unix, this may include file names and associated inode numbers. In Windows, it is a table.
- Per file FCB (File control block)
  - Metadata about a file. Unique identifier to associate it with a directory.

file system : in-memory contents

- Mount table : contains information about each mounted volume

```
$ cat /etc/fstab
```

- In memory directory structure cache holds recently accessed directories
- System wide open file table
- Per process open file table
- Buffer cache to hold file system blocks

File operations (create)

- File Creation
  1. Create FCB for the new file
  2. Update directory contents
- 3. Write new directory contents to disk (and may cache it as well)

Opening a File

- Steps involved
  - Resolve Name : search directories for file names and check permissions
  - Read file metadata into open file table
  - Return index in the open file table (this is the familiar file descriptor)

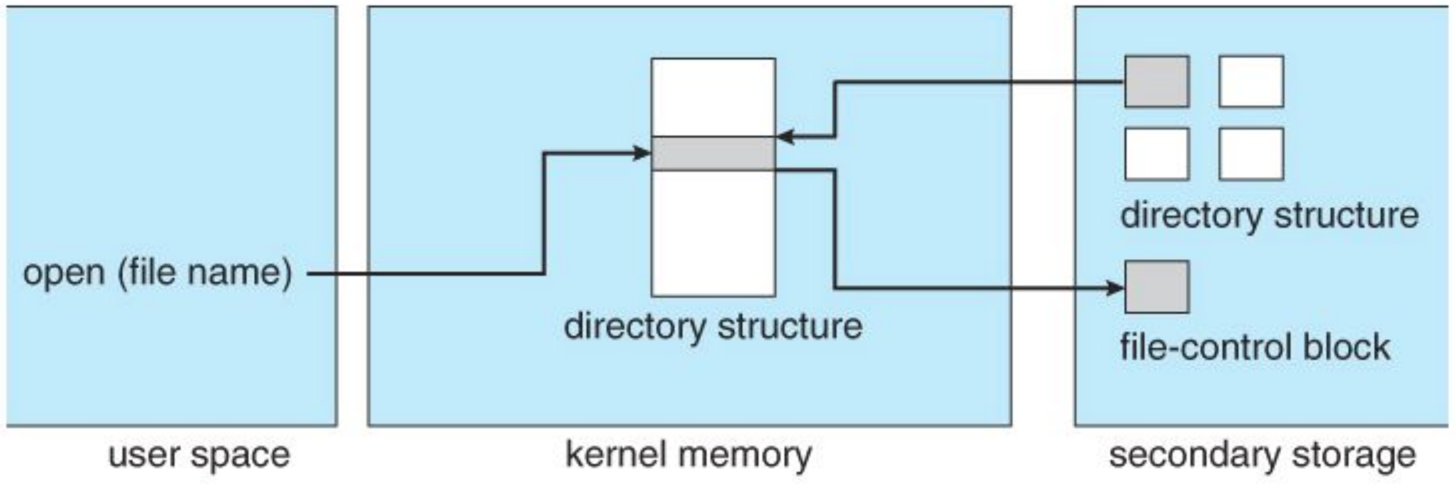
Open file tables

- Two open file tables used
  - system wide table
    - Contains information about inode, size, access dates, permission, location, etc.
    - Reference count (tracks number of processes that have opened the file)
  - per process table
    - Part of PCBs proc structure
    - Pointer to entry in the system wide table

File Operations (open)

- File Open

1. Application passes file name through open system call
2. sys_open searches the system-wide open file table to see if the file is already in use by another process
   - If yes, then increment usage count and add pointer in per-process open file table
   - If no, search directory structure for file name (either in the cache or disk) add to system-wide open file table and per-process open file table
3. The pointer (or index) in the per-process open file table is returned to application. This becomes the file descriptor

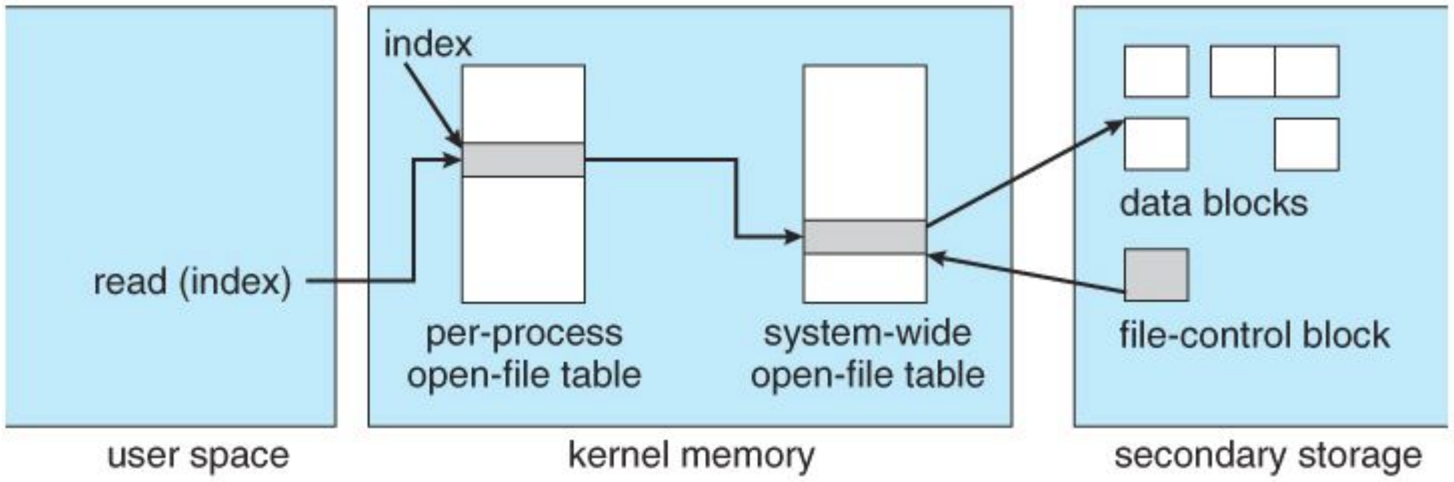


File operations (close)

- file close
  - Per process open table entry is removed
  - System wide open table reference count decremented by 1.
    - If this value becomes 0 then updates copied back to disk (if needed)
    - Remove system wide open table entry

File Operations (read/write)

- File Read



File Access Methods

- Sequential Access
  - Information processed one block after the other
  - Typical usage
- Direct Access
  - Suitable for database systems
  - When query arrives, compute the corresponding block number, and directly access block

Tracking Free Space

- Bitmap of blocks
  - 1 indicates used, 0 indicates free
- Linked list of free blocks

- File systems may use heuristics
  - eg. A group of closely spaced free blocks

Allocation Methods

- How does the OS allocate blocks in the disk?
  - Contiguous allocation
  - Linked allocation
  - Indexed allocation

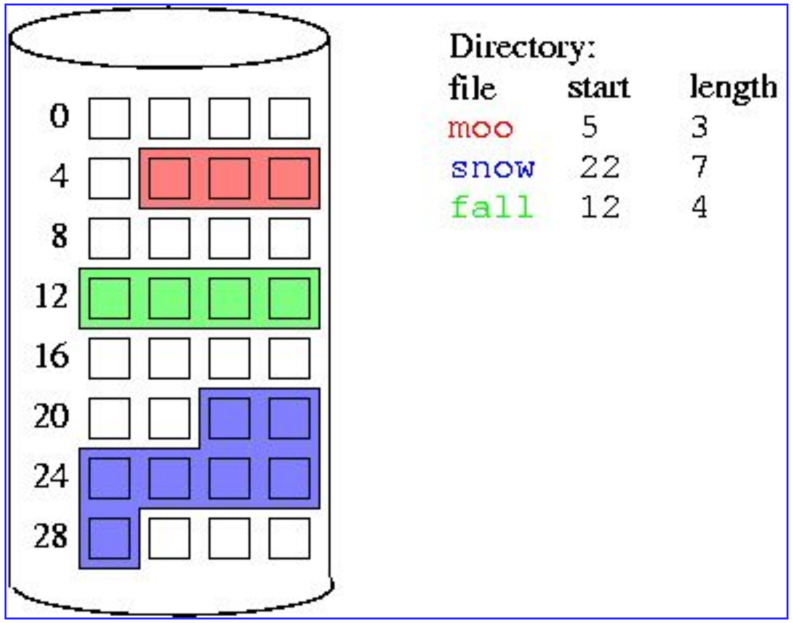
Contiguous Allocation

- Each file is allocated contiguous blocks on the disk
- Directory entry keeps the start and length
- Allocation
  - First fit / best fit ?
- Advantages
  - Easy / simple
- Disadvantages
  - External fragmentation

    (may need regular defagmentation)
  - Users need to specify the

    maximum file size at creation

    (may lead to internal fragmentation – a file may request a much large space and not use it)

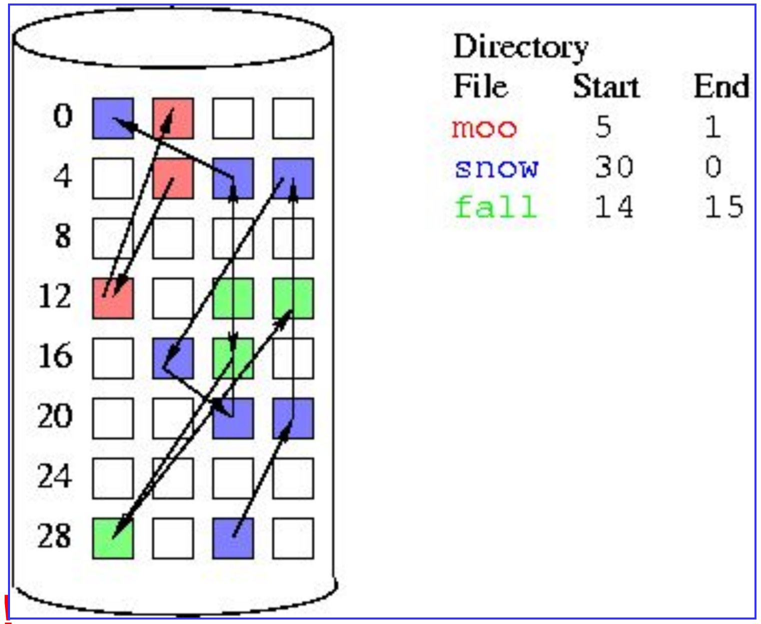


Directory:

file	start	length
moo	5	3
snow	22	7
fall	12	4

Linked Allocation

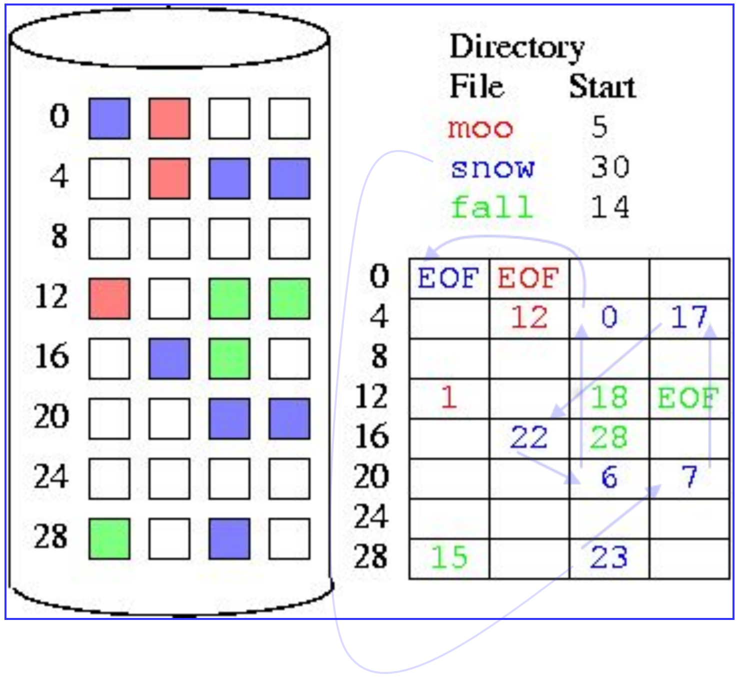
- Directory stores link of start and end block (optionally)
- Pointer in block store link to next block
- Advantages
  - Solves external fragmentation problems
- Disadvantages
  - Not suited for direct access of files (all pointers need to be accessed)
  - Pointer needs to be stored .. **overheads!!**
    - Overheads reduced by using clusters (ie. cluster of sequential blocks associated with one pointer)
  - Reliability.
    - If a pointer is damaged (or lost), rest of file is lost.
    - A bug in the OS may result in a wrong pointer being picked up.



FAT File

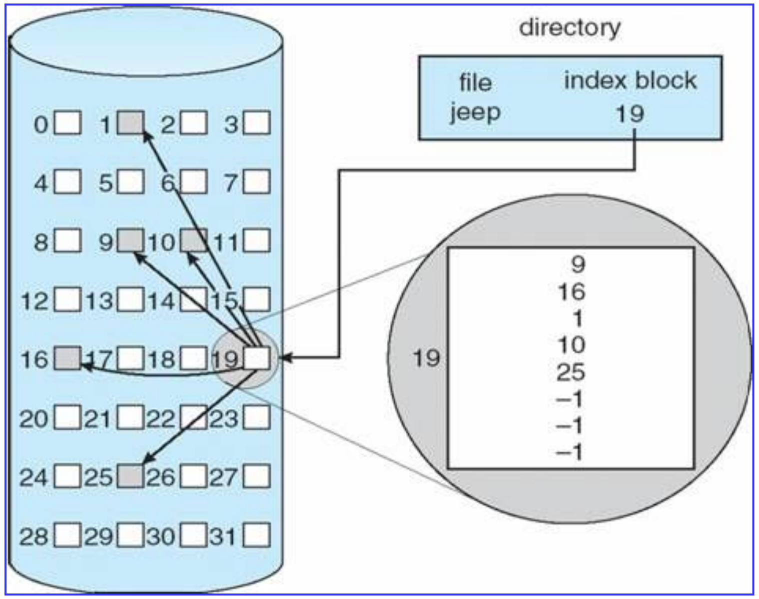
(a variation linked allocation scheme)

- Invented by Marc McDonald and Bill Gates
- FAT is a table that
  - contains one entry for each block
  - and is indexed by block number.
- Files represented by linking pointers in the FAT
- FAT table generally cached
- Advantages,
  - Solves direct access problems of linked allocation
  - Easy to grow files
  - Greater reliability
    - A bad block implies only one block is corrupted
- Disadvantages,
  - Volume size determined by FAT size



Indexed Allocation

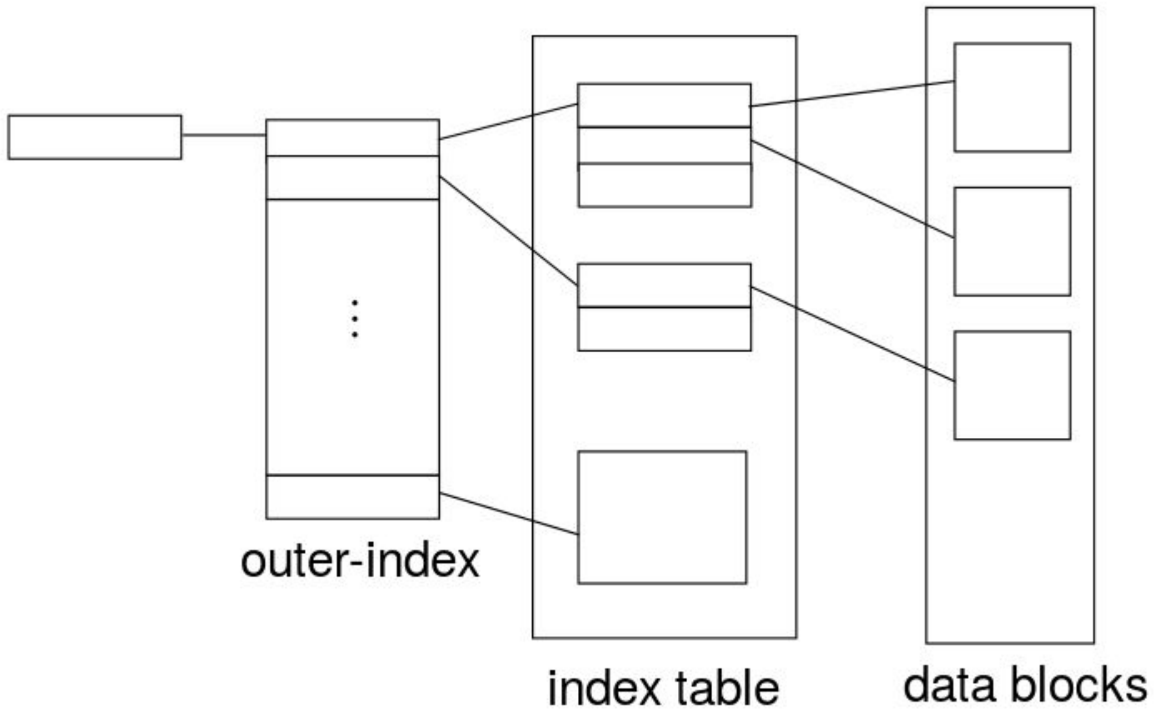
- Advantages,
  - Supports direct access
  - No external fragmentation
  - Easy to grow files
- How large should the index block be?
  - Files typically, one or two blocks long
    - The index block will therefore have only one or two entries
    - A large index block huge wastage
  - A small index block will limit the size of file
    - Need an additional mechanism to deal with large files
- Disadvantage,
  - Sequential access may be slow
    - **May use clusters**



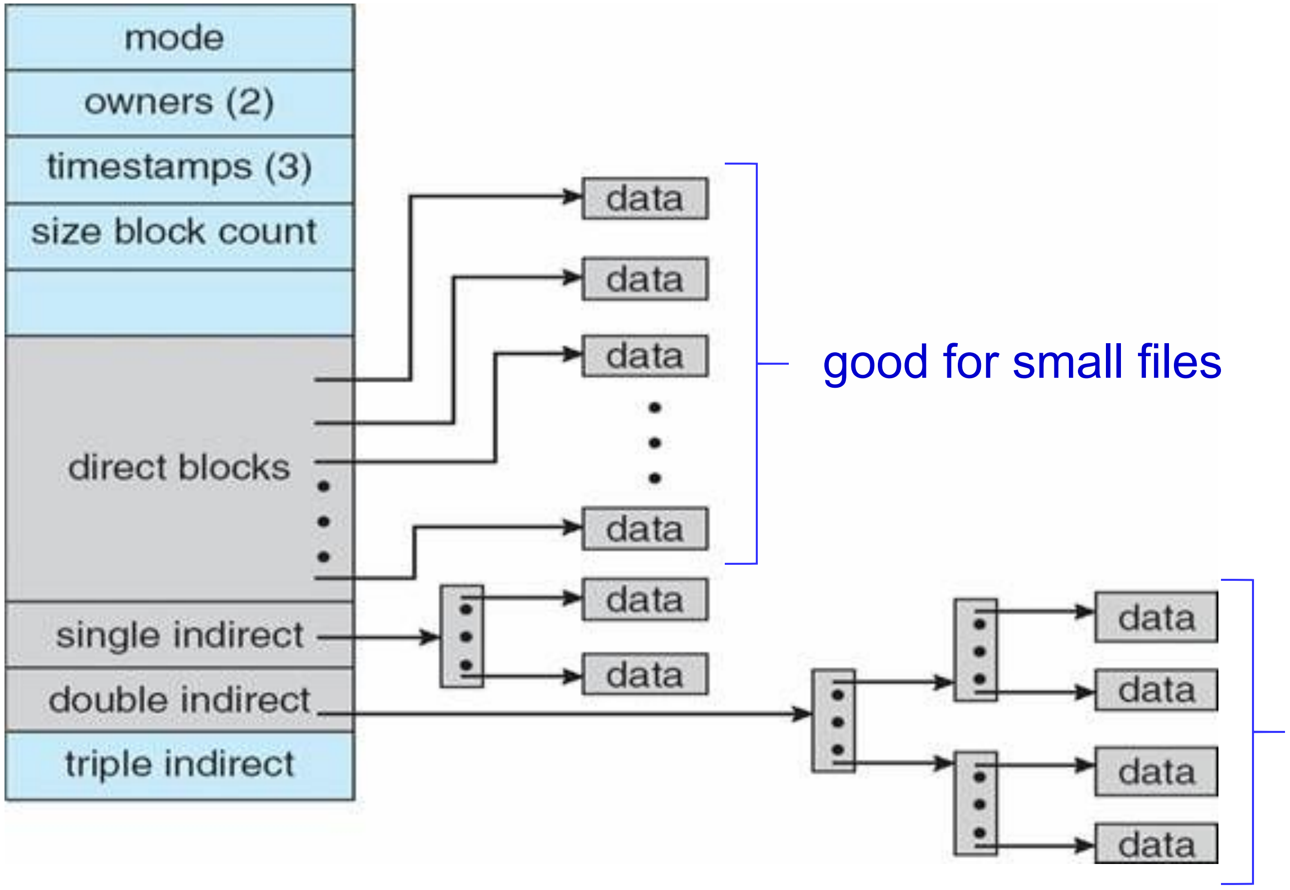
Use disk blocks as index blocks that don't hold file data, but hold pointers to the disk blocks that hold file data.

Multi Level Indexing

- Block index has multiple levels



Multi Level Indexing Linux (ext2/ext3), xv6

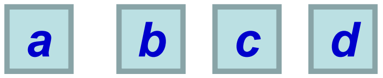


Performance Issues

- Disk cache
  - In disk controller, can store a whole track
- Buffer cache
  - In RAM, maintained by OS
- Synchronous / asynchronous writes
  - Synchronous writes occur in the order n which the disk subsystem receives them.
    - Writes are not buffered
  - In asynchronous writes, data is buffered and may be written out of order
    - Generally used

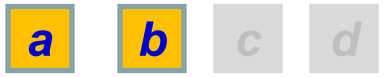
System Crashes

A system call wants to modify 4 blocks in the file system



Block ***a*** corresponds to a file's FCB
Block ***b*** corresponds to the corresponding directory
Block ***c*** and d corresponds to the file data

What happens if the system crashes after ***a*** and ***b*** are written?



Files system state is inconsistent

Block ***a*** indicates that there exists a directory entry for the file
Block ***b*** indicates that the file should be present
BUT the file is not present

Recovery

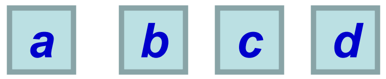
- System crashes can cause inconsistencies on disk
  - Eg. System crashed while creating a file
- Dealing with inconsistencies
  - Consistency checking
    - scan all data on directory and compares with data in each block to determine consistency ….. slow!!
    - fsck in Linux, chkdsk in Windows
      - Checks for inconsistencies and could potentially fix them
    - Disadvantages
      - May not always be successful in repairing
      - May require human intervention to resolve conflicts

Journaling File Systems (JFS)
(Log based recovery techniques)

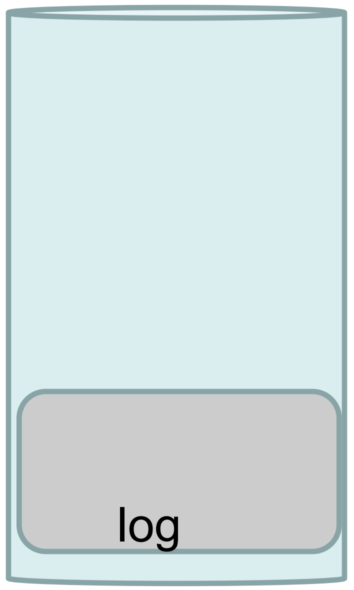
1. When system call is invoked, all metadata changes are written sequentially to a log and the system call returns.
2. Log entries corresponding to a single operation is called transactions.
3. Entries are played sequentially to actual file system structures.
   - As changes are made to file systems, a pointer is updates which actions have completed
   - When an entire transaction is completed; the corresponding log entries are removed
4. If the system crashes, and if log file has one or more entries (it means OS has committed the operations but FS is not updated)
   - continue to play entries; as in step 3

Journaling File Systems (JFS)

- Allocate a small portion of the file system (on disk) as a log
- System call modifies 4 blocks (as before)

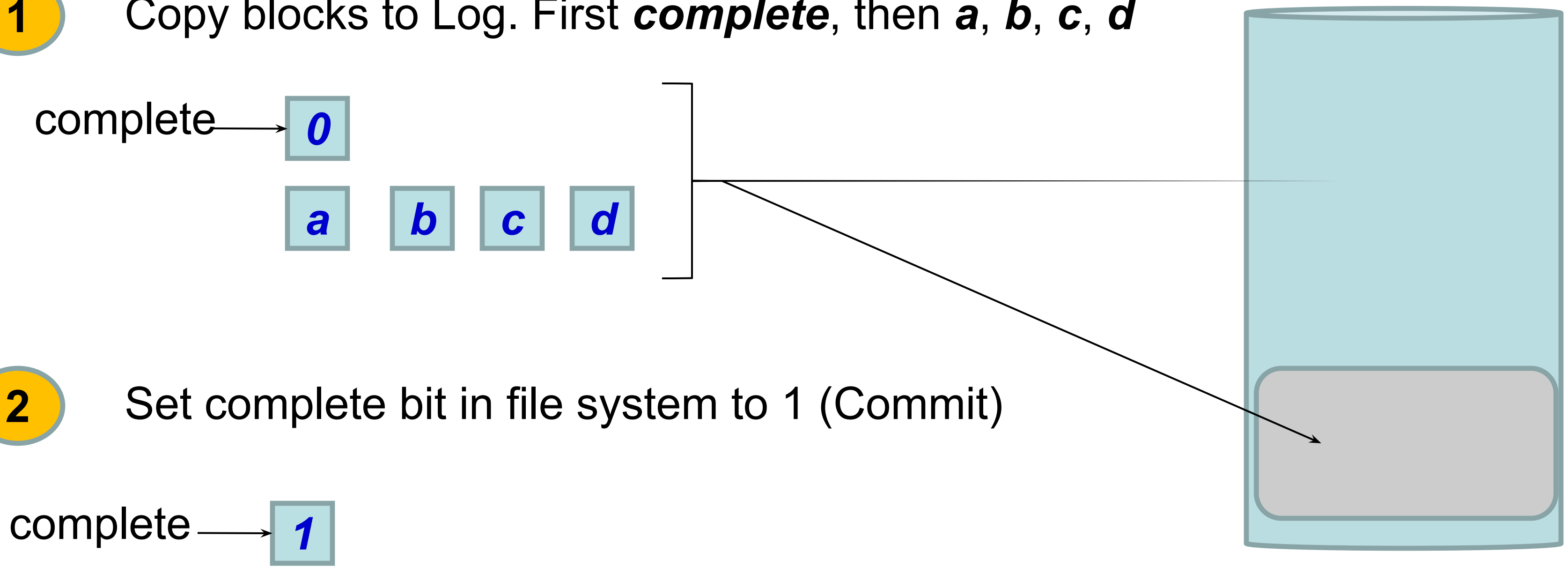


  - This we call as a **transaction**
  - JFS ensures that this transaction is done atomically
    (either all 4 blocks are modified or none)



JFS working (1)

1 Copy blocks to Log. First ***complete***, then ***a***, ***b***, ***c***, ***d***



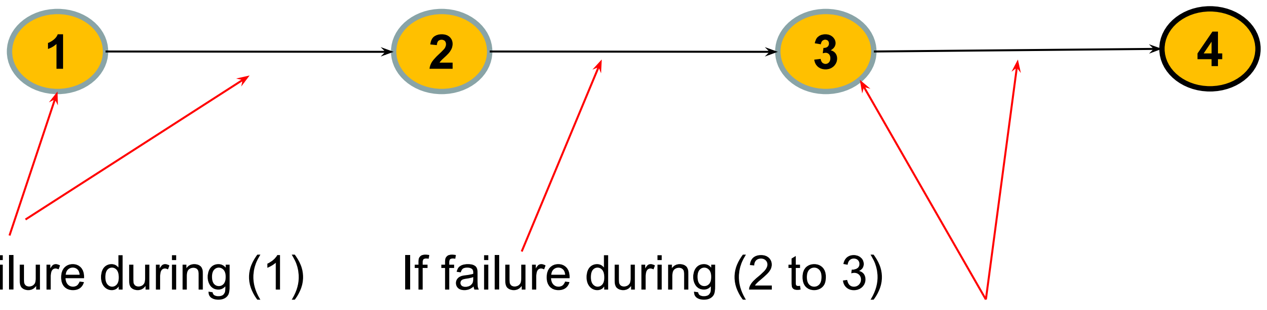
2 Set complete bit in file system to 1 (Commit)

3 Copy from Log to file system

4 Set complete bit in file system to 0
Remove ***a***, ***b***, ***c***, ***d*** from the log

JFS Working (2)

1. Copy blocks to Log. First ***complete***, then ***a***, ***b***, ***c***, ***d***
2. Set complete bit in file system to 1
3. Copy from Log to file system
4. Set complete bit in file system to 0. Remove ***a***, ***b***, ***c***, ***d*** from the log



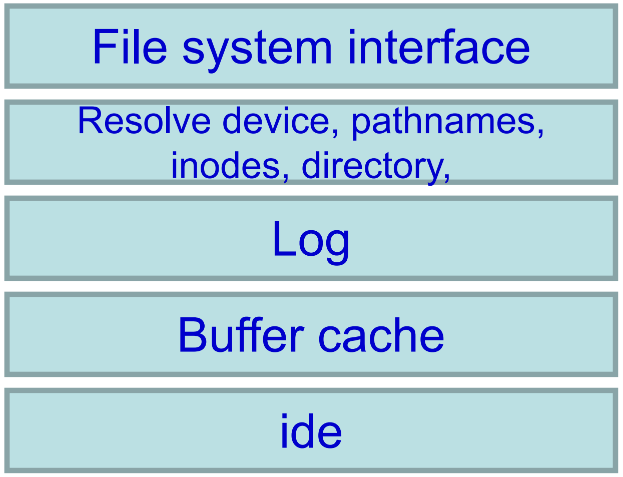
If failure during (1)
on restart, C = 0
Ignore transaction
update

If failure during (2 to 3)
On restart, C = 1
Proceed with (3) as
usual

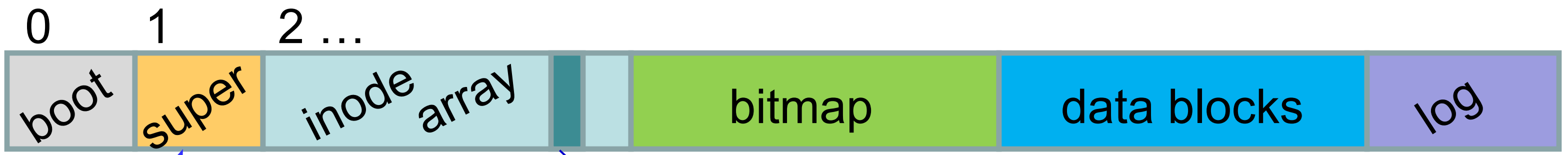
If failure during (3) or between (3 to 4)
On restart, C = 1
Redo (3) from
beginning

xv6 File System

xv6 File System Layers



On disk file system format



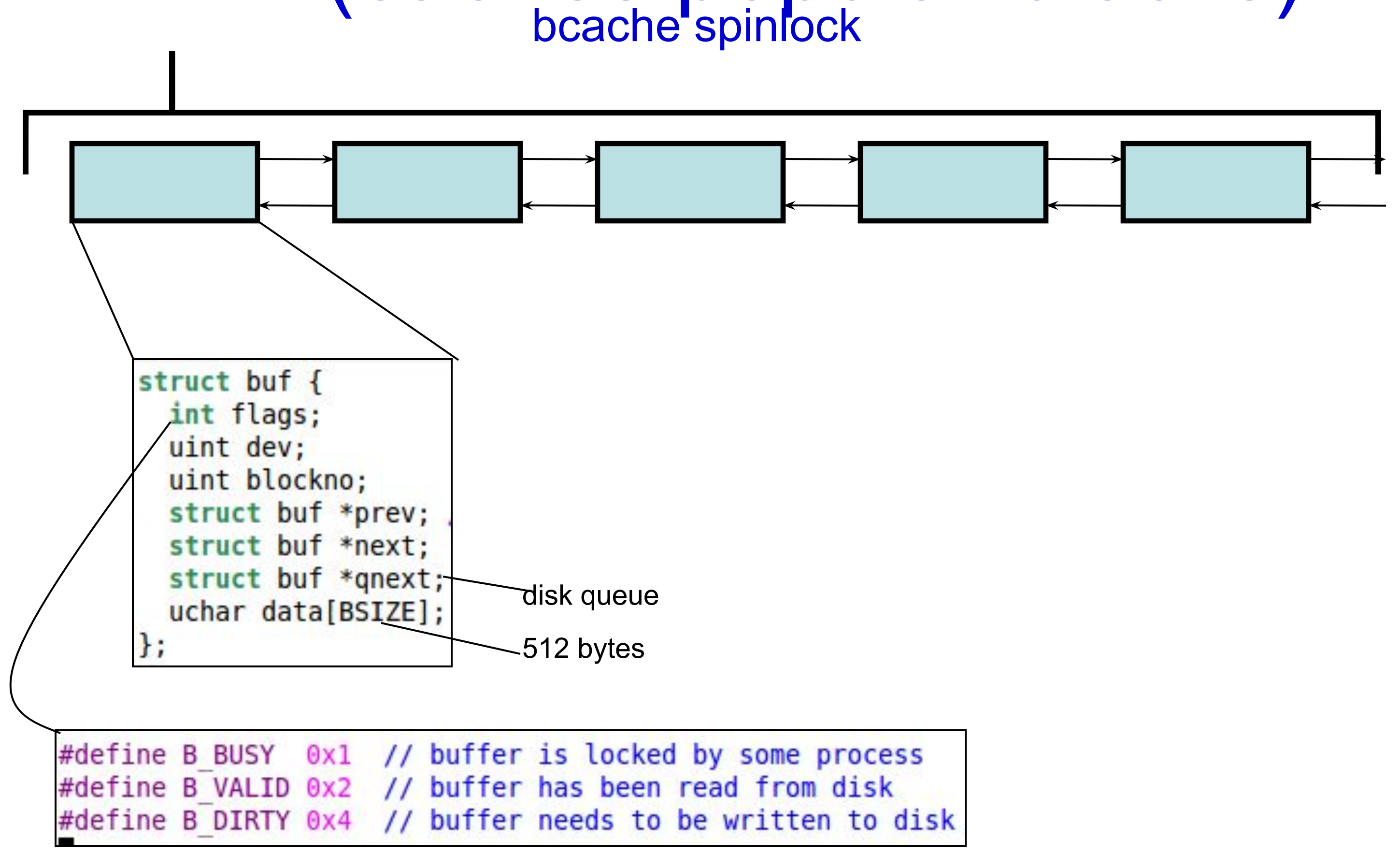
```
struct superblock {
  uint size;          // Size of file system image (blocks)
  uint nblocks;       // Number of data blocks
  uint ninodes;       // Number of inodes.
  uint nlog;          // Number of log blocks
};
```

Block size 512 bytes

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEV only)
  short minor;          // Minor device number (T_DEV only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

ref: fs.h

Buffer Cache (caches popular blocks)



```
struct buf {
  int flags;
  uint dev;
  uint blockno;
  struct buf *prev;
  struct buf *next;
  struct buf *qnext;
  uchar data[BSIZE];
};
```

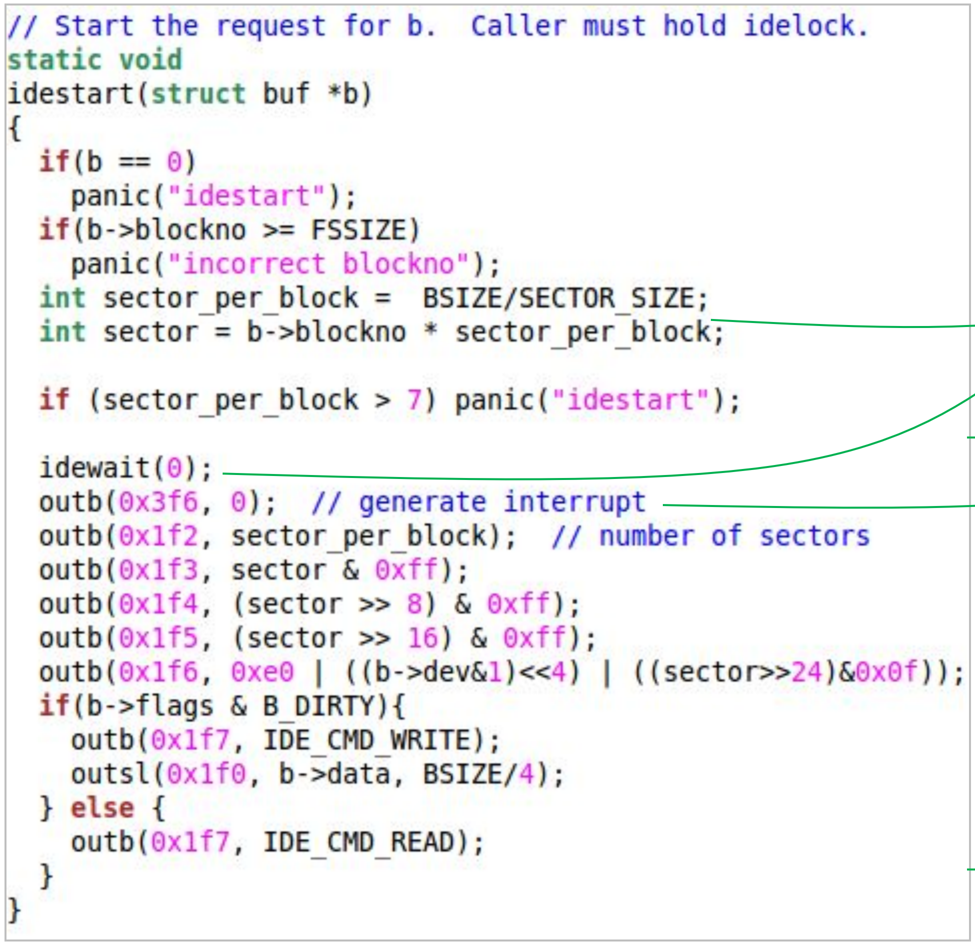
```
#define B_BUSY  0x1  // buffer is locked by some process
#define B_VALID 0x2  // buffer has been read from disk
#define B_DIRTY 0x4  // buffer needs to be written to disk
```

- Size of list is fixed
- Recycling done by LRU

ref: buf.h

IDE disk driver

- Has a queue of bufs that need to be read / written from disk



```c
// Start the request for b.  Caller must hold idelock.
static void
idestart(struct buf *b)
{
  if(b == 0)
    panic("idestart");
  if(b->blockno >= FSSIZE)
    panic("incorrect blockno");
  int sector_per_block =  BSIZE/SECTOR_SIZE;
  int sector = b->blockno * sector_per_block;

  if (sector_per_block > 7) panic("idestart");

  idewait(0);
  outb(0x3f6, 0);  // generate interrupt
  outb(0x1f2, sector_per_block);  // number of sectors
  outb(0x1f3, sector & 0xff);
  outb(0x1f4, (sector >> 8) & 0xff);
  outb(0x1f5, (sector >> 16) & 0xff);
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);
    outsl(0x1f0, b->data, BSIZE/4);
  } else {
    outb(0x1f7, IDE_CMD_READ);
  }
}
```

Convert from block no. to sector no.

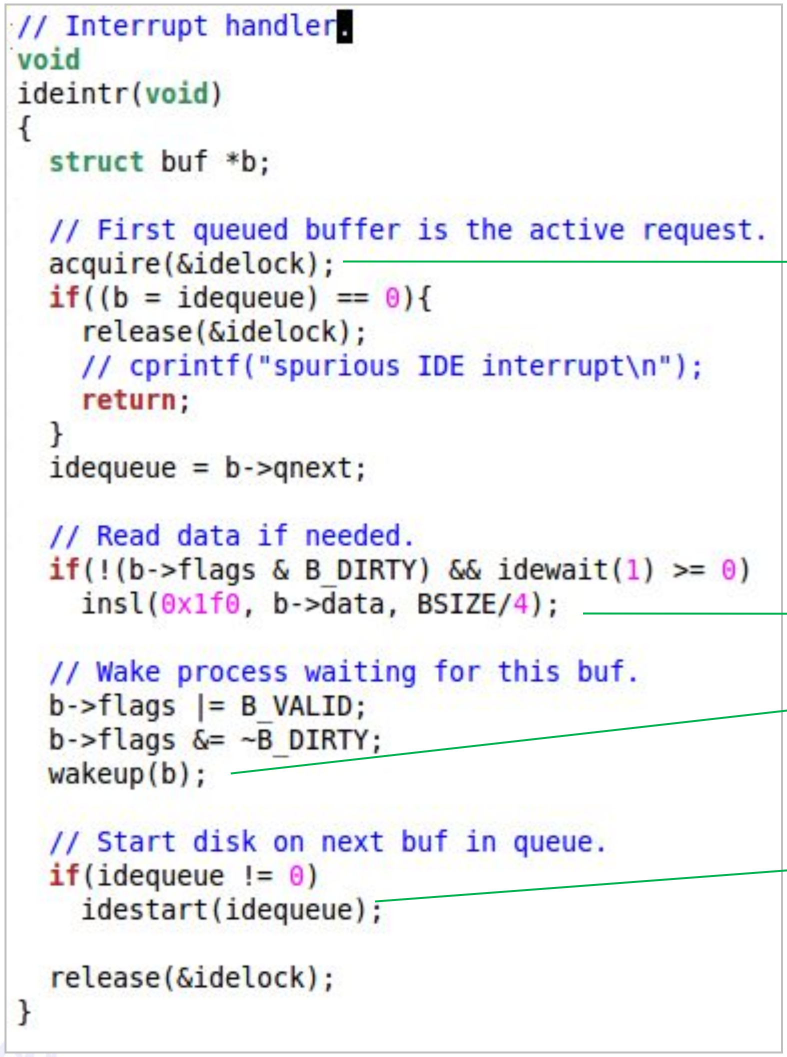
wait until disk is not busy

Interrupt when ready

specify the sector to read / write

ref: ide.c

IDE interrupt handler



```
// Interrupt handler.
void
ideintr(void)
{
  struct buf *b;

  // First queued buffer is the active request.
  acquire(&idelock);
  if((b = idequeue) == 0){
    release(&idelock);
    // cprintf("spurious IDE interrupt\n");
    return;
  }
  idequeue = b->qnext;

  // Read data if needed.
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, BSIZE/4);

  // Wake process waiting for this buf.
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);

  // Start disk on next buf in queue.
  if(idequeue != 0)
    idestart(idequeue);

  release(&idelock);
}
```

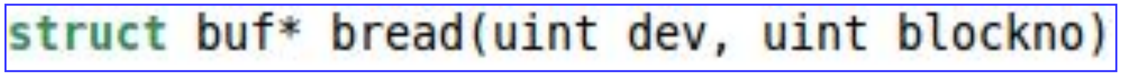
Serialize access to disk
(only the first buf in queue must be serviced)

Loop 512/4 times to read data from disk (if needed)

Wakeup any process that is sleeping for this buffer

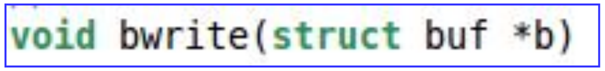
Trigger the next buf on queue to be serviced

Buffer Cache Access



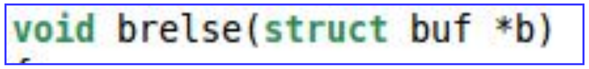
```
struct buf* bread(uint dev, uint blockno)
```

(1) Find block number (blockno) in buffer cache;
(2) If not found, need to query the hard disk and allocate a block from buffer cache. (may need to use LRU policy to allocate block)
(3) Returns pointer to the buf queried for



```
void bwrite(struct buf *b)
```

Trigger disk to read/write buf from cache to disk



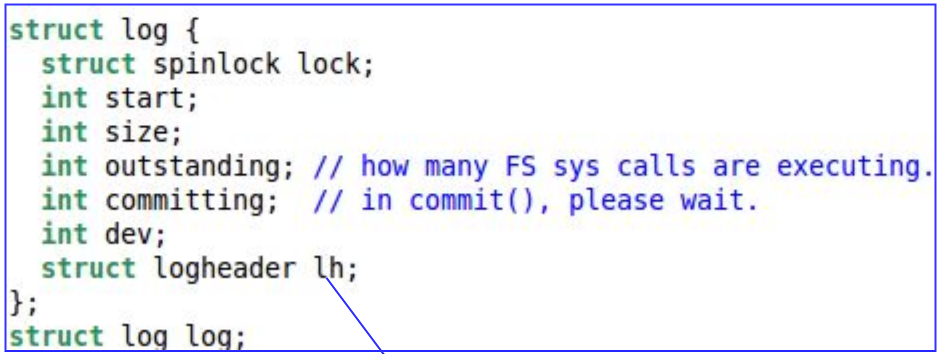
```
void brelse(struct buf *b)
```

Release buffer and mark as most recently used

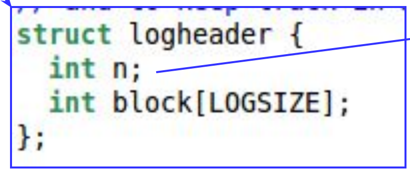
Typical Usage for modifying a block

```
bp = bread(xxx, yyy)
// modify bp[offset]
bwrite(bp)
brelse(bp)
```

Log based Recovery



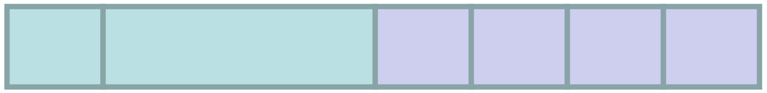
```
struct log {
  struct spinlock lock;
  int start;
  int size;
  int outstanding; // how many FS sys calls are executing.
  int committing;  // in commit(), please wait.
  int dev;
  struct logheader lh;
};
struct log log;
```



```
struct logheader {
  int n;
  int block[LOGSIZE];
};
```

Number of blocks in log

Log Structure on Disk



```
bp = bread(xxx, yyy)
// modify bp[offset]
log_write(bp)
…
…
commit()
```