# Sheets for MIEEC's SOPE

*based on teaching material supplied by A. Tanenbaum for book: Modern Operating Systems, ed...*

# Chap 6: File Systems

# Chapter 6

# File Systems

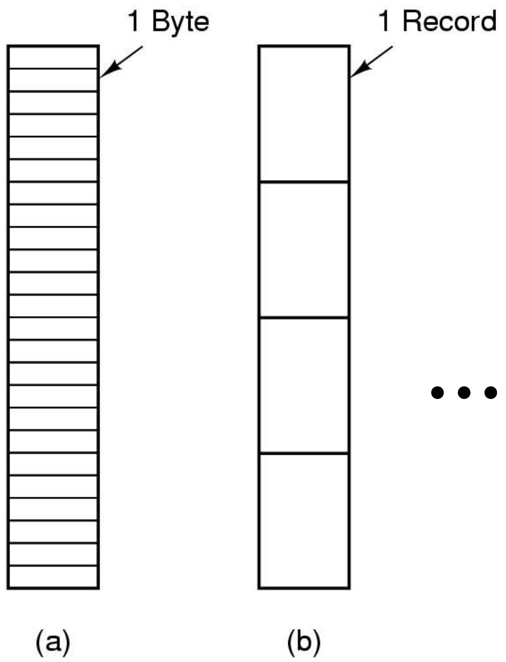
Files
Directories
File system implementation
Examples

# Information Storage: some goals

Storage system should be able to:

- allow easy access to data: (human) file naming
- keep data safe beyond life of process that created it
- store large amounts of data
- support concurrent access to data

# Files: Structure




Structure can be

- independent of data stored
    - current generalist systems (a)
        - Unix, MsWindows…
    - data should be interpreted by programs
        - but should be easily portable (if wanted!)
- reflect data stored
    - specialized systems (b)
        - e.g. some data base management systems
    - data is closely bound to those systems

# Files: Naming

Naming should provide:

- unequivocal (human) identification
  - organization, aggregation
  - class (type) identification (use of "extension")
- efficient access to data

# File Types

## File types

- defined by content
- interpreted by software
- help for humans: file extension

| 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 |

## Other classification of files:

- regular (text, binary): hold "real" data
- directory: manage "real" data
- Unix special's (character/block, pipes, links...): implementation issues?...

# File Access

## Sequential access

- read all bytes/records from the beginning
- cannot jump around (sometimes, could rewind or back up)
- essential when medium was inherently serial (e.g. magnetic tape)

## Random access

- read bytes/records in any order
  - "file marker" can be placed anywhere
- essential for database systems
- normal automatism:
  - on reading, file marker is updated (moved forward)

# File Attributes

## Besides Identification* (i.e. name), a file has attributes (metadata)

* won't identification be an attribute?...

Possible 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 |
| 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 |

# File Operations

Create
Delete
Open
Close
Read
Write

Append
Seek
Get attributes
Set Attributes
Rename
...

(change of attribute?) ;-)

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

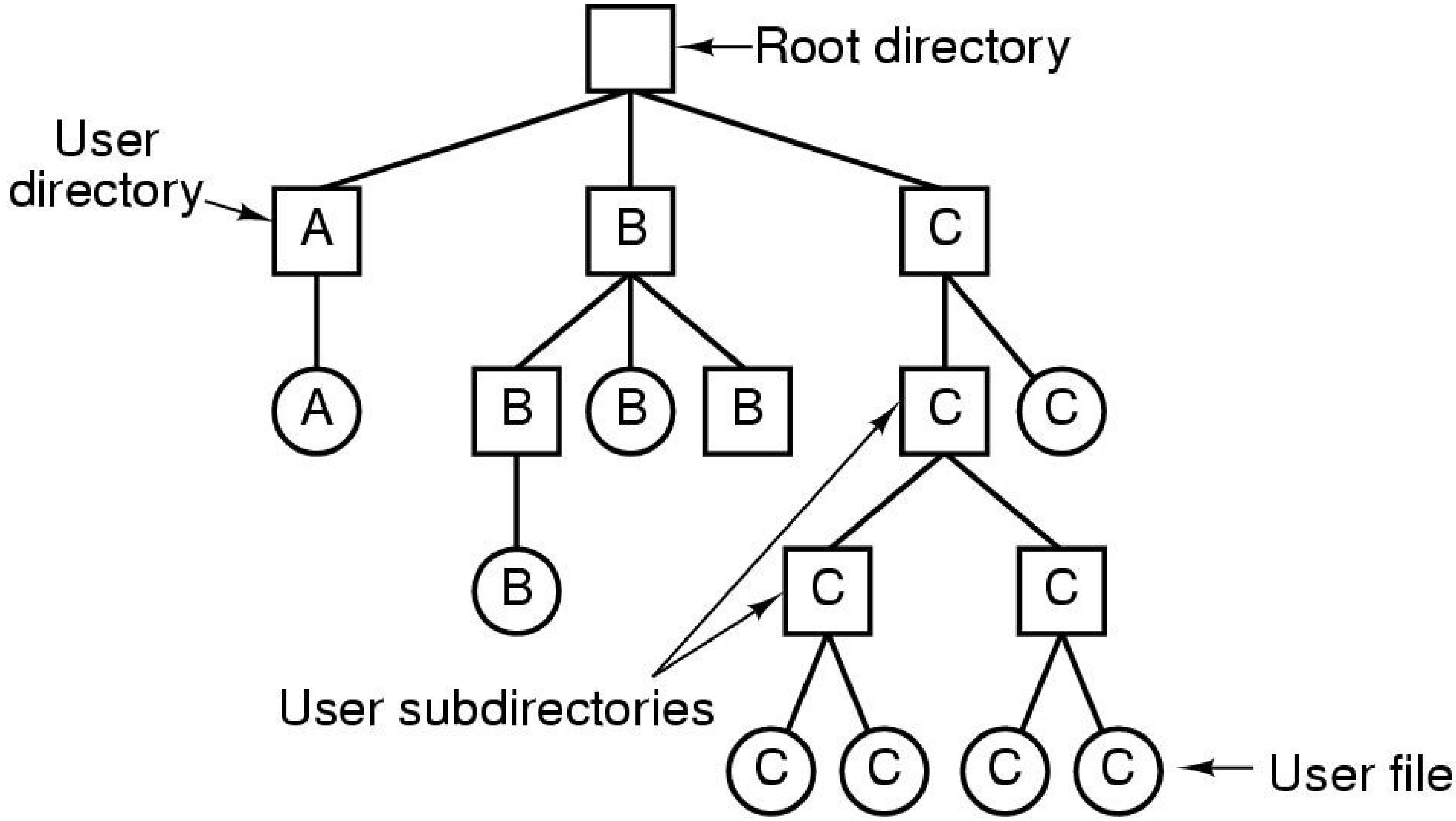
Usage: copyfile abc xyz

```
/* 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 */
}
```

# Directories: Hierarchical Systems

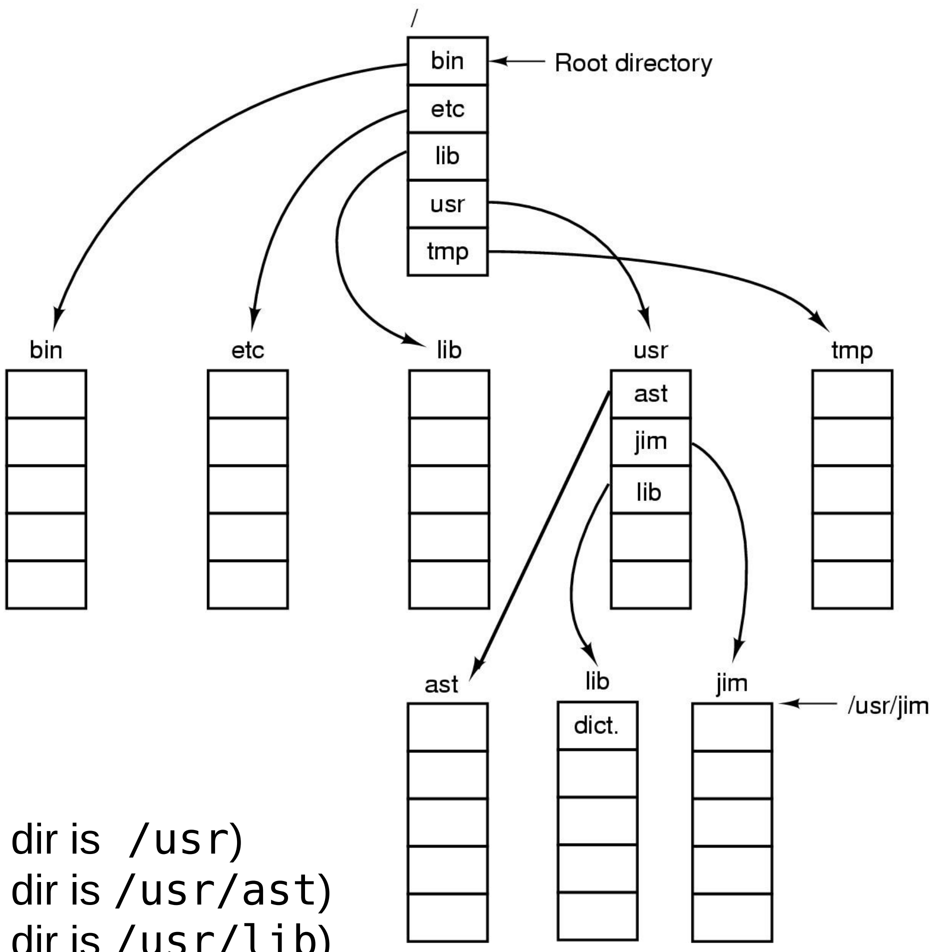



## A hierarchical directory system

(Letters indicate *owners* of the directories and files!)

# Path Names

## A UNIX directory tree:




Absolute path name:
    `/usr/lib/dict.`
Relative path name:
    `lib/dict.`        (current working dir is `/usr`)
    `../lib/dict.`     (current working dir is `/usr/ast`)
    `./dict.`          (current working dir is `/usr/lib`)
    `dict.`            (current working dir is `/usr/lib`)

# Directory Operations

Create

Delete

Opendir

Closedir

Readdir

Rename

Link

Unlink

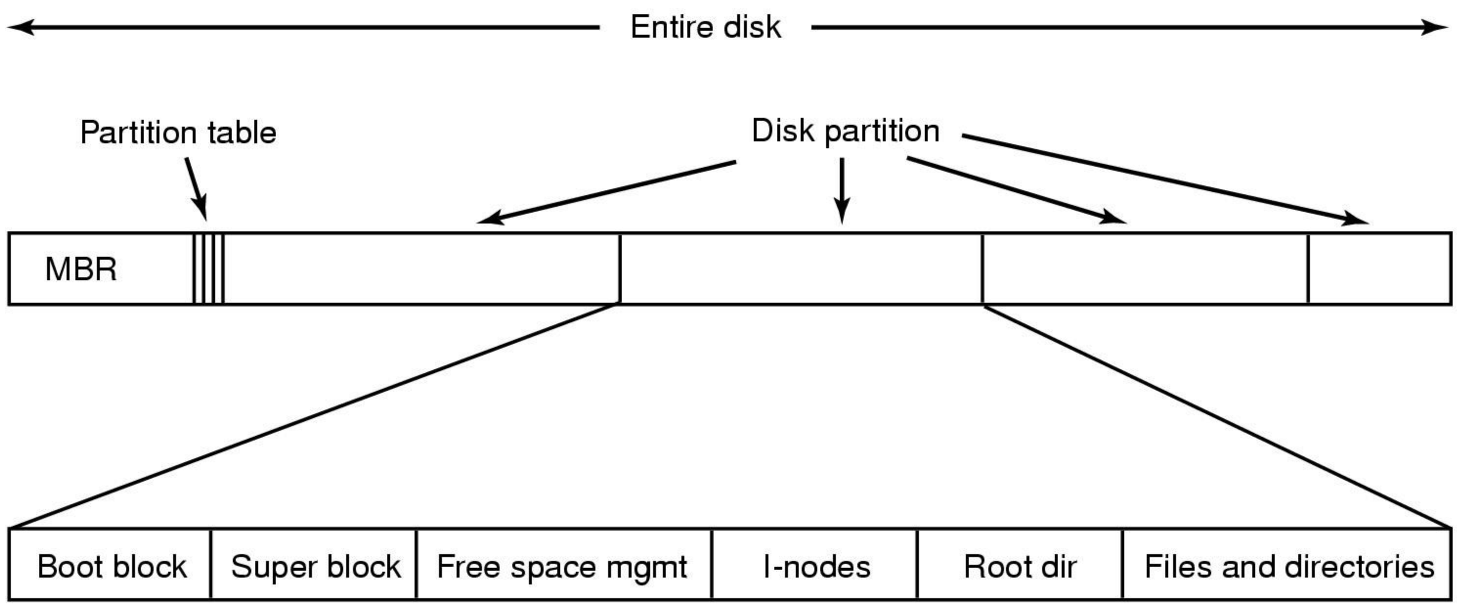
...

# Directory Programming

## Main code line (example):

```
//  Listing directory's regular file names

    DIR *od; struct dirent * ds; struct stat st;
    char filename[1024] = {'\0'};

if ((od = opendir(dirname)) == NULL) {
    perror("opendir");  exit(1);
}

while((ds = readdir(od)) != NULL) {

    if (!strcmp(ds->d_name, ".") || !strcmp(ds->d_name, ".."))
         continue;

    sprintf(filename, "%s/%s", dirname, ds->d_name);

    if (stat(filename, &st) < 0) {
         perror("stat");   continue;
    }

    if ((st.st_mode & S_IFMT) == S_IFREG)
         printf("%s\n",  ds->d_name);

} // while()
```

# File System Implementation

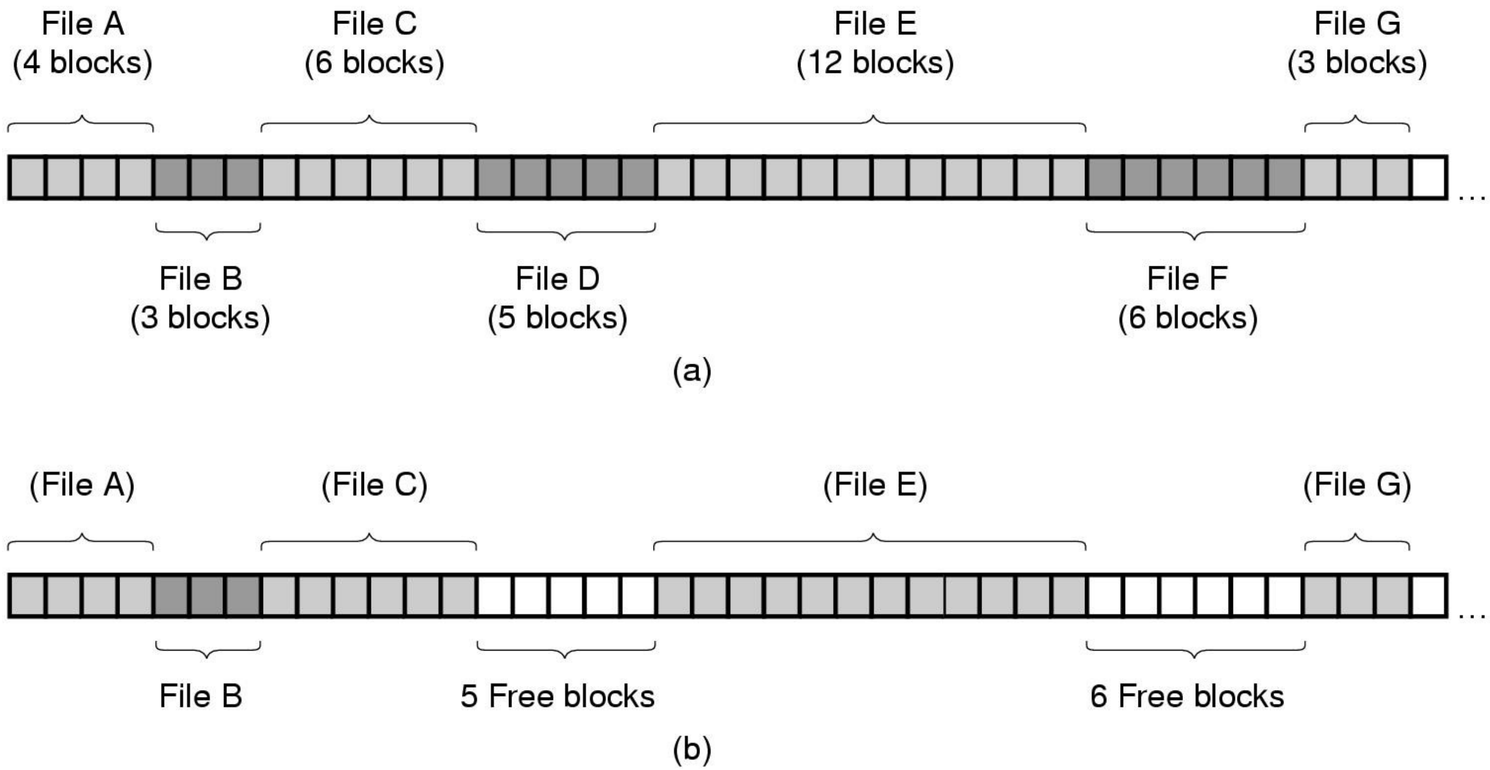



## A possible file system layout

Disk block:
- unit of storage
- typically 4kiB

Example of Super block info:
- file system type
- max number of blocks of partition
- other administrative data...

# Implementing Files (1)



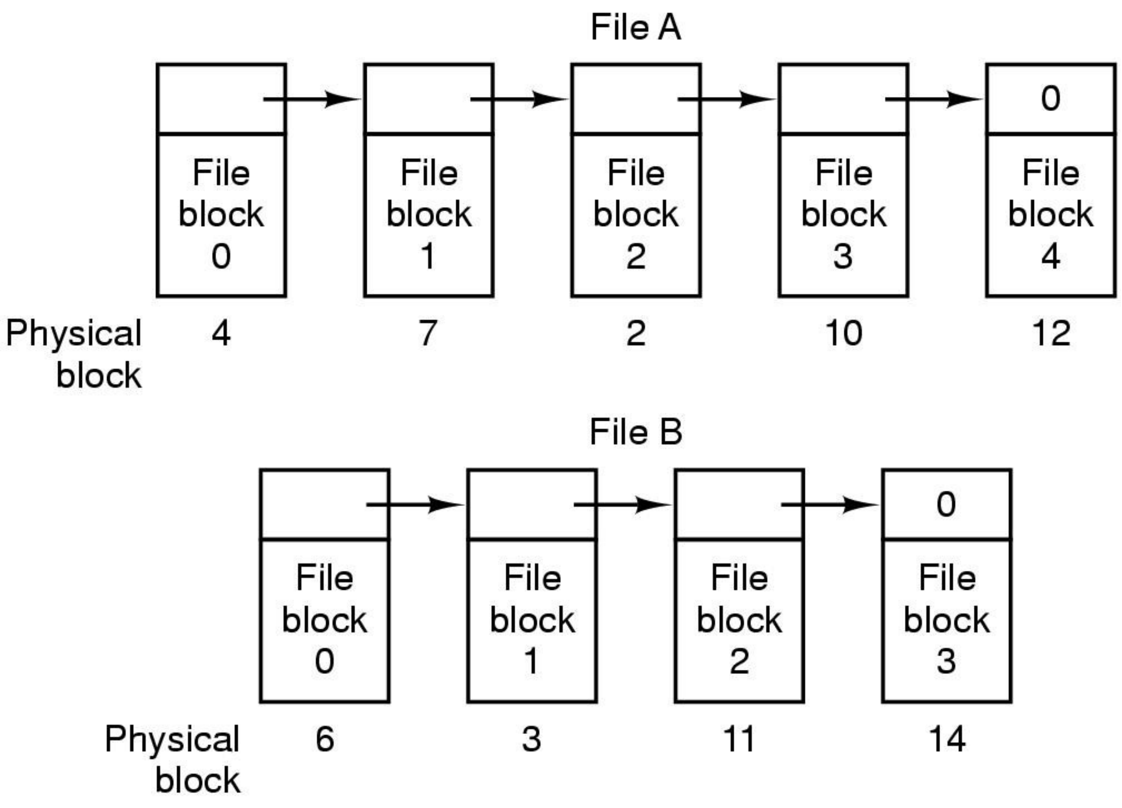

(a) Contiguous allocation of disk space for 7 files

(b) State of the disk after files *D* and *E* have been removed

Needed:

- Table of files showing starting block and number of blocks

# Implementing Files (2)



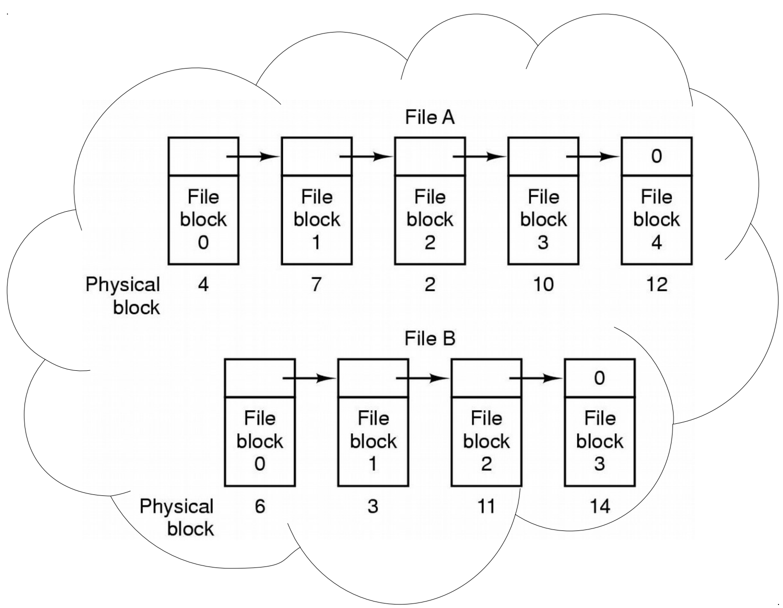

Storing a file as a linked list of disk blocks:

- no space is wasted
- random access is slow

# Implementing Files (3)

Linked list allocation using a <u>file allocation table</u> in RAM:

- quick, but
- table grows linearly with disk...



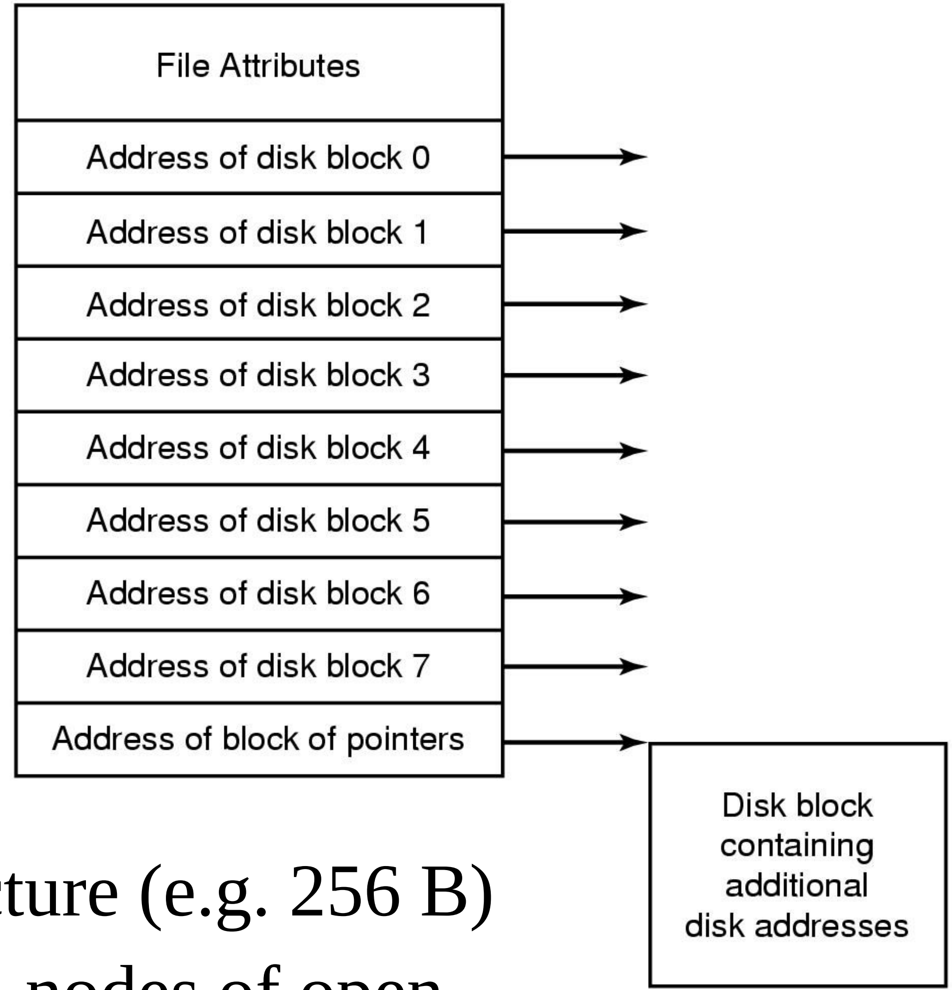

File Allocation Table

| | *nxt blck* | |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | *10* | |
| 3 | *11* | |
| 4 | *7* | File A starts here |
| 5 | | |
| 6 | *3* | File B starts here |
| 7 | *2* | |
| 8 | | |
| 9 | | unused block |
| 10 | *12* | |
| 11 | *14* | |
| 12 | *-1* | File A's last block |
| 13 | | |
| 14 | *-1* | |
| 15 | | |

index is physical block #

# Implementing Files (4)

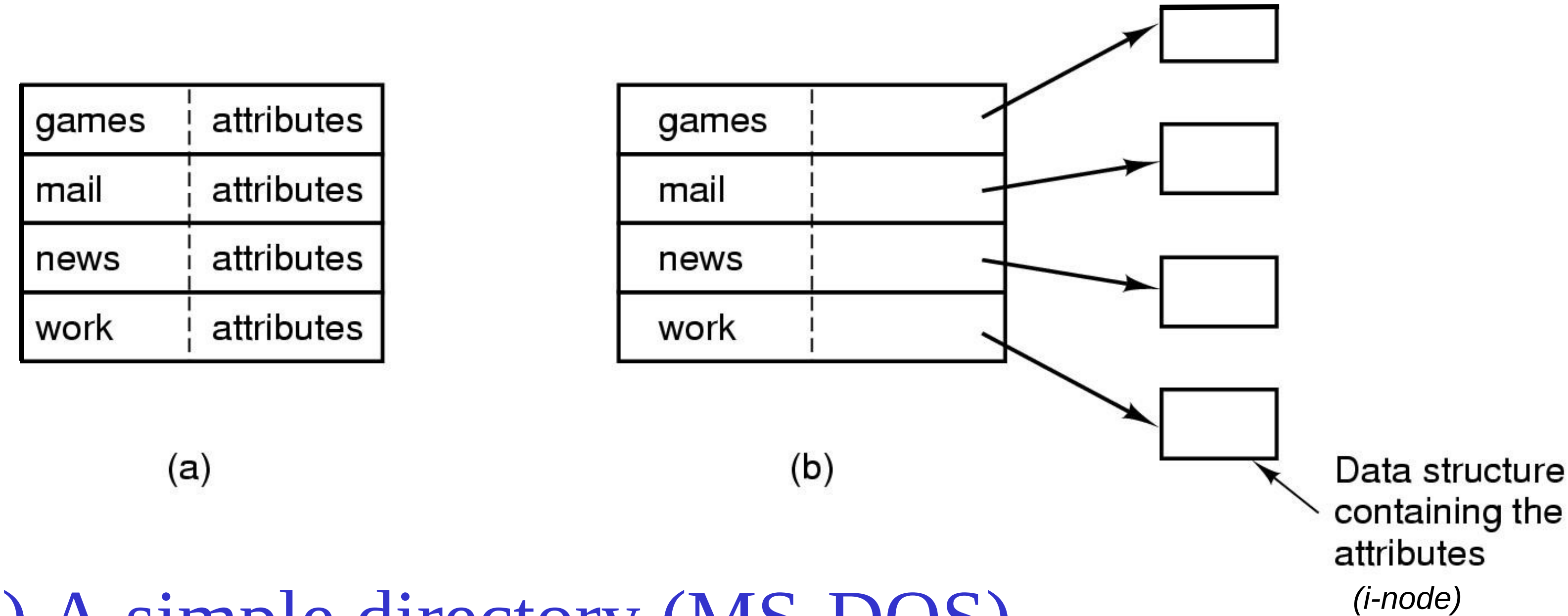
i-node:




## Using i-nodes

- additional data structure (e.g. 256 B)
- but with plus: only i-nodes of open files are needed in memory!

# Implementing Directories



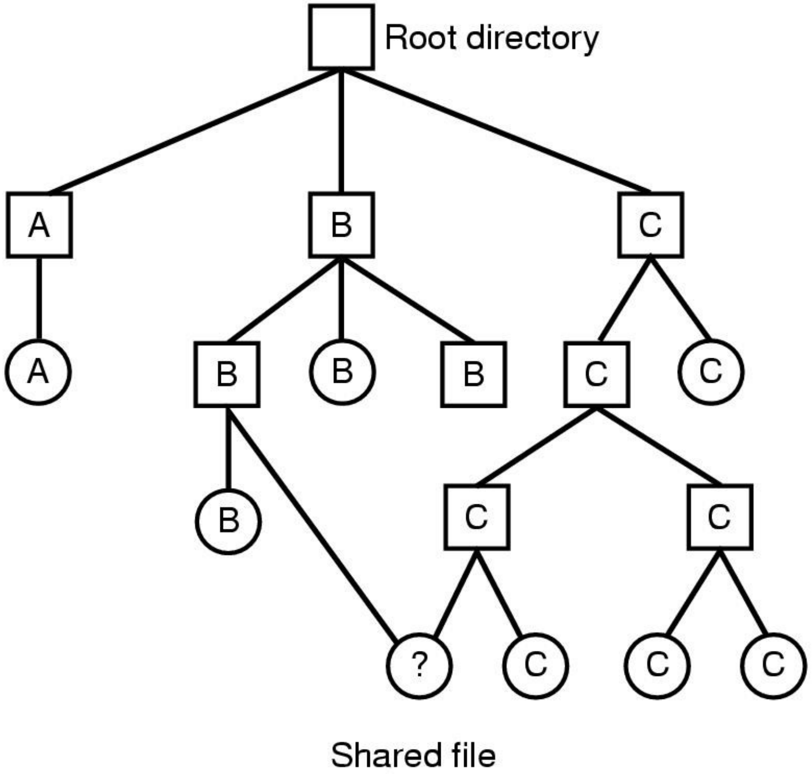

(a) A simple directory (MS-DOS)

- fixed size entries
- 1st disk address and attributes in directory entry

(b) Directory in which each entry just refers to an i-node (Unix)

Attributes: file size, time of creation and others, owner, protection...

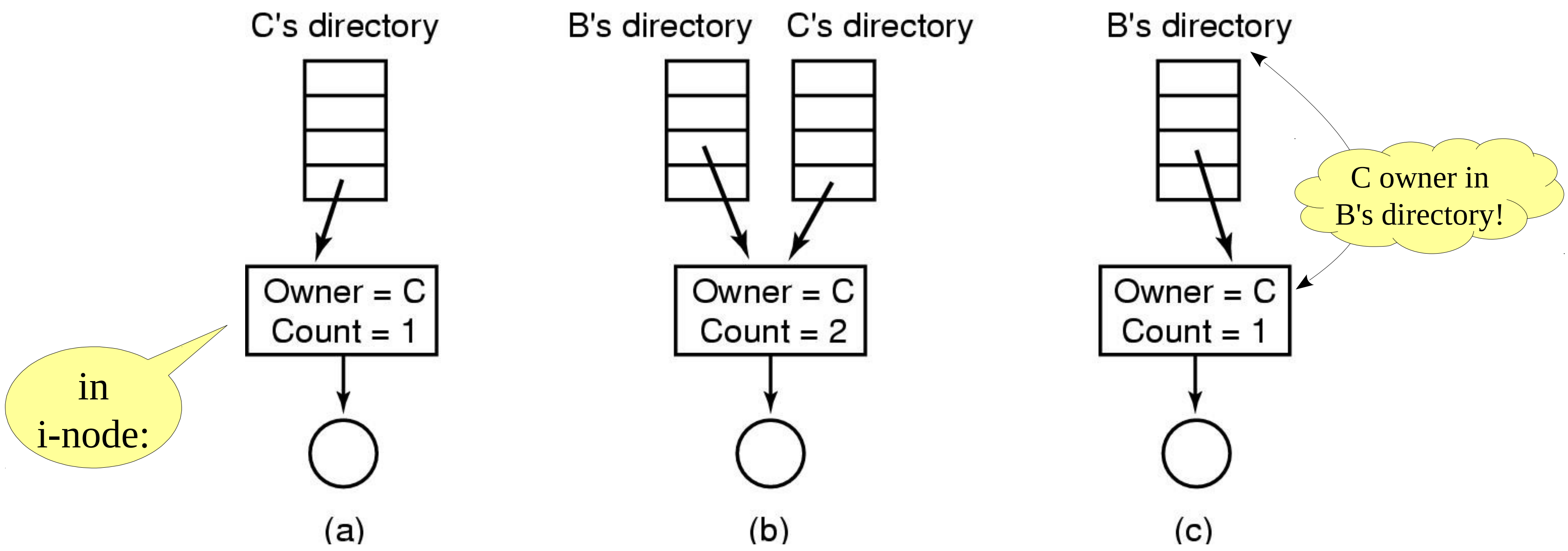
# Shared Files (1)




File system containing a shared file

- normal link
- symbolic link

# Shared Files (2)



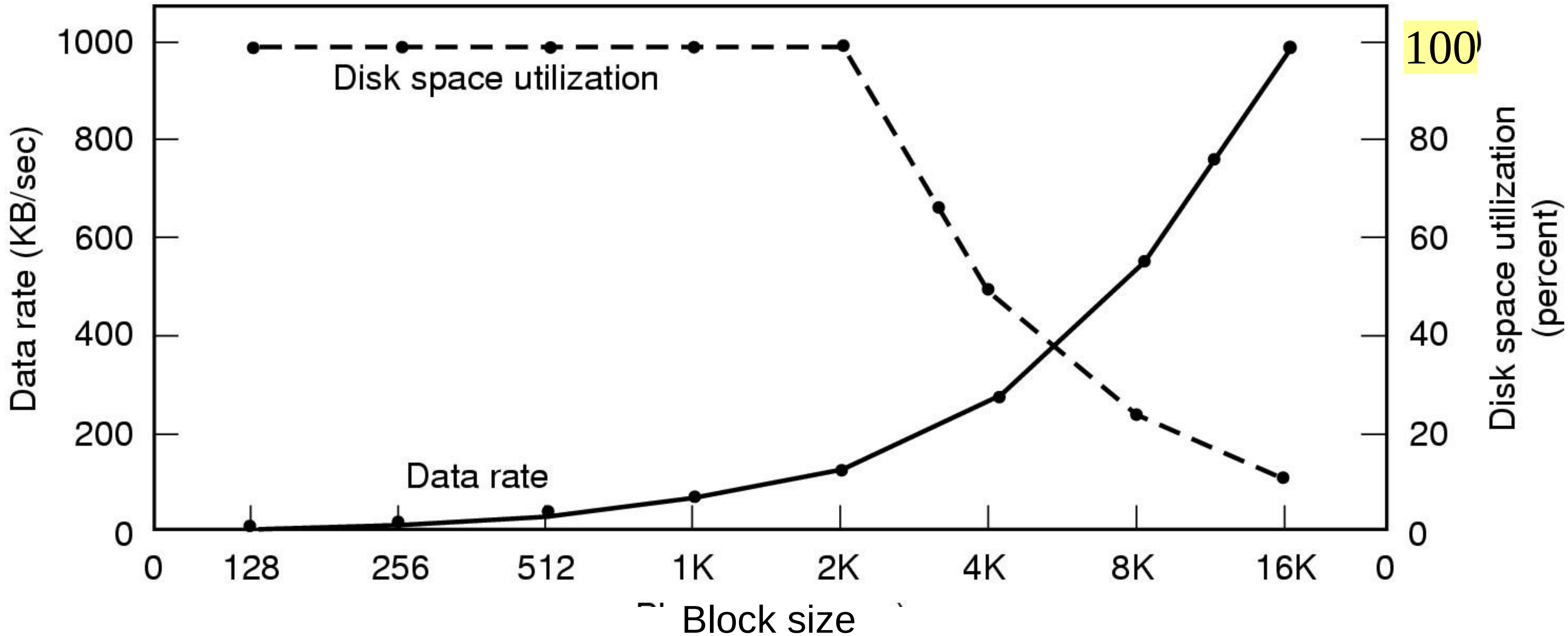

(a) Situation prior to (normal) linking

(b) After the (normal) link is created

(c) After the original owner removes the file

(Link) Count: one of the file's attributes...

# Disk Space Management (1)



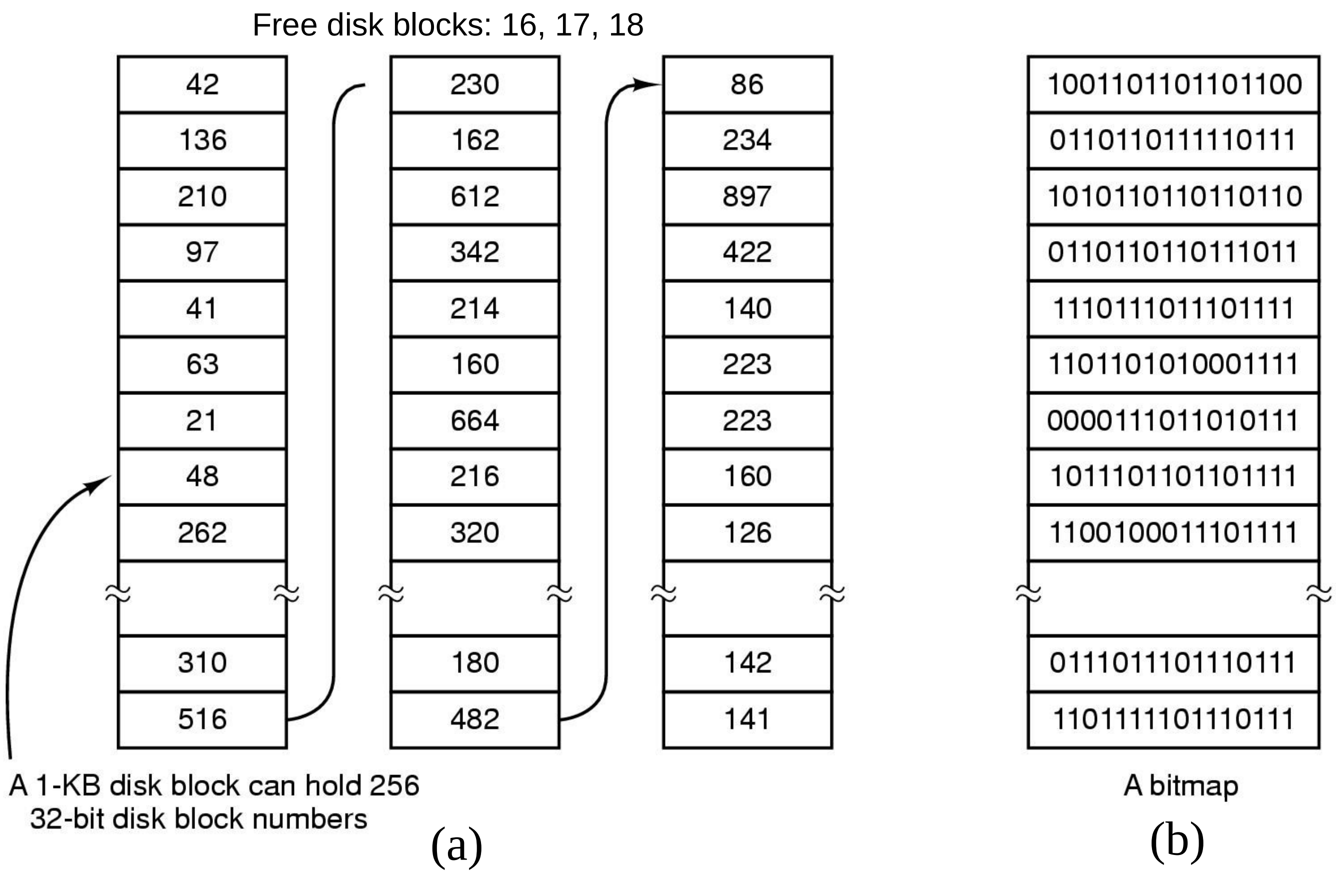

Dark line (left hand scale) gives data rate of disk access

Dotted line (right hand scale) gives disk space efficiency

All files 2KB

- block size: 4kB is typical in Linux and MsWindows NT

# Disk Space Management (2): tracking the free blocks




(a) Storing the free list on a linked list (in free blocks!)

(b) A bit map

# Disk Space: typical file system characteristics

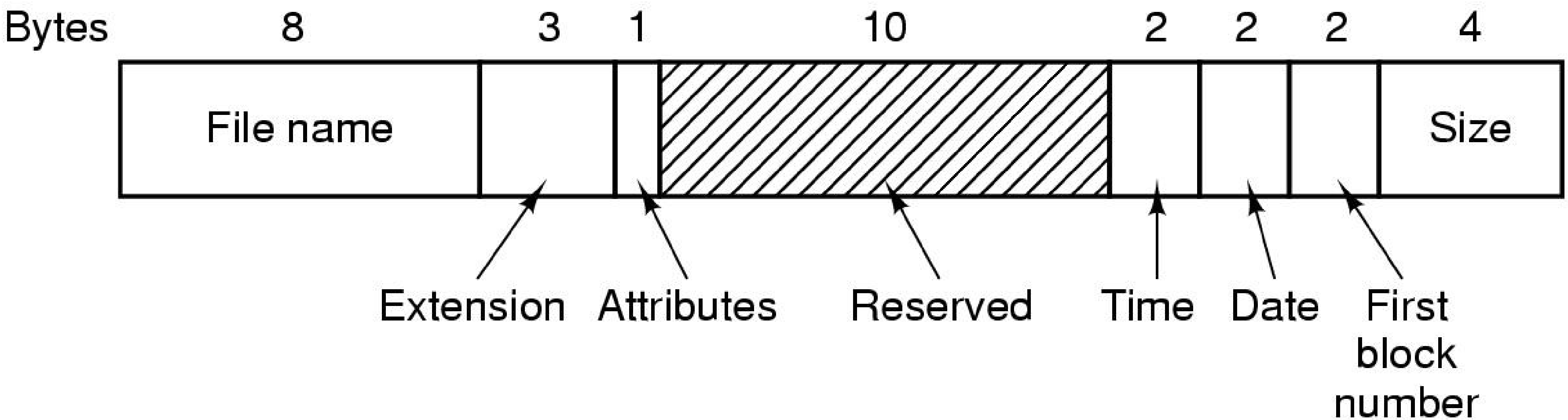
| | |
|---|---|
| **Most files are small** | Roughly 2K is the most common size |
| **Average file size is growing** | Almost 200K is the average |
| **Most bytes are stored in large files** | A few big files use most of the space |
| **File systems contains lots of files** | Almost 100K on average |
| **File systems are roughly half full** | Even as disks grow, file systems remain ~50% full |
| **Directories are typically small** | Many have few entries; most have 20 or fewer |

Figure 40.2: **File System Measurement Summary**

Agrawal et al., 2007

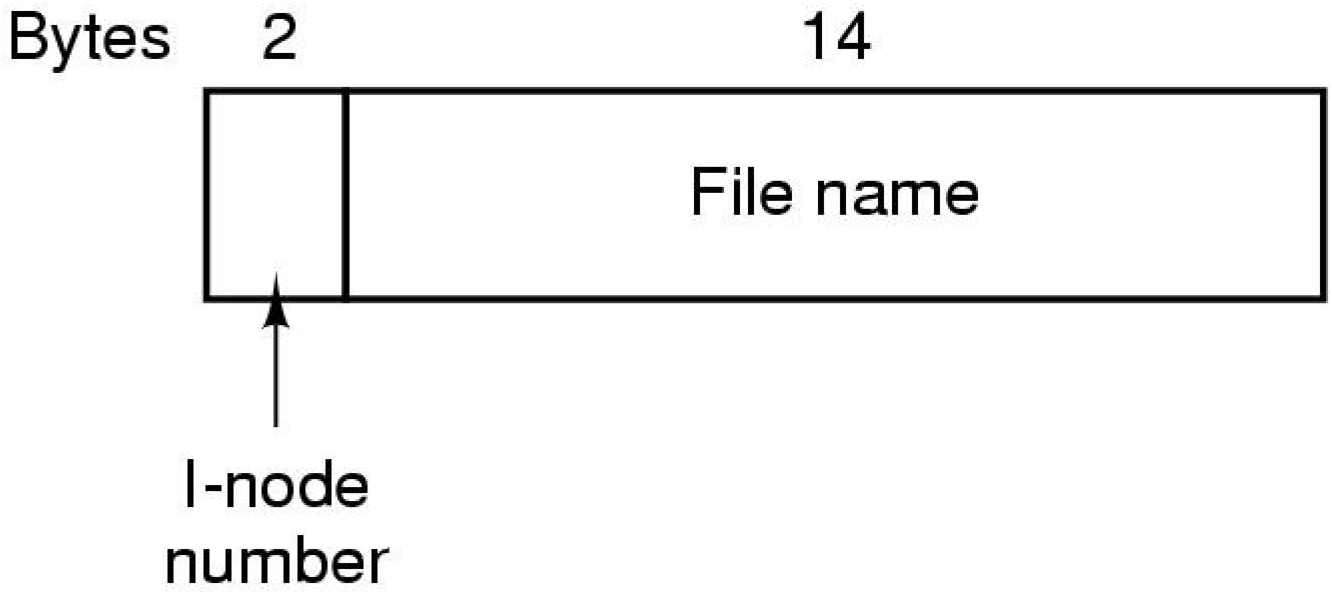
*in Arpaci-Dusseau's OSTEP*

# Some examples: MS-DOS File System



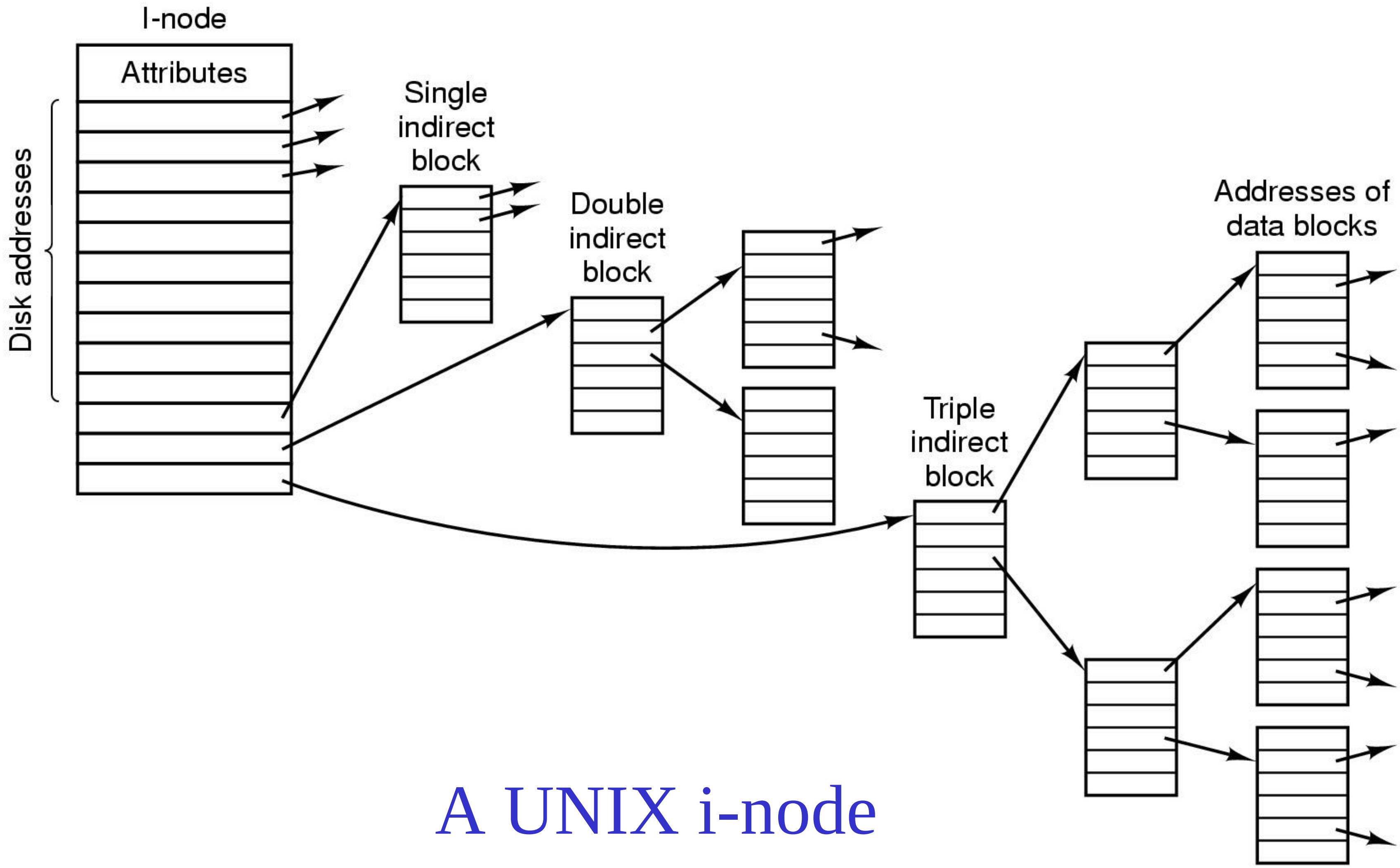

The MS-DOS directory entry

# Some examples: UNIX V7 File System (1)




A UNIX V7 directory entry

# Some examples:
# UNIX V7 File System (2)




A UNIX i-node

# Some examples:
# Linux ext2 i-node

| Size | Name | What is this inode field for? |
|---|---|---|
| 2 | mode | can this file be read/written/executed? |
| 2 | uid | who owns this file? |
| 4 | size | how many bytes are in this file? |
| 4 | time | what time was this file last accessed? |
| 4 | ctime | what time was this file created? |
| 4 | mtime | what time was this file last modified? |
| 4 | dtime | what time was this inode deleted? |
| 2 | gid | which group does this file belong to? |
| 2 | links_count | how many hard links are there to this file? |
| 4 | blocks | how many blocks have been allocated to this file? |
| 4 | flags | how should ext2 use this inode? |
| 4 | osd1 | an OS-dependent field |
| 60 | block | a set of disk pointers (15 total) |
| 4 | generation | file version (used by NFS) |
| 4 | file_acl | a new permissions model beyond mode bits |
| 4 | dir_acl | called access control lists |

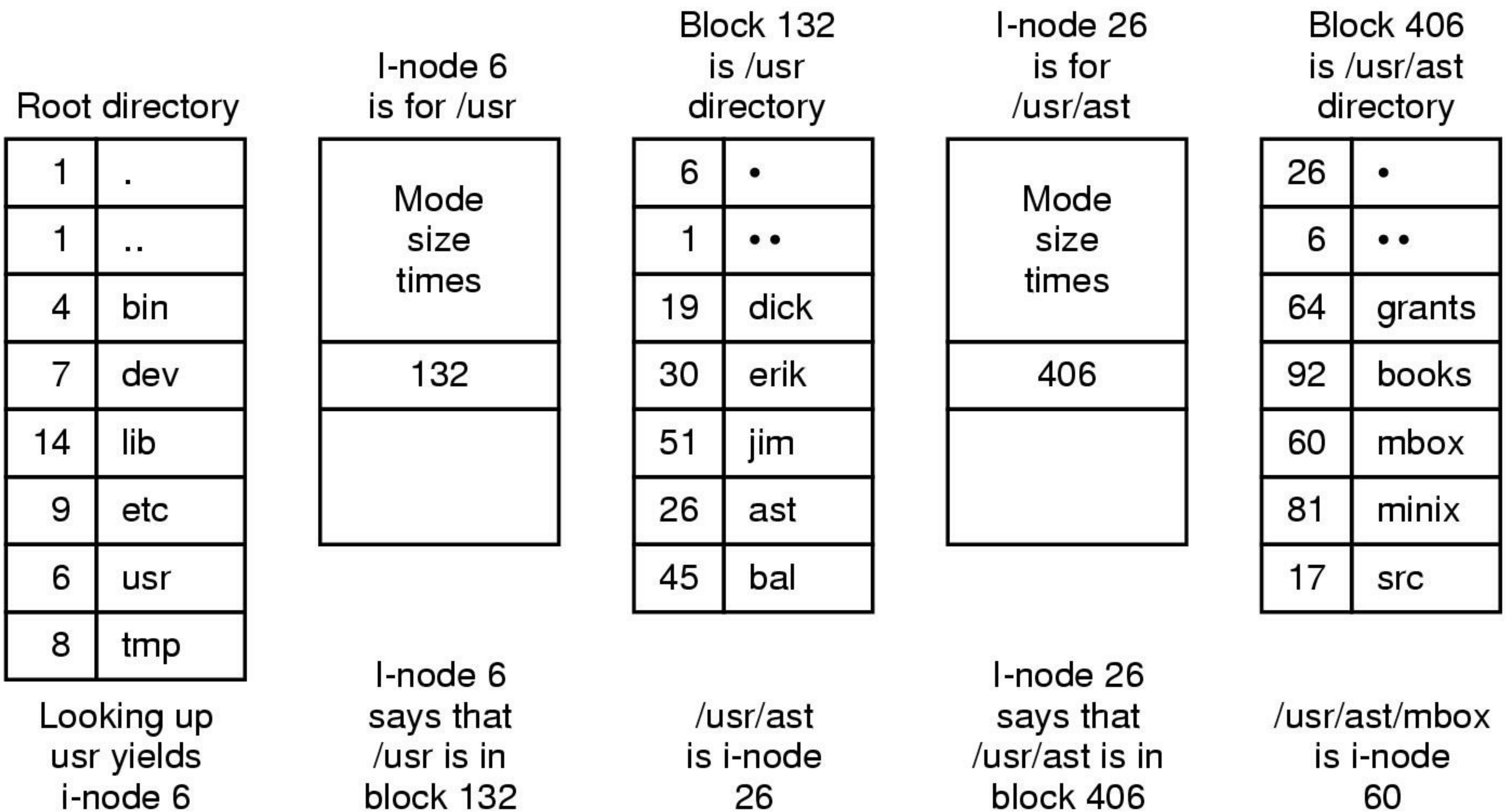
*not created: changed file's i-node status! (e.g. `uid` changed)*

Figure 40.1: **Simplified Ext2 Inode**

*in Arpaci-Dusseau's OSTEP*

## Attributes of a file (example of Linux ext2)

# Some examples: UNIX V7 File System (3)




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

# Log-Structured File Systems

With CPUs faster, memory larger:

- disk caches can also be larger, improving access speed
- increasing number of read requests can come from cache
- thus, most disk accesses will be writes
- BUT care must be exercised to minimize data loss on case of faults (e.g. power failure)
- So came log-structured file systems...