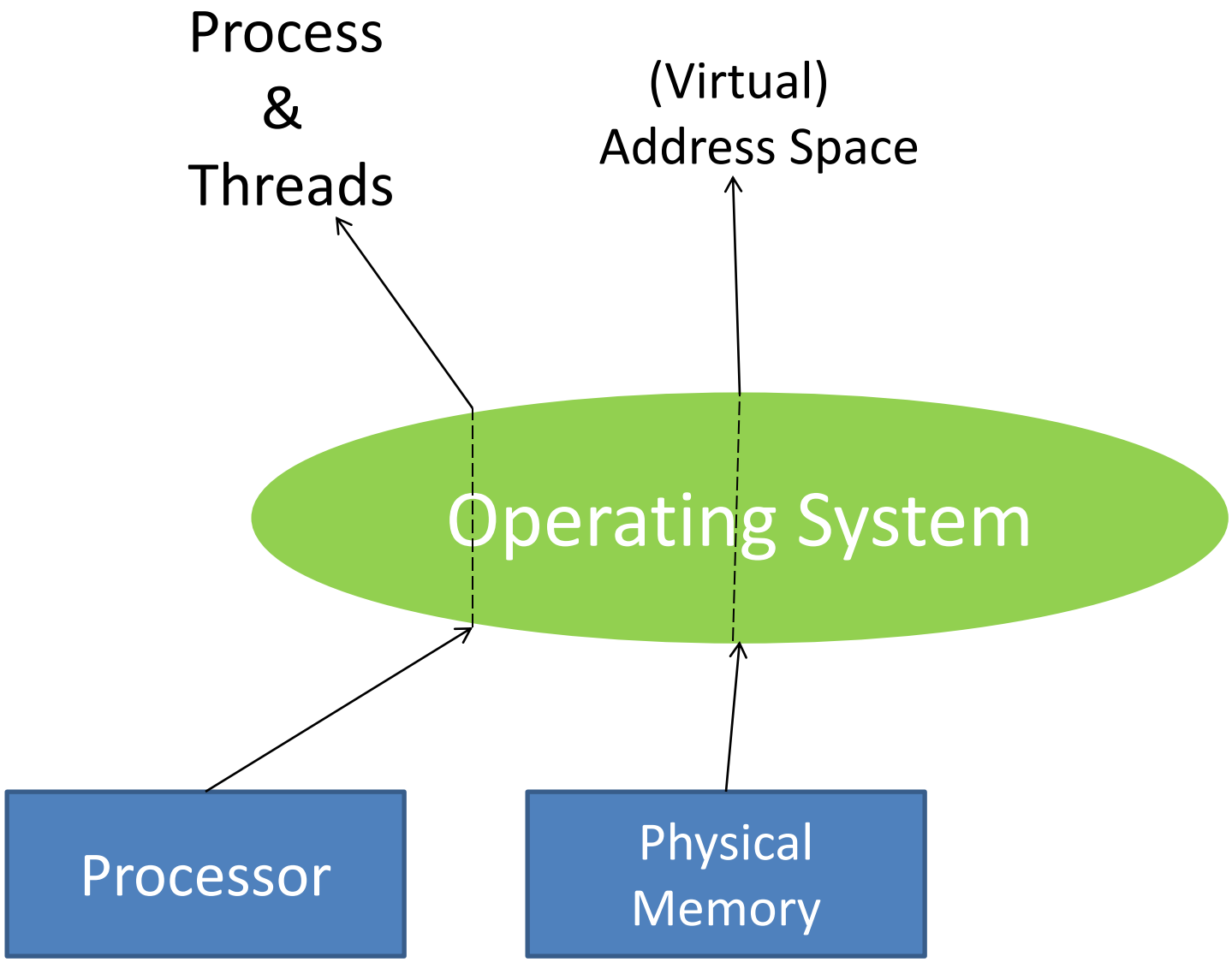
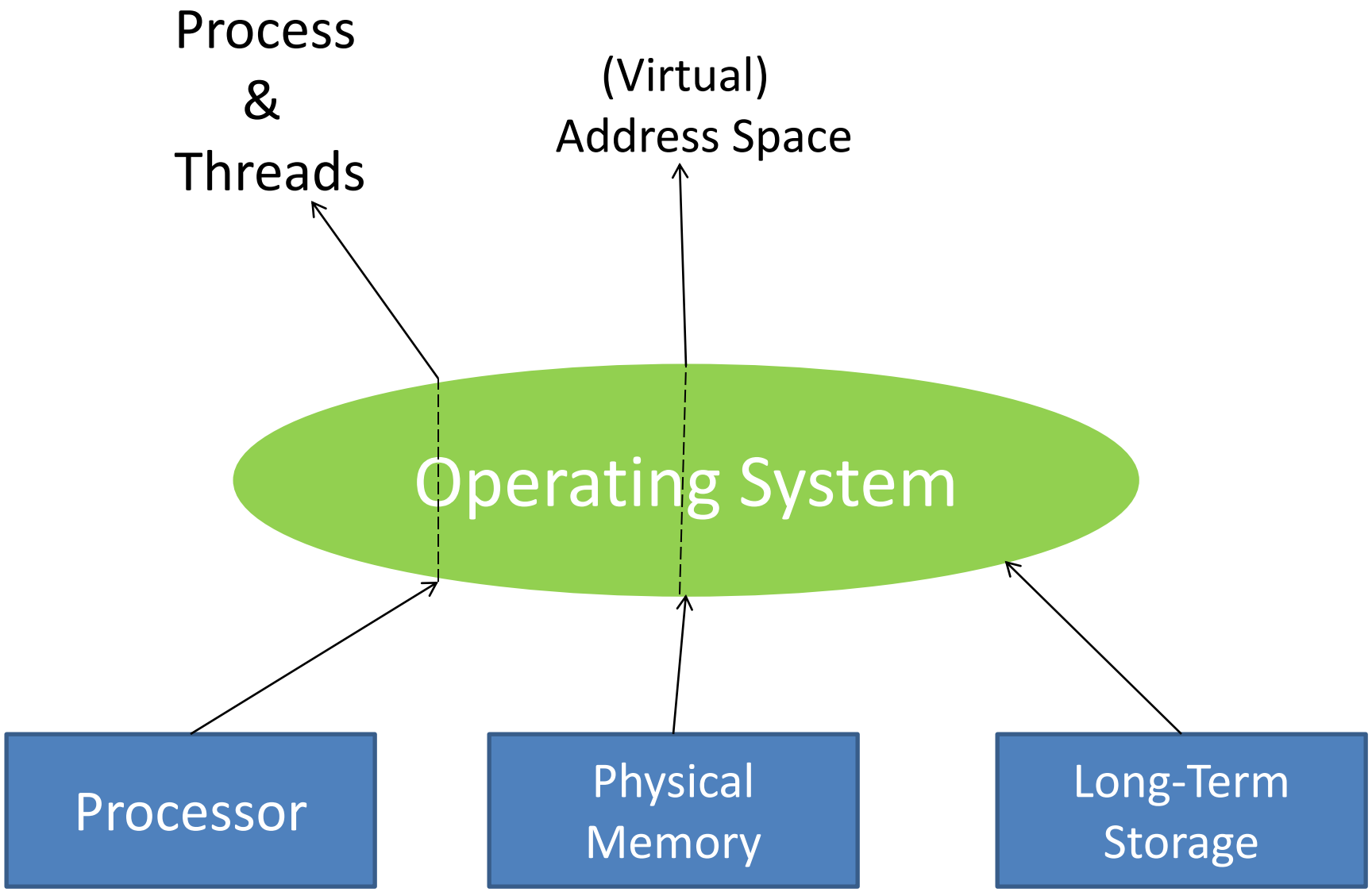
CSCI-GA.2250-001

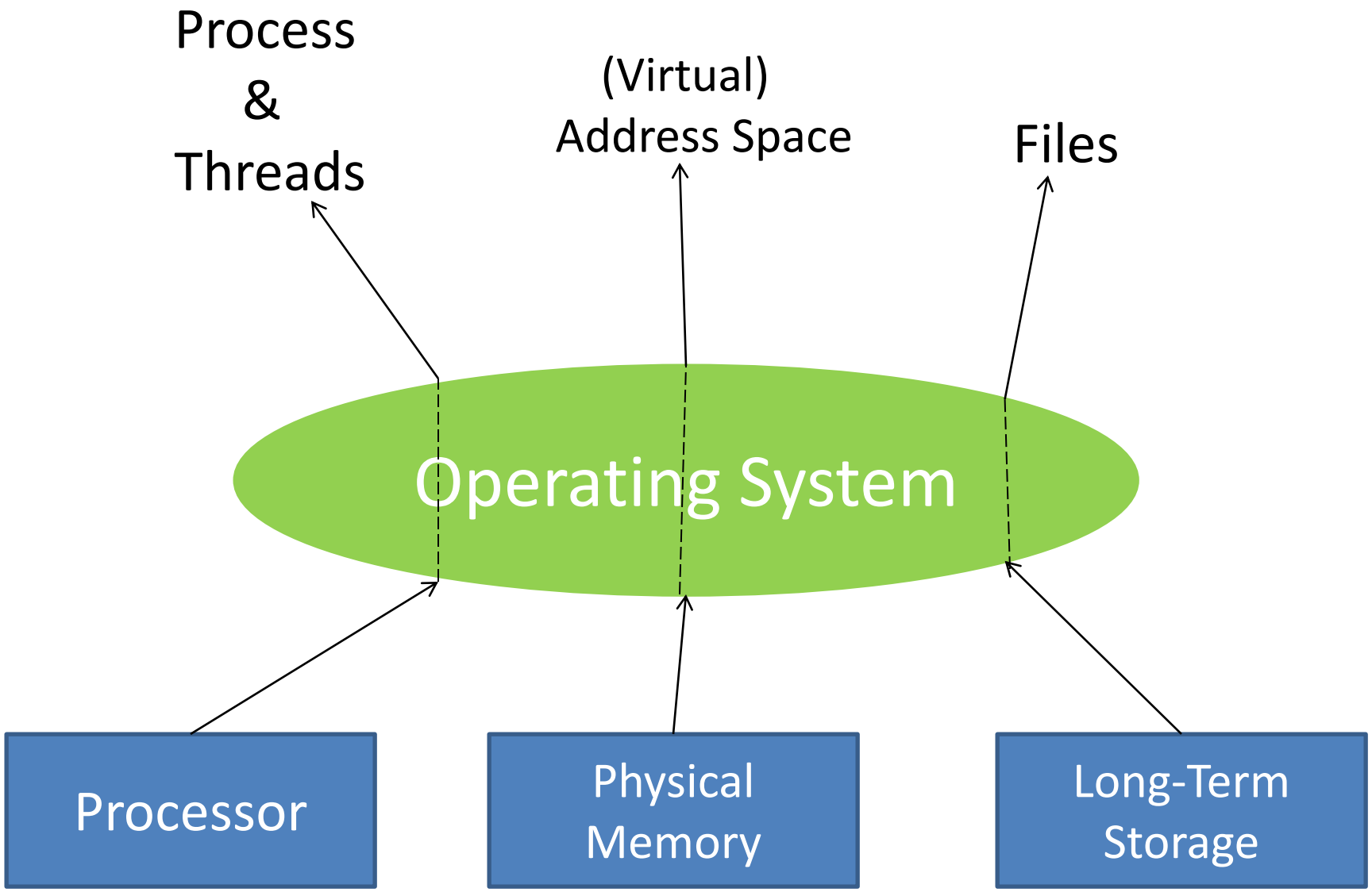
Operating Systems

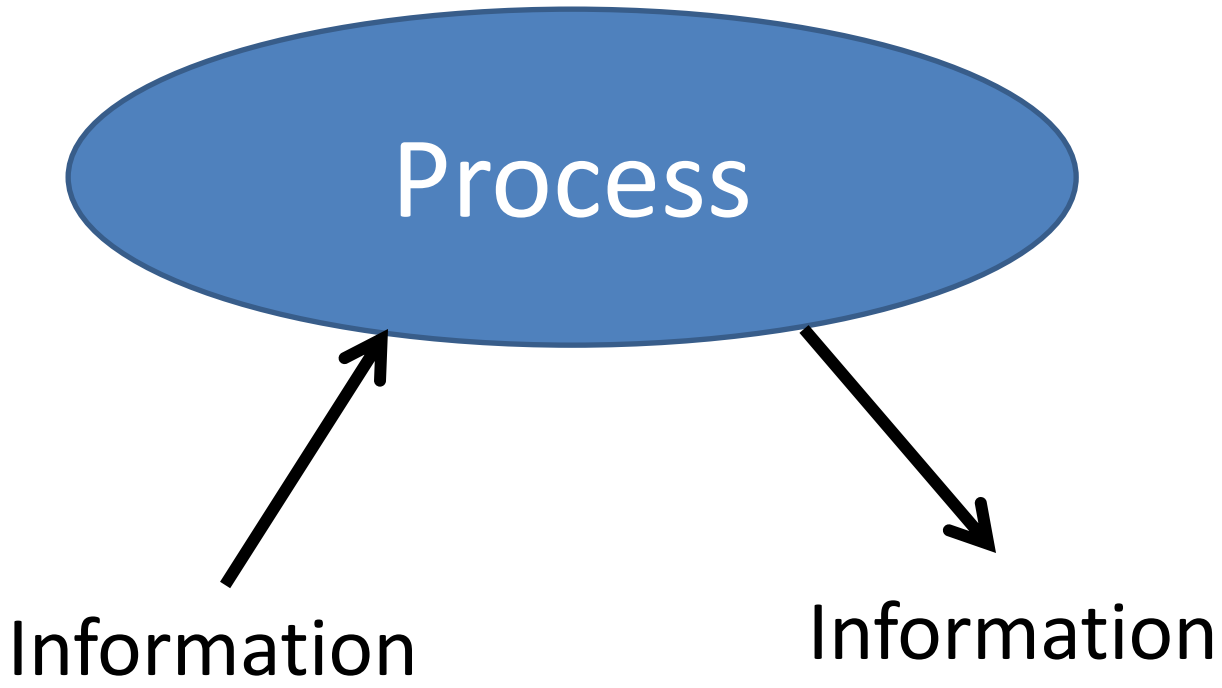
File Systems

Hubertus Franke
frankeh@cs.nyu.edu


Process
&
Threads
(Virtual)
Address Space
Operating System
Processor
Physical
Memory


Process
&
Threads
(Virtual)
Address Space
Operating System
Processor
Physical
Memory
Long-Term
Storage


Process
&
Threads
(Virtual)
Address Space
Files
Operating System
Processor
Physical
Memory
Long-Term
Storage



Is it OK to keep this information only in the process address space?

Shortcomings of Process Address Space

- Virtual address space may not be enough storage for all information
- Information is lost when process terminates, is killed, or computer crashes.
- Multiple processes may need the information at the same time
- Information might be obtained from 3rd sources

Requirements for Long-term Information Storage

- Store very large amount of information
- Information must survive the termination of the process using it
- Multiple processes must be able to access the information concurrently

Files

- Data collections created by users
- The File System is one of the most important parts of the OS to a user
- Desirable properties of files:

Long-term existence

- files are stored on disk or other secondary storage and do not disappear when a user logs off

Sharable between processes

- files have names and can have associated access permissions that permit controlled sharing

Structure

- files can be organized into hierarchical or more complex structure to reflect the relationships among files

Files

- Logical units of information created by processes
- Used to model disks instead of RAM (memory)
- Information stored in files must be persistent (i.e. not affected by processes creation and termination)
- Managed by OS
- The part of OS dealing with files is known as the file system

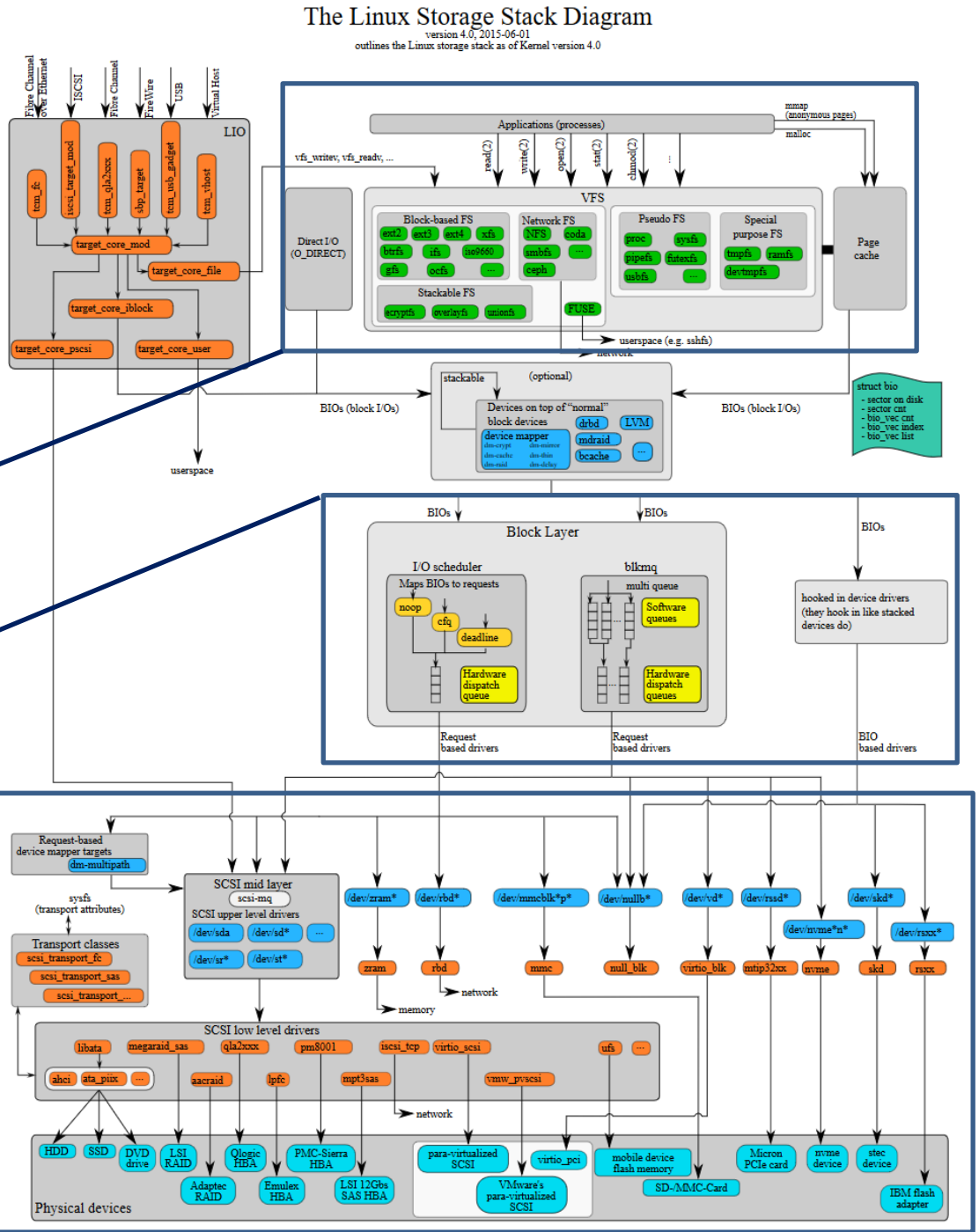
Full Linux I/O Stack

File System

IO Schedulers

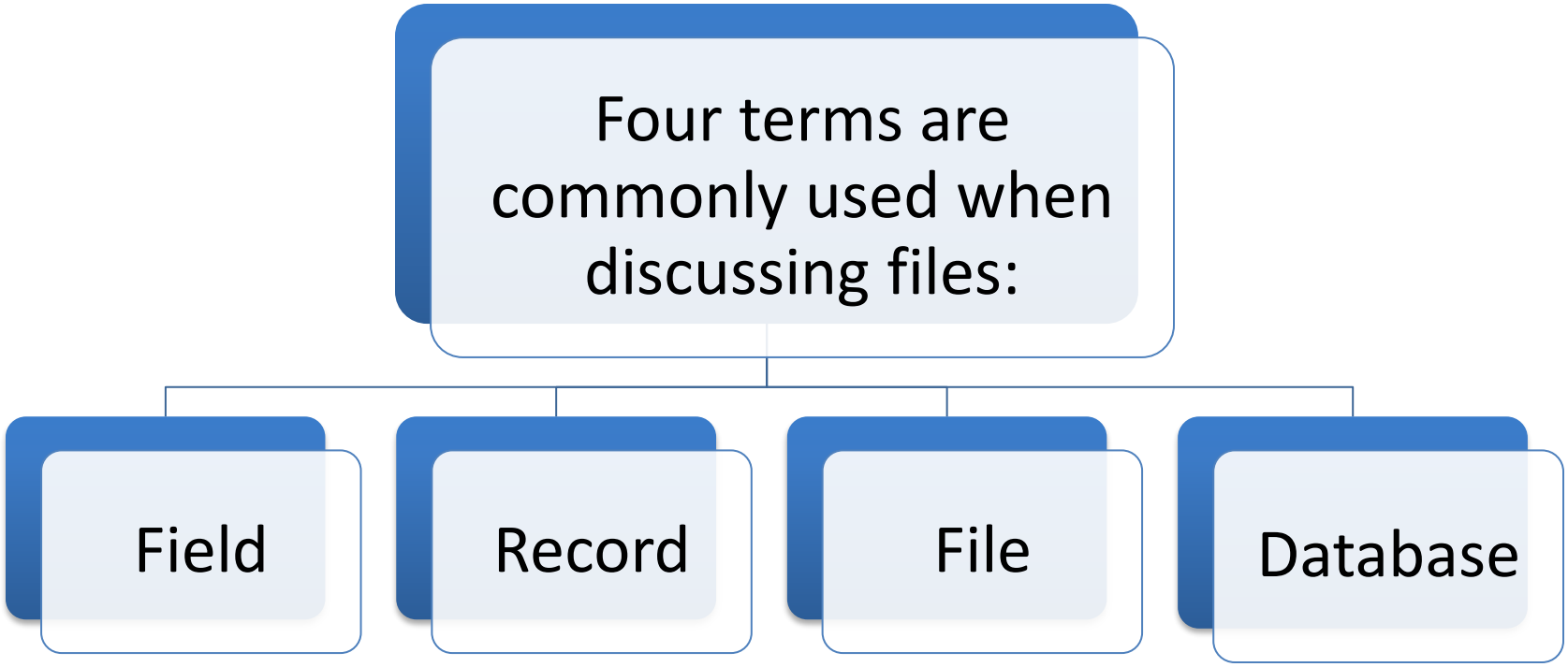
Device Drivers

Filesystem maps file content and hierarchy to blocks and I/O requests



File Structure
(a walk down memory lane)

From the early days: most of terms related to databases



Structure Terms

Field

- collection of related data
- basic element of data
- contains a single value
- fixed or variable length

Record

- collection of related data
- collection of related fields that can be treated as a unit by some application program
- fixed or variable length

Database

- collection of related data
- relationships among elements of data are explicit
- designed for use by a number of different applications
- consists of one or more types of files

File

- collection of similar records
- treated as a single entity
- may be referenced by name
- access control restrictions usually apply at the file level

Example: (1) Telephone Books
(2) Airlines

Issues

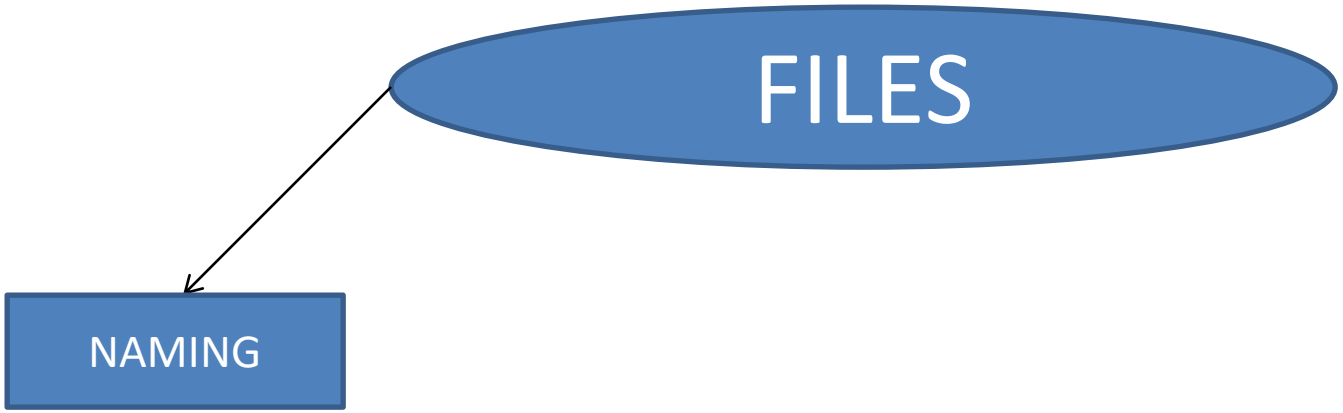
- How do you find information?
- How do you keep one user from reading another user's data?
- How do you assign files to disk blocks?
- How do you know which disk blocks are free?

Files from The User's point of View

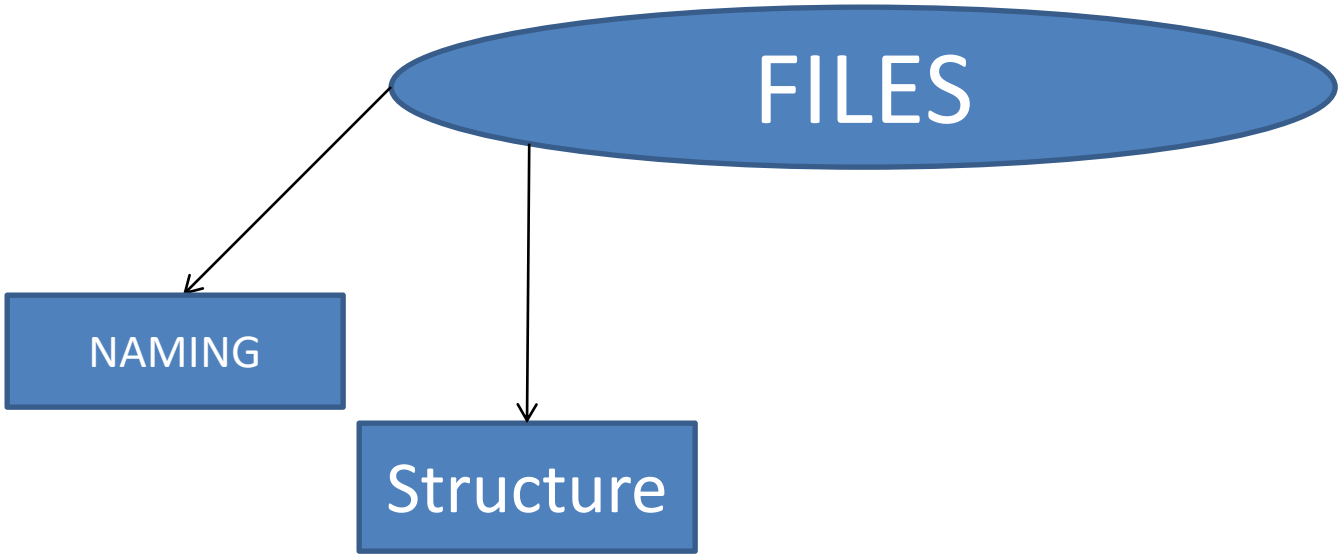
- Files
  - Naming
  - Structure
  - Types
  - Access
  - Attributes
  - Operations
- Directories
  - Single-level
  - Hierarchical
  - Path names
  - Operations

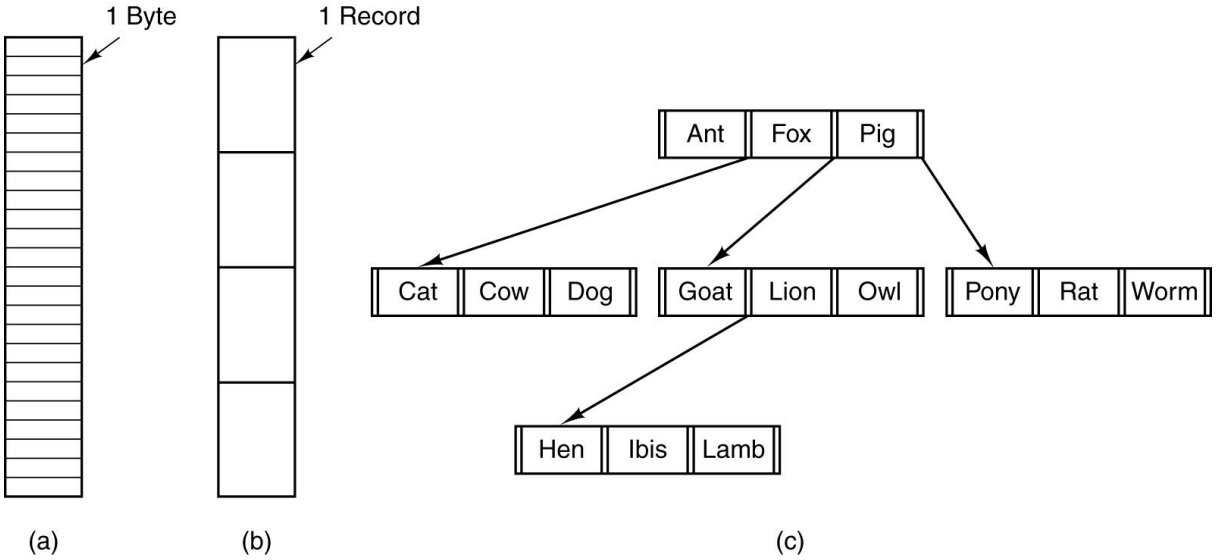
File Systems

- Provide a means to store data organized as files as well as a collection of functions that can be performed on files
- Maintain a set of attributes associated with the file
- Typical operations include:
  - Create
  - Delete
  - Open
  - Close
  - Read
  - Write



- Shields the user from details of file storage.
- In general, files continue to exist even after the process that creates them terminates.

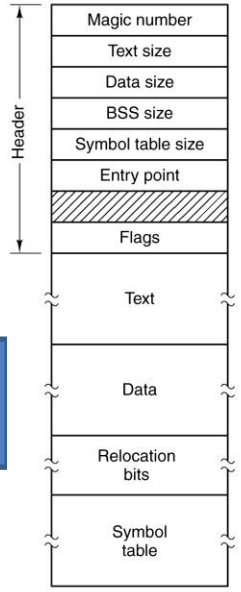


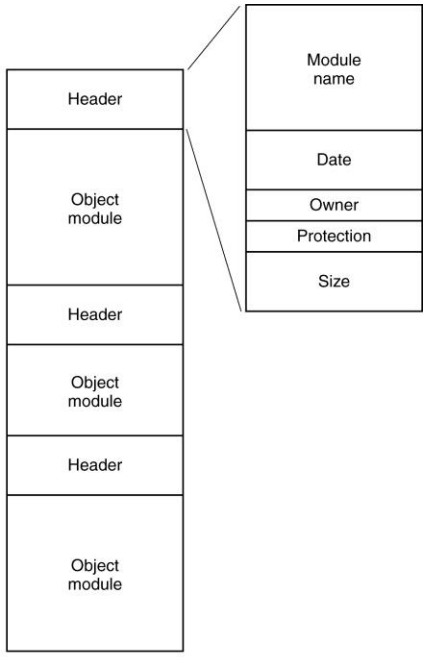


OS view Historical Still Used in Some Mainframes

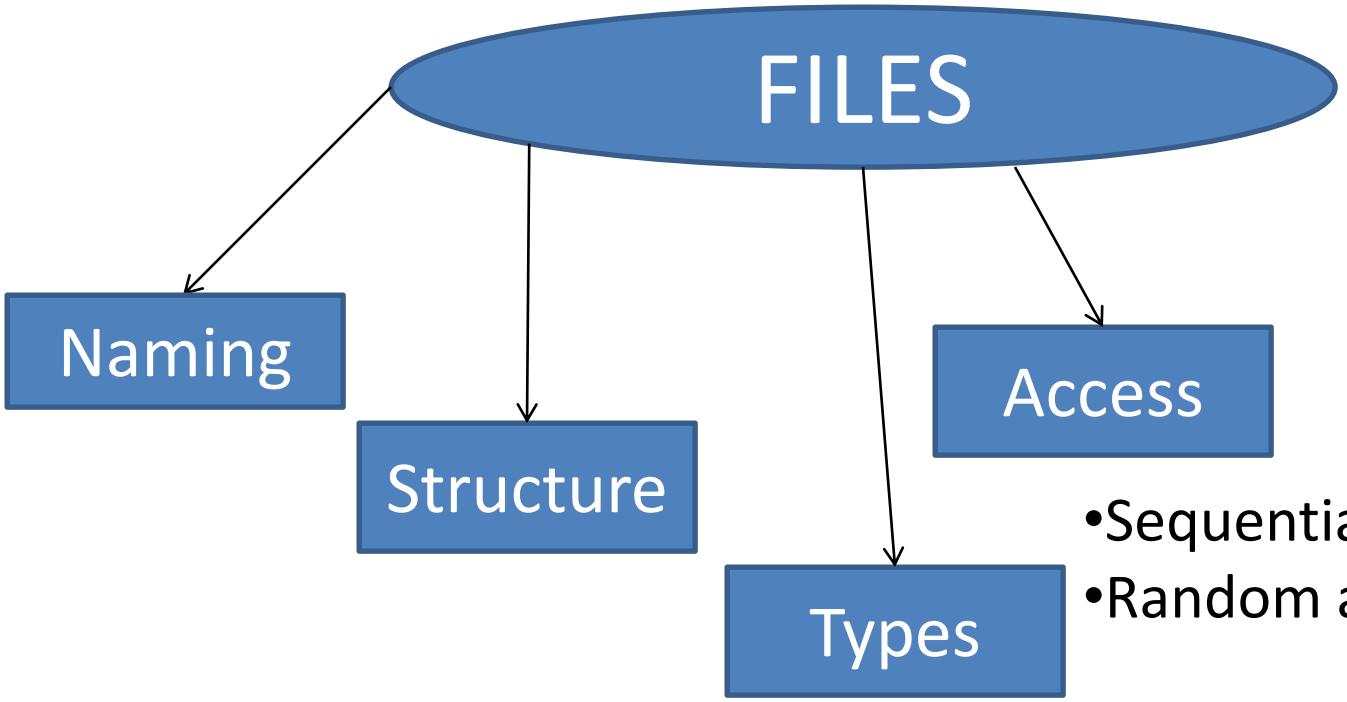
FILES

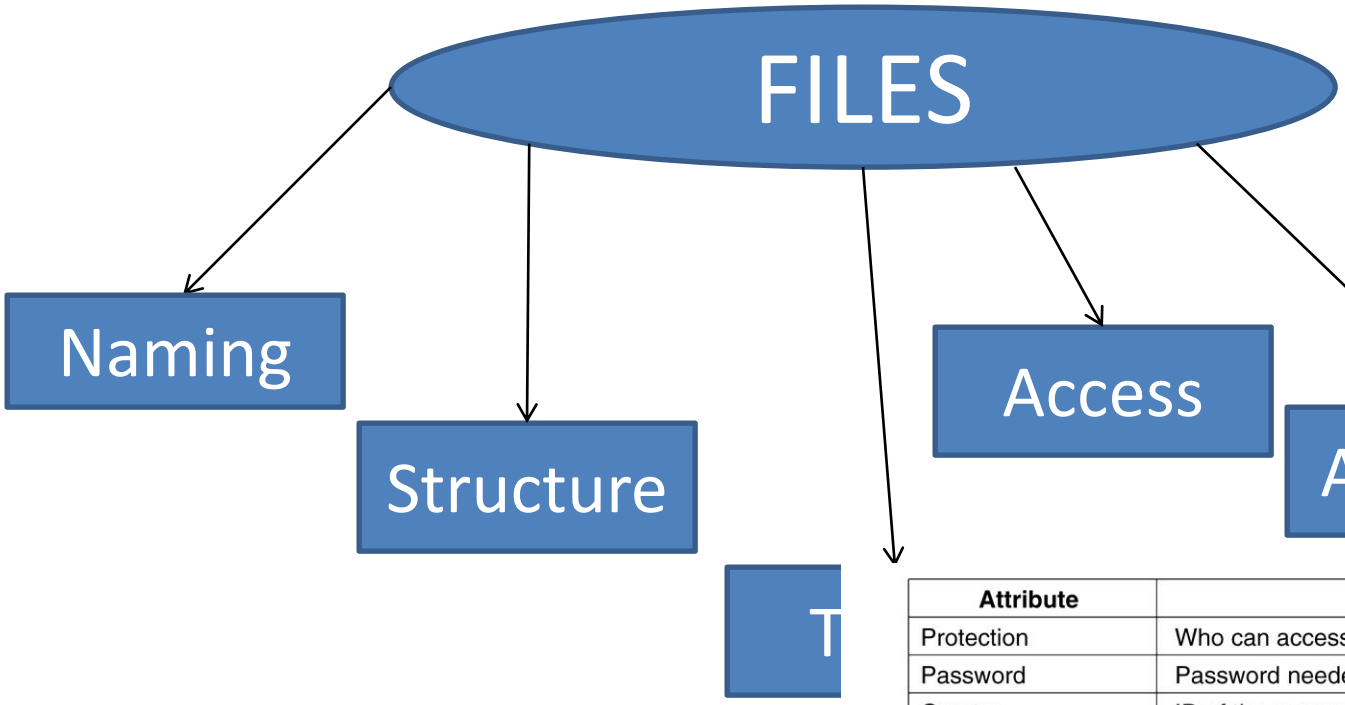
- Naming
- Structure
- Types



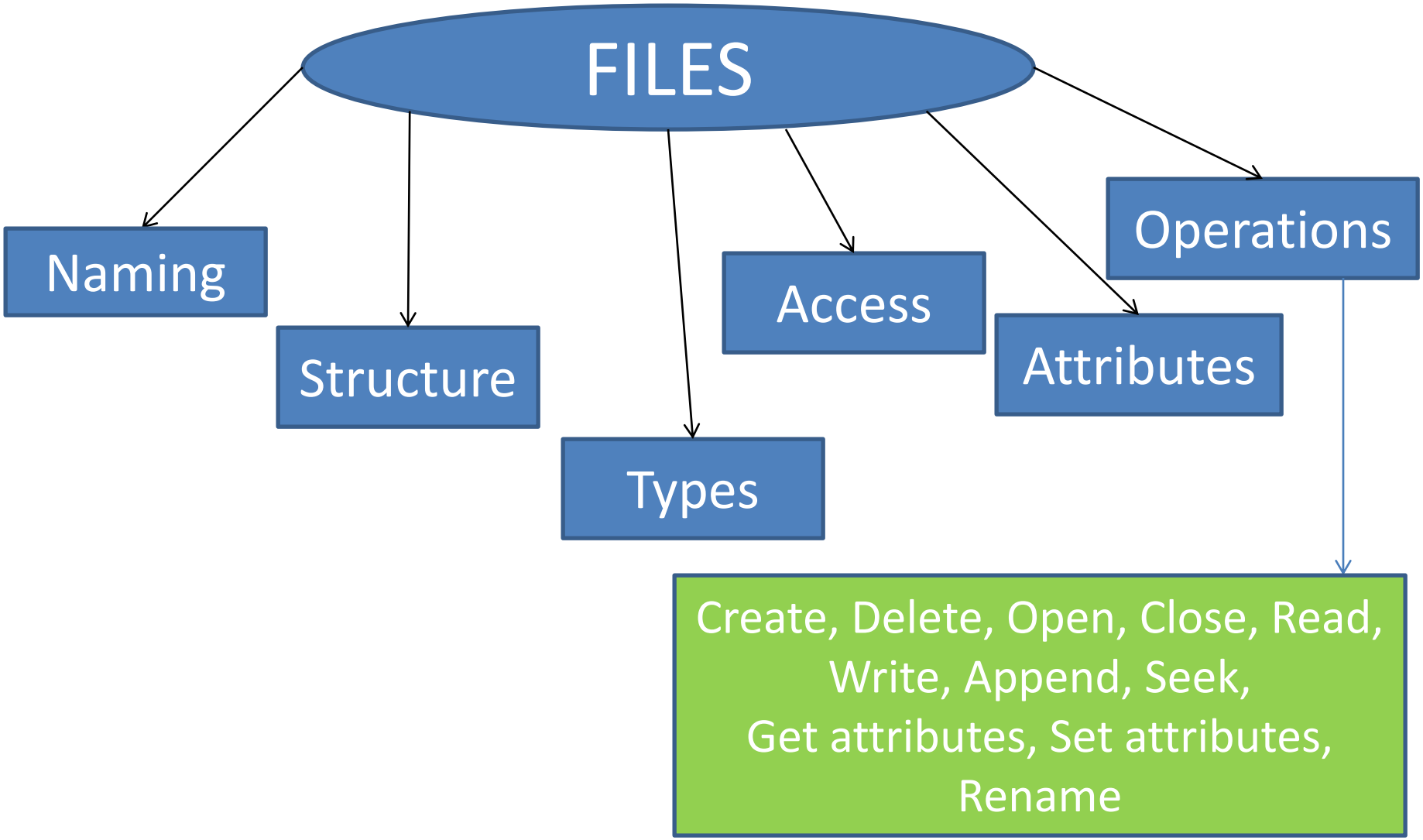


- Regular files
  - ASCII
  - Binary
- Character special
  - to model serial devices (printers, networks, …)
- Block special
  - to model disks


FILES
Naming
Structure
Types
Access
•Sequential
•Random access

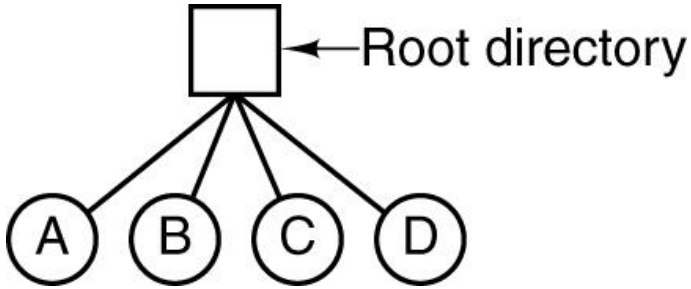


Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to


FILES
Naming
Structure
Types
Access
Attributes
Operations
Create, Delete, Open, Close, Read,
Write, Append, Seek,
Get attributes, Set attributes,
Rename

```
/* File copy program. Error checking and reporting is minimal. */

#include <sys/types.h>                  /* include necessary header files */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);       /* ANSI prototype */

#define BUF_SIZE 4096                   /* use a buffer size of 4096 bytes */
#define OUTPUT_MODE 0700                /* protection bits for output file */

int main(int argc, char *argv[])
{
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];

    if (argc != 3) exit(1);             /* syntax error if argc is not 3 */

    /* Open the input file and create the output file */
    in_fd = open(argv[1], O_RDONLY);    /* open the source file */
    if (in_fd < 0) exit(2);             /* if it cannot be opened, exit */
    out_fd = creat(argv[2], OUTPUT_MODE);  /* create the destination file */
    if (out_fd < 0) exit(3);            /* if it cannot be created, exit */

    /* Copy loop */
    while (TRUE) {
        rd_count = read(in_fd, buffer, BUF_SIZE); /* read a block of data */
    if (rd_count <= 0) break;           /* if end of file or error, exit loop */
        wt_count = write(out_fd, buffer, rd_count); /* write data */
        if (wt_count <= 0) exit(4);     /* wt_count <= 0 is an error */
    }

    /* Close the files */
    close(in_fd);
    close(out_fd);
    if (rd_count == 0)                  /* no error on last read */
        exit(0);
    else
        exit(5);                        /* error on last read */
}
```

Directories: Single-Level Directory Systems



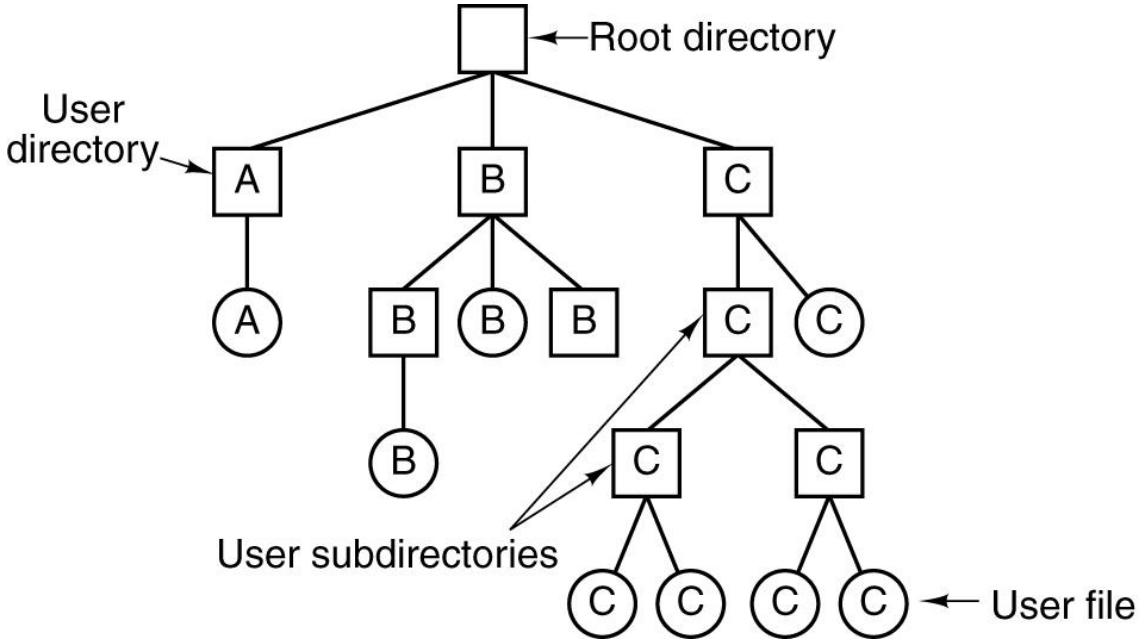
+ Simplicity

-Not adequate for large number of files.

Used in simple embedded devices

Directories: Hierarchical Directory Systems

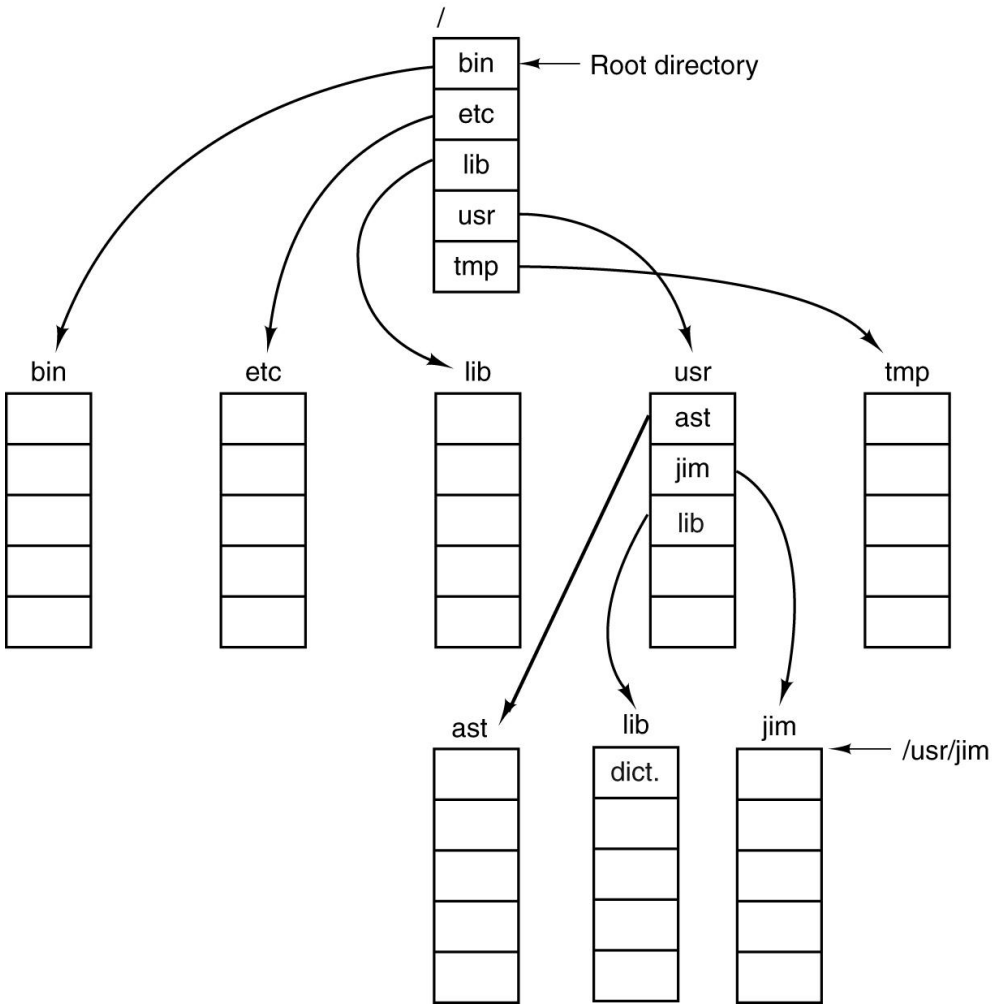
- Group related files together
- Tree of directories



Directories: Path Names

- Needed when directories are used
- Absolute path names
  - Always start at the root
  - A path from the root to the specified file
  - The first character is the separator
- Relative path names
  - Relative to the working directory
  - Each process has its own working directory

Directories: Path Names



Directories: Operations

- More variations among OSes than file operations
- Examples (from UNIX):
  - Create, deleted
  - Opendir, closedir
  - Readdir
  - Rename
  - Link, unlink

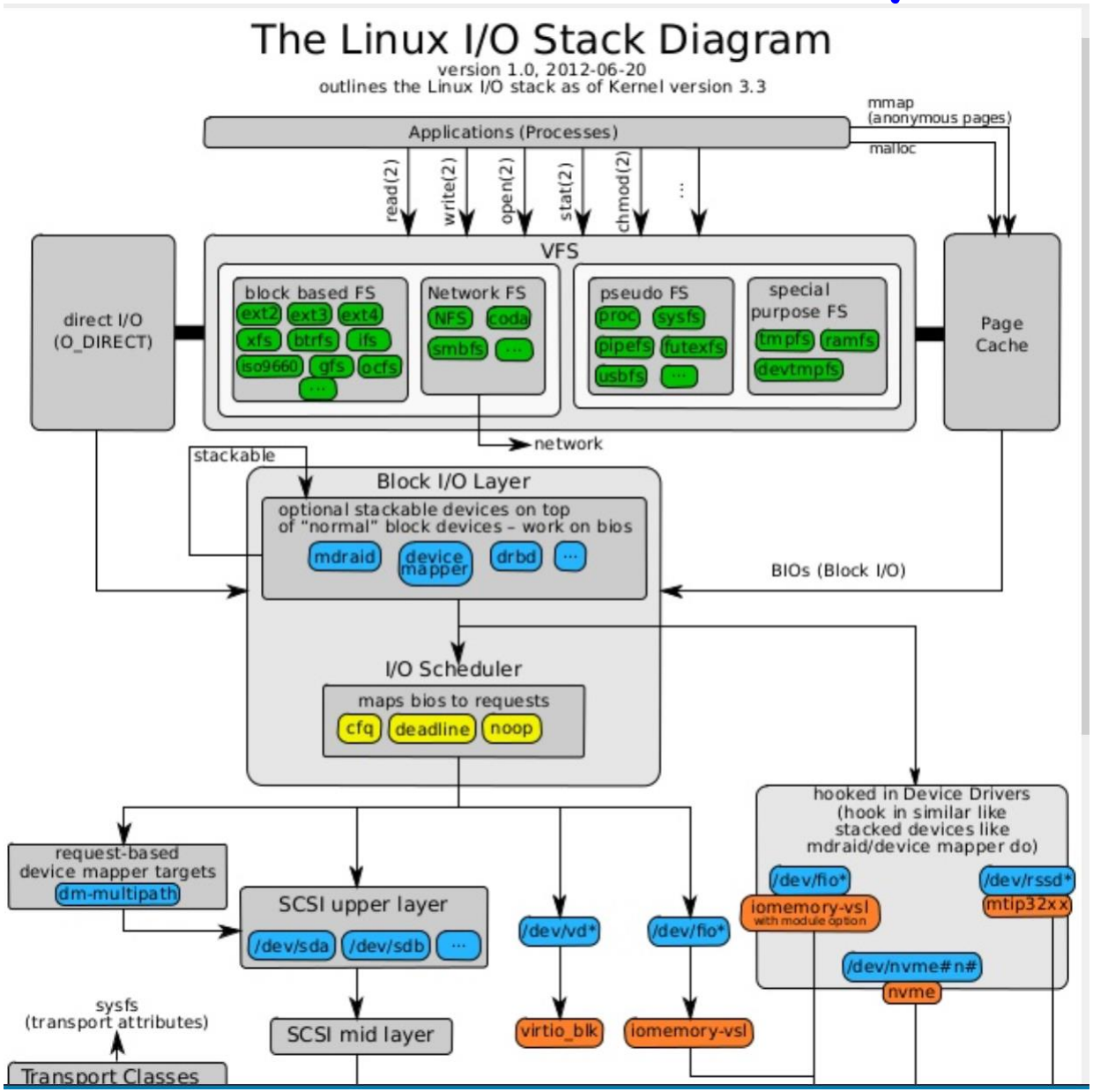
Reminder of the hierachy

Application

Filesystem

Block I/O

Device Drivers



Some Example Syscalls

```
access modes: O_RDONLY, O_WRONLY, or O_RDWR.
              O_CLOEXEC,  O_CREAT, O_DIRECTORY, O_EXCL,
              O_NOCTTY, O_NOFOLLOW, O_TMPFILE, and O_TRUNC. …
```

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);

int creat(const char *pathname, mode_t mode);
```

0x777 (rwx , rwx , rwx)

Intermediate Conclusions

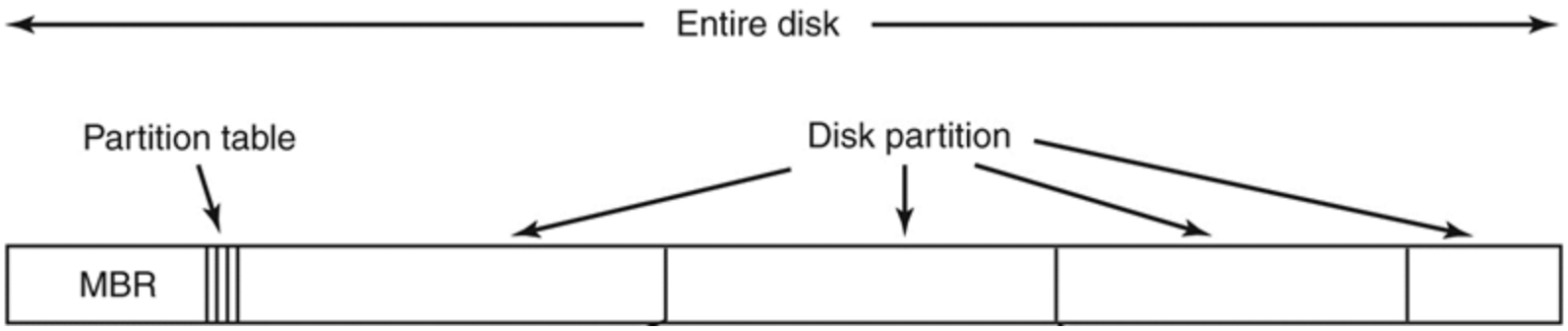
- Files are OS abstraction for storage, same as address space is OS abstraction for physical memory, and processes (& threads) are OS abstraction for CPU.
- So far we discussed files from user perspective.
- Next we will discuss the implementation.

Questions that need to be answered:

- How does the system boot
- How files and directories are stored ?
- How disk space is managed ?
- How to make everything work efficiently and reliable ?

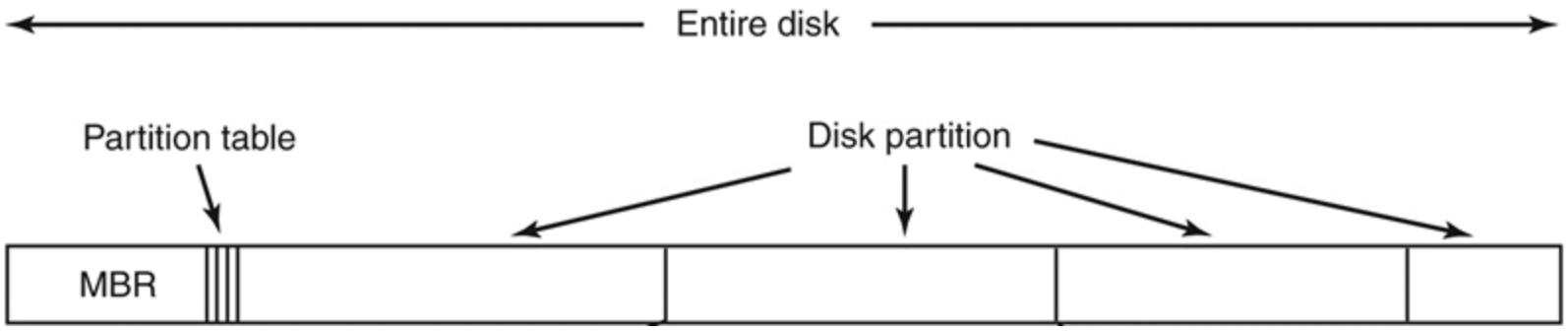
Disk Layout

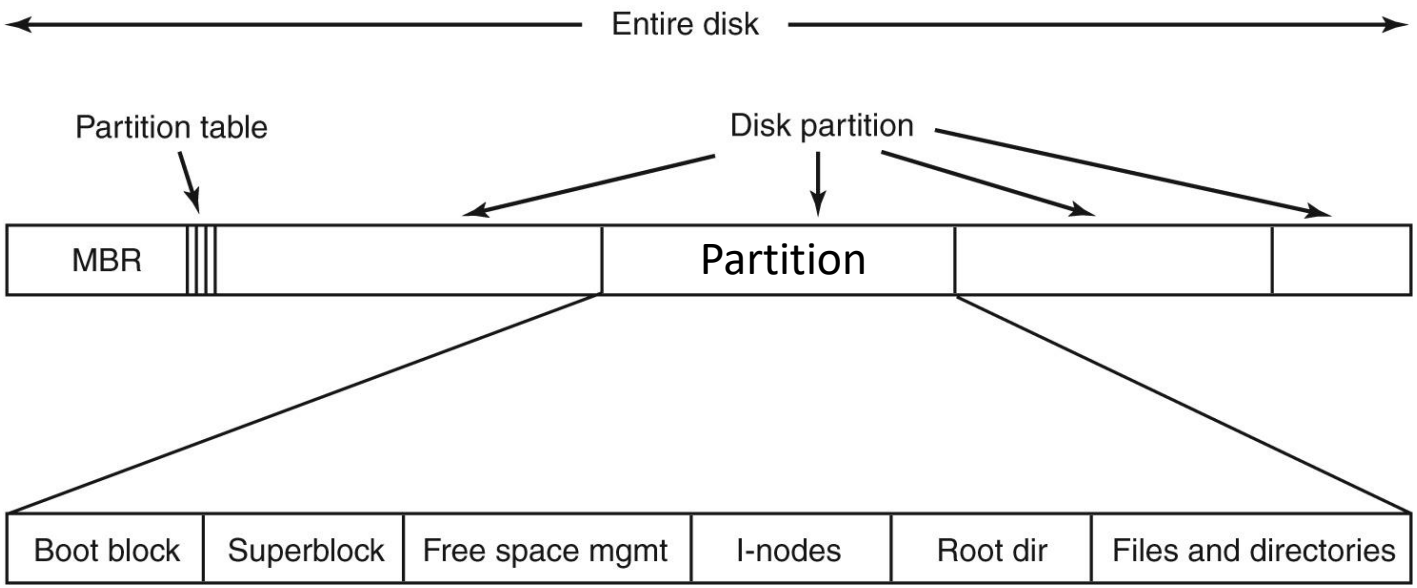
- Stored on disks
- Disks can have partitions with different file systems
- Sector 0 of the disk called MBR (Master Boot Record)
- MBR used to boot the computer
- The end of MBR contains the partition table

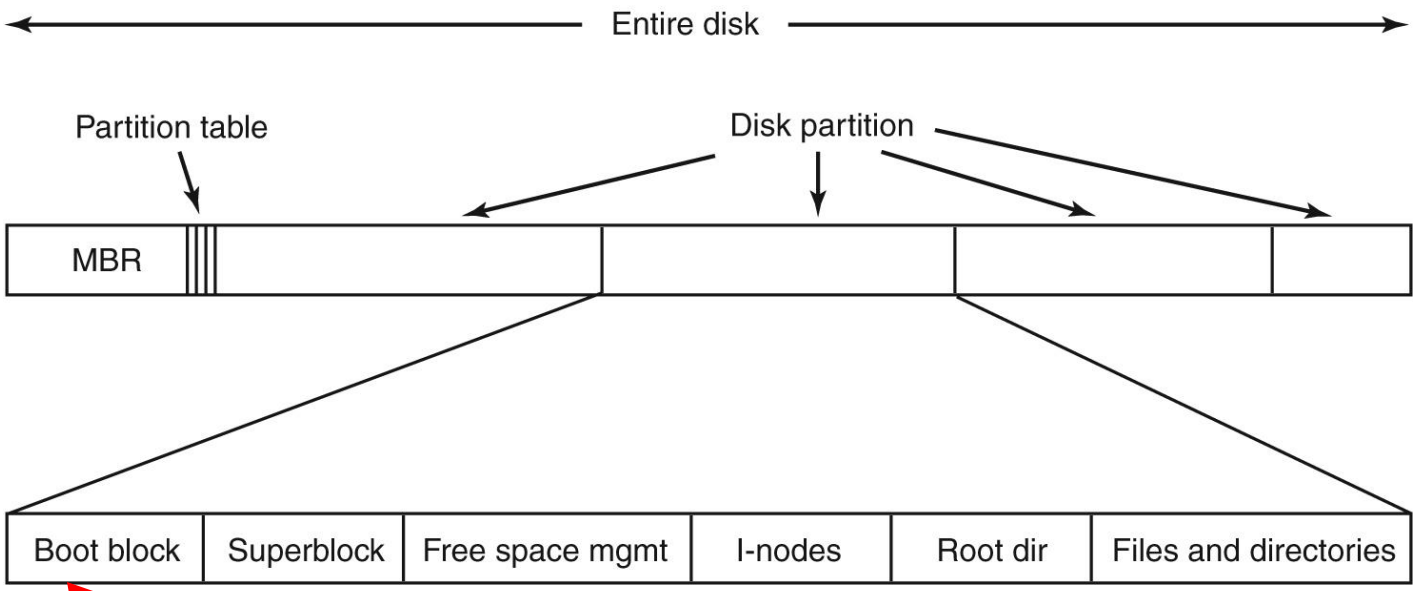


MBR and Partition Table

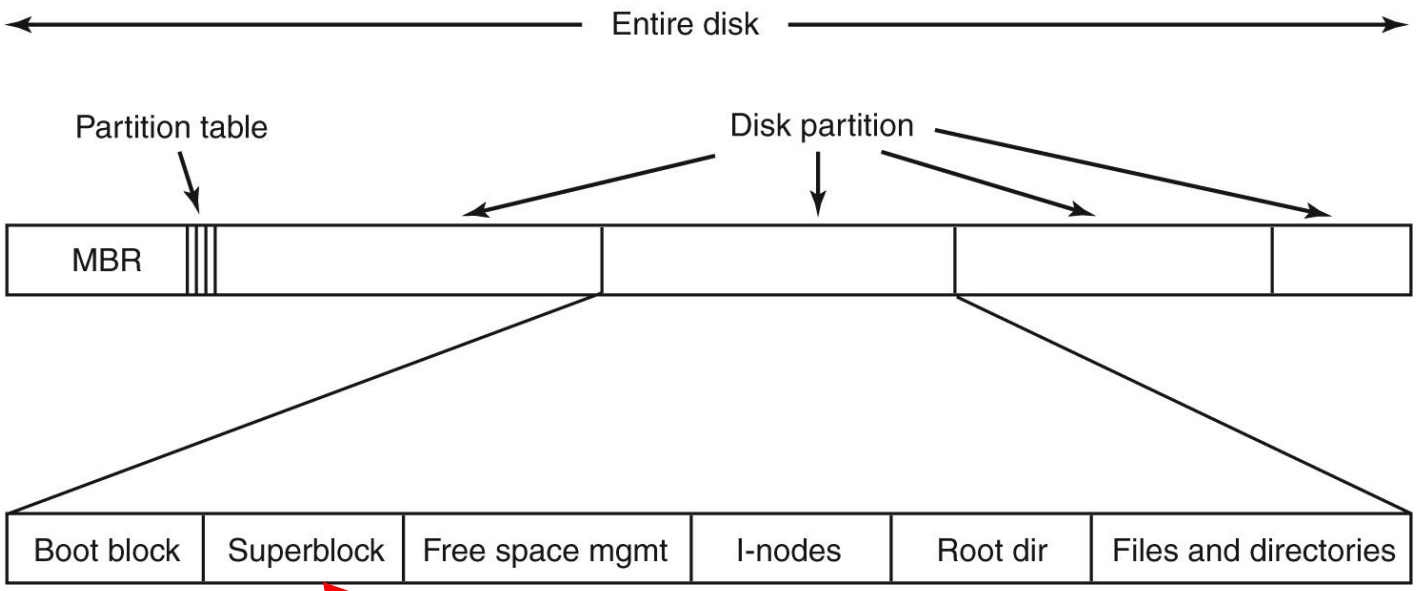
- Gives the starting and ending addresses of each partition
- One partition in the table is marked as active.
- When the computer is booted, BIOS executes MBR.
- MBR finds the active partition and reads its first block (called boot block) and executes it.
- Boot block loads the "OS" contained in that partition.



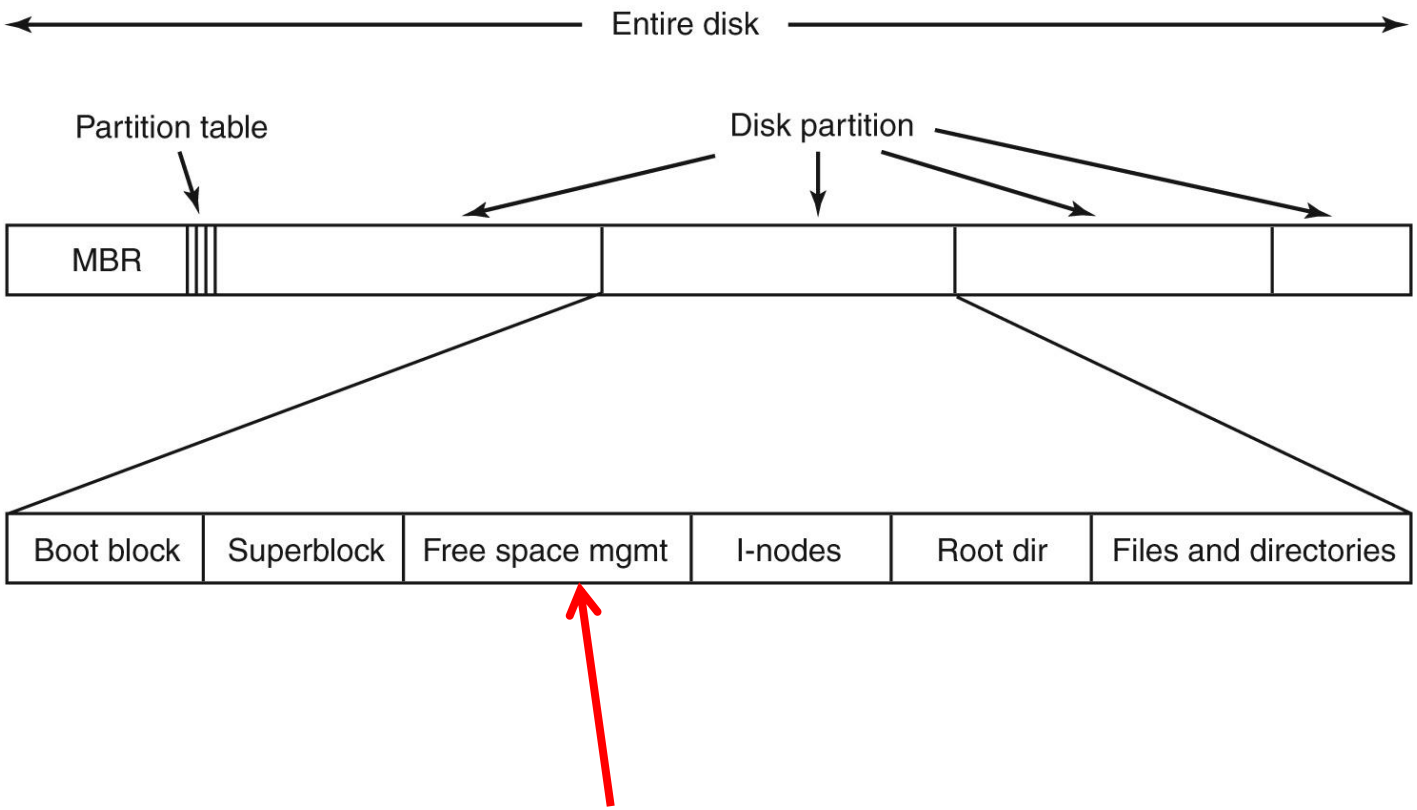

Entire disk
Partition table
Disk partition
MBR
Partition
Boot block
Superblock
Free space mgmt
I-nodes
Root dir
Files and directories



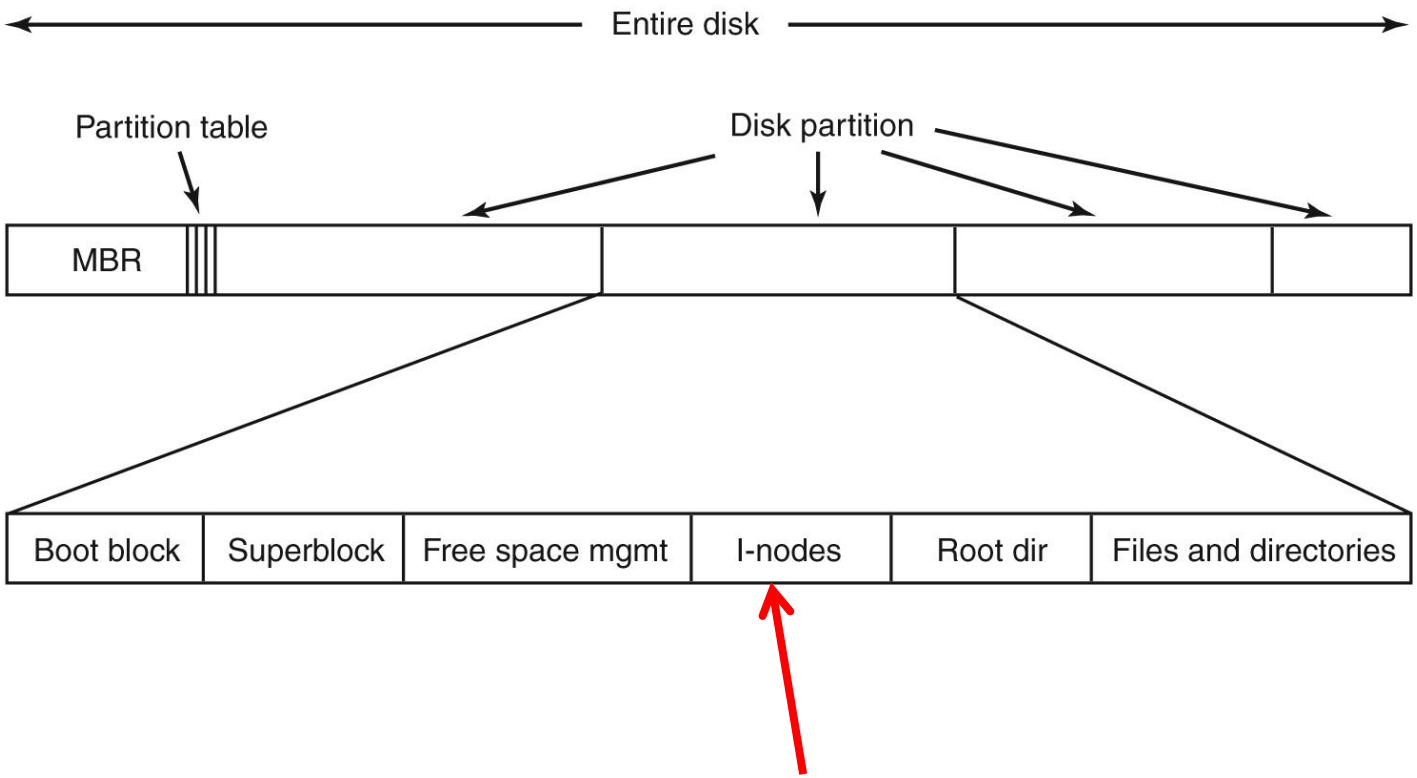
- Contains the bootable code (e.g. operating system)



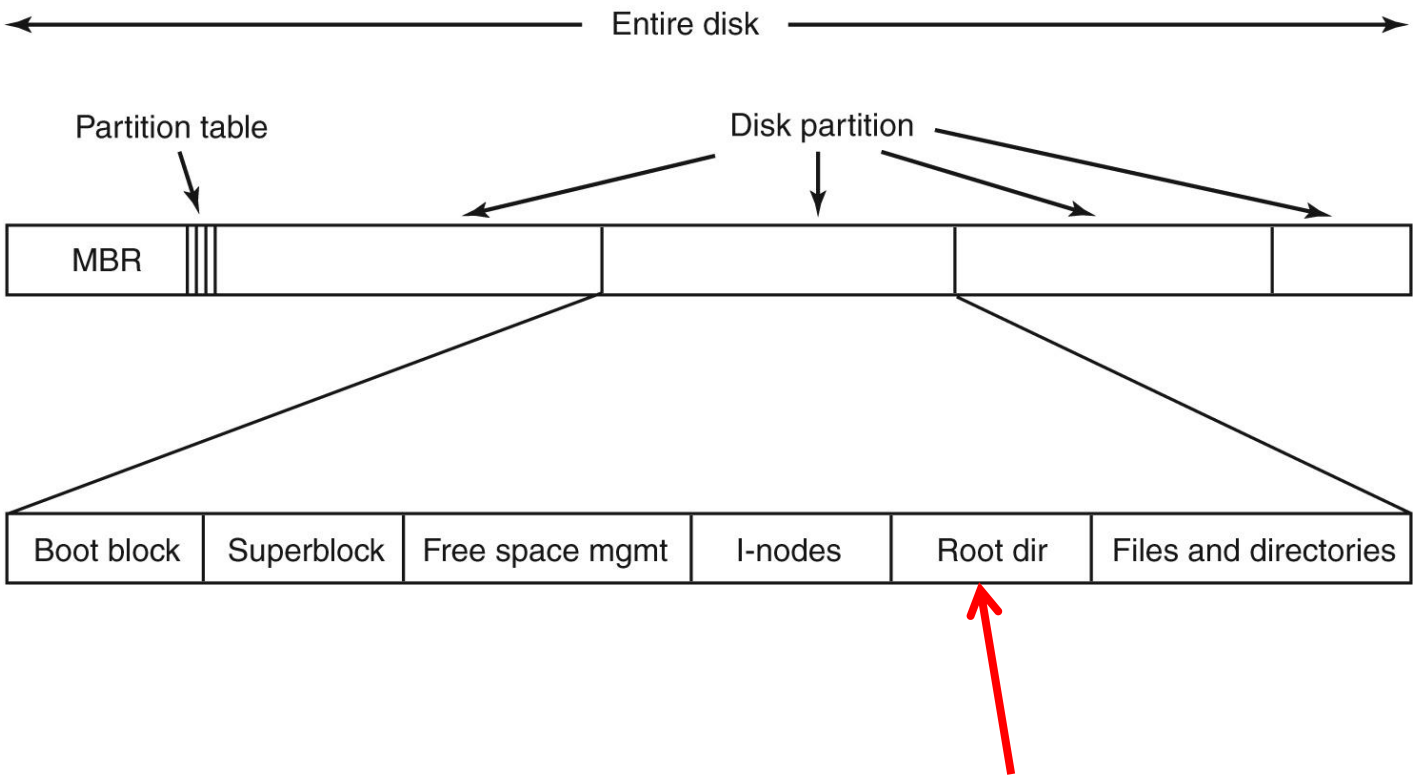
- Contains all key parameters about the file system
  - (e.g. filesystem type (magic, #-blocks, ..)
- Is read into memory when computer is booted or file system is touched.



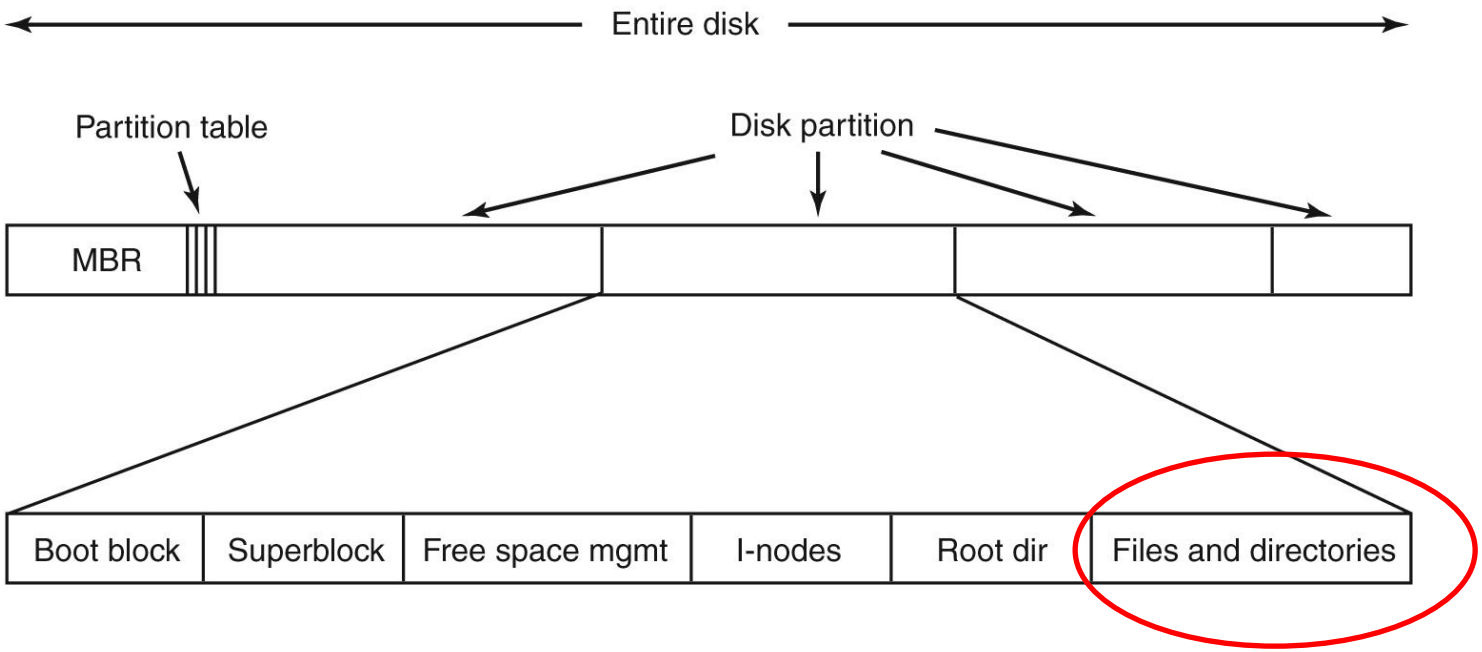
Bitmap or linked list



An array of data structures, one per file, telling about the file



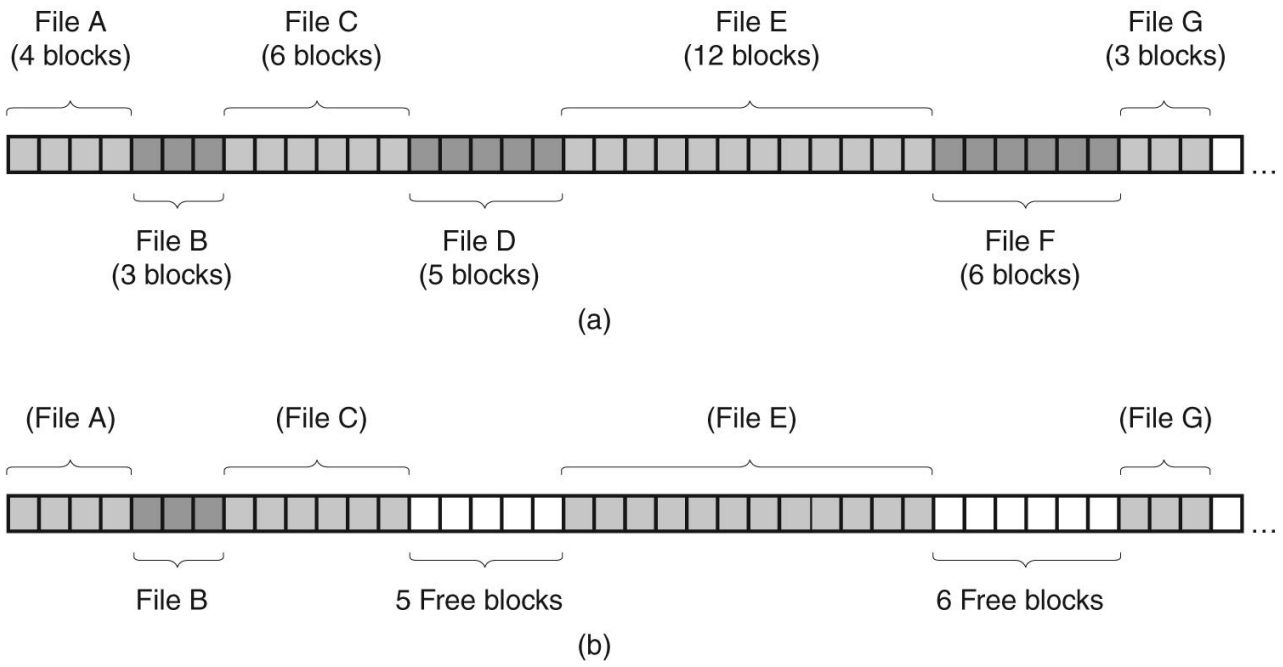
Root Directory .. Think '/' (as in '/home/user/jamesbond')



Which disk blocks go with which files?

Implementing Files: Contiguous Allocation

- Store each file as a contiguous run of disk blocks



After files D and F were deleted

Implementing Files: Contiguous Allocation

+ Simple to implement:
  Need to remember starting block address of the file and number of blocks

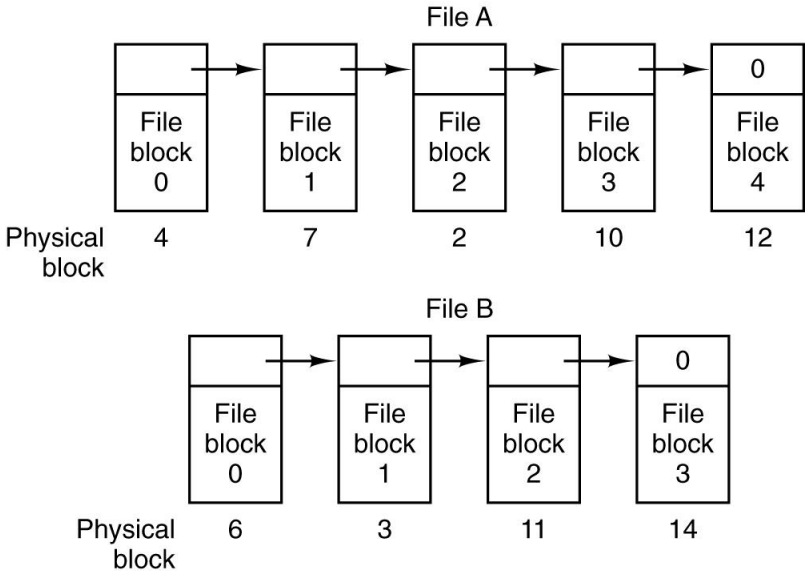
+ Read performance is excellent
  The entire file can be read from disk in a single operation.

- Disk becomes fragmented

- Need to know the final size of a file when the file is created

Implementing Files: Linked List Allocation

- Keep a file as a linked list of disk blocks
- The first word of each block is used as a pointer to the next one.
- The rest of the block is for data.



Implementing Files: Linked List Allocation

+ No (external) fragmentation

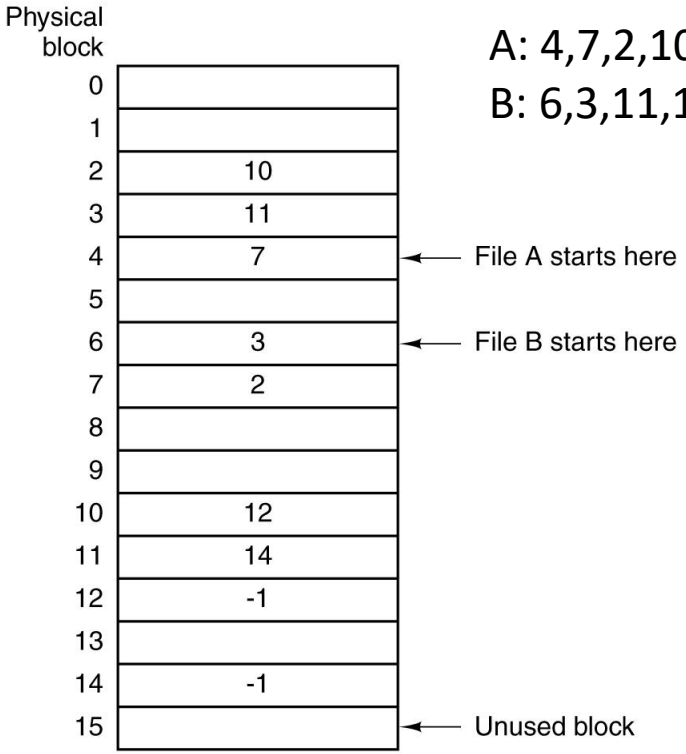
+ The directory entry needs just to store the disk address of the first block.

- Random access is extremely slow because we need to start a random request from the beginning half of the time.

- The amount of data storage is no longer a power of two, because the pointer takes up a few bytes (not to bad though)

Implementing Files:
Linked List Allocation Using a Table in Memory

- Take the pointer word from each block and put it in a table in memory.



A: 4,7,2,10,12
B: 6,3,11,14

- This table is called:
  File Allocation Table (FAT)
- Directory entry needs to keep a single integer:
  (the start block number)

Main drawback: Does not scale to large disks because the table needs to be in memory all the time.

FAT12, FAT16, FAT32

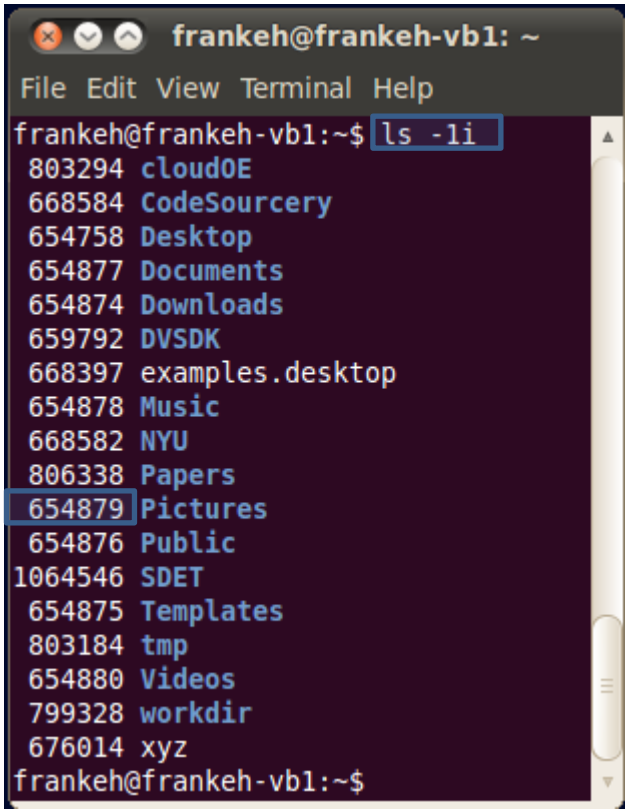
Limits of FAT

- Limits:
  http://en.wikipedia.org/wiki/Comparison_of_file_systems#cite_note-note-7-14

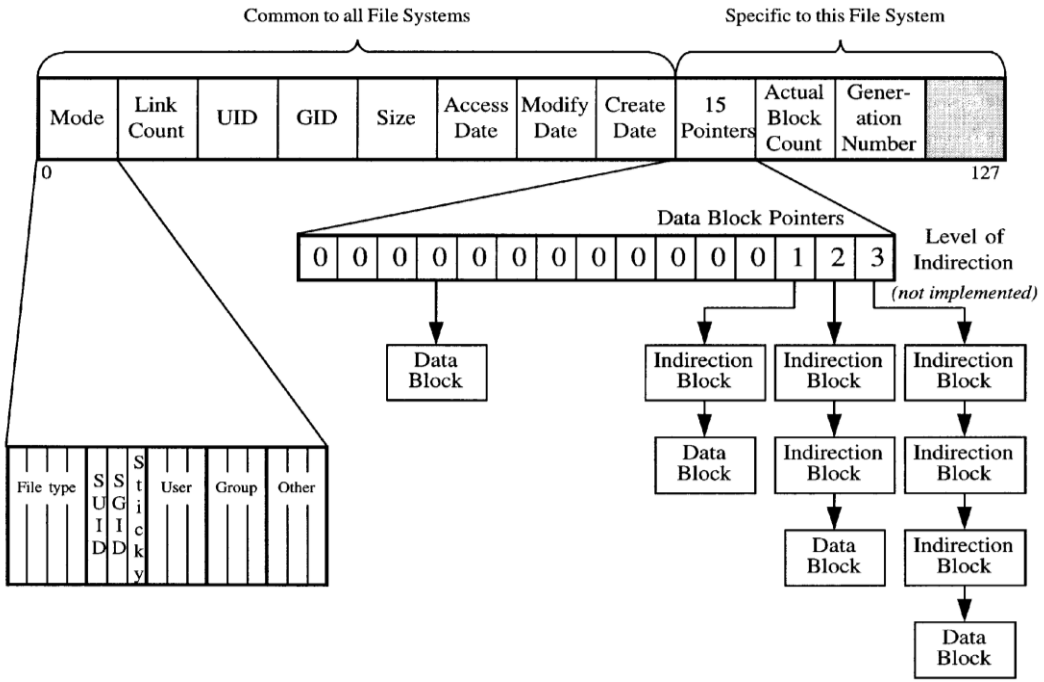
File system	Maximum filename length	Allowable characters in directory entries[5]	Maximum pathname length	Maximum file size	Maximum volume size[6]
FAT12	8.3 (255 UTF-16 code units with LFN)[14]	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \ | + , . ; = [] (lowcase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL[14][15]	No limit defined[16]	32 MB (256 MB)	32 MB (256 MB)
FAT16	8.3 (255 UTF-16 code units with LFN)[14]	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \ | + , . ; = [] (lowcase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL[14][15]	No limit defined[16]	2 GB (4 GB)	2 GB or 4 GB
FAT32	8.3 (255 UTF-16 code units with LFN)[14]	Any byte except for values 0-31, 127 (DEL) and: " * / : < > ? \ | + , . ; = [] (lowcase a-z are stored as A-Z). With VFAT LFN any Unicode except NUL[14][15]	No limit defined[16]	4 GB (256 GB[22])	2 TB[23] (16 TB)
ext2	255 bytes	Any byte except NUL[15] and /	No limit defined[16]	2 TB[6]	32 TB
ext3	255 bytes	Any byte except NUL[15] and /	No limit defined[16]	2 TB[6]	32 TB
ISO 9660:1988	Level 1: 8.3, Level 2 & 3: ~ 180	Depends on Level[51]	~ 180 bytes?	4 GB (Level 1 & 2) to 8 TB (Level 3)[52]	8 TB[53]
XFS	255 bytes[57]	Any byte except NUL[15]	No limit defined[16]	8 EB[58]	8 EB[58]
ZFS	255 bytes	Any Unicode except NUL	No limit defined[16]	16 EB	16 EB

Implementing Files: I-nodes

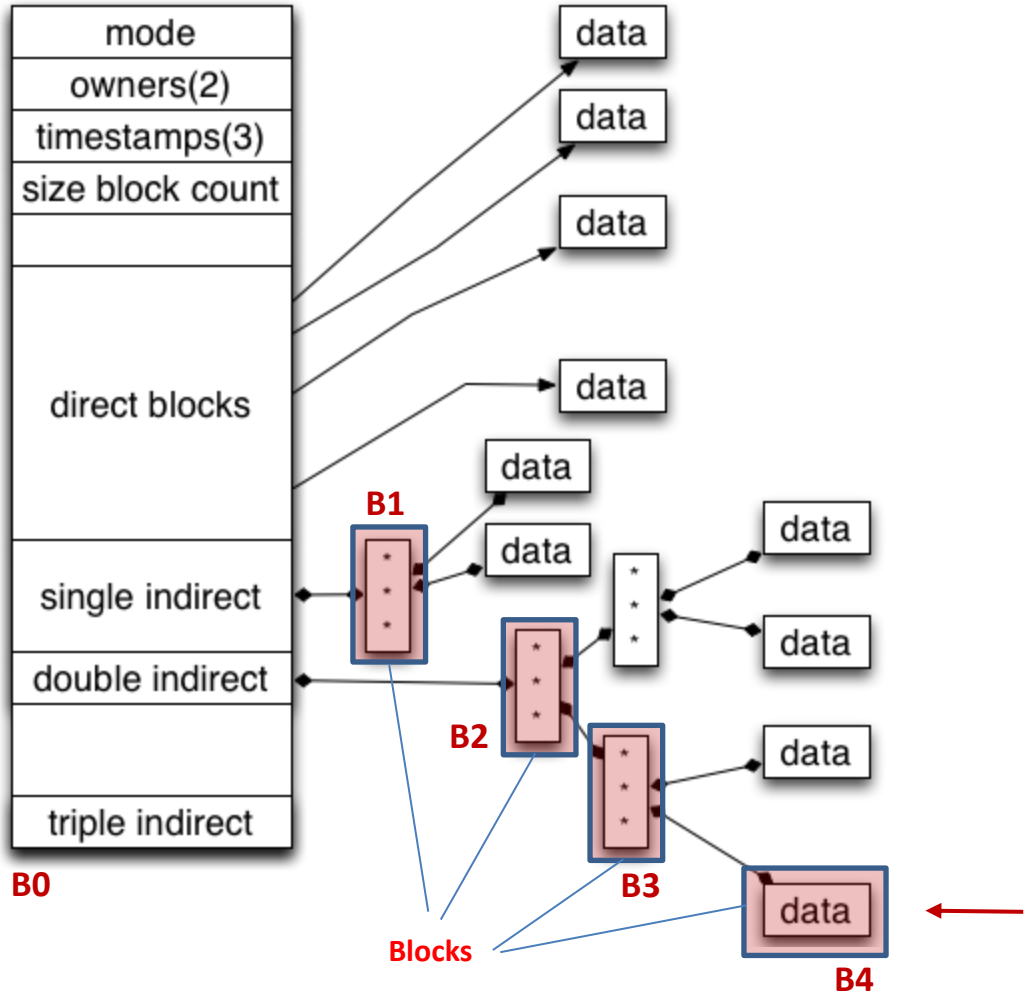
- A data structure associated with each file (and directory)
- Lists the attributes and disk addresses of the file blocks
- Need only be in memory when the corresponding file is open
- Aka FILE META DATA (typically 128 bytes)



i-node=#



Implementing Files: i-nodes



Properties:

- Small file access fast
- Everything a block
- Huge files can be presented
- Overhead (access, space) proportional to file size

To get to this data we must read
1) inode (B0)
2) double indirect block (B2) and (B3)
3) data block (B4)

→ if uncached, 4 disk block reads

Example for File access fixed size blocks

- Assume blksz=1K (2^10) and block number a 4 byte integer
  so the data is stored in 1 KB blocks and 256 (==2^8) block id's can fit in an indirection block

How much data range is covered at each level ??

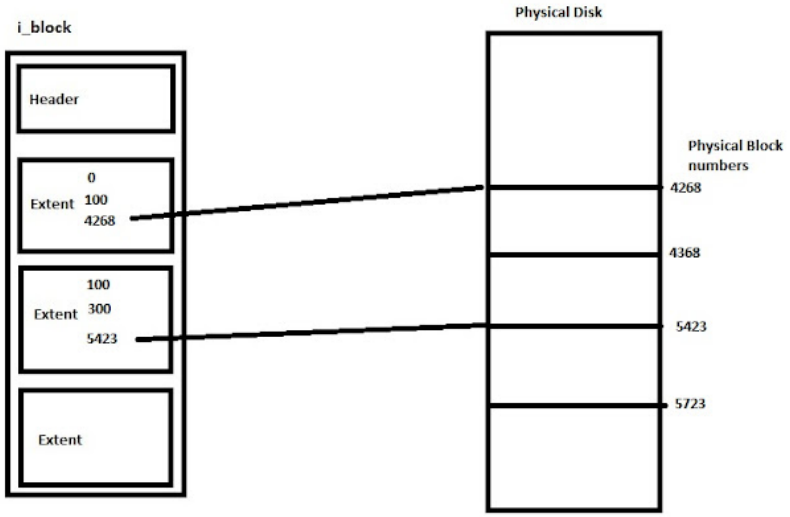
- Direct Access (12 entries) cover 12* 2^10 = 12KB
- Indirect: 2^8 * 2^10 = 2^18 = 256KB
- Double Indirect: 2^8 * 2^8 * 2^10 = 2^26 = 64MB
- Triple Indirect: 2^8 * 2^8 * 2^8 * 2^10 = 2^34 = 16GB
- Quad Indirection: 2^8 * 2^8 * 2^8 * 2^8 * 2^10 = 2^42 = 4TB

- Then: fileofs=260KB is covered by Indirect as it covers 12KB – 268KB

- Note: the math is different if you go to quad pointers (only 11 direct then) or if blksz is different.

Extent based Filesystems

- Inodes, but instead of fixed size blocks, each entry (direct,indicrect,..) is an extent: [file-offset, phys-block, length]
- The extent is phys contigous
- Significantly drives down fragmentation and disk access time
- Example: ext4




```
root@lnx2:~# filefrag -v /boot/vmlinuz-5.15.0-69-generic
Filesystem type is: ef53
File size of /boot/vmlinuz-5.15.0-69-generic is 11570216 (2825 blocks of 4096 bytes)
 ext:     logical_offset:        physical_offset: length:   expected: flags:
   0:        0..    2047:     927744..    929791:   2048:
   1:     2048..    2824:     933888..    934664:    777:     929792: last,eof
/boot/vmlinuz-5.15.0-69-generic: 2 extents found
```

syscalls to retrieve meta data

```
int stat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
```


`ls -ls` translates to

```
struct stat {
    dev_t     st_dev;         /* ID of device containing file */
    ino_t     st_ino;         /* Inode number */
    mode_t    st_mode;        /* File type and mode */
    nlink_t   st_nlink;       /* Number of hard links */
    uid_t     st_uid;         /* User ID of owner */
    gid_t     st_gid;         /* Group ID of owner */
    dev_t     st_rdev;        /* Device ID (if special file) */
    off_t     st_size;        /* Total size, in bytes */
    blksize_t st_blksize;     /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;      /* Number of 512B blocks allocated */

    /* Since Linux 2.6, the kernel supports nanosecond
       precision for the following timestamp fields.
       For the details before Linux 2.6, see NOTES. */

    struct timespec st_atim;  /* Time of last access */
    struct timespec st_mtim;  /* Time of last modification */
    struct timespec st_ctim;  /* Time of last status change */

#define st_atime st_atim.tv_sec      /* Backward compatibility */
#define st_mtime st_mtim.tv_sec
#define st_ctime st_ctim.tv_sec
};
```

Log-Structured File Systems

- Disk seek time does not improve as fast as relative to CPU speed, disk capacity, and memory capacity.
- Disk caches can satisfy most requests (e.g. buffer cache)

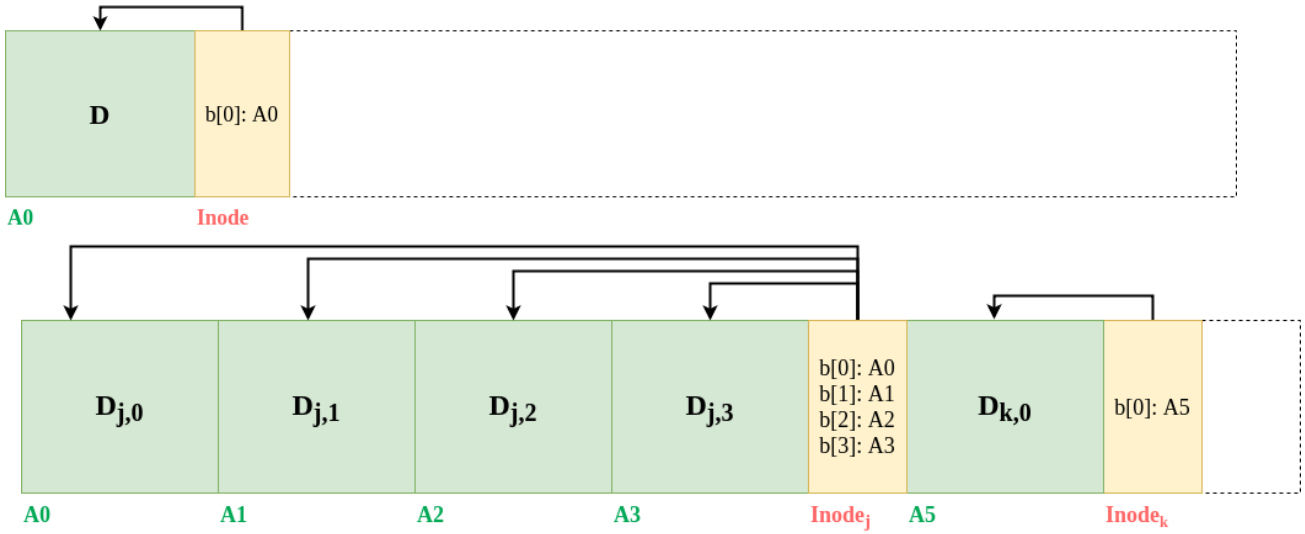
➔ In the future:
most disk accesses will be writes

➔ LSF optimizes for writes !

Log-Structured File Systems

- Basic idea: Buffer all writes (data + metadata) using an in-memory segment; once the segment is full, write the segment to a log (aka checkpoint)
- The segment write is one sequential write, so its fast
- We write one large segment (e.g., 1 or 4 MB) instead of a bunch of block-sized chunks to ensure that, at worst, we pay only one seek and then no rotational latencies (instead of one seek and possibly many rotational latencies)
- There are no in-place writes as in previous discussed filesystem implementation
- Reads still require random seeks, but physical RAM is plentiful, so buffer/page cache hit rate should be high (more on that later)

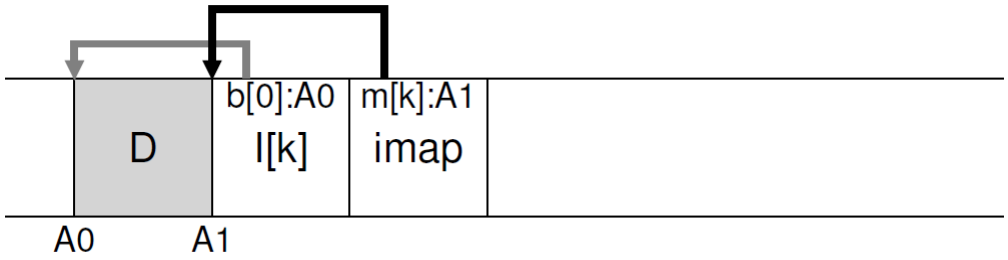
LSF: Segments



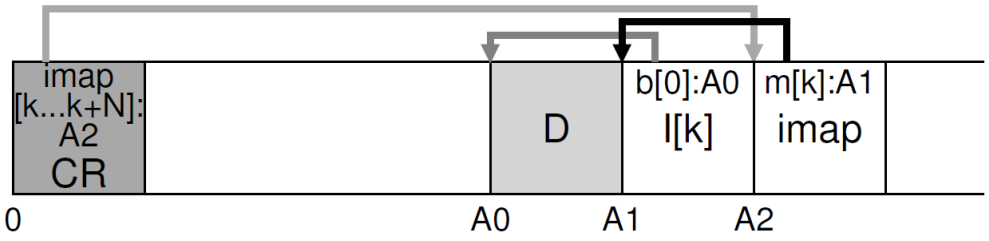
In order to determine whether a block is stale, you need to know its identity. This is stored in an additional part of the segment called the **segment table**. The segment table contains an entry for each block in the segment, which identifies which file (inode number) the block is part of, and which part it is (e.g. "direct block 37", or "the inode").

Log-Structured File Systems

- i-nodes now scattered over the disk
- Part of the i-node map is written with inode/block to log



- An i-node map, indexed by i-number, is maintained.
- The map is kept on disk at fixed location and is also cached.



- Checkpoint region (CR) only update periodically (30secs)

Log-Structured File Systems

- Disks are not infinite, hence at some point no further segment can be written to the log

- But many segments contains blocks that are no longer needed.
  - Block overwritten with a new block (now in a different segment)
  - E.g. file created then deleted (think compile)

➔ Cleaner Thread (kernel)

LSF (Cleaner Thread)

- Scan the log circularly to compact it
- Reads in the summary of the next segment to identify inodes and blocks
- Looks up inode map to check whether
  - inode is still in the current inode-map (deleted ?)
  - inode is still current (current pending write , so will be overwritten)
- Inodes and blocks still in use will go into memory and written to next segment
- Original Segment (and now compacted) is marked free
- This way disk is a big circular buffer
  - Writer Thread adds new segments to the front
  - Cleaner Thread removing old ones from the back
- Crash in period between checkpoints results in data loss or inconsistency

Improving strict LSF

- This sort of garbage collection of stale inodes and data blocks can lead to:
  - Increased I/O load and write amplification
- To reduce the overhead, most implementations avoid a strict circular log, but simply as a list of segments with order to inspect where the head of the log
  - Advances to a (non-adjacent) segment that is already free
  - Inspect the least-full segment first
- Improvement is increasingly ineffective as the file system fills up and approaches full capacity.

Journaling File System

- LSF not widely used, though principles continue in JFS
- Example: Microsoft NTFS, Linux ext3
- Deals with consistency issues
- Consider "rm /home/franke/nofreelunch"
  - Release the i-node
  - Release all the disk blocks to the free block pool
  - Remove the file from its directory
- Order of operation matters in the presence of crashes
  - Regardless of order, resources might become orphaned

Journaling File System

- The JFS first writes an entry of the three actions to be taken to the journal.
- Commit (write) journal to disk (now we have a record)
- Only after the journal is written, do the operations begin
- After operations are completed the journal entry is released
- If the system crashes before it is done, then after rebooting the journal is checked and the operations are rerun.
- Note: the journaled operations must be idempotent (i.e. can be repeated as often as necessary without harm).

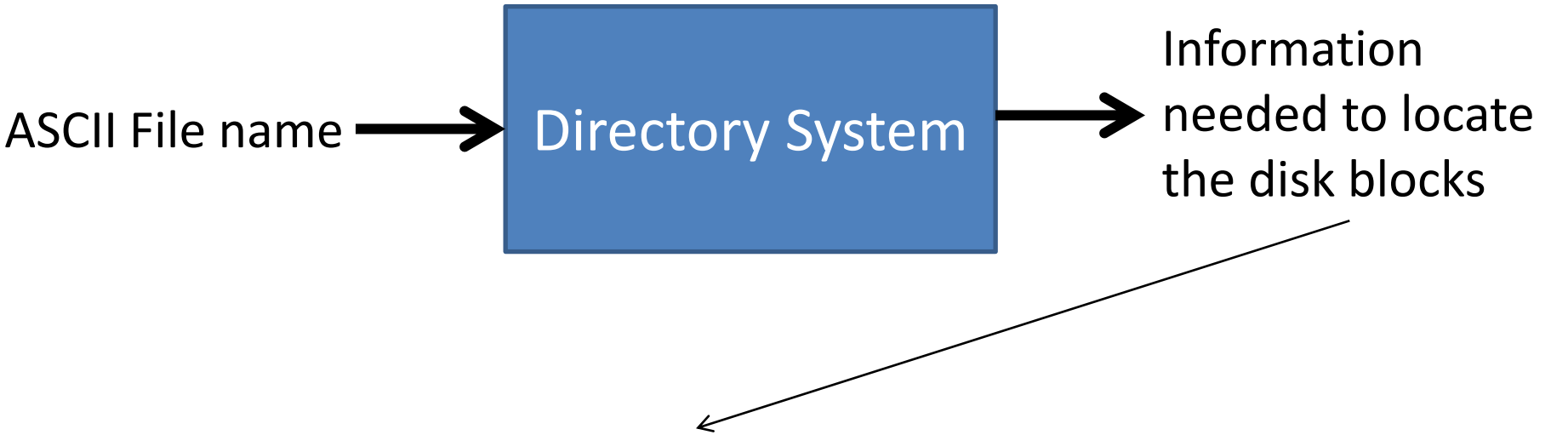
Journaling File Systems

- Journal writes must be guarded against corruption (e.g. checksum)
- Physical Journals
  - Logs every block to be written into log
  - Performance penalty → 2x writes
- Logical Journals
  - Only meta data is written (inode, append space)
  - Can lead to data corruption

Copy-on-Write Filesystems

- Copy on write filesystems avoid in place updates
- First write the data
- Then update the meta data that point at new data block
- Preserves correctness but no 2x write overhead
- Examples:
  - ZFS (zettabyte FS): unifies volume mgmt and filesystem implementation, feature rich with snapshotting capabilities, cloning
  - BTRFs (better fs): - " -

Implementing Directories

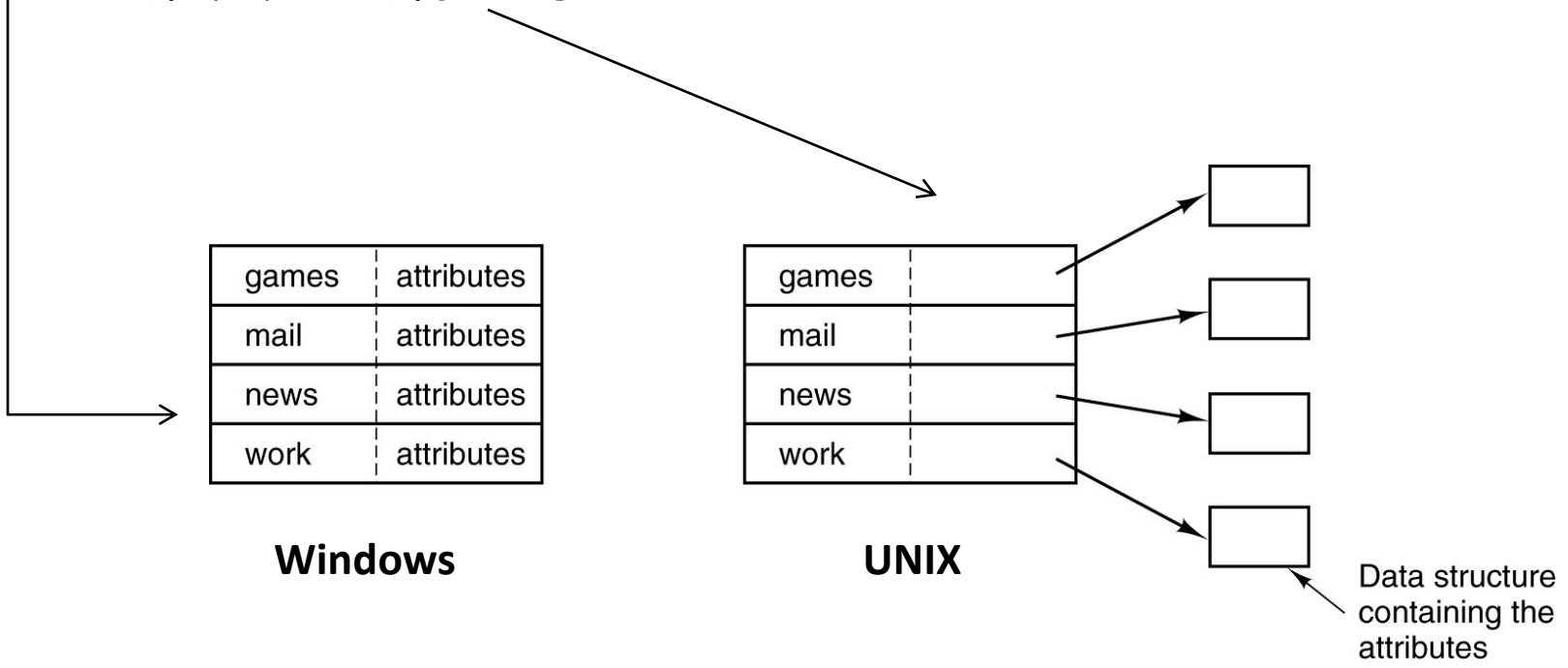


Example:

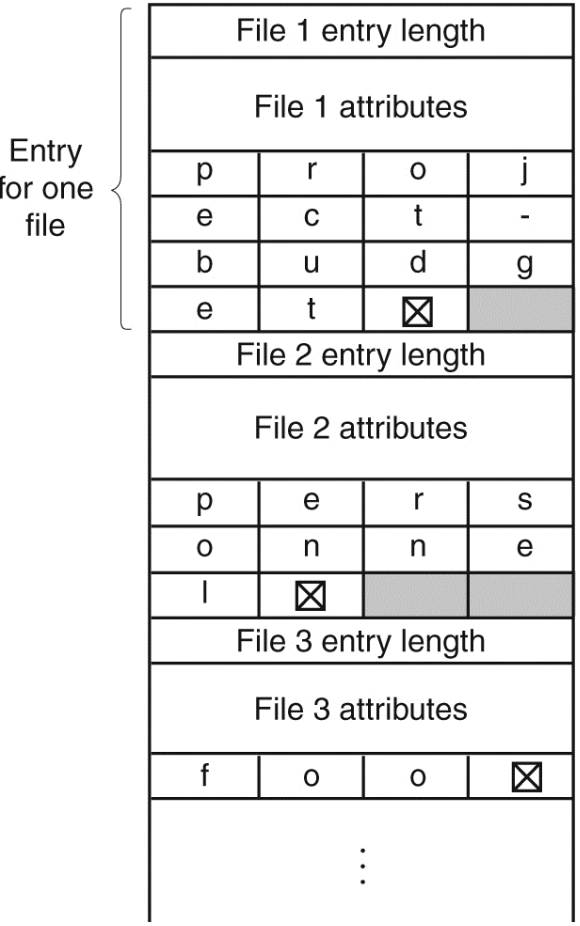
- Disk address of the file (in contiguous scheme)
- Number of the first block (in linked-list schemes)
- Number of the i-node,

Implementing Directories

- Where the attributes should be stored?
  - Directly in the directory entry
  - In the i-nodes



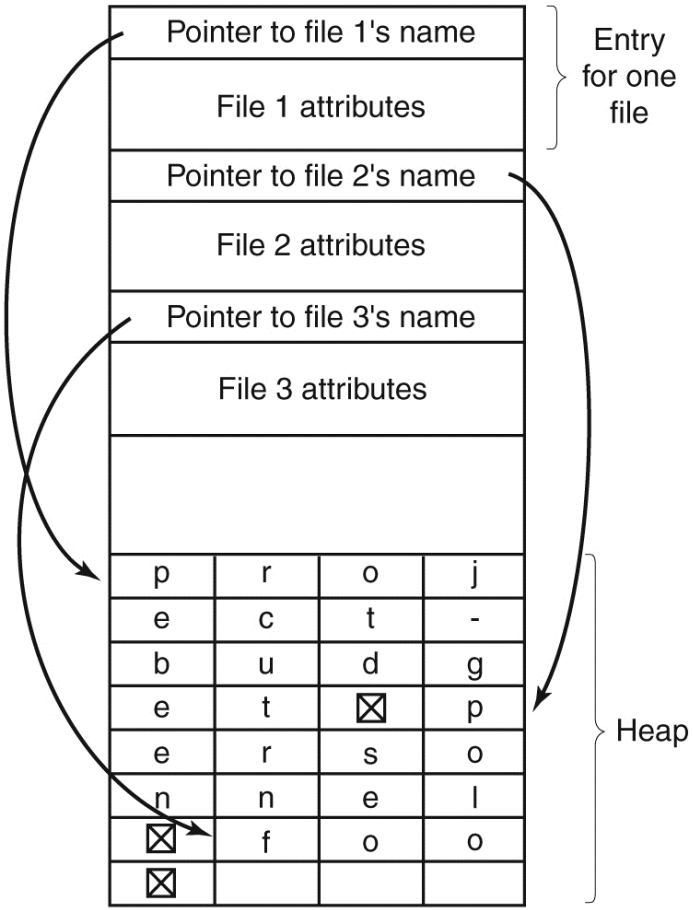
Implementing Directories: Variable-Length Filenames



Disadvantages:

- Entries are no longer of the same length.
-Variable size gaps when files are removed
- A big directory may span several pages which may lead to page faults.

Implementing Directories: Variable-Length Filenames



- Keep directory entries fixed length
- Keep filenames in a heap at the end of the directory.
- Page faults can still occur while accessing filenames.

Implementing Directories

- For extremely long directories, linear search can be slow.
  - Hashing can be used
  - Caching can be used

- Note: A directory is nothing but a file, where the filedata represents the lookup table and the "directory bit" is set in the inode

Listing content of directory

```
DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
```

```
struct dirent {
    ino_t          d_ino;       /* Inode number */
    off_t          d_off;       /* Not an offset; see below */
    unsigned short d_reclen;    /* Length of this record */
    unsigned char  d_type;      /* Type of file; not supported
                                   by all filesystem types */
    char           d_name[256]; /* Null-terminated filename */
};
```

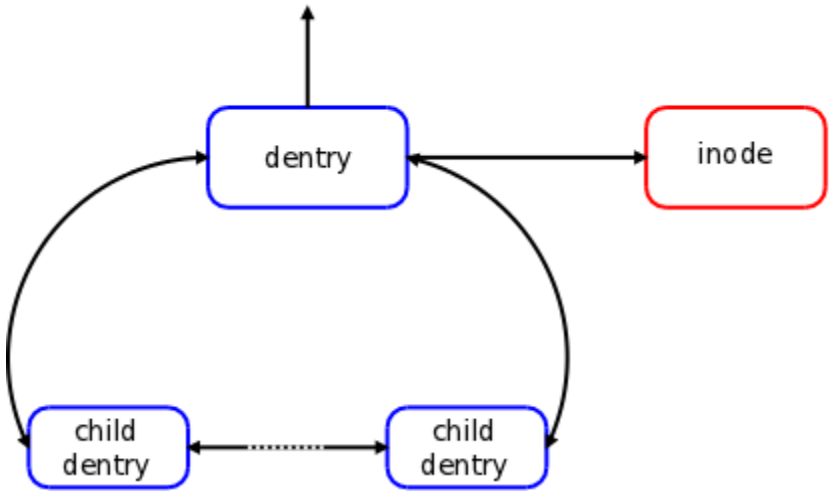
```
main() {
  DIR *dir;
  struct dirent *entry;

  if ((dir = opendir("/")) == NULL)
    perror("opendir() error");
  else {
    puts("contents of root:");
    while ((entry = readdir(dir)) != NULL)
      printf("  %s\n", entry->d_name);
    closedir(dir);
  }
}
```

Speeding it up

- Continuously going to the disk is expensive
  e.g. "/home/frankeh/nyu/best/class/ever"
- Root -> home -> frankeh -> nyu -> best -> class -> ever
  multiple inodes/blocks and dentries (directories) need to be read

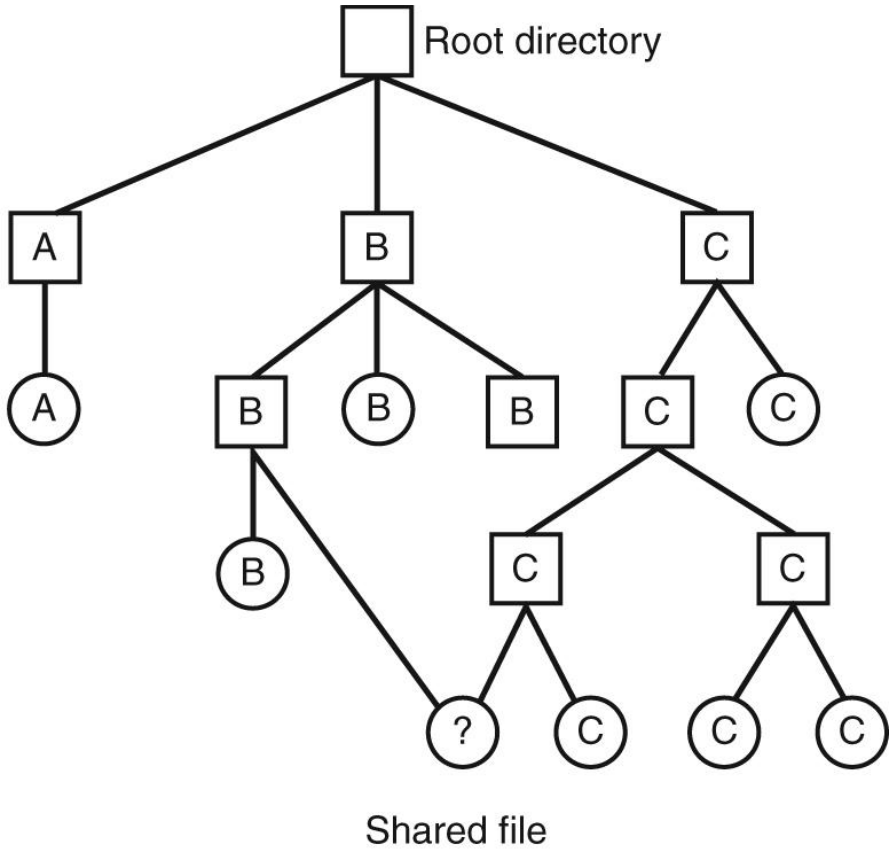
- Same old story → caching, caching, caching

- *DENTRY cache*
  paths that are walked are stored as an in-memory directory entry tree



Shared Files

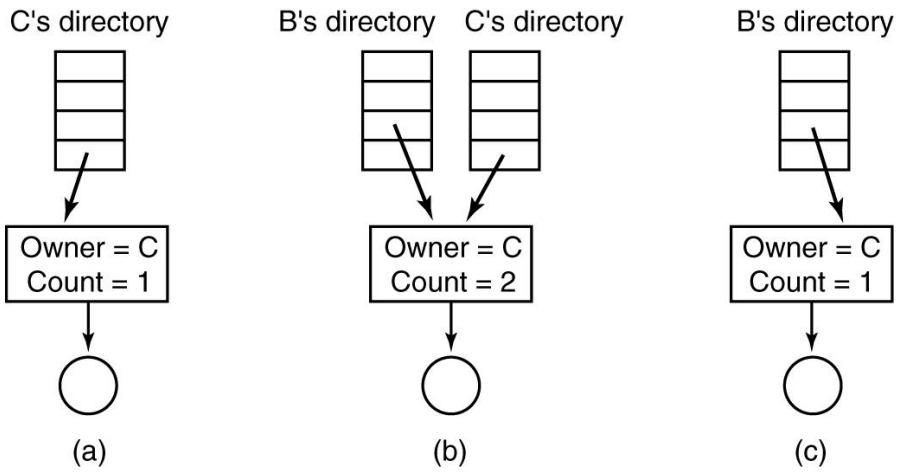
- Appear simultaneously in different directories



Shared Files: Method 1

- Disk blocks are not listed in directories but in a data structure associated with the file itself (e.g. i-nodes in UNIX).
- Directories just point to that data structure.
- This approach is called: static linking
- "`ln <origfile> <newfilename>`" or "`ln -P`"

Problematic Scenario



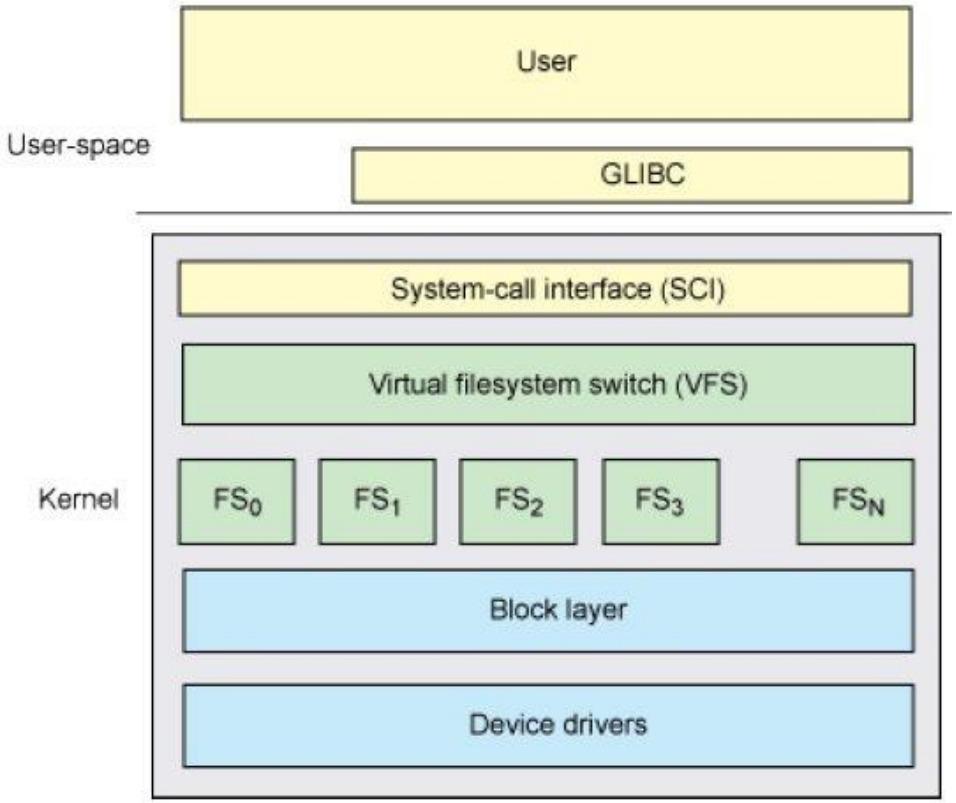
Shared Files: Method 2

- Have the system create a new file (of type LINK). This new file contains the path name of the file to which it is linked.
- This approach is called: symbolic linking
- The main drawback is the extra overhead.

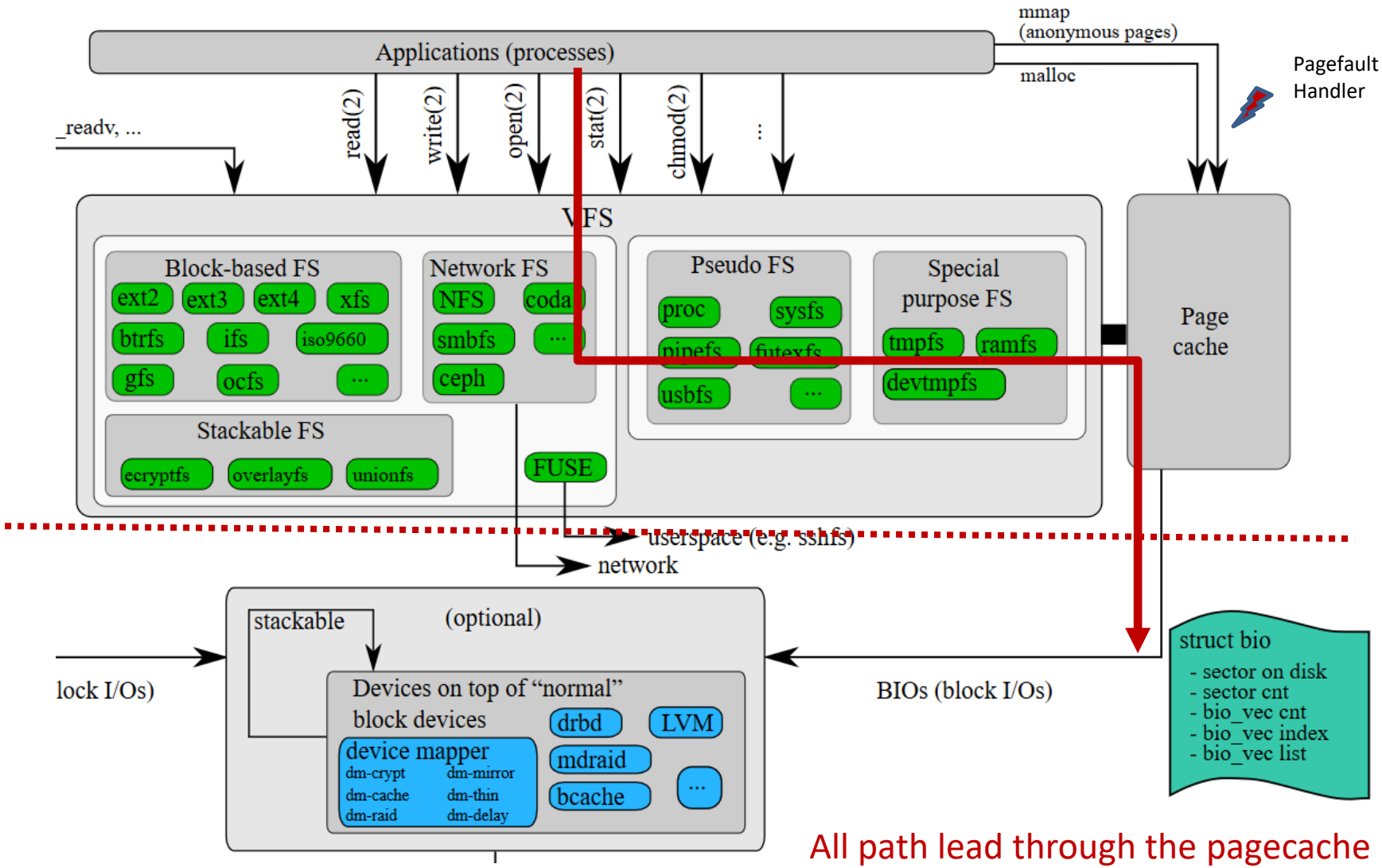
- "ln -s <origfile> <newfilename>"

Virtual File Systems

- Integrating multiple file systems into an orderly structure.
- Provides a common layer to the SCI and services to switch to different filesystems underneath
- Said filesystems then take advantage of common OS service such as caching (e.g. block cache) and (block I/O layer)



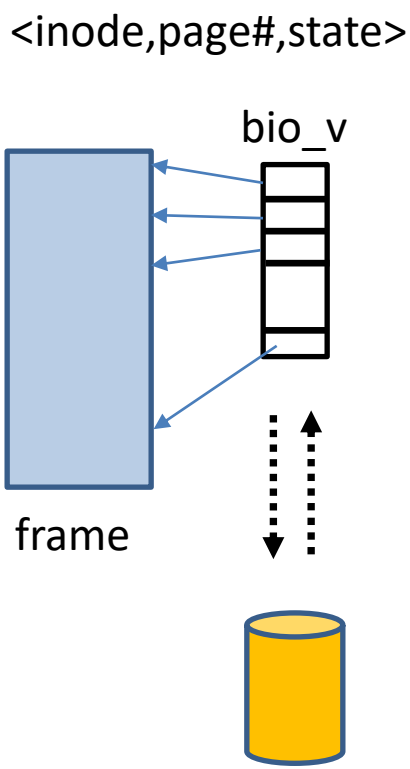
Page Cache and Other types of FileSystems



All path lead through the pagecache

Page Cache

- Integrates I/O caching (buffer cache) and memory mapped file
- Page cache indexed by <inode,vpageofs> vpageofs is the virtual page relative to the beginning of the file.
- If no hit, frame is allocated and I/O is issued as set of bio's (block IOs), when all are completed, the data is in cache and the file can be mapped or data can be provided to application.
- Order of the bio's can depend on whether data is requested via mmaps or read/write.
- Page Cache responsible for flushing data (sync) in due time (write backs)



Page Cache

- PageCache pervasively used in OS
- If there is no other use for memory, why not for cache ?
- Freshly booted system (4GB), mostly free but already 710MB cached

```
root@lnx1:~# free -m
              total        used        free      shared  buff/cache   available
Mem:           3944         263        2970           1         710        3459
Swap:          1942           0        1942
```

- Search the entire disk for something -> access all data

```
root@lnx1:/# find . -type f -print | xargs grep some_string_i_wont_find
```

- PageCache now heavily loaded, almost no memory free

```
root@lnx1:/# free -m
              total        used        free      shared  buff/cache   available
Mem:           3944         292         149           1        3502        3392
Swap:          1942           4        1938
```

- When memory is required for apps (2GB) the page cache will be reduced

```
root@lnx1:/# free -m
              total        used        free      shared  buff/cache   available
Mem:           3944        2306         144           1        1493        1385
Swap:          1942          11        1930
```

Effects of PageCache

- Significantly speeds up searching through 2.4GB disk space
- Start with a freshly booted system and page cache smal

```
frankeh@lnx1:~/NYU$ free -m
              total        used        free      shared  buff/cache   available
Mem:           3944         258        2971           1         714        3464
Swap:          1942           0        1942
frankeh@lnx1:~/NYU$ time grep -nr some_bizarre_foobar_garbage

real    0m14.858s
user    0m1.632s
sys     0m5.444s
frankeh@lnx1:~/NYU$ free -m
              total        used        free      shared  buff/cache   available
Mem:           3944         257         467           1        3219        3438
Swap:          1942           0        1942
frankeh@lnx1:~/NYU$ time grep -nr some_bizarre_foobar_garbage

real    0m4.523s
user    0m4.074s
sys     0m0.443s
frankeh@lnx1:~/NYU$ du -ms .
2435    .
```

Cache now loaded

Network File System

- Files are located on a different server - CIFS, NFS, ….
- Requests are sent over to server with name and block requests
- Data can be cached, but be aware of sharing

Pseudo FileSystem

- /proc or /sys
- Means to access/manipulate OS internals
  (state, parameters, algos, settings)

- Allows scripting (vs. a special syscall interface)

- The hierarchy is created dynamically and on access, there is no real disk !

Proc fs (inspecting state)

- Inspecting a single process or system statistics
- Registers read/write functions per "node"

```
frankeh@lnx1:~$ ps -edf | grep 1692
frankeh   1692  1509  0 Apr27 ?        00:00:19 x-terminal-emulator
```

```
frankeh@lnx1:~$ ls /proc/1692/
attr        coredump_filter  gid_map    mountinfo   oom_score      sched         stat          uid_map
autogroup   cpuset           io         mounts      oom_score_adj  schedstat     statm         wchan
auxv        cwd              limits     mountstats  pagemap        sessionid     status
cgroup      environ          loginuid   net         patch_state    setgroups     syscall
clear_refs  exe              map_files  ns          personality    smaps         task
cmdline     fd               maps       numa_maps   projid_map     smaps_rollup  timers
comm        fdinfo           mem        oom_adj     root           stack         timerslack_ns
```

```
frankeh@lnx1:/proc/1692$ cat stat
1692 (x-terminal-emul) S 1509 1397 1397 0 -1 4194304 13687 13059 12 49 1472 604 61 132 20 0 3 0 3444 524214272 10142
18446744073709551615 94007262363648 94007262436208 140727902463216 0 0 0 0 4096 65536 0 0 0 17 2 0 0 4 0 0 9400726453
5656 94007264542176 94007273050112 140727902464728 140727902464748 140727902464748 140727902466011 0
frankeh@lnx1:/proc/1692$ cat statm
127982 10142 5828 18 0 12981 0
```

```
frankeh@lnx1:~$ cat /proc/1692/maps | head
557fc57ac000-557fc57be000 r-xp 00000000 08:01 1194341                    /usr/bin/lxterminal
557fc59be000-557fc59bf000 r--p 00012000 08:01 1194341                    /usr/bin/lxterminal
557fc59bf000-557fc59c0000 rw-p 00013000 08:01 1194341                    /usr/bin/lxterminal
557fc61dd000-557fc72da000 rw-p 00000000 00:00 0                          [heap]
7fc988000000-7fc988021000 rw-p 00000000 00:00 0
7fc988021000-7fc98c000000 ---p 00000000 00:00 0
7fc98c000000-7fc98c021000 rw-p 00000000 00:00 0
```

Sys-fs

Read and modify system status and behavior

```
frankeh@lnx1:/sys$ ls
block  bus  class  dev  devices  firmware  fs  hypervisor  kernel  module  power
frankeh@lnx1:/sys$ cd bus
frankeh@lnx1:/sys/bus$ ls
ac97         container     gpio  iscsi_flashnode  mipi-dsi  nvmem        platform  sdio    usb        xen
acpi         cpu           hid   machinecheck     mmc       pci          pnp       serial  virtio     xen-backend
clockevents  edac          i2c   mdio_bus         nd        pci-epf      rapidio   serio   vme
clocksource  event_source  isa   memory           node      pci_express  scsi      spi     workqueue
frankeh@lnx1:/sys/bus$ cd pci
frankeh@lnx1:/sys/bus/pci$ ls
devices  drivers  drivers_autoprobe  drivers_probe  rescan  resource_alignment  slots  uevent
frankeh@lnx1:/sys/bus/pci$ ls devices/
0000:00:00.0  0000:00:01.1  0000:00:03.0  0000:00:05.0  0000:00:07.0  0000:00:0d.0
0000:00:01.0  0000:00:02.0  0000:00:04.0  0000:00:06.0  0000:00:08.0
```

Example: change the scheduling algorithm for a particular disk device (/dev/sda):

```
root@lnx1:~# cat /sys/block/sda/queue/scheduler
noop deadline [cfq]
root@lnx1:~# echo noop > /sys/block/sda/queue/scheduler
root@lnx1:~# cat /sys/block/sda/queue/scheduler
[noop] deadline cfq
```

Special Purpose Filesystem

- RamFS (builds a ram disk)
- Backed by memory not disk
- Instead of issuing block I/O requests you memcpy(4K) to from memory
- Obviously
  - significantly faster than going to any real device
  - But no persistence
  - Reduces available RAM on your system
- Linux : tmpfs

File System Performance

- Caching:
  - Block cache: a collection of blocks kept in memory for performance reasons
  - Page cache: pages of files are kept in memory
- Block Read Ahead:
  - Get blocks into the cache before they are needed
  - See (page cache)
  - Recognize sequential access and pre-read.
- Reducing Disk Arm Motion:
  - Putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder
- Defragmentation

Example: fs creation and mounting

- Create a new filesystem on a particular device:
  - Allocate inodes and blocks

```
# mkfs -t ext3 /dev/sda6
mke2fs 1.42 (29-Nov-2011)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
1120112 inodes, 4476416 blocks
223820 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=0
137 block groups
32768 blocks per group, 32768 fragments per group
8176 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208,
        4096000

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

- Associating a device with a particular filesystem and providing an access point "mount point"

```
mount -t type device directory
```

```
~]# mount -t vfat /dev/sdc1 /media/flashdisk
```

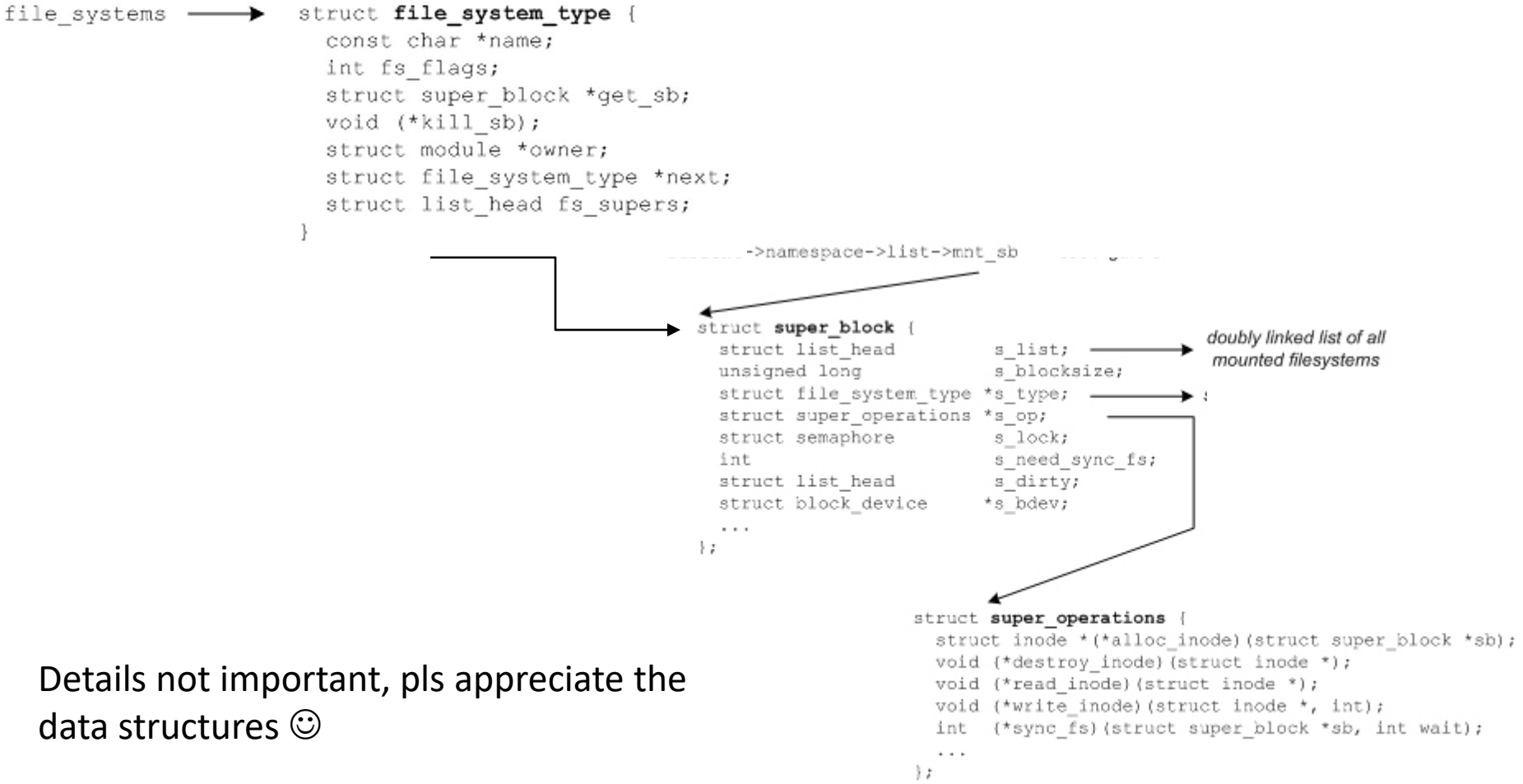
Type	Description
ext2	The **ext2** file system.
ext3	The **ext3** file system.
ext4	The **ext4** file system.
iso9660	The **ISO 9660** file system. It is commonly used by optical media, typically CDs.
jfs	The **JFS** file system created by IBM.
nfs	The **NFS** file system. It is commonly used to access files over the network.
nfs4	The **NFSv4** file system. It is commonly used to access files over the network.
ntfs	The **NTFS** file system. It is commonly used on machines that are running the Windows operating system.
udf	The **UDF** file system. It is commonly used by optical media, typically DVDs.
vfat	The **FAT** file system. It is commonly used on machines that are running the Windows operating system, and on certain digital media such as USB flash drives or floppy disks.

"mount" Example

```
root@lnx1:~# mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
udev on /dev type devtmpfs (rw,nosuid,relatime,size=1986852k,nr_inodes=496713,mode=755)
devpts on /dev/pts type devpts (rw,nosuid,noexec,relatime,gid=5,mode=620,ptmxmode=000)
tmpfs on /run type tmpfs (rw,nosuid,noexec,relatime,size=403920k,mode=755)
/dev/sda1 on / type ext4 (rw,relatime,errors=remount-ro,data=ordered)
```

- Note the top entry "/" is mapped to disk
- At various mount points different filesystems are "attached", e.g. all pseudo files systems or network file systems.

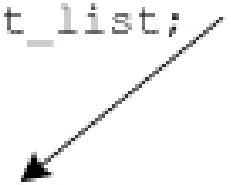
Example: Linux Internals



Details not important, pls appreciate the data structures ☺

Example: Linux Internals

```
current->namespace->list ──►  struct vfsmount {
                                struct list_head mnt_hash;
                                struct vfsmount *mnt_parent;
                                struct dentry *mnt_mountpoint;
                                struct dentry *mnt_root;
                                struct super_block *mnt_sb;
                                struct list_head mnt_mounts;
                                struct list_head mnt_child;
                                atomic_t mnt_count;
                                int mnt_flags;
                                char *mnt_devname;
                                struct list_head mnt_list;
                              }
```



mounted filesystem list

Example: Linux Internals

```
struct inode {
  unsigned long            i_ino;
  umode_t                  i_mode;
  uid_t                    i_uid;
  struct timespec          i_atime;
  struct timespec          i_mtime;
  struct timespec          i_ctime;
  unsigned short           i_bytes;
  struct inode_operations *i_op;
  struct file_operations  *i_fop;
  struct super_block      *i_sb;
  ...
}
```

```
struct inode_operations {
  int (*create)(struct inode *, struct dentry *,
                struct nameidata *);
  struct dentry *(*lookup)(struct inode *,
                           struct dentry *,
                           struct nameidata *);
  int (*mkdir)(struct inode *, struct dentry *, int);
  int (*rename)(struct inode *, struct dentry *,
                struct inode *, struct dentry *);
  ...
}
```

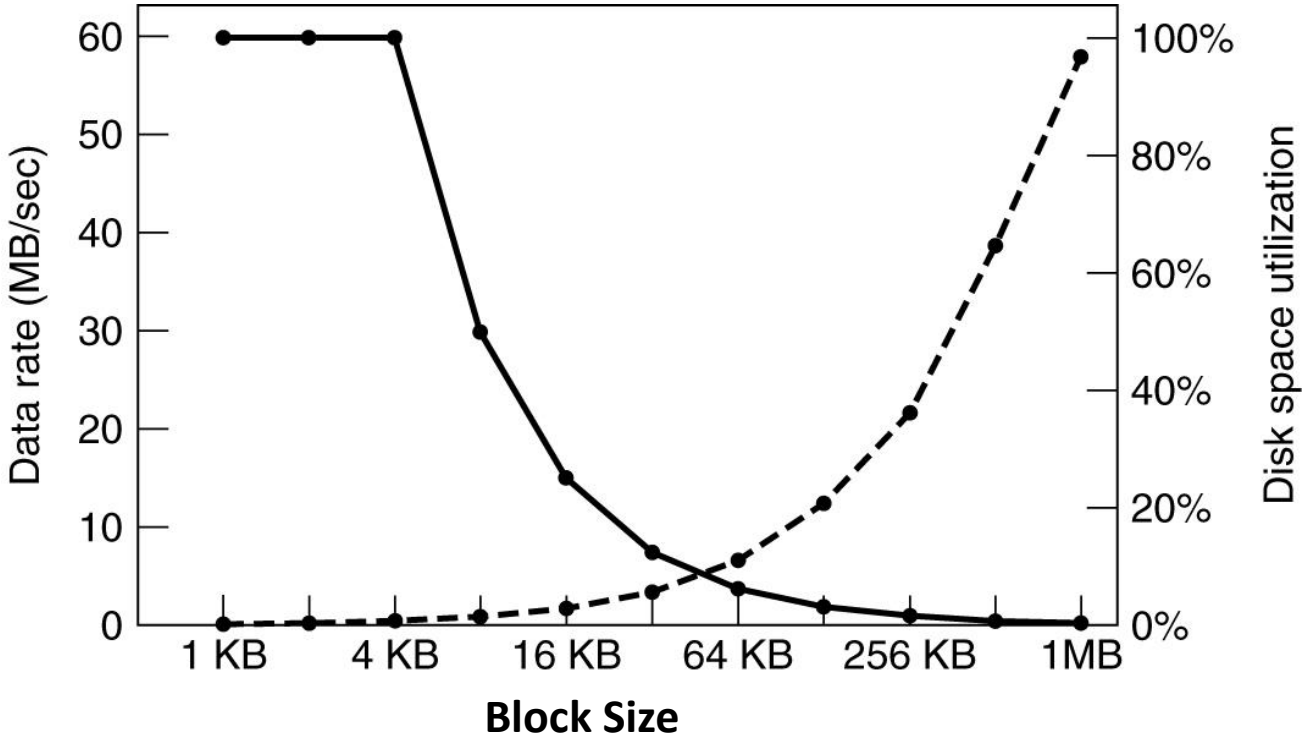
```
struct file_operations {
  struct module *owner;
  ssize_t (*read)(struct file *, char __user *,
                  size_t, loff_t *);
  ssize_t (*write)(struct file *, const char __user *,
                   size_t, loff_t *);
  int (*open)(struct inode *, struct file *);
  ...
}
```

Disk Space Management

- All file systems chop files up into fixed-size blocks that need not be adjacent.
- Block size:
  - Too large -> we waste space
  - Too small -> we waste time

Access time for a block is completely dominated by the seek time and rotational delay.
So … The more data are fetched the better.

(dotted)

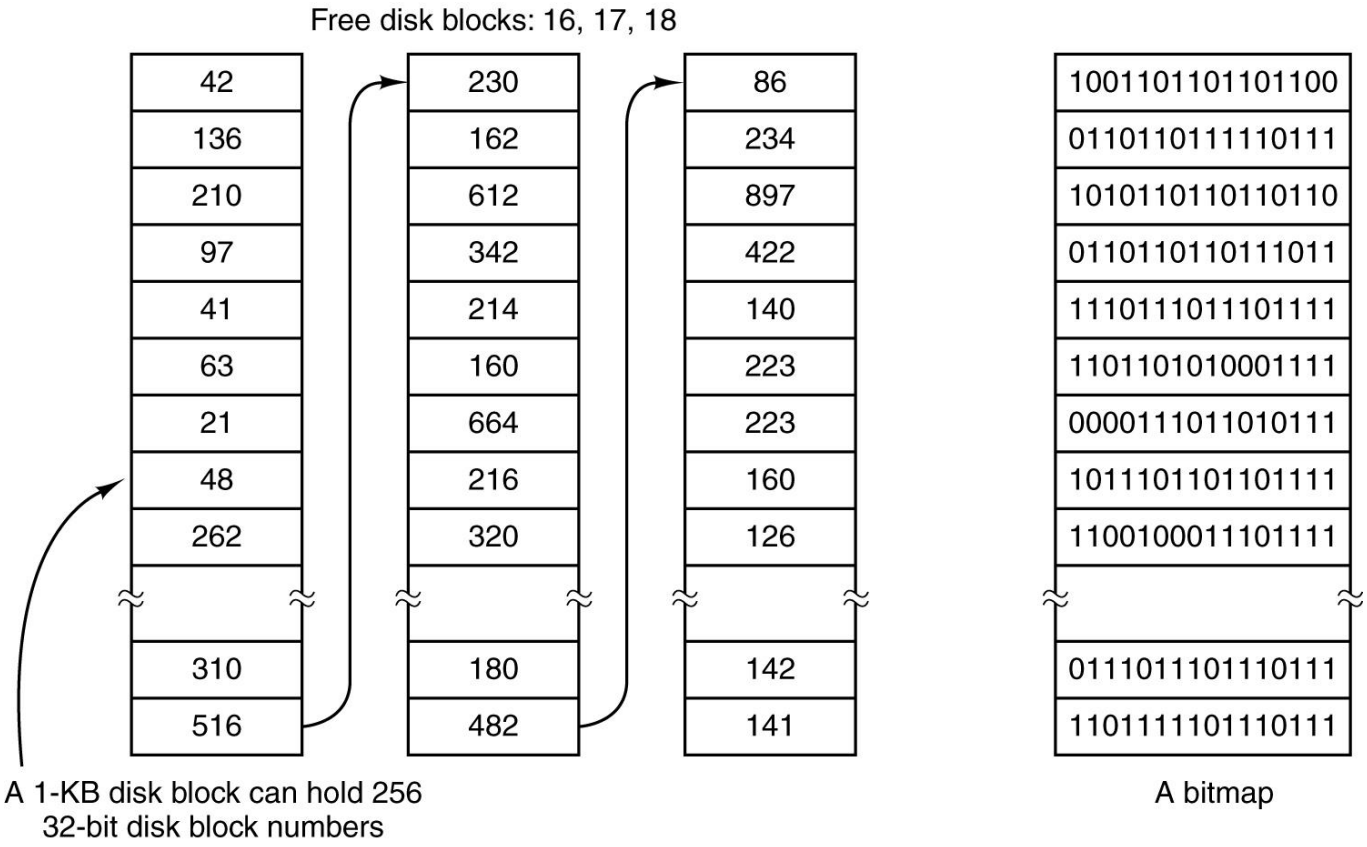


Disk Space Management: Keeping Track of Free Blocks

- **Method 1**: Linked list of disk blocks, with each block holding as many free disk block numbers as possible.

- **Method 2**: Using a bitmap

Disk Space Management: Keeping Track of Free Blocks

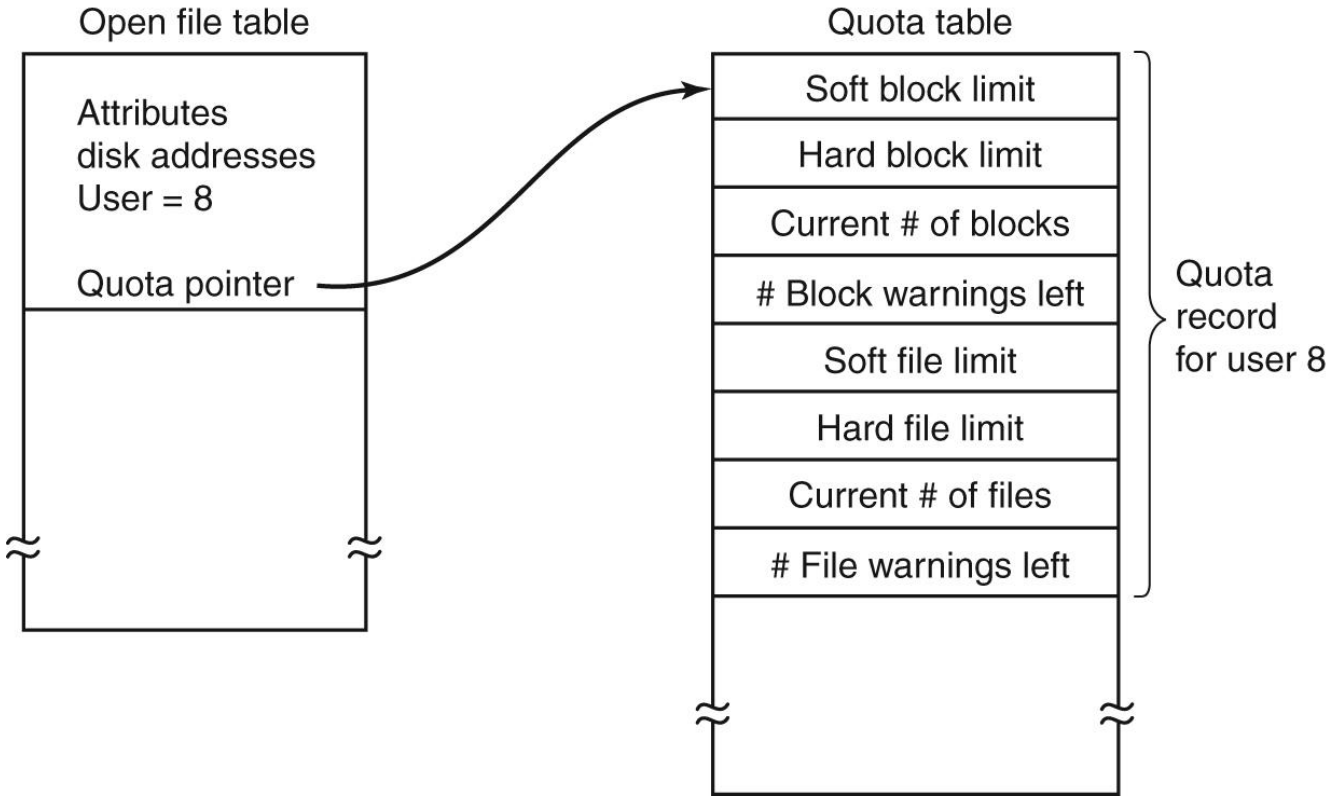


Free blocks are holding the free list.

Disk Space Management: Disk Quotas

- When a user opens file
  - The attributes and disk addresses are located
  - They are put into an open file table in memory
  - A second table contains the quota record for every user with a currently open file.

Disk Space Management: Disk Quotas



File System Backups

- It is usually desirable to back up only specific directories and everything in them than the entire file system.
- Since immense amounts of data are typically dumped, it may be desirable to compress them.
- It is difficult to perform a backup on an active file system.
- Incremental dump: backup only the files that have been modified from last full-backup

File System Backups: Physical Dump

- Starts at block 0 of the disk
- Writes all the disk blocks onto the output tape (or any other type of storage) in order.
- Stops when it has copied the last one.

\+ Simplicity and great speed

\- Inability to skip selected directories and restore individual files.

File System Backups: Logical Dump

- Starts at one or more specified directories
- Recursively dumps all files and directories found there and have changed since some given base date.

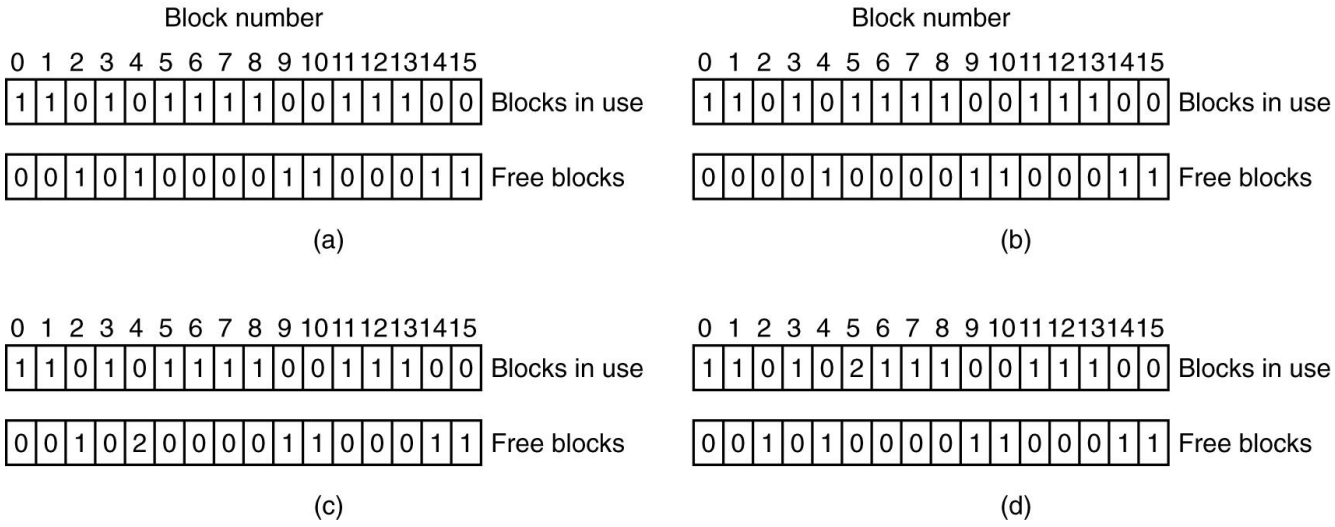
File System Consistency

- Two kinds of consistency checks
  - Blocks
  - Files

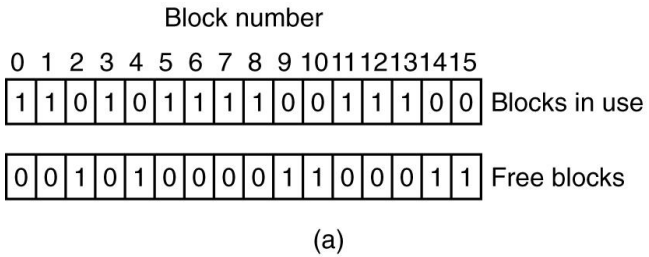
File System Consistency: Blocks

- Build two tables, each one contains a counter for each block, initially 0
- Table 1: How many times each block is present in a file
- Table 2: How many times a block is present in the free list
- A consistent file system: each block has 1 either in the first or second table

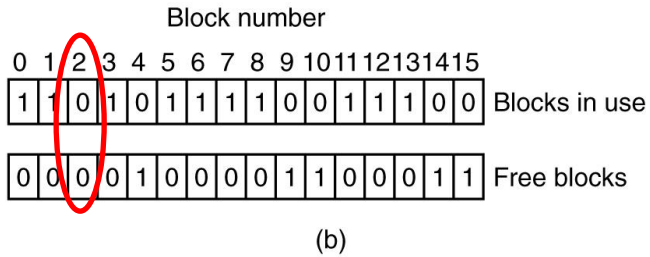
File System Consistency: Blocks

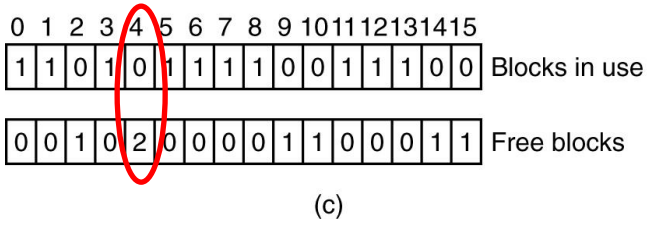


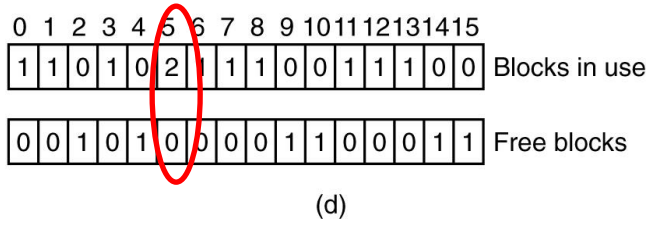
File System Consistency: Blocks



Add the block to the free list







Rebuild the free list

Allocate a free block, make a copy of that block and give it to the other file.

File System Consistency: Files

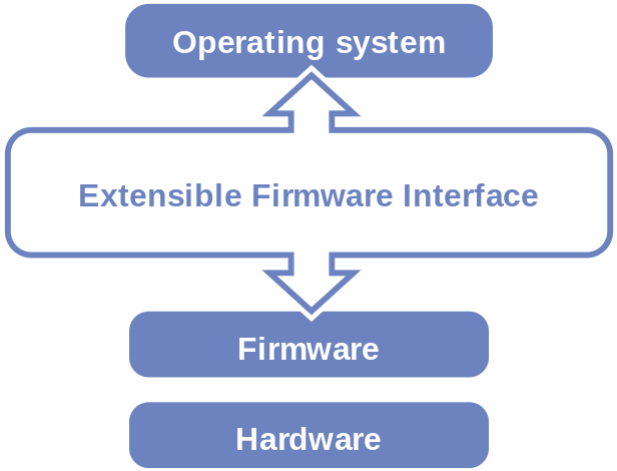
- Table of counters; a counter per file
- Counts the number of that file's usage count.
- Compares these numbers in the table with the counts in the i-node of the file itself.
- Both counts must agree.

File System Consistency: Files

- Two inconsistencies:
  - count of i-node > count in table
  - count of i-node < count in table
- Fix: set the count in i-node to the correct value

Booting an Operating System

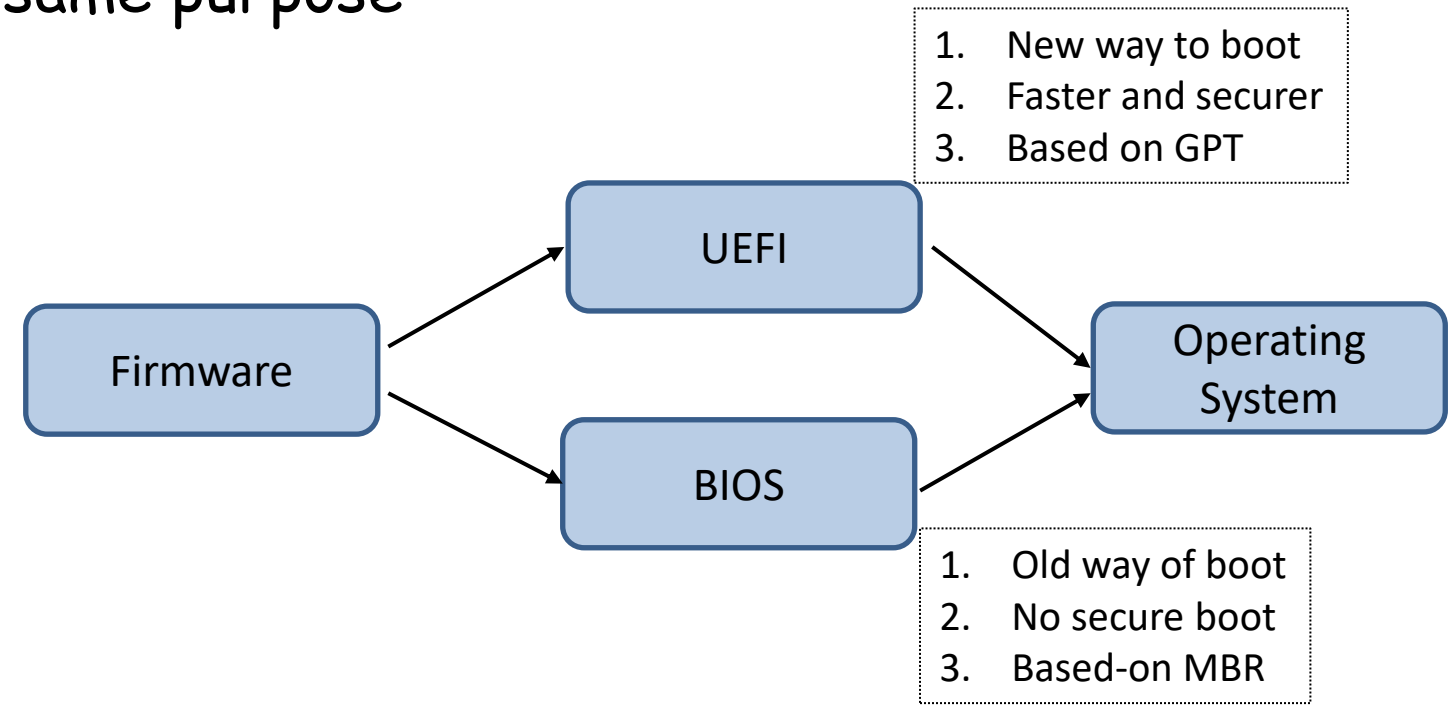
- **Firmware** is a generic term for embedded software (and its included data) to run something. System controllers in large computer systems that control power up etc have a **mini operating system** (typically a **mini** linux) that's referred to as **firmware**.
- Firmware provides standard API to operating system
- Shields Operating System from low level specifics of a platform
- Assists bringup of physical system to enable the boot process
- Enables settings of the platform (boot order, CPU features)



src: wikipedia

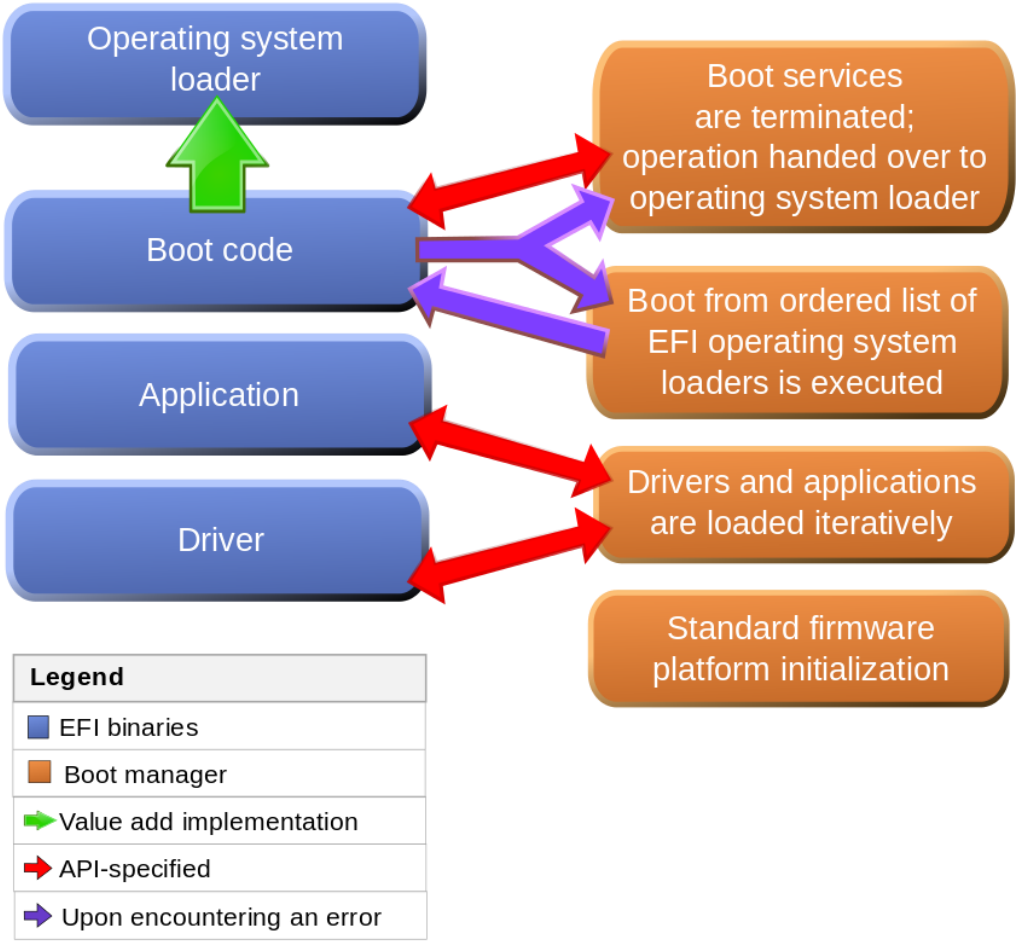
Booting an Operating System

- Two types of firmware based on features
  - BIOS (legacy till about 2012) : **B**asic **I**nput **O**utput **S**ystem
  - UEFI (modern system): **U**nified **E**xtensible **F**irmware **I**nterface
- different layouts of the disk and components, though same purpose



UEFI

(Universal Extensible Firmware Interface)

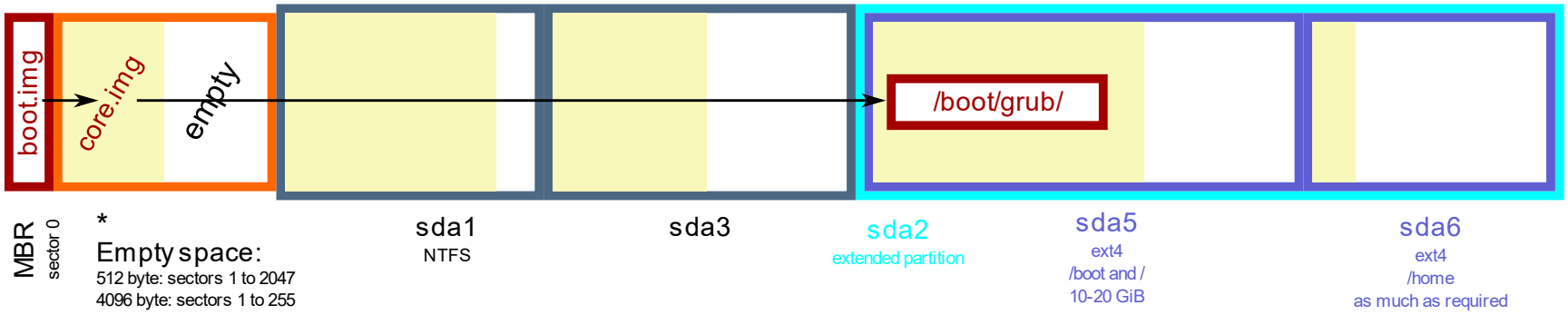


Disk Layout of bootable Disk

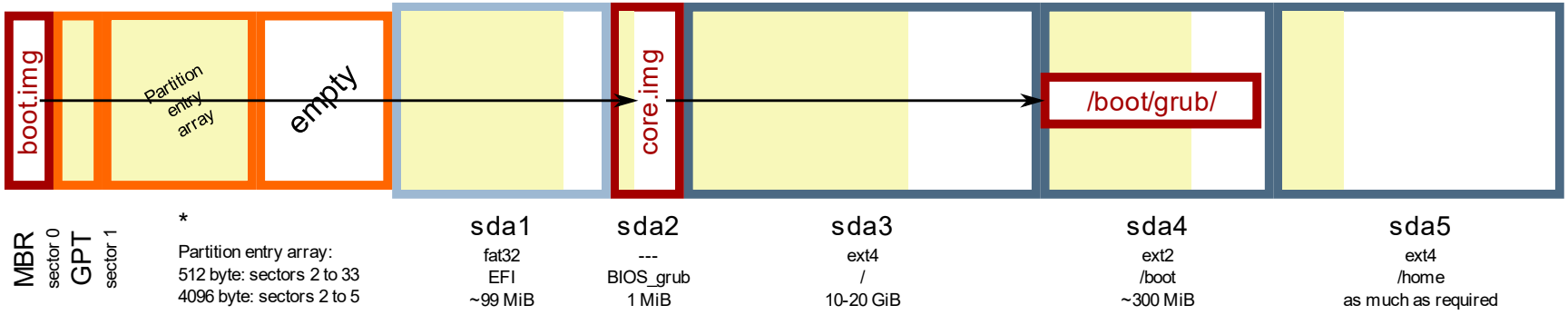
- Need to get to the bootloader (grub) which ultimately loads the OS

Locations of *boot.img*, *core.img* and the */boot/grub* directory

Example 1: An MBR-partitioned hard disk with sector size of 512 or 4096 bytes



Example 2: A GPT-partitioned hard disk with sector size of 512 or 4096 bytes



Bootstrapping (booting in steps)

- boot.img:
  This image is the first part of GRUB to start. It is written to a master boot record (MBR) or to the boot sector of a partition. Because a PC boot sector is 512 bytes, the size of this image is exactly 512 bytes. The sole function of boot.img is to read the first sector of the core image from a local disk and jump to it. Because of the size restriction, boot.img cannot understand any file system structure, so grub-install hardcodes the location of the first sector of the core image into boot.img when installing GRUB.

- core.img:
  This is the core image of GRUB. It is built dynamically from the kernel image and an arbitrary list of modules by the grub-mkimage program. Usually, it contains enough modules to access /boot/grub, and loads everything else (including menu handling, the ability to load target operating systems, and so on) from the file system at run-time. The modular design allows the core image to be kept small, since the areas of disk where it must be installed are often as small as 32KB.

- /boot/grub:
  holds all the modules required for boot including reading disk, menus, graphics handling ..

Booting (kernel)

- The boot loader (grub) identifies the kernel that is to be loaded and copies this kernel image and any associated initrd into memory.
- vmlinuz : compressed kernel code
- The initial RAM disk (initrd) is an initial file system that is mounted prior to when the real root file system is available. The initrd is bound to the kernel and loaded as part of the kernel boot procedure. The kernel then mounts this initrd as part of the two-stage boot process to load the modules to make the real file systems available and get at the real root file system.
- The initrd contains a minimal set of directories and executables to achieve this, such as the insmod tool to install kernel modules into the kernel.

```
frankeh@lnx1:/boot$ ls -ls | egrep "0-91"
  216 -rw-r--r-- 1 root root   217457 Feb 28 05:45 config-4.15.0-91-generic
60784 -rw-r--r-- 1 root root 62235093 Apr 15 19:27 initrd.img-4.15.0-91-generic
 3976 -rw------- 1 root root  4069388 Feb 28 05:45 System.map-4.15.0-91-generic
 8180 -rw------- 1 root root  8375960 Feb 28 05:51 vmlinuz-4.15.0-91-generic
```

- Config = kernel compile configuration ; System.map = symboltable

Conclusions

- Files and file systems are major parts of an OS
- Files and File system are the OS way of abstracting storage.
- There are different ways of organizing files, directories, and their attributes.
  - However, VFS is a unifying way to represent a common API
- Remember Murphy's law: What can go wrong will.
  - Please backup your System / Data !!!!!