Chapter 4

File Systems

Chapter 4. File Systems

Long-term Information Storage

1. Must store large amounts of data
2. Information stored must survive the termination of the process using it - persistent
3. Multiple processes must be able to access the information concurrently

➔ solution: store in units called files

➔ Information in files must be persistent

➔ File system: The study of how files are structured, named, accessed, used, protected, implemented, and managed

File Systems (2)

Think of a disk as a *linear sequence of fixed-size blocks* and supporting reading and writing of blocks. Questions that quickly arise:

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

4.1 Files

- **User point of view of a file system**
  - **Files and Directories**

File Naming

- Files are abstraction mechanism to store information
- Hide details of secondary storage from user
- Naming is essential for management of stored information

File extension

(Unix : just conventions, Windows : aware of its meaning)

Extension	Meaning
file.bak	Backup file
file.c	C source program
file.gif	Compuserve Graphical Interchange Format image
file.hlp	Help file
file.html	World Wide Web HyperText Markup Language document
file.jpg	Still picture encoded with the JPEG standard
file.mp3	Music encoded in MPEG layer 3 audio format
file.mpg	Movie encoded with the MPEG standard
file.o	Object file (compiler output, not yet linked)
file.pdf	Portable Document Format file
file.ps	PostScript file
file.tex	Input for the TEX formatting program
file.txt	General text file
file.zip	Compressed archive

File Structure

(a) Byte sequence

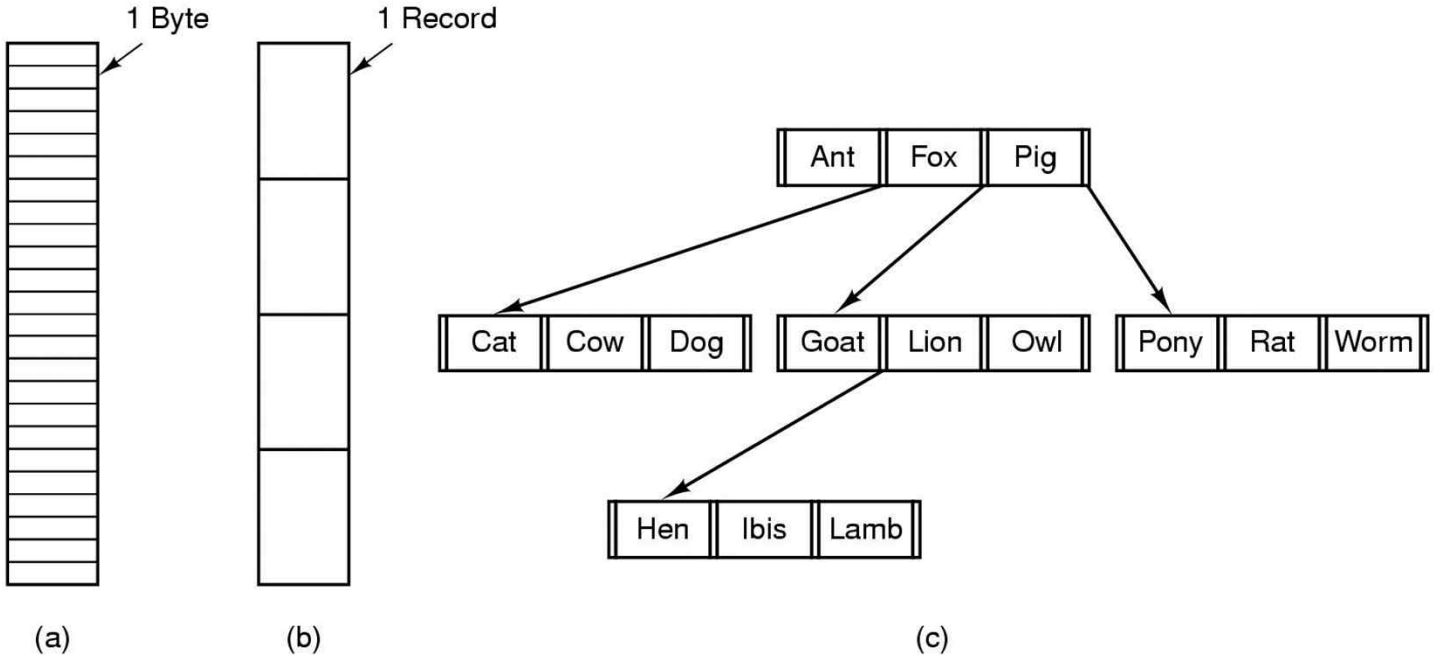
- Files are nothing but a sequence of bytes → meaning imposed by user
- UNIX and Windows

(b) Record sequence

- Fixed-length records have internal structure

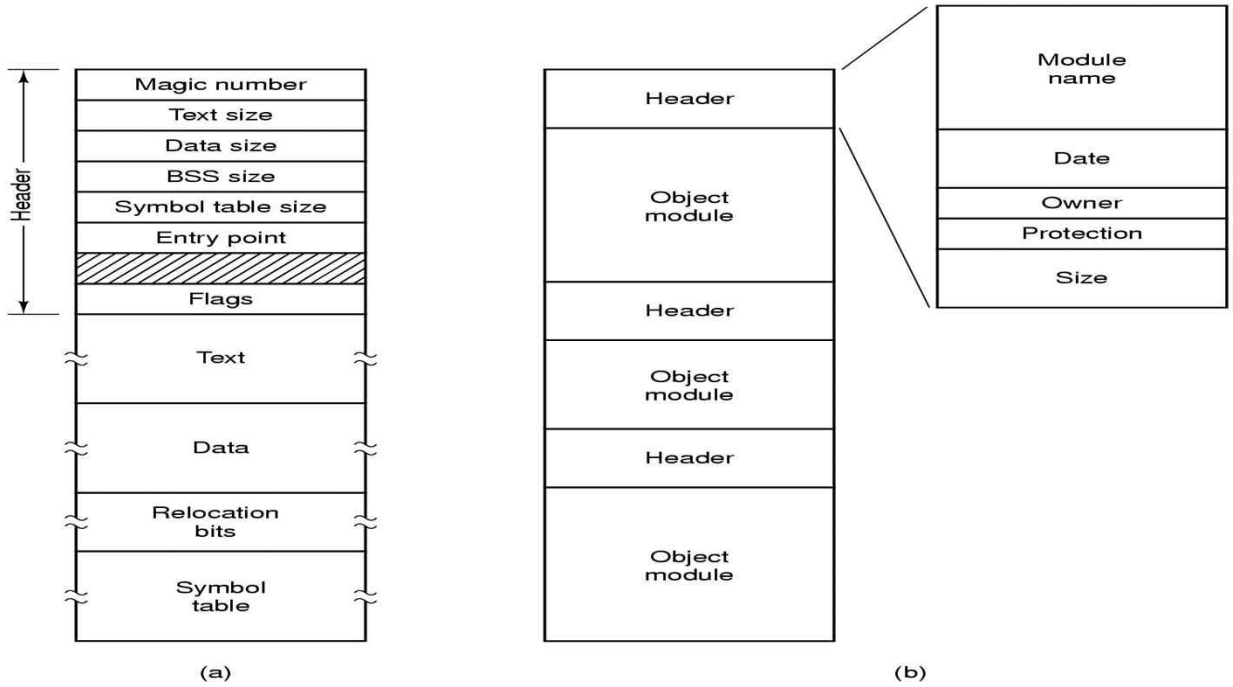
(c) Tree

- Tree of records referenced through a key
- large mainframe computers



File Types

- **OS supports various types of files**
  - **Regular file vs directories**
  - **Character special files vs block special files**
- **Regular files**
  - **ASCII : displayed and printed as is**
  - **binary files**
    - » Magic number : identify file types
    - » (a) executable (b) archive



File Access

- **Sequential access**
  - **read all bytes/records from the beginning**
  - **cannot jump around, could rewind or back up**
  - **convenient when medium was mag tape**
- **Random access**
  - **bytes/records read in any order**
  - **essential for data base systems**
  - **read can be …**
    - » move file marker (**seek**), then read or …
    - » read and then move file marker

File Attributes

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 has last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Possible file attributes

Basic operations

File Operation

1. Create
2. Delete
3. Open
4. Close
5. Read
6. Write
7. Append
8. Seek
9. Get attribute
10. Set attribute
11. Rename

Unix

- create(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)
- rename(old, new)
- fcntl(fd, cmd, *lock)

NT

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

An Example Program Using File System Calls (1/2)

copyfile abd xyz

```
/* 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 */
```

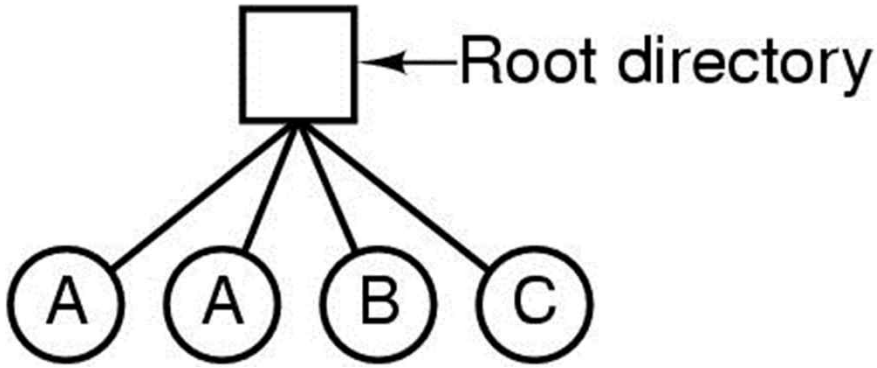
An Example Program Using File System Calls (2/2)

```
    /* 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 */
}
```

4.2 Directories
Single-Level Directory Systems

- **Directories(folders) provide:**
  - **a way for users to organize their files**
  - **a convenient file name space for both users and FS's**

- **A single level directory system**
  - **contains 4 files**
  - **owned by 3 different people, A, B, and C**



Hierarchical Directory Systems

- **Most file systems support multi-level directories**
  - **naming hierarchies (/, /usr, /usr/local, /usr/local/bin, …)**

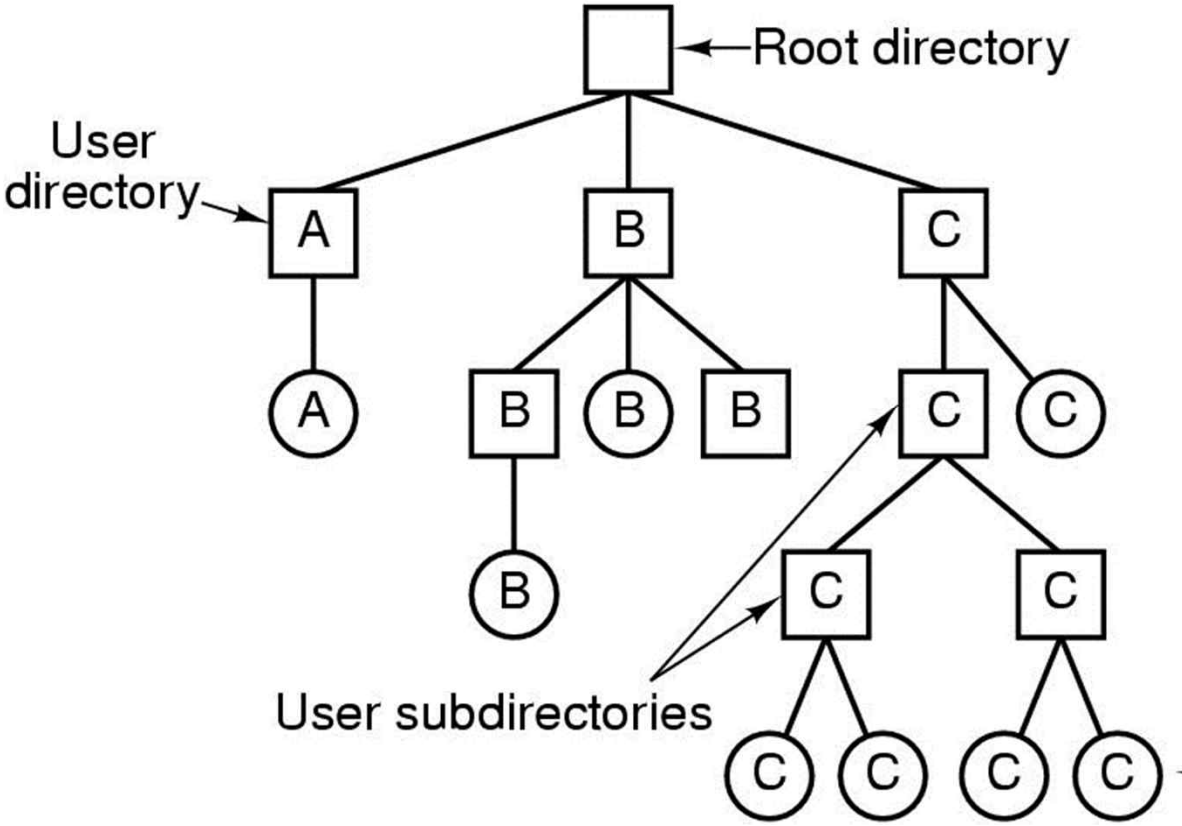
- **Path Names**
  - Most file systems support the notion of current directory (working directory/current directory)
    - » **“.”** - current directory, **“..”** – parent directory
  - absolute path names: fully-qualified starting from root of FS

    ```
    bash$ cd /usr/local
    ```

  - relative path names: specified with respect to current directory

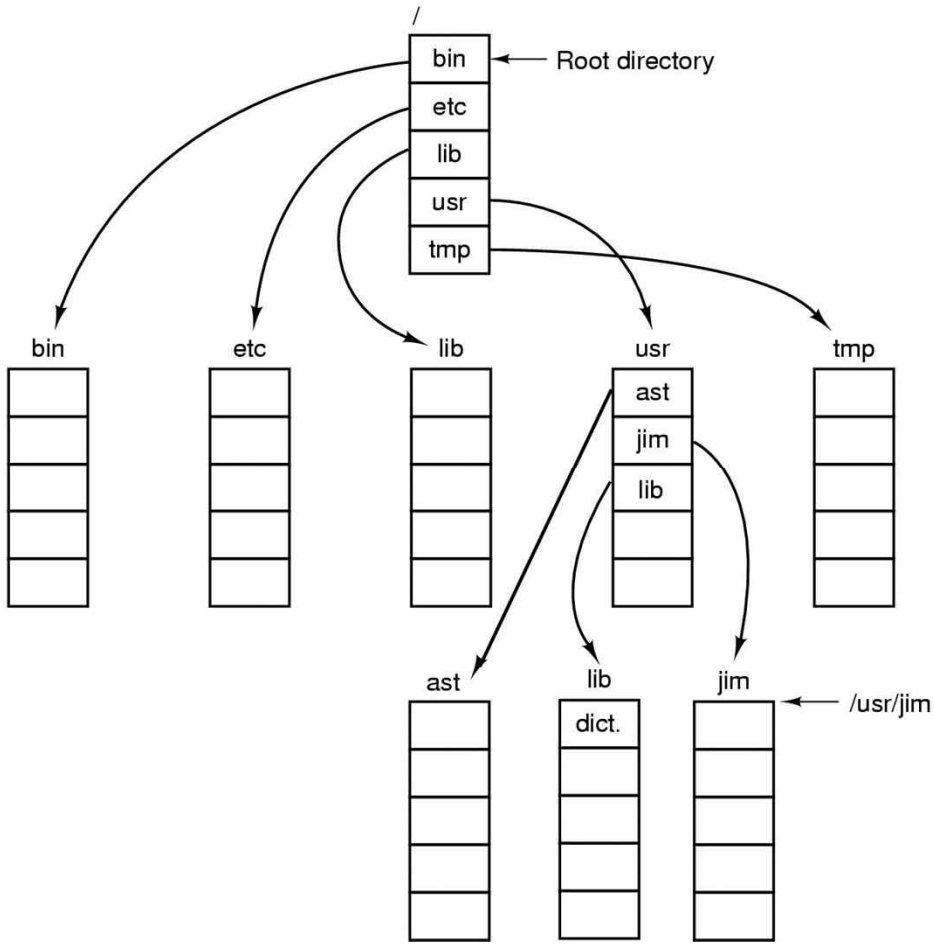
    ```
    bash$ cd /usr/local   (absolute)
    bash$ cd bin          (relative, equivalent to cd /usr/local/bin)
    ```

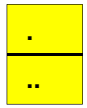
Hierarchical Directory Systems



A hierarchical directory system

Path Names





A UNIX directory tree

Directory Operations

1. Create
2. Delete
3. Opendir
4. Closedir
5. Readdir
6. Rename
7. Link
8. Unlink

Hard link
Symbolic link

Directory Internals

- **A directory is typically just a file that happens to contain special metadata**
  - **directory = list of (name of file, file attributes)**
  - **attributes include such things as:**
    - » size, protection, location on disk, creation time, access time, …
  - **the directory list is usually unordered (effectively random)**
    - » when you type “ls”, the “ls” command sorts the results for you

Path Name Translation

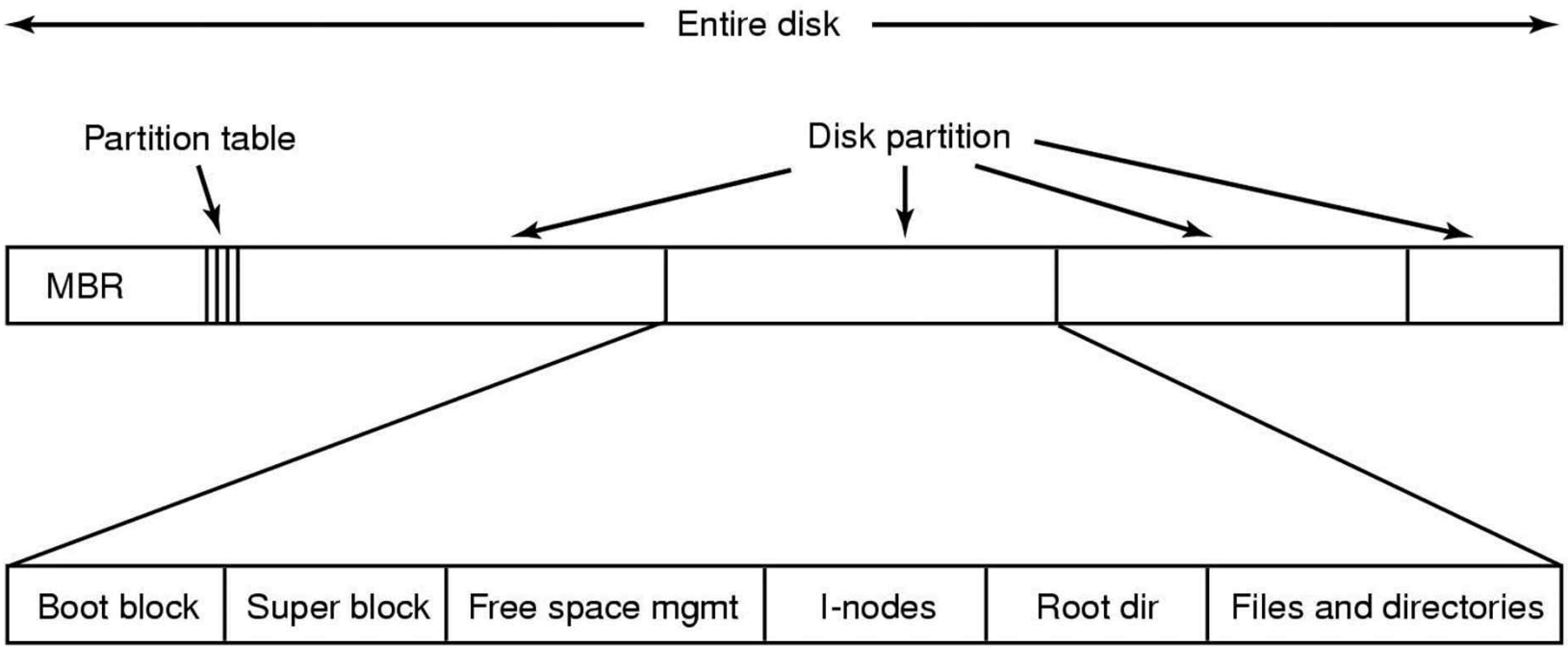
- Let's say you want to open "/one/two/three"

```
fd = open("/one/two/three", O_RDWR);
```

- What goes on inside the file system?
  - open directory "/" (well known, can always find)
  - search the directory for "one", get location of "one"
  - open directory "one", search for "two", get location of "two"
  - open directory "two", search for "three", get loc. of "three"
  - open file "three"
  - (of course, permissions are checked at each step)
- FS spends lots of time walking down directory paths
  - this is why open is separate from read/write (session state)
  - OS will cache prefix lookups to enhance performance
    - » /a/b, /a/bb, /a/bbb all share the "/a" prefix

File System Implementation

File System Layout



A possible file system layout

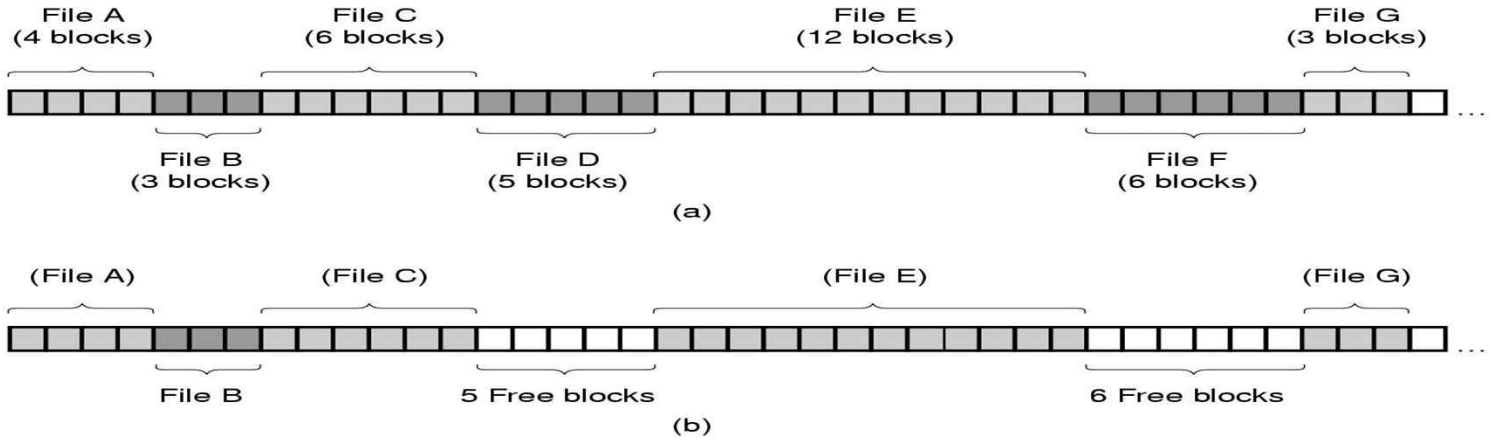
File System Layout

- **Each partition has an independent file system.**
- **MBR(Master Boot Record)**
  - **Used to boot the computer**
  - **Partition table**
    - Starting and ending addresses of each partition
    - One of the partitions is marked as active
- **Boot sequence**
  1. **BIOS reads in and executes the MBR.**
  2. **MBR locates the active partition, reads in its first block, called the boot block, and executes it.**
  3. **The program in the boot block loads the operating system contained in that partition.**

Layout of a Disk Partition

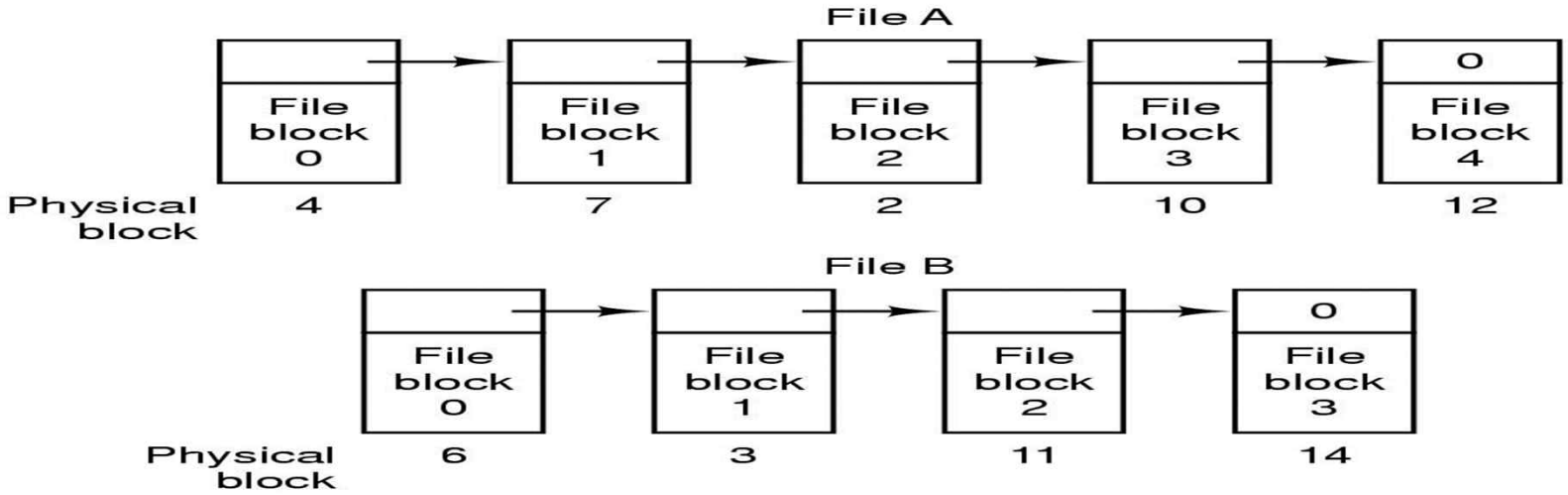
- **Boot block**
- **Superblock**
  - **Key parameters of the file system**
    - Magic number to identify the file system type
    - Number of blocks etc.
- **Free space mgmt**
  - **A bitmap or a list of pointers**
- **I-nodes**
  - **Attributes and disk addresses of the file's blocks**
- **Root dir**
- **Files and directories**

4.3.2 Implementing Files
Contiguous Allocation



- **Pros**
  - Simple to implement: one disk address + number of blocks
  - Excellent read performance: only one seek
- **Cons**
  - Disk fragmentation → impractical
- **Applicability**
  - CM-ROM file system
  - File size known in advance
  - No change in size

Linked List Allocation

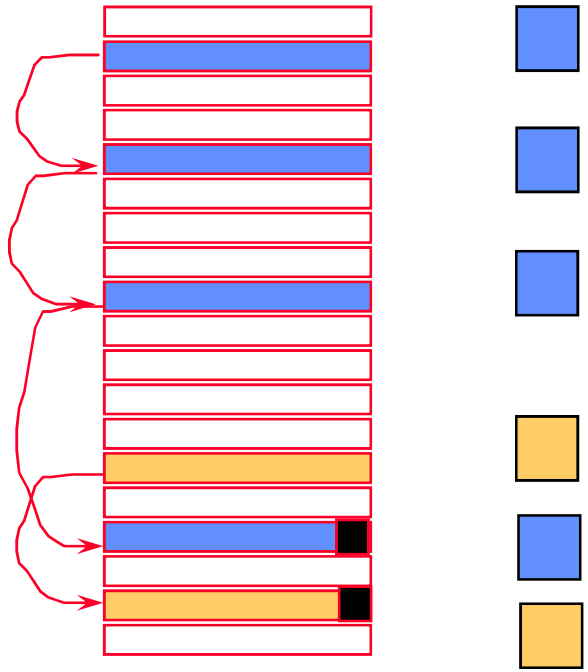


- **Pros**
  - **No disk fragmentation**
  - **All blocks can be used**
- **Cons**
  - **Slow for both sequential and, especially for random access**
  - **Lost bytes for pointer → degrades performance**
    - **Reads of the full block size require two block reads and concatenation**

Linked List Allocation using a Table in Memory

File Allocation Table

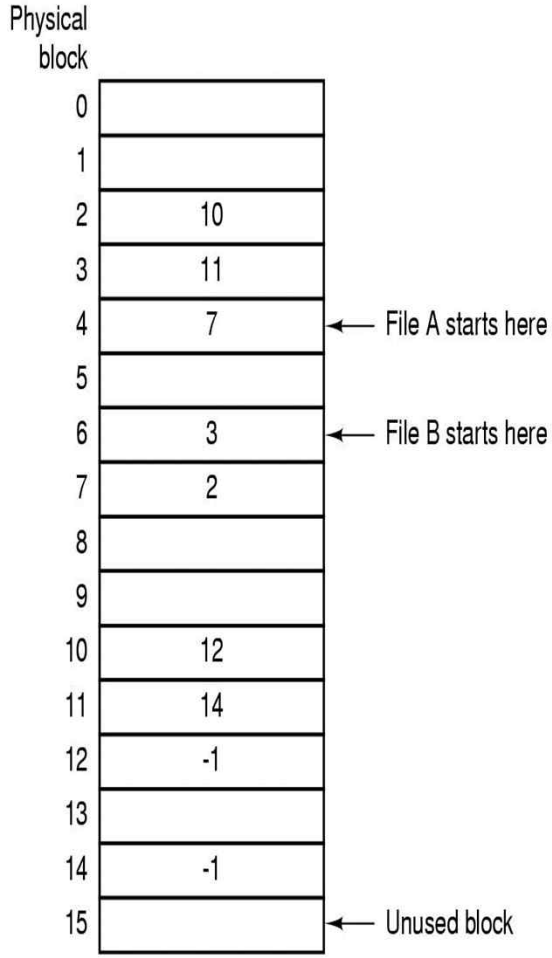
- One file allocation table in memory describes the layout of the entire disk.
- Each entry in the table refers to a specific cluster within a file.
  - zero says cluster not used.
  - not zero says where the next FAT entry for the file is.
- A file's directory entry points to the first FAT entry for the file.



The FAT is on disk in a special sequence blocks.

Limitations of FAT

- FAT index is 16 bits.
  - 1 disk can have up to 64K clusters.
- As disks get bigger, cluster size must increase.
- Big clusters yield internal fragmentation.
  - 10 to 20% wastage for 16KB clusters not uncommon.
- Minimum of one file per cluster.
  - limitation to 64K files.
- The FAT itself is a critical resource.
  - You lose the FAT on disk, you've lost the whole disk.



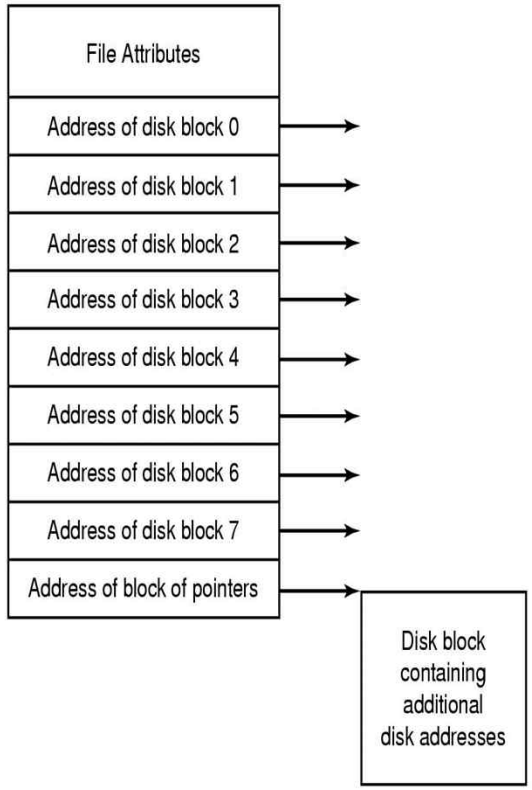
FAT (File Allocation Table)

- **Advantages**
  - **Random access is much easier.**
  - **The chain can be followed without making any disk references.**
  - **The dir entry keeps the starting block number.**

- **Disadvantages**
  - **The entire table must be in memory all the time : the table is proportional in size to the disk.**
    - 200G disk and a 1-KB block size => 200M entries take up 800MB (4bytes/entry) in main memory!

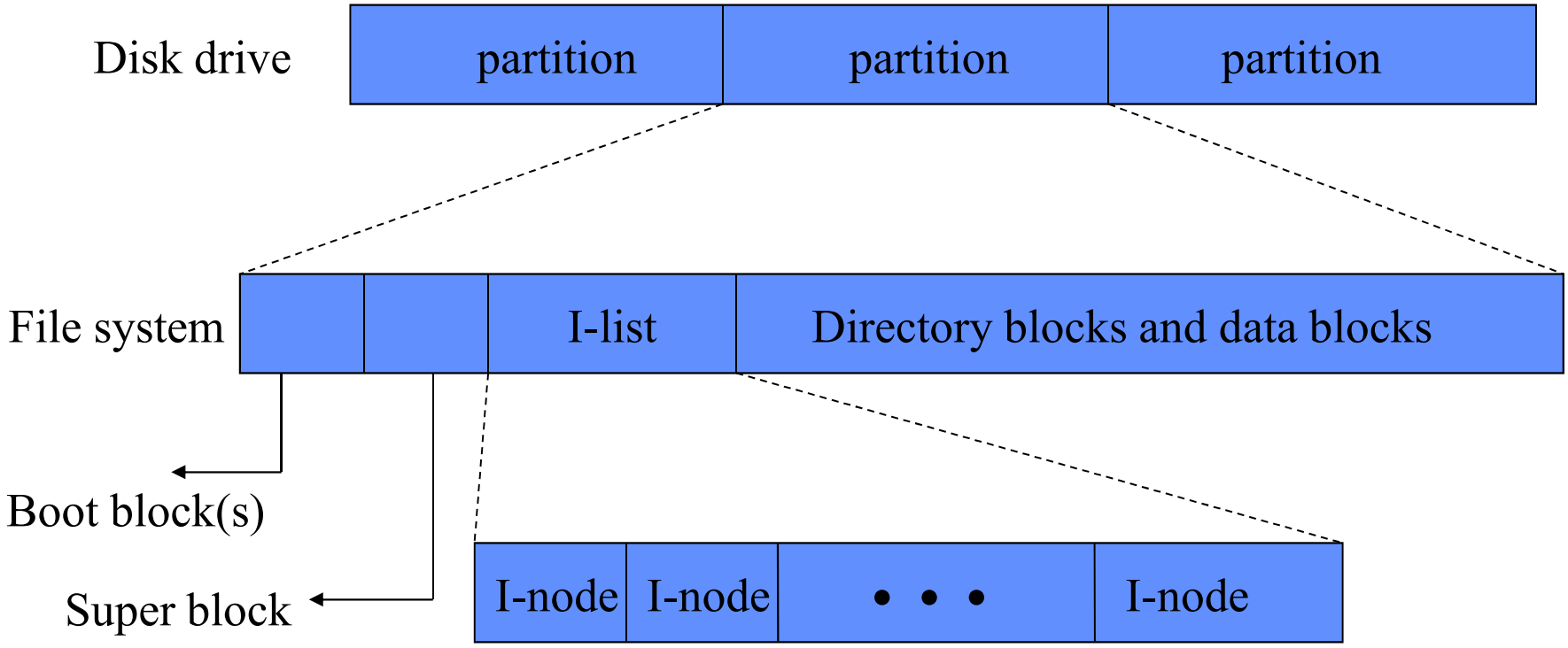
I-nodes (index-node)

- **index-node**
  - List attributes and disk addresses of a file
  - Last disk address for the address of a block containing more disk block addrs
- **Only need to keep inodes of files *that are open* → occupies small main memory**
  - FAT requires the whole table in memory



Overview of the Unix File Systems

- **File system : consist of a sequence of *logical* blocks**

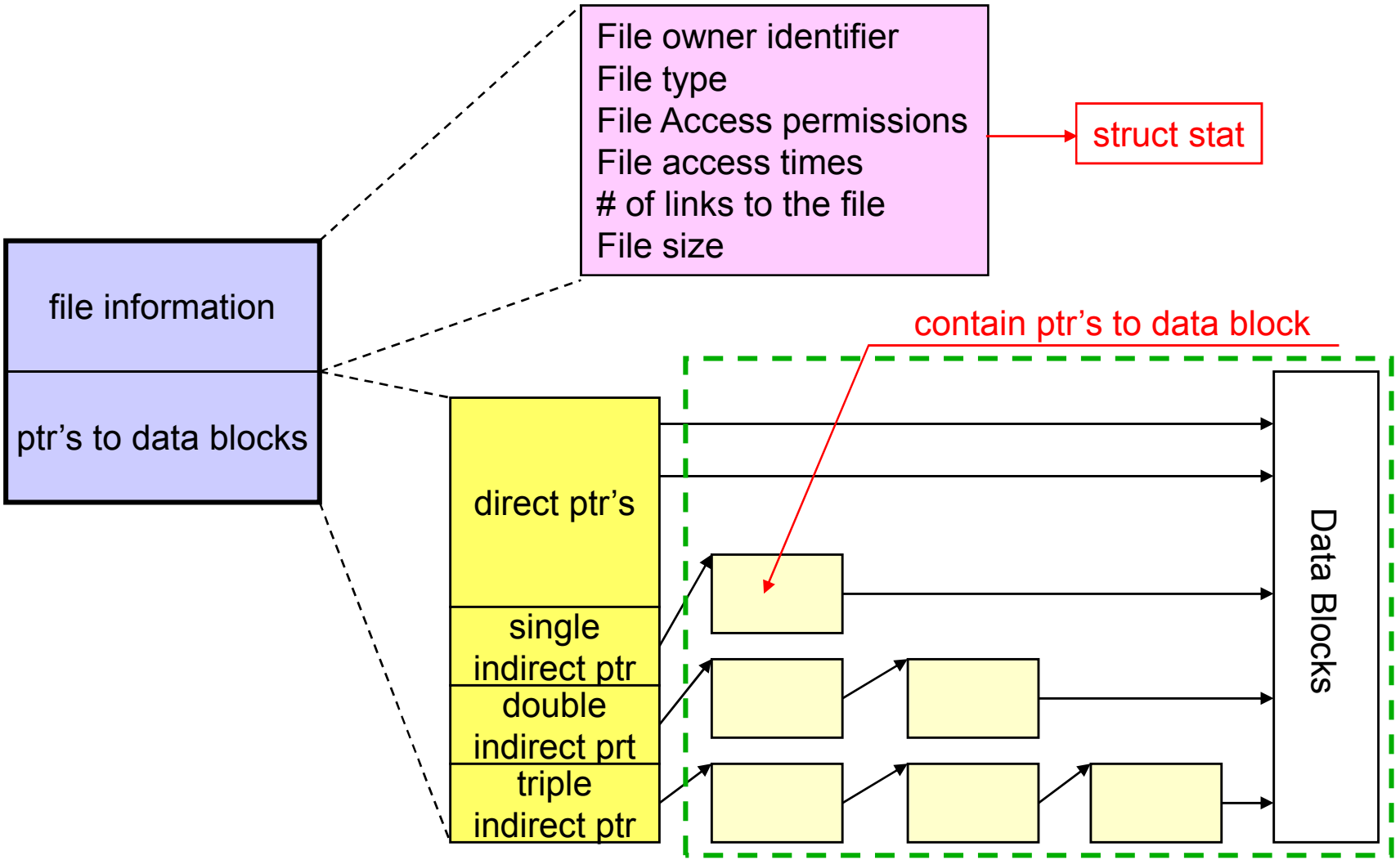


Disk layout of System V File System (s5fs)

Overview of the Unix File Systems

- **Super block**
  - **contains**
    - The size of the file system, the size of the I-node list
    - The number of free blocks/I-nodes in the file system
    - A list of free blocks/I-nodes in the file system
    - The index of the next free block/I-node in the free block/I-node list
- **I-node : internal representation of a file**
  - **contains**
    - description of the disk layout of the file data
    - file information (e.g. owner, access permission, access time, a link count…)
  - **one-to-one mapping with a file (cf. link())**
- **Directory**
  - **file name**
  - **I-node number (in the *same* file system, not in the different file system)**
- **Data blocks**

I-node


File owner identifier
File type
File Access permissions
File access times
# of links to the file
File size
struct stat
file information
ptr's to data blocks
contain ptr's to data block
direct ptr's
single indirect ptr
double indirect prt
triple indirect ptr
Data Blocks

The UNIX V7 File System (2)

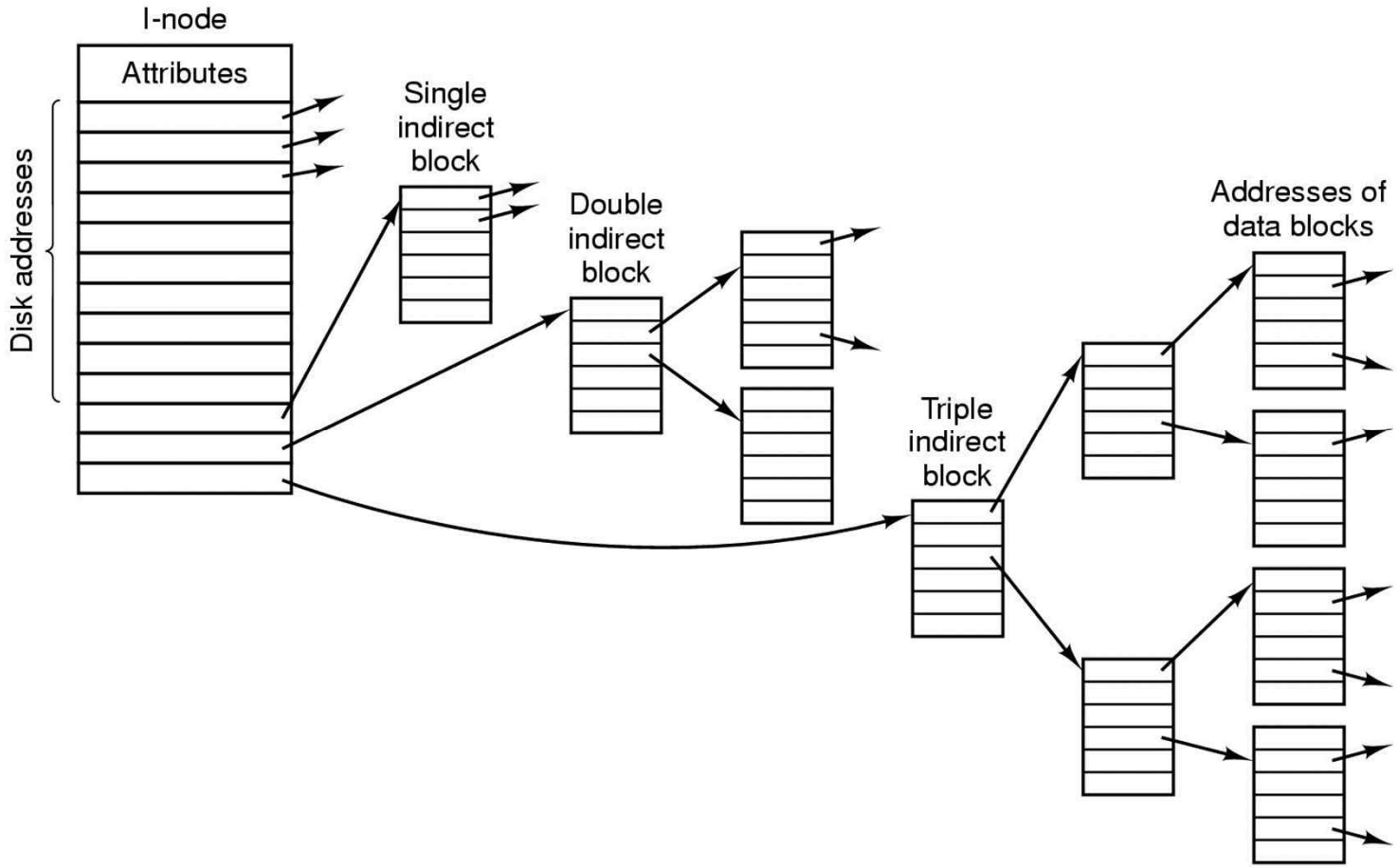
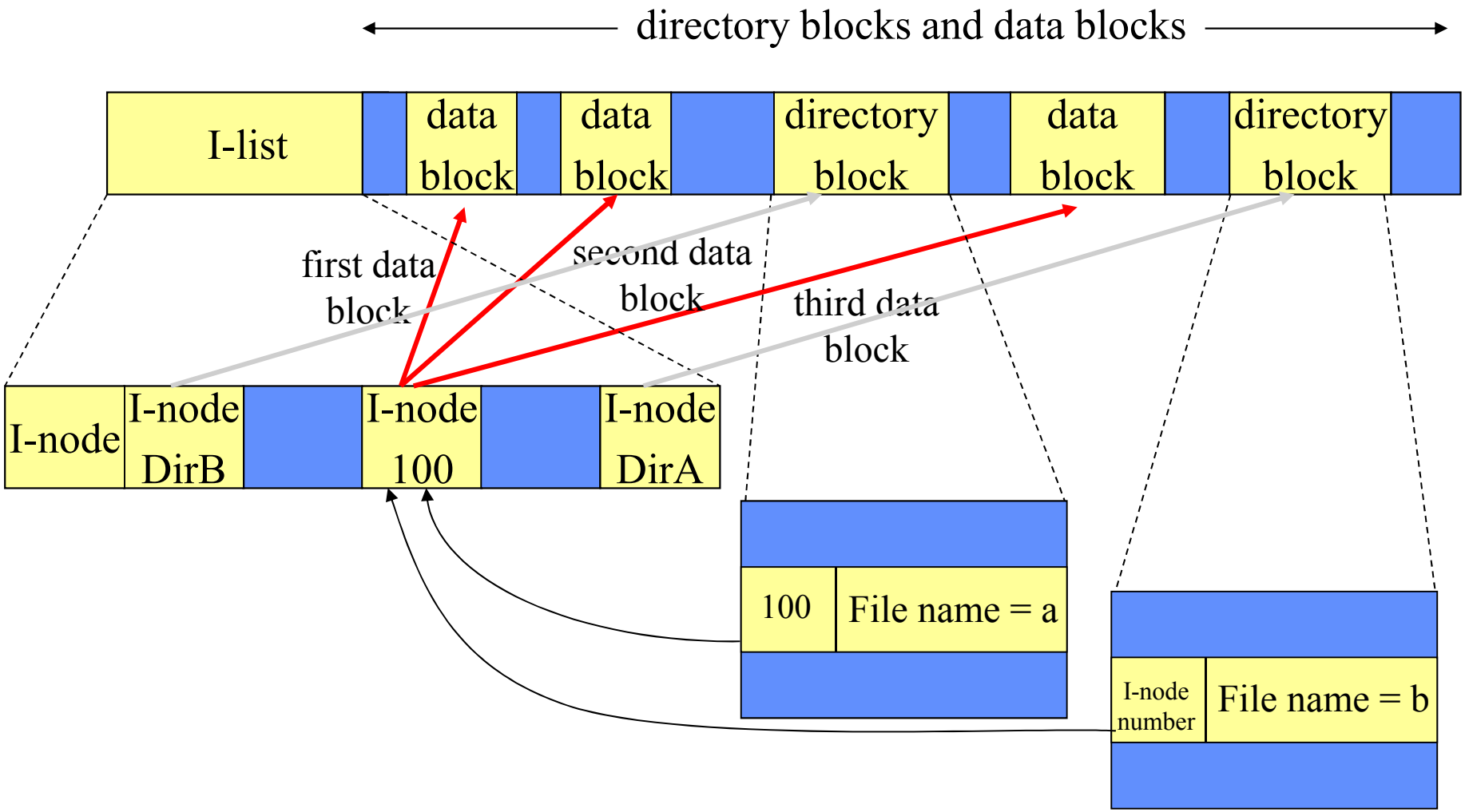


Figure 4-34. A UNIX i-node.

File system in more detail



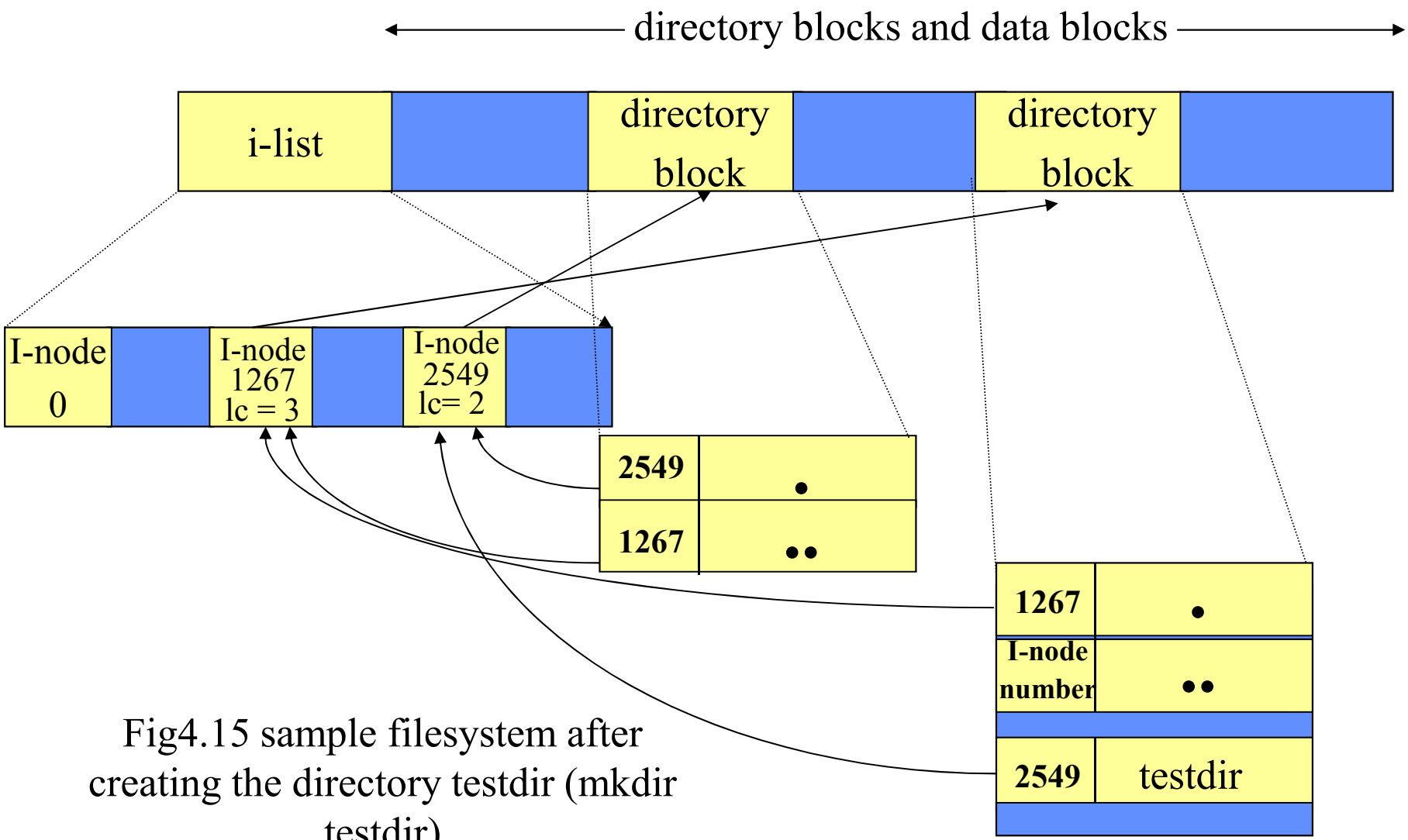
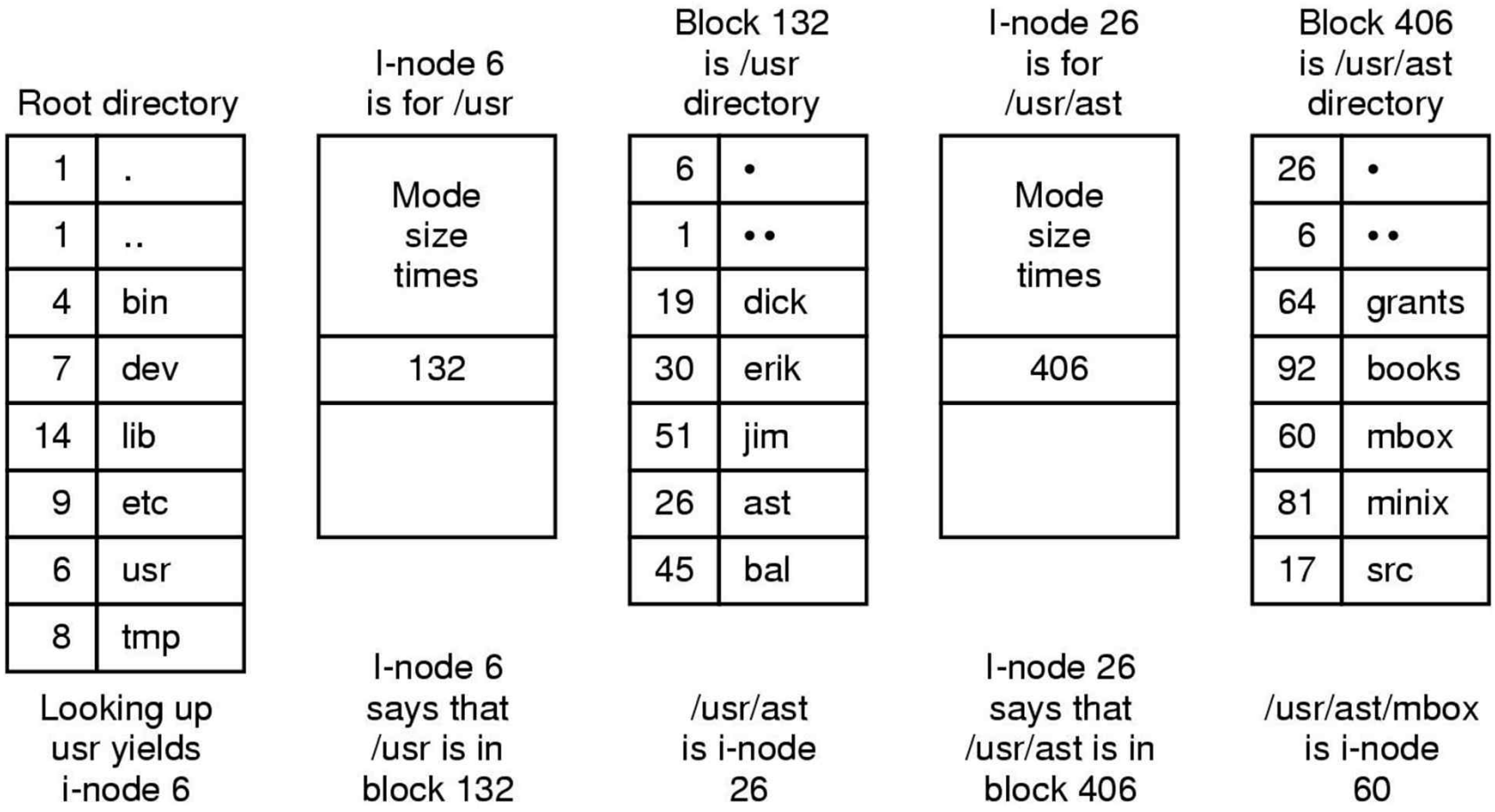


Fig4.15 sample filesystem after creating the directory testdir (mkdir testdir)

Example: UNIX V7



The steps in looking up */usr/ast/mbox*

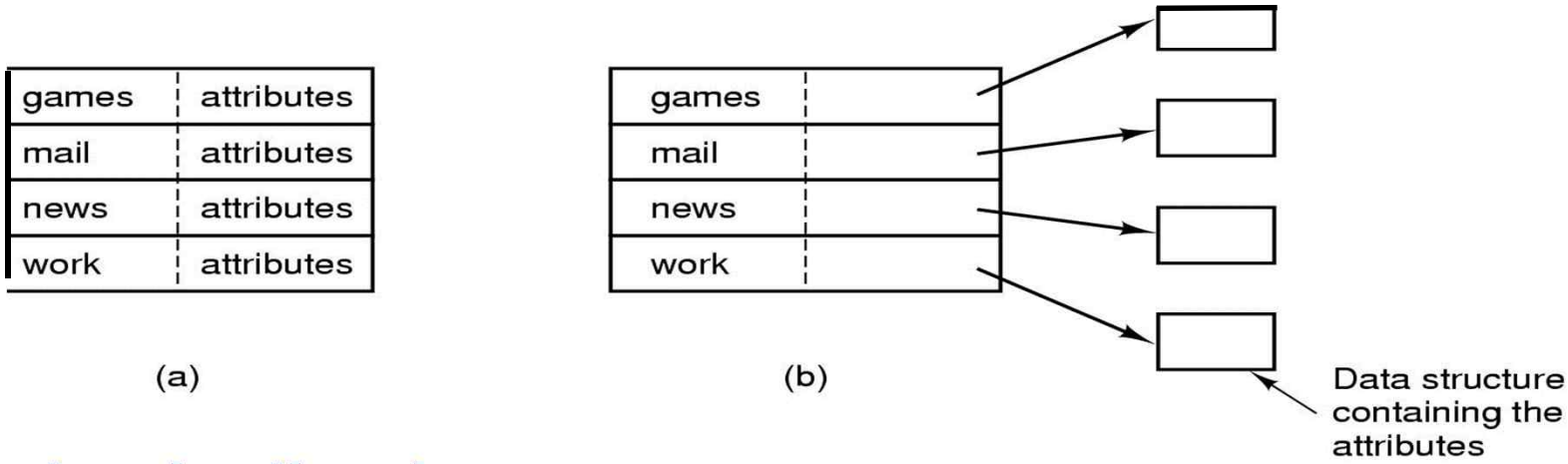
4.3.3 Implementing Directories

- **Main function of directory system**
  - Map ASCII name of file to information needed to locate data
- **Need to consider attributes of files**
  - Where the attributes should be stored?

Directory Internals

- **A directory is typically just a file that happens to contain special metadata**
  - **directory = list of (name of file, file attributes)**
  - **attributes include such things as:**
    - » size, protection, location on disk, creation time, access time, …
  - **the directory list is usually unordered (effectively random)**
    - » when you type “ls”, the “ls” command sorts the results for you

Fixed Length Directory Entry



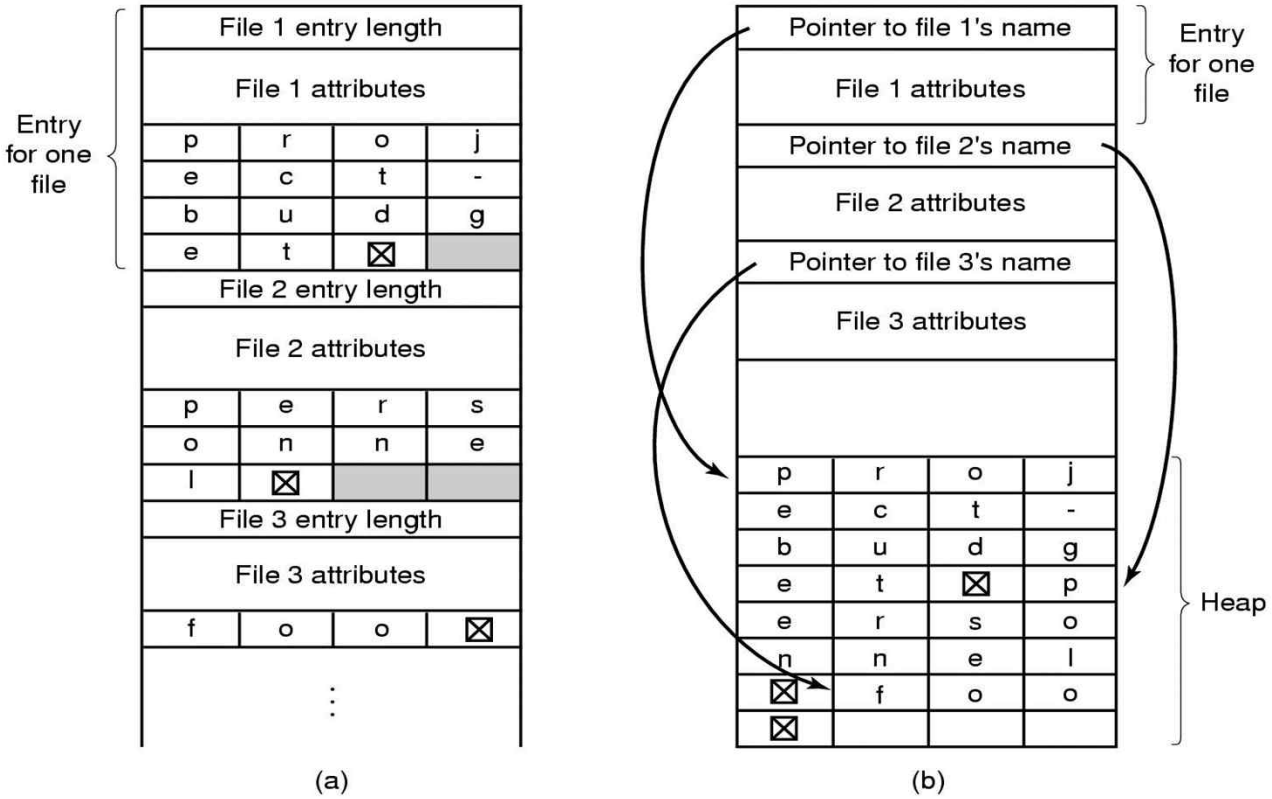
(a) A simple directory

- fixed size entries
- disk addresses and attributes in directory entry
- MS-DOS/Windows

(b) Directory with i-nodes

- fixed size entries + i-node number
- disk addresses and attributes in i-nodes
- UNIX

Handling Long File Names



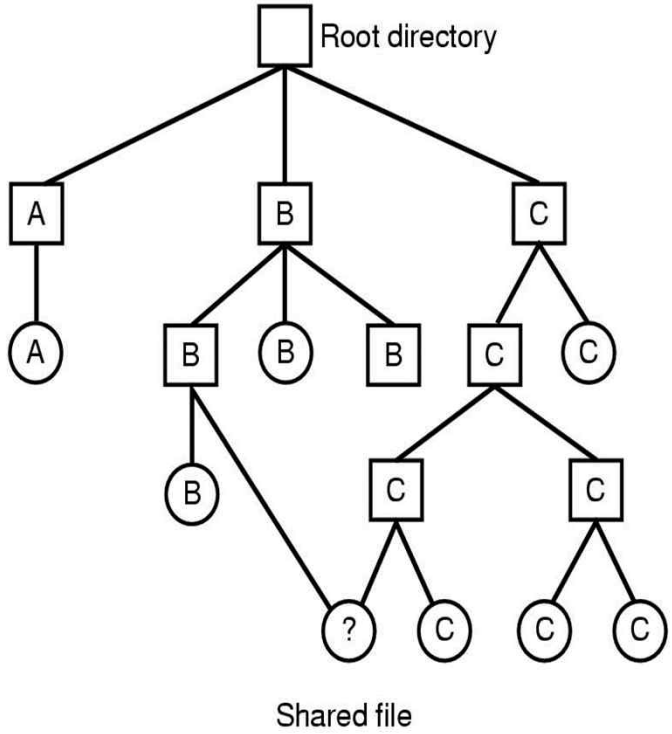
- **In-line**
  - **When a file is removed, a hole can be introduced.**
- **In a heap**
  - **No need for file names to begin at word boundaries**

Speeding up the File Name Search

- **Hash table**
  - Hashes by the file name
  - More complex administration
  - Works if directories contain a large number of files
- **Caching**
  - Caches the results of searches
  - Works if a small number of files comprise the majority of the lookups

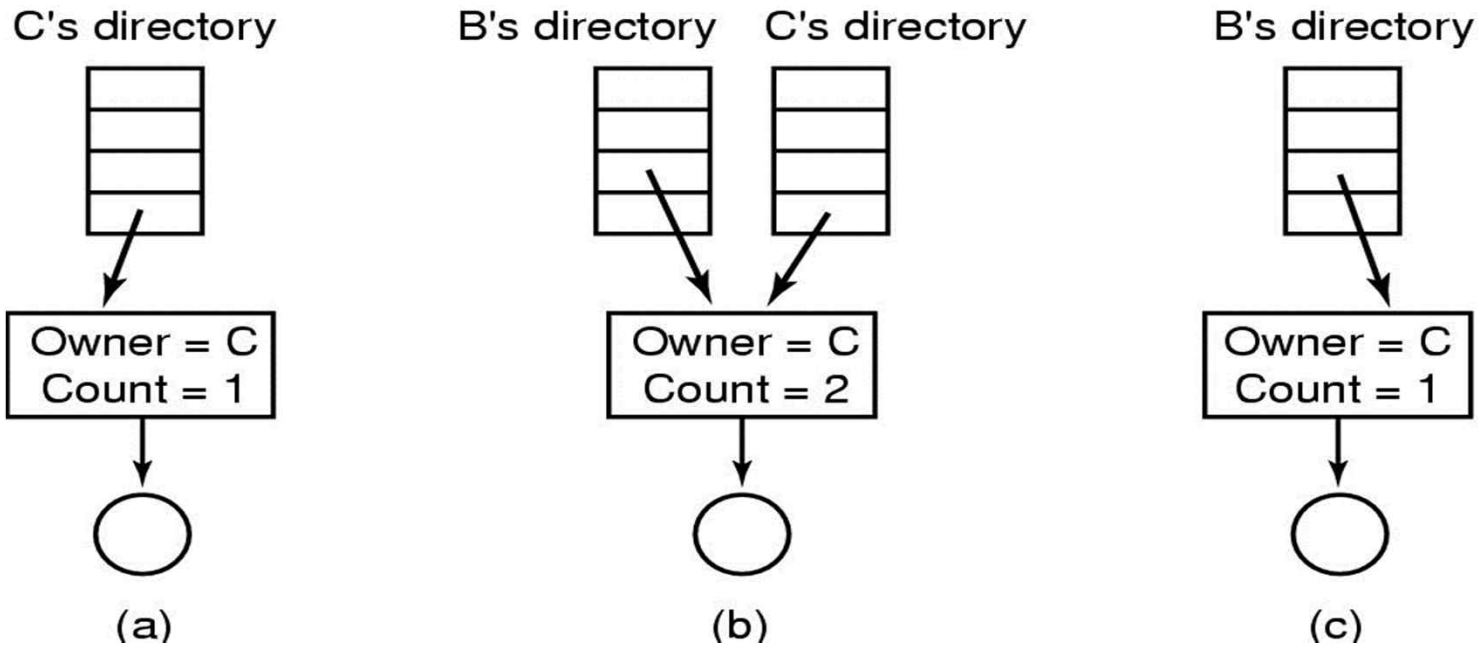
4.3.4 Shared Files

- **Directories sharing same file cannot both have disk addresses**
  - **Change of disk address will not be reflected on the other**
- **Solution**
  - **Using i-nodes: point to same i-node**
  - **Symbolic link**
    - » Create new file of type LINK
    - » Enter the path name in this file
      - File /B/B/B1 has path name "/C/C/C/C1"



Shared file

Problems with Sharing Files



- **Problem with i-nodes**
  - **Diagram above, when C deletes the shared file**
    - » B has C's file that C wants to delete, causing prob. such as quotas
- **Problem with symbolic links**
  - **Performance overhead for following path in LINK file**
    - » Have to access the link file first; more disk accesses

Shared Files (3)

- **Hard Link**
  - A directory entry points to a data structure associated with the file.
  - Does not change the ownership, but increases the link count
  - Hard to find all the directory entries for a file to be deleted

- **Symbolic Link**
  - A directory entry for a new file of type LINK contains the path name of the file to be linked.
  - Extra disk access to reach a file
  - Extra i-node (and disk block to store the path)
  - Can be used to link files on different machines

4.3.5 Log-Structured File Systems

- **Assumption – most disk accesses will be writes**
  - **Most reads are satisfied by big file system caches**
  - **Writes are done in very small chunks**
    - » Creating a new file needs writes of i-node for the directory, the directory block, i-node for the file, and file itself.
  - **To achieve the full disk bandwidth even with many small random writes**
- **The entire file system structure is a log**
  - **All writes buffered in memory are collected into a single segment and written to the disk as a log**
  - **A log may contains i-node, directory blocks, data blocks and a segment summary**
  - **I-node map to find i-nodes scattered all over the log**
    - » Kept in disk and usually cached in memory
  - **Cleaner frees obsolete segments for next logs**
    - » Active data in a segment is collected into a new segment (log)
    - » Bookkeeping is not trivial – find a i-node for data block moved!!

4.3.6 Journaling File Systems

- **Improve file system robustness in the face of failure**
- **Keep a log of what the file system is going to do before it does**
  - **Keeps track of the changes it intends to make in a journal before committing them to the main file system**
  - **After operations logged is completed, the log entry is erased**
  - **The logged operations must be idempotent**
    - » Can be repeated as often as necessary without harm
    - » Ex) Update the bitmap to mark block N as free
- **Removing a file**
  1. **Remove the file from its directory**
  2. **Release the i-node to the pool of free i-nodes**
  3. **Return all the disk blocks to the pool of free disk blocks**
  - **What happens if a system crashes between each steps?**
  - **Log 1,2,3 first and performed operations.**

4.3.7 Virtual File Systems

- **Integrate multiple file systems into a single structure**
  - Abstract out that part of the file system that is common to all file sytems
  - A separate layer for underlying concrete file systems
- **VFS**
  - Upper interface : POSIX interface (open,read,write,lseek etc)
  - Lower interface : VFS interface to concrete file systems
  - Includes the superblock, V-node, directory
  - Includes some internal data structures – mount table, array of file descriptors and so on

Virtual File Systems (1)

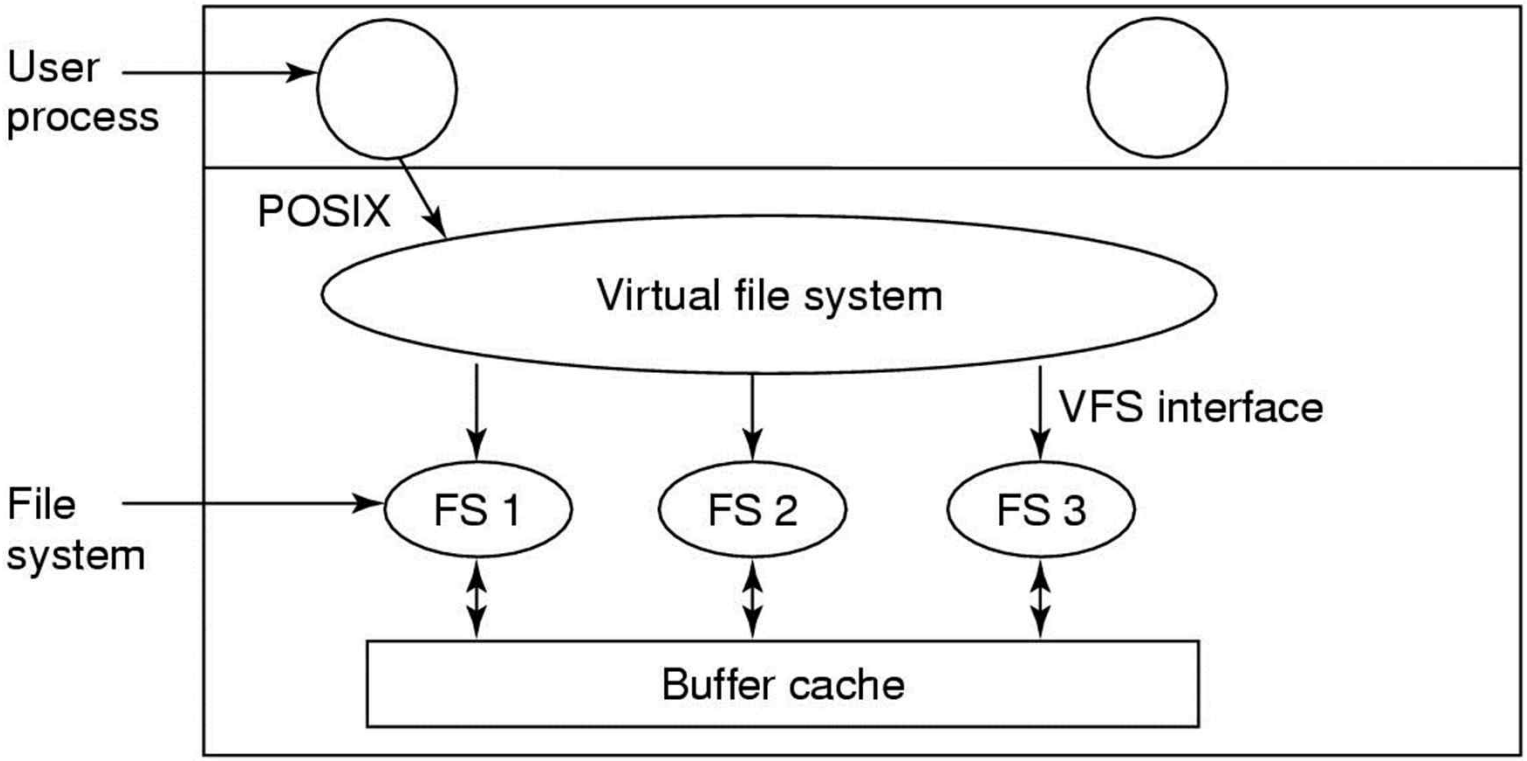


Figure 4-18. Position of the virtual file system.

Virtual File Systems (2)

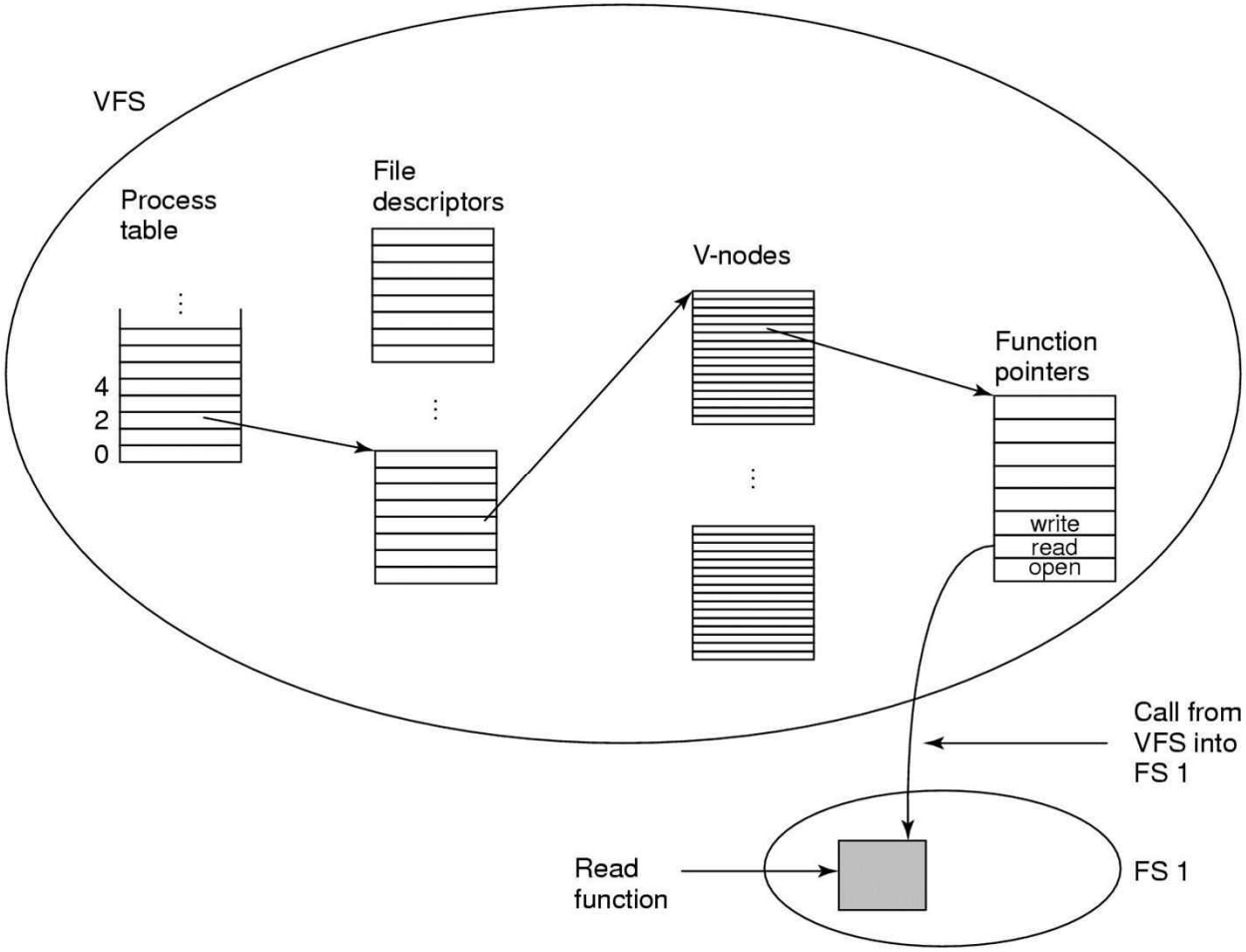


Figure 4-19. A simplified view of the data structures and code used by the VFS and concrete file system to do a read.

4.4.1 Disk Space Management

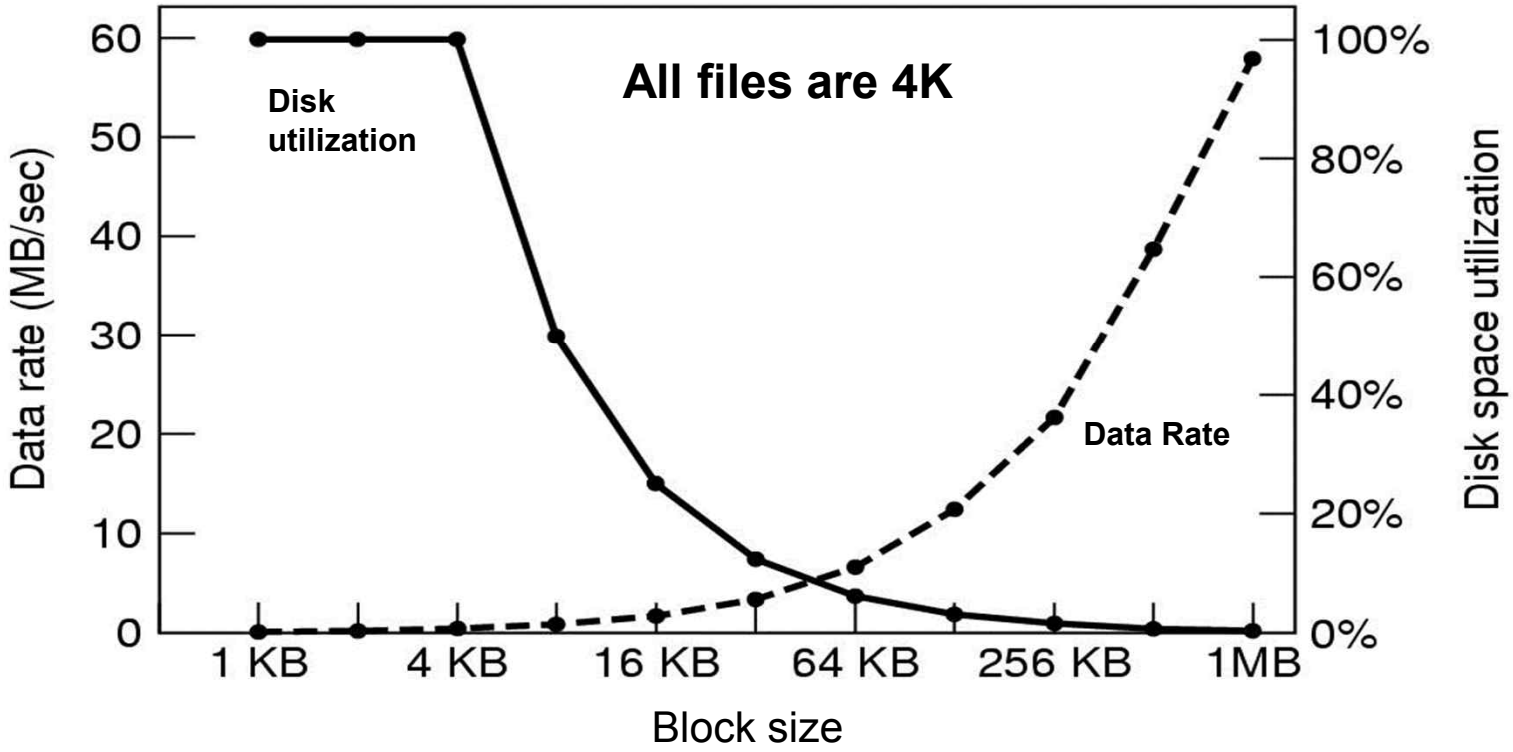
Block Size

Length	VU 1984	VU 2005	Web
1	1.79	1.38	6.67
2	1.88	1.53	7.67
4	2.01	1.65	8.33
8	2.31	1.80	11.30
16	3.32	2.15	11.46
32	5.13	3.15	12.33
64	8.71	4.98	26.10
128	14.73	8.03	28.49
256	23.09	13.29	32.10
512	34.44	20.62	39.94
1 KB	48.05	30.91	47.82
2 KB	60.87	46.09	59.44
4 KB	75.31	59.13	70.64
8 KB	84.97	69.96	79.69

Length	VU 1984	VU 2005	Web
16 KB	92.53	78.92	86.79
32 KB	97.21	85.87	91.65
64 KB	99.18	90.84	94.80
128 KB	99.84	93.73	96.93
256 KB	99.96	96.12	98.48
512 KB	100.00	97.73	98.99
1 MB	100.00	98.87	99.62
2 MB	100.00	99.44	99.80
4 MB	100.00	99.71	99.87
8 MB	100.00	99.86	99.94
16 MB	100.00	99.94	99.97
32 MB	100.00	99.97	99.99
64 MB	100.00	99.99	99.99
128 MB	100.00	99.99	100.00

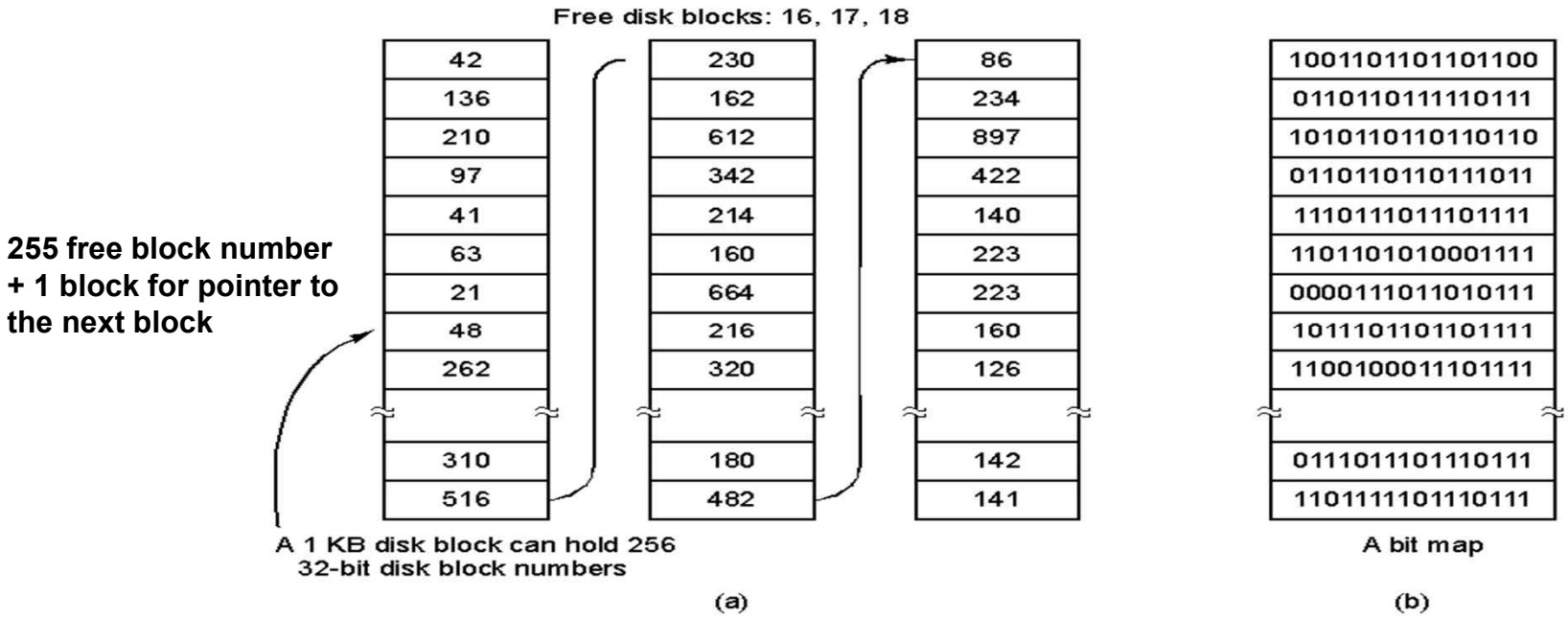
Figure 4-20. Percentage of files smaller than a given size (in bytes).

4.4.1 Disk Space Management



- **Data normally stored in disk → importance of disk management**
- **What is the "right" size of a disk block?**
  - **Small blocks are bad for performance; seek and rotation for each block access**
  - **Larger block, more internal fragmentation wastes**
- **Median file size is roughly 2KB (VU 2005)**

Keeping Track of Free Blocks



- **Storing the free list on a linked list**
  - Keep only one block in memory
  - 500GB disk (488M blocks) requires 1.9M blocks
  - Free blocks are used, so the storage is essentially free
- **A bit map**
  - Keep allocated blocks together
  - 500GB disk (488M bits) requires about 60,000 1KB blocks

4.4.2File System Backups

- **Importance of file system reliability**
- **Files contain information pertaining to personal life**

- **Why backups?**
  - **Recovery from disasters**
  - **Recovery from stupidity**
- **What to backup**
  - **Not the whole file system**
- **How to backup**
  - **Incremental dumps**
  - **Compressed format**
  - **Offline or on-line**
    - » Offline backup is not always acceptable
  - **Realistic issues (administration Issues)**
    - » e.g. where to physically place the backup tape – security, safety

Backups: Physical dumps

- **Start at block 0 on disk and write all blocks onto tape**

- **Simple and fast**
- **Some problems with physical dump**
  - **dump unused blocks?**
  - **dump bad blocks?**
    - » If bad blocks are visible to OS, it can cause endless disk read errors
  - **Inability to be selective, make incremental dumps, restore individual files**

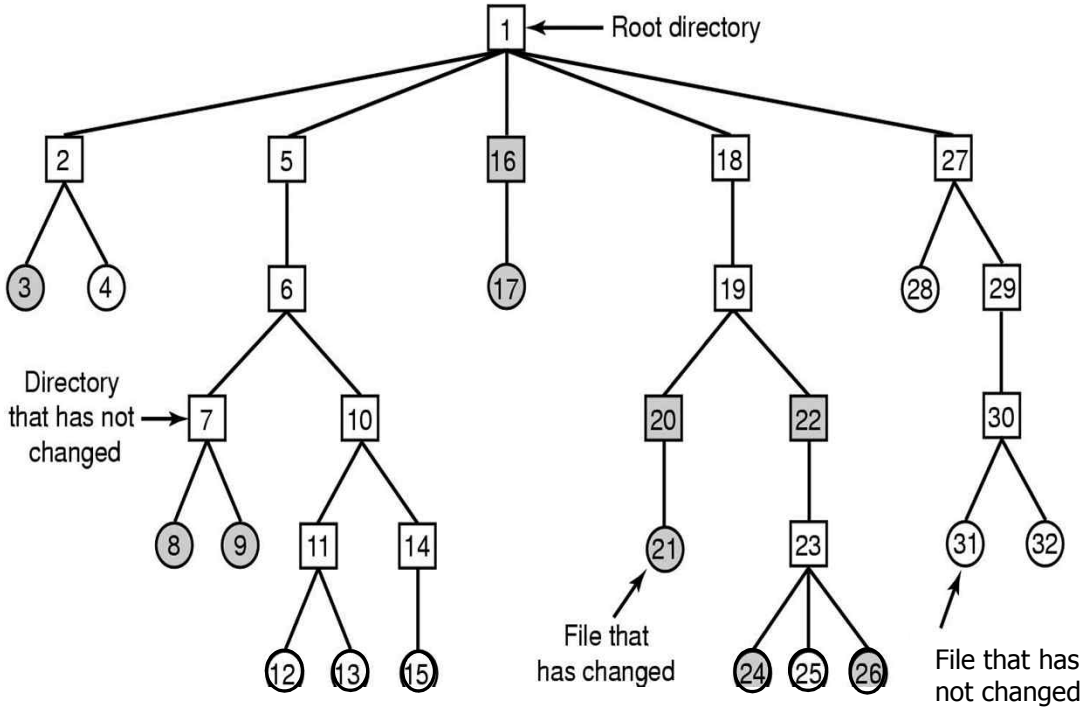
Backups: Logical dumps

- **Start at specified directory and dump recursively**
- **Dumps/restores specified files and directories changed**
  - **Full dump and incremental dump**
- **Dumps all directories (*even unmodified one*) that lie on the path to a modified file or directory**
  - **Can transport entire file systems between computers**
  - **Make it possible to incrementally restore a single file**
    - » Eg. /usr/jhs/proj/nr3 is removed and but we want to restore /usr/jhs/proj/nr3/plans/summary

    => The directory nr3 and plans must be restored first

    => need to incrementally dump all directories in the path to file "summary"

Backups: Logical dumps



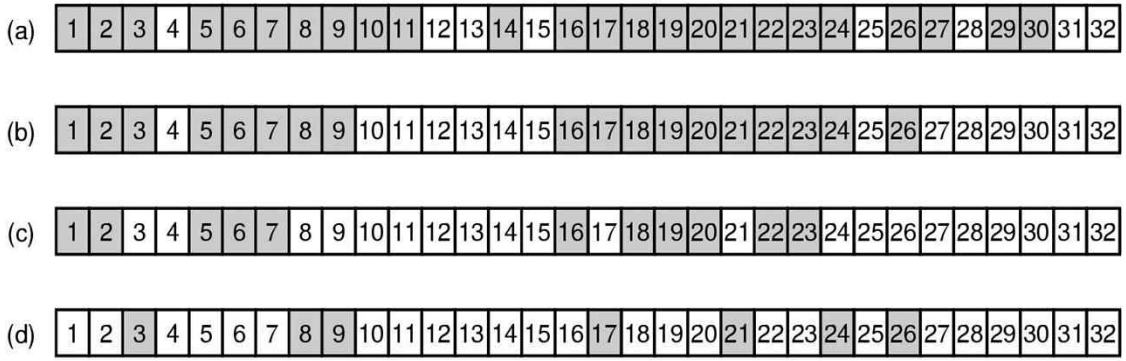
(a) All modified files and all directories are marked (I-node #)

(b) Unmark any directories that have no modified files or directories (10,11,14,27,29,30)

(c) dump all the directories that are marked for dumping

(d) Files marked in (d) are also dumped

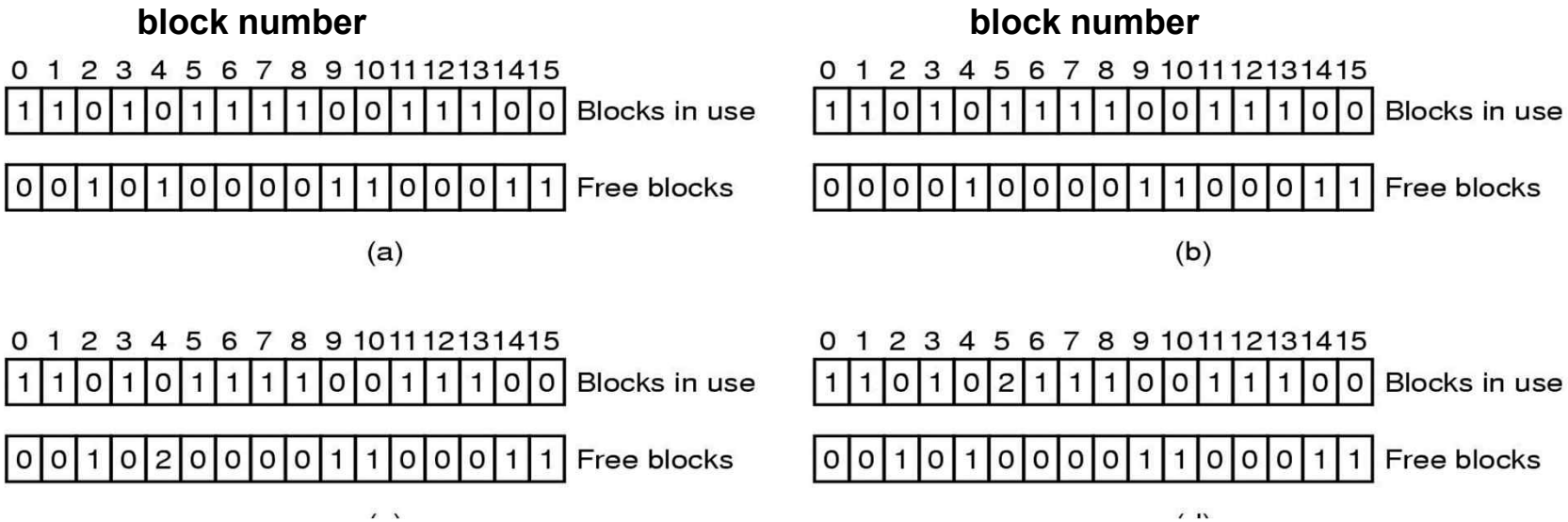
Bit maps used by the logical dumping algorithm



File System Consistency

- **File system reads, modifies, writes**
- **Inconsistency may arise due to crash**
- **Consistency check is necessary**
  - **Two kinds of consistency cheks**
    - » Blocks and files consistency checks
  - **fsck in UNIX, scandisk in Windows**

Block Consistency Check



- **Use two counters for each block**
  - **Count how many times each block is present in a file**
    - » Examine I-nodes
  - **Count how many times each block is present in the free list**
    - » Examine the free list or bitmap
- **File system states**

  (a) consistent

  (b) missing block (block 2) - add to the free list

  (c) duplicate block in free list (block 4) - rebuild the free list

  (d) duplicate data block (block 5) - copy the block

Directory (file) Consistency Check

- **Use a counter per file**
  - Increment the counter for a file if the file is found in a directory while recursively inspecting the file tree
- **Compare the usage counter with the link counter of the file**

  (a) link counter > usage counter - fix the link counter

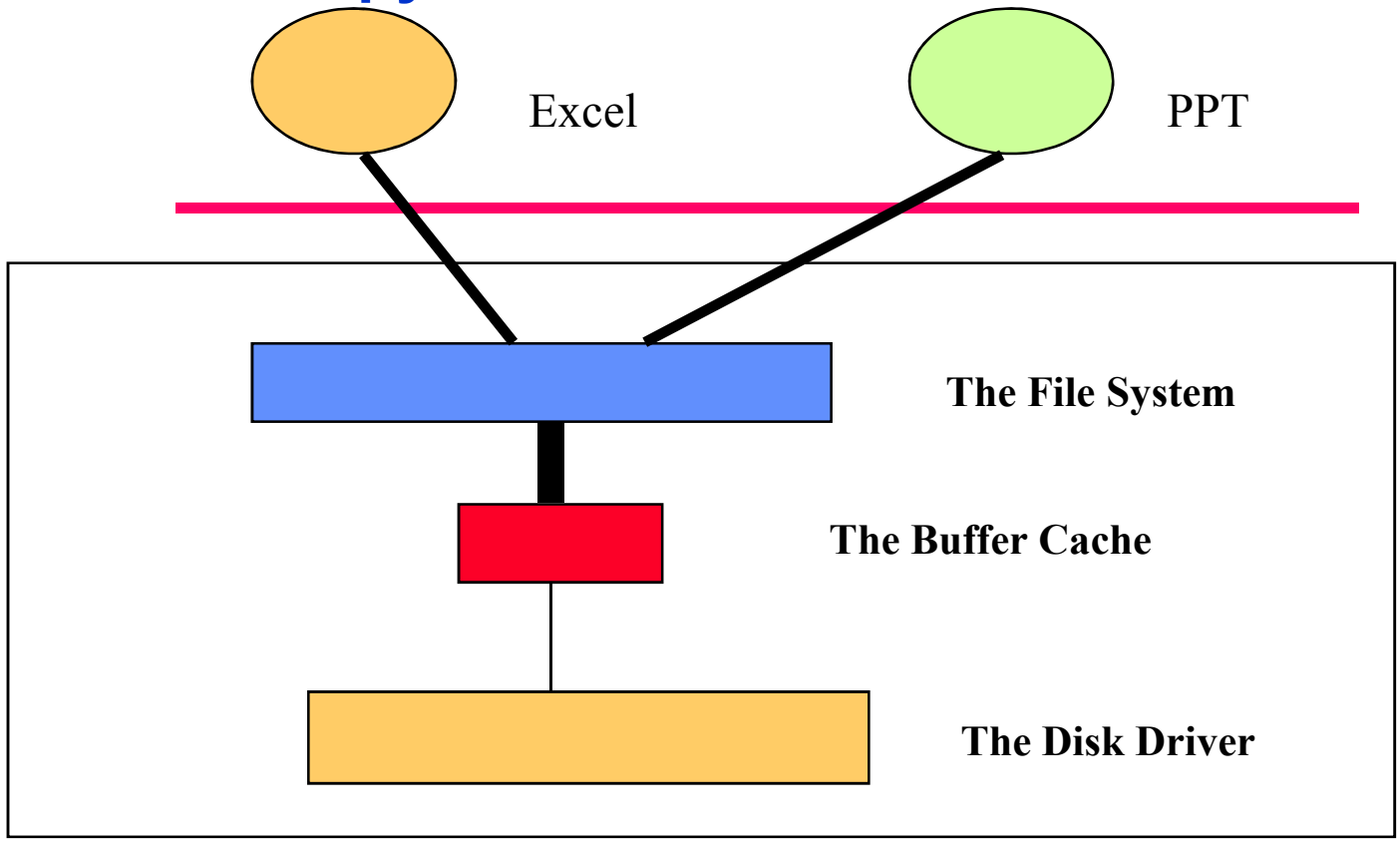
  (b) link counter < usage counter - set the link counter to usage counter value

4.4.4 File System Performance

- Caches
- Block Read Ahead
- Reducing Disk Arm Motion (FFS)

Block/Buffer/File Caching

- The idea is that data accessed recently is likely to be needed again (locality)
- Buffer in main memory for disk sectors
- Contains a copy of some of the sectors on the disk

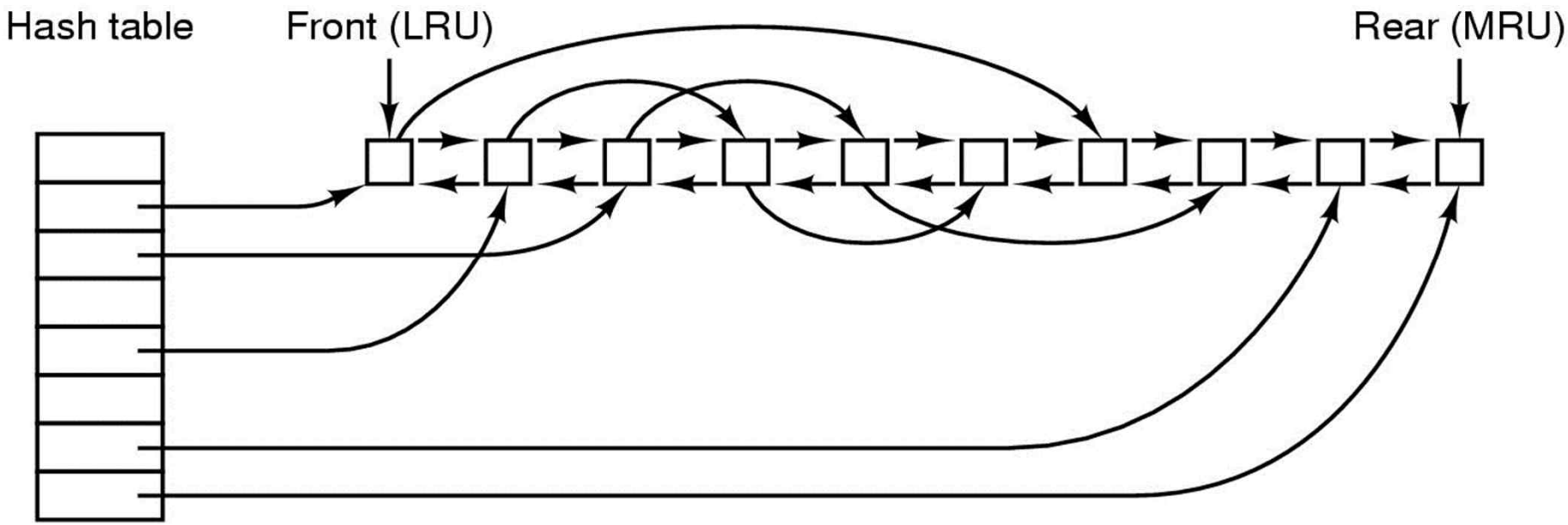


Buffer Cache

- **The buffer cache has only a finite number of blocks in it.**
- **On a cache miss, one block must be discarded in favor of the fresh block that is being brought in.**
- **Typical systems operate in an LRU order**
  - **"Full" LRU may be used**
    - » Why is the full implementation of LRU possible? Remember LRU paging problems?
  - **Policy issue → replacement algorithm**
- **Some workloads LRU is really bad for**
  - **sequential scans**
    - » video, audio
  - **random access**
    - » large database
- **Some blocks LRU is bad for**
  - **I-node : rarely referenced two times within a short interval**
  - **Blocks essential to file system consistency (everything except for data block)**
    - » Need to be written into disk as soon as possible (file system metadata)
  - **Even keeping data blocks long in the cache is bad**
    - » Unix : sync or update daemon; MS-DOS : write-through cache

Least Recently Used

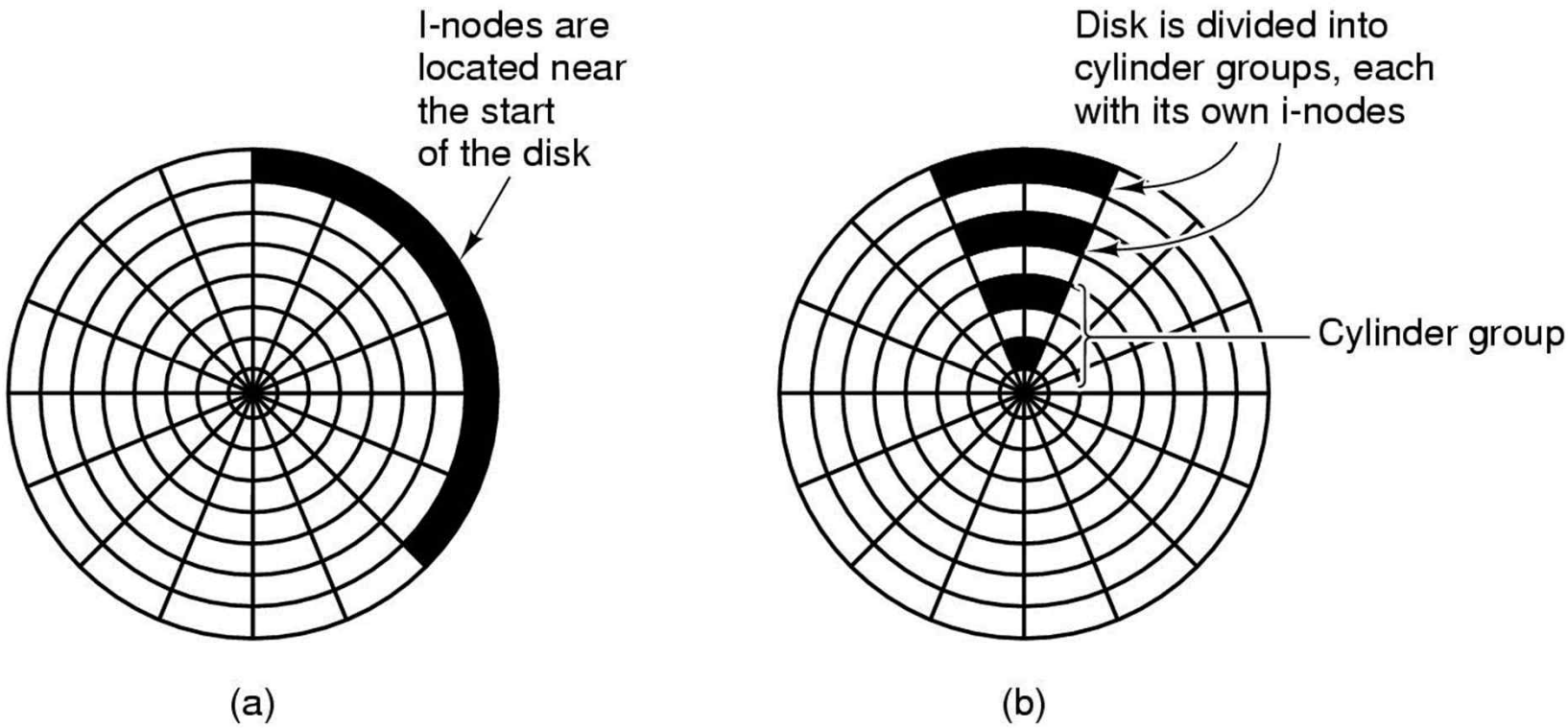
- Buffer caches are hashed by device number and disk address
- The block that has been in the cache the longest with no reference to it is replaced
- The cache consists of a stack of blocks
- Blocks don't actually move around in main memory
- A stack of pointers is used



Caching works for reads, what about writes?

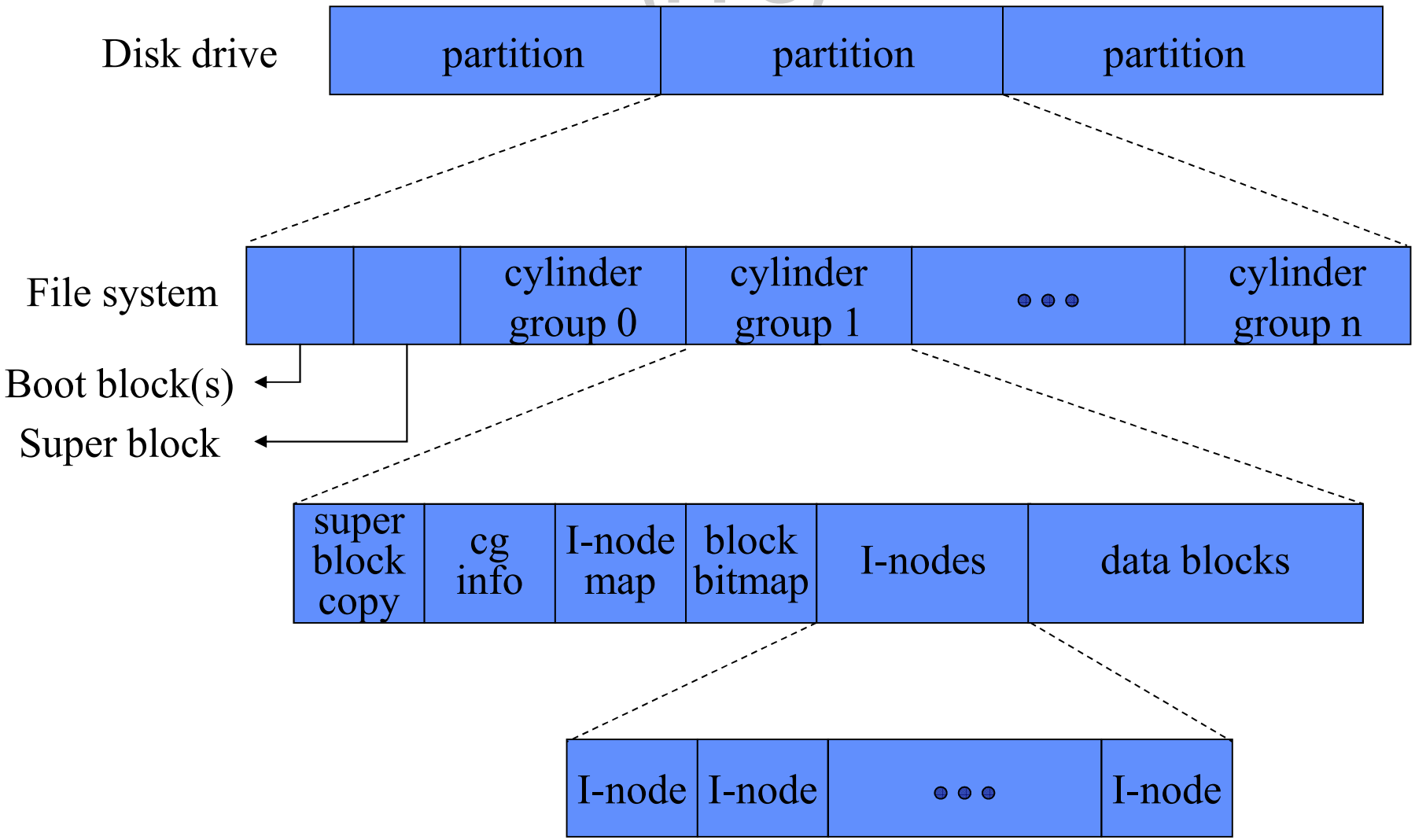
- **Ensure data makes it through the buffer cache and onto the disk**
- **Consequently, writes, even with caching, can be slow**
  - **Write-through**
- **Systems do several things to compensate for this**
  - **Write-behind (write-back)**
    - » maintain a queue of uncommitted blocks
    - » periodically flush the queue to disk
    - » unreliable
  - **battery backed up RAM**
    - » as with write-behind, but maintain the queue in battery backed up memory
    - » expensive
  - **log structured filed system**
    - » always write the next block on disk the one past where the last block was written
    - » Treat the disk like a tape
      - **Complicated to get right (always need a fresh 'tape')**

Reducing Disk Arm Motion



- **Optimize I-node placement**
  - **I-nodes placed at the start of the disk - (a)**
  - **Disk divided into cylinder groups (FFS) - (b)**
    - » each with its own blocks and i-nodes
- **Allocate blocks in groups of consecutive blocks**
  - **E.g. Allocate blocks in units of 4 blocks**

Disk layout of Fast File System (FFS)



4.4.5 Defragmentation Disks

- **As file system get aged, the blocks used for a file may be spread all over the disk**
  - **Bad file system performance**
  - **Some file systems need defragmentation**
    - » Linux file systems (ext2 and ext3) suffer less from defragmentation