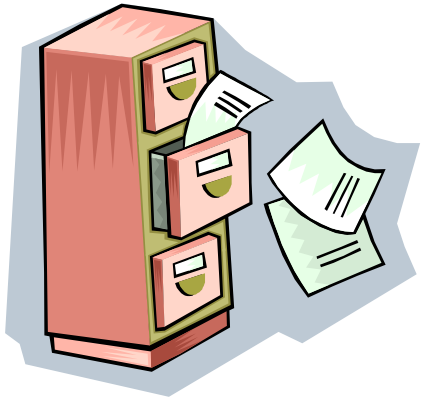
CS 520, *Operating Systems Concepts*

Lecture 9

File Systems



Issues

- Long-term storage of large amount of information
- Survival of the termination of any process using this information
- Concurrent access to the information by multiple processes
- Protection of the information from
  - Involuntary destruction
  - Unauthorized access

General Solutions

- The information is stored on external (auxiliary) media: typically, disks, and tapes.
- The *persistent* information object is called a *file*. (Only a file's owner can remove the file.)
- The issues of user interface (*naming*, *structure*, and *access methods* are separated from the implementation issues)

Naming Issues

- Restriction on the number of characters in a file name (none in *Unix*; 8+3, in *MS-DOS*)
- Restriction on the use of special characters (“.”, “?”, “*”). (After you have created a file called ‘*’, consider the effect of the `'rm *'` command in *Shell*.)
- The meaning of special characters inserted in a special place. (Can you see *“.visible”* with the *UNIX ls* command?)
- Distinction between the lower case and upper case letters (*Unix* distinguishes; *MS-DOS* does not.)
- Allocation of suffixes (or file *extensions*) for special purposes: `random.h`, `bus.c`, etc.)
- Acceptance and structure of the path name

(`cs-520/homework_5/buses.c` or `cs520\hw5\buses.c`)

Example: File Extensions

file.bak	Backup file
file.bas	BASIC source program
file.wav	Wave (sound) file
file.mid	MIDI file
file.man	Online manual page (*Unix*)
file.ftn	FORTRAN source program
file.tex	Input to TEX formatting program
file.doc	Input to Microsoft Word for Windows program
file.ppt	Input to Power Point program

File Structure

- A file can be a
  1. Just a sequence of bytes
  2. A sequence of structured fixed-length records (an 80-column punched card, a 132 line-printer line)
  3. A tree of records identifiable by a key

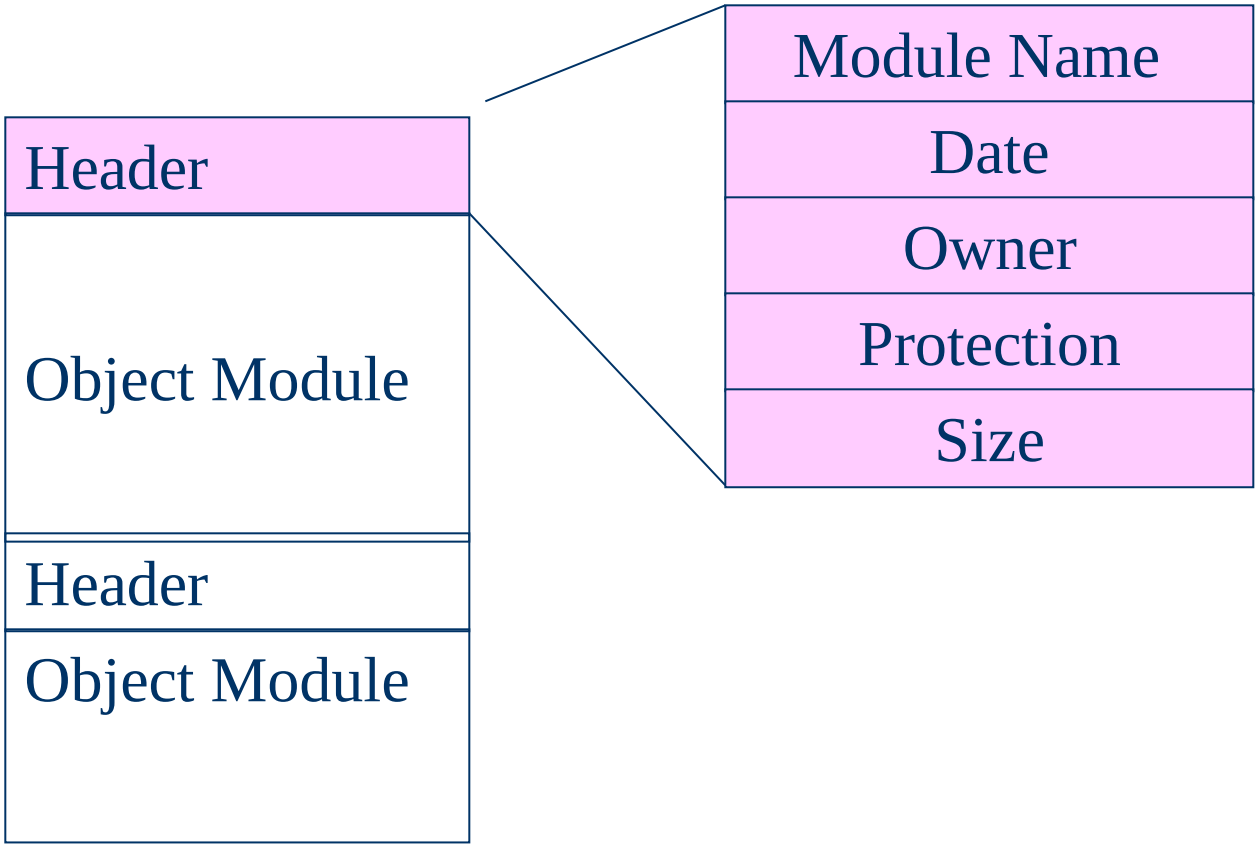
File Types

- Regular files (i.e., the ones we are going to deal with today)
- Directories (system files for organizing the file system)
- Character-special files (related to I/O and used to model terminals and printers)
- Block-special files (related to I/O and used to model disks)

Regular Files

- ASCII Files
  - "Lines" of text separated by a *line feed (LF)* character or a *carriage return (CR)* character or a combination of LF and CR. These are easy to display and print or edit with any text editor. (For this reason, only ASCII-formatted document files used to be accepted and published by the Internet Engineering Task Force [IETF] before the HTML became ubuquitous.)
- Binary Files
  - Non-ASCII files (even though they may be entirely dedicated to text—like *.doc* or *.rtf* files)

An Example: *UNIX* archive file



An Observation

- Mandatory name extensions and otherwise strong typing do help the OS and its applications to make many decisions automatically (consider the *make* program)
- Strong typing systematically enforced by the OS (like copying a *.xxx* file only to a *.xxx* file) proved to drive normal people crazy

File Access

- Early systems (and even text editors!) provided only sequential access—the file could be read only in one direction, from the beginning to the end
- Virtually all database applications need to have *random access* to files (which does require a different mechanism for storing them). This is achieved using *Read* operation with the exact record number, or a combination of *Seek* and *Read* operations
- Older operating systems required specification of *random* or *sequential* at the time of the creation of a file
- In modern operating systems, all files are random-access

File Attributes

- Except for surely providing the file name and its contents, there is hardly a prescription for absolutely necessary attributes
- It is not easy to list *all* possible attributes
- At least one of the following attributes or groups of attributes is present in one or another operating systems

Attributes (examples)

- Protection Attributes
  - Passwords
  - Flags (or bits sequences) displaying allowed access to groups for writing, reading, or executing, or even *displaying* (or any combination of the above)
  - File creator
  - File owner

Attributes (examples) (cont.)

- *Creation time* (including the date, of course)
- *Size*
- *Time of last access*
- *Time of last modification* (for example, for the use of *make*)
- *Maximum size allowed* (so that an operating system can allocate just as much space on the disk)
- *In-use flag*

Attributes (examples) (cont.)

- *Archive flag* (cleared by the back-up program, and set when the file is changed)
- *Lock flag*
- *Random access flag*
- *Temporary flag* (if set, the file must be deleted on the exit of a program)
- *Record* length
  - *Key offset* (within the record)
  - *Key length*

File Operations

- *Create* (at the least, the file attributes are set)
- *Delete*
- *Open* (to allow the system to get the attributes, and—when required—register the use of the file; some implementations [e.g., MUSS] use interesting semantics)
- *Close* (to register the end of use and free internal memory resources needed to deal with an open file)

File Operations (cont.)

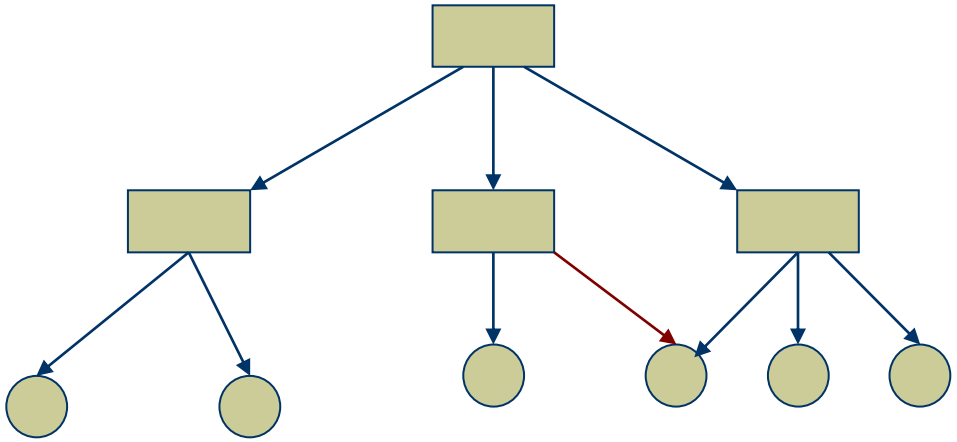
- *Read* (from a present [or specified] position a specified amount of data into a given buffer)
- *Write* (from a present [or specified] position a specified amount of data from a given buffer)
- *Append* (a specified amount of data from a given buffer)
- *Seek* (position the record pointer in a specified place for random access)

File Operations (cont.)

- *Get attributes*
- *Set attributes*
- *Copy* (to another file)
- *Rename* (a special case of *set attributes*)
- *Move* (may be the same as *rename*, or actually be the result of a sequence of *copy* and *move*)

Directories

- If there is one concept well understood, it is this one: directories are organized into trees (well, with the *link*, they become *Direct Acyclic Graphs [DAGs]* whose leaves are files.



Names

- Typically, directory names do not have extensions
- Each file (or directory) has an *absolute* path name (like, `/usr/igorf/cs-520/Lecture_8.ppt`)
- *Relative* path names, specify names relative to a *current* directory. (This could be quite involved, as in `../../../../cs-520/Lecture_9.ppt`)

Directory Operations

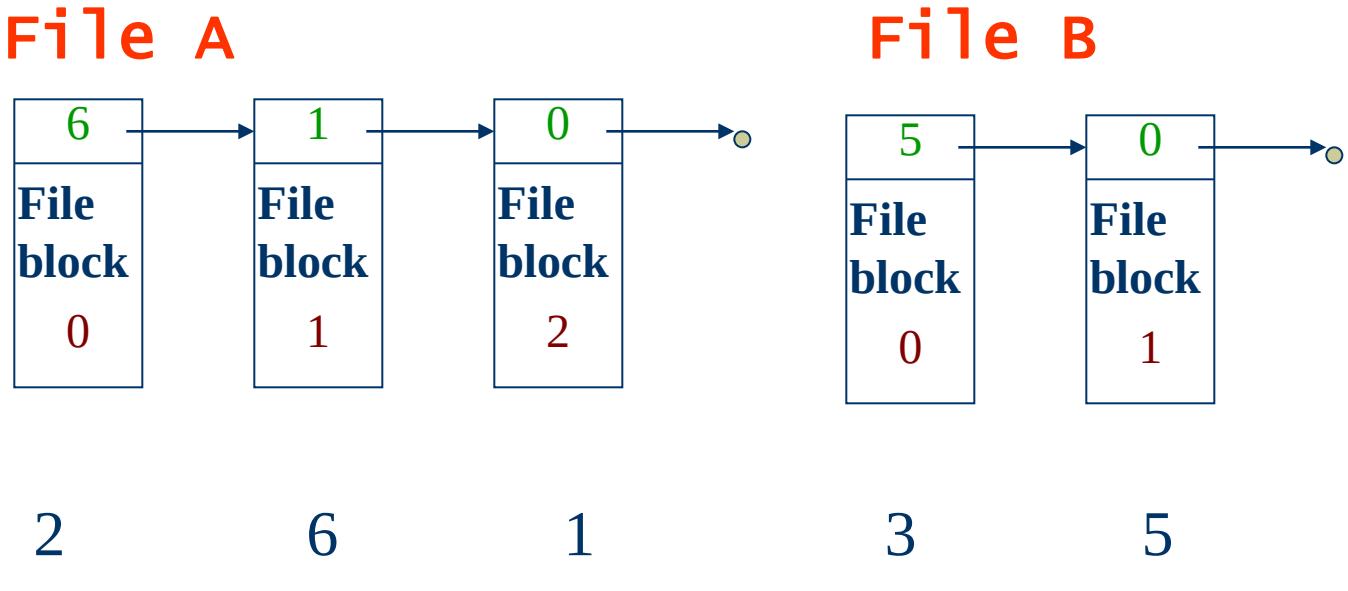
- *Create*
- *Delete*
- *Open*
- *Close*
- *Read*
- *Rename*
- *Link* (show a specified file in the specified directory)
- *Unlink* (a *delete* has the same effect in *UNIX*)

File System Implementations

The Key issue: Which disk blocks go with which file?

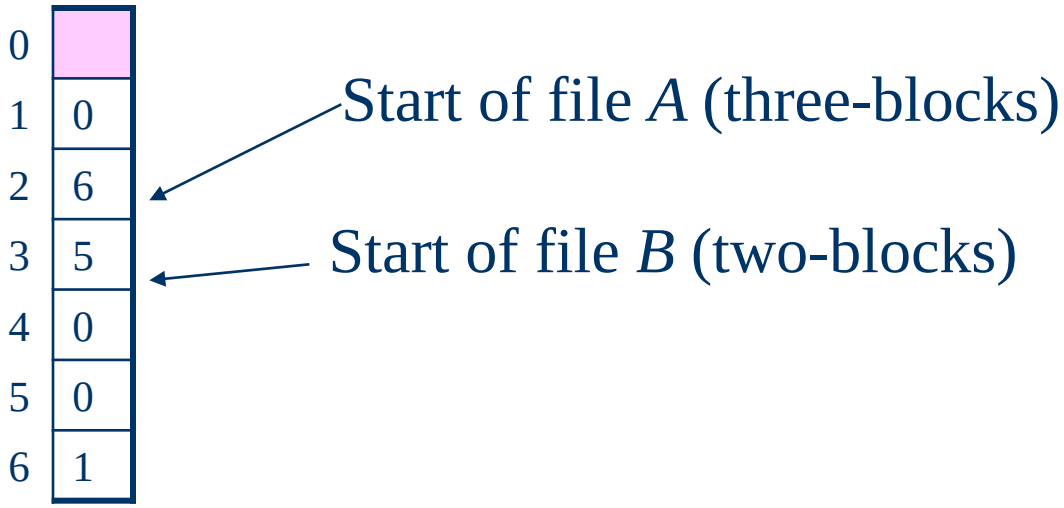
- Options
  1. Contiguous allocation
     - *Pros*: simple to implement, excellent performance
     - *Contras*: impossible to implement unless the file size is known at the creation, causes disk fragmentation
  2. Linked list allocation
     - *Pros*: no (external) fragmentation, easy to enter into a directory
     - *Contras*: random access is very slow, some block space is given to a pointer making the usable size no longer a power of two

Files as Linked Lists



File System Implementations (cont.)

3. Linked list allocation with an *index* table
   - *Pro*: eliminates all previous disadvantages
   - *Contra*: the entire index table must be in memory, which is expensive: A 500-MB disk would require a 1.5-2 megabytes of memory for the index tables

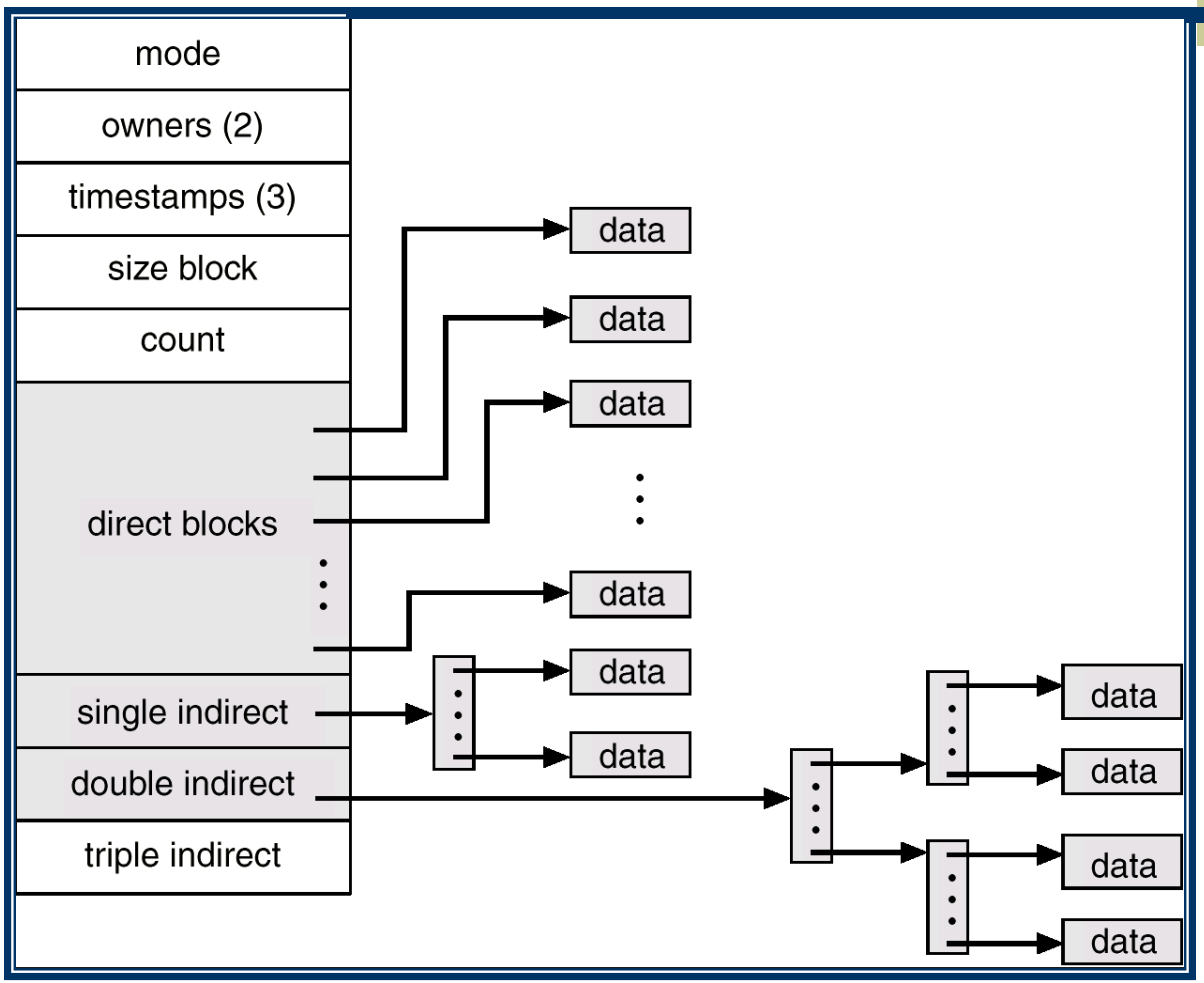


File System Implementations (cont.)

3. The best of all worlds—Index Nodes (*I-Nodes*)
   - With each file is associated a table called *i-node*
   - The table lists the file attributes and the addresses of a few blocks of the file (or the whole file, if it is appropriately small)
   - In larger files, one of the i-node addresses is that of a *single indirect block*, which contains additional addresses
   - In even larger files, there is a pointer to a *double indirect block*, which contains addresses of *single indirect blocks*
   - And then there could even be a *triple indirect block*, but that is it—no more indirection!

A file's i-node is read into memory when the file is opened!

I-Node (4K block) in *Unix (from the Book)*



Implementation of Directories

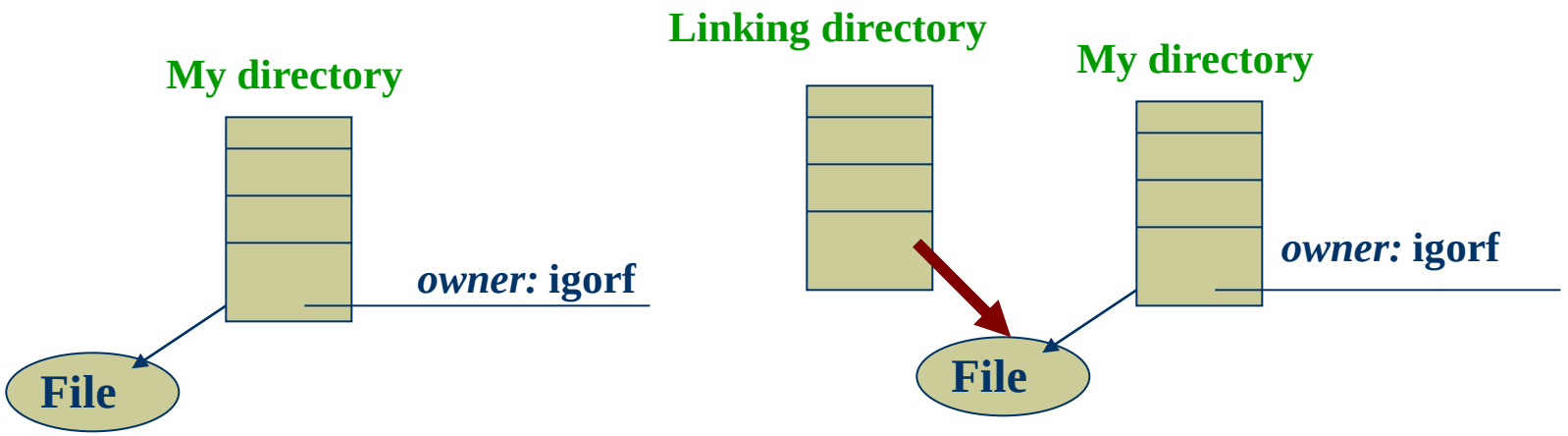
- The directory entry provides the information necessary to find the disk blocks of each file in the directory
- Depending on the system, the directory may point to the whole file block (contiguous allocation), the first block (linked list schemes), or the i-node
- The file attributes for each file may be stored in the directory, but it makes a lot of sense to keep them in the i-node and not duplicate it in the directory

Shared Files

- Linking is the most transparent way of making the same file appear in more than one directory
- Some directory structures (e.g., those used in CP/M) require that all file blocks be listed in a directory. In such cases, it is necessary to copy a file to another directory (and then it really does become another file, thus defeating the whole purpose of *sharing*...)
- One solution (*symbolic linking*) is to point to a new file, of a type *link*, which has the full path to the linked file
- The other solution is to point to a data structure (such as i-node), which deals with the file—this is how *Unix* does it

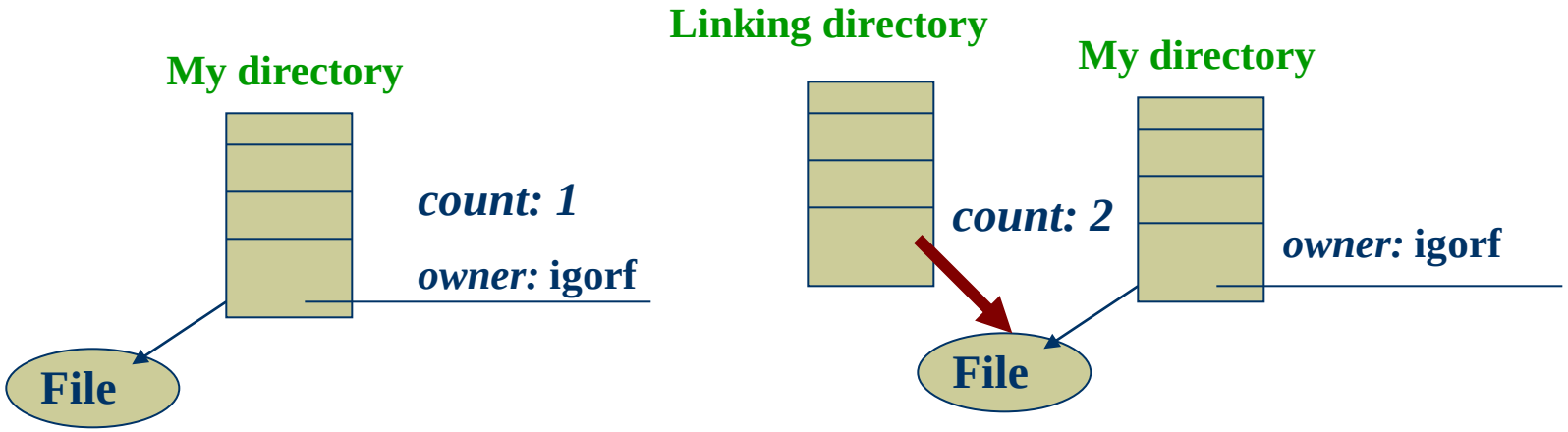
Linking Files

- Symbolic linking is straightforward; its only problem is the extra overhead (reading the path and following it component-by-component until the file is reached)
- The direct link poses a problem: After the owner deletes the file, the ghost link to a wrong (or non-existent) i-node still exists



Linking Files (cont.)

- Solution: a *counter* as a directory attribute. Until the count of links is zero, the system does not remove the file.

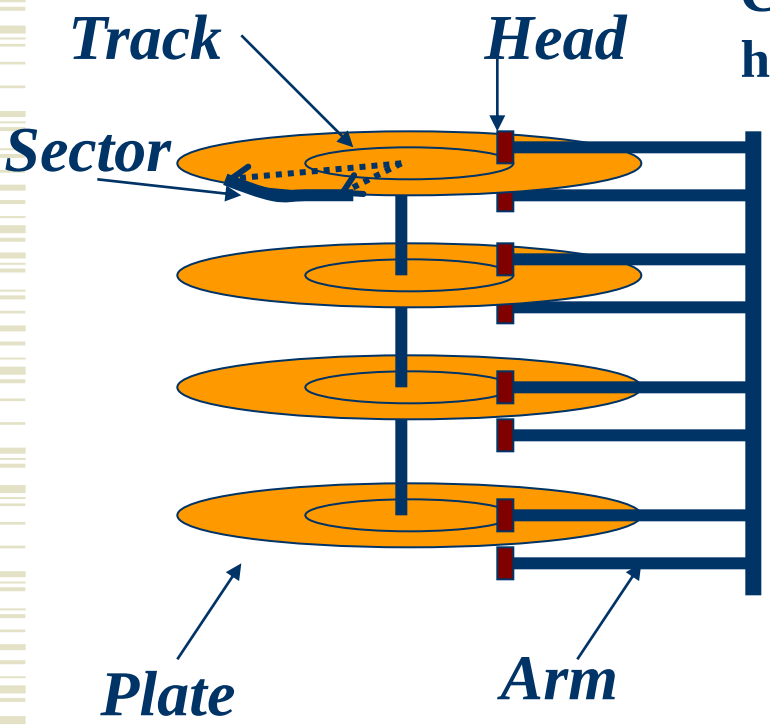


- And then there is a general problem with linking: several copies of the same file are fished out by the programs that look for it

Disk Space Management

- The tradeoff is the same as in segmentation versus paging; the solution: keep the file in non-contiguous, fixed-size blocks
- How big should the block be? From what we know about the disk, the candidates sizes are those of a *sector*, *track*, or *cylinder*. Then, in paging systems, a page size is another good candidate (Why?)

Disk (Reminder)



***Cylinder* = set of all *tracks* under the heads in a present position**

Commands:

- ***SEEK cylinder***
- ***READ track, [sector]***
- ***WRITE track, [sector]***
- ***RECALIBRATE***

Block Size

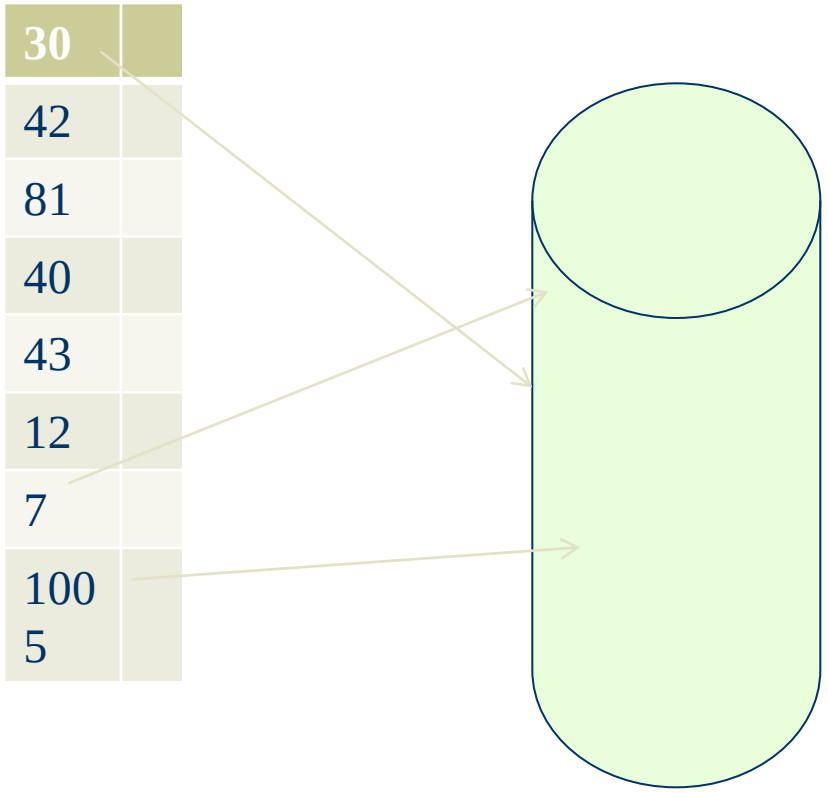
- Studies have shown that the median file size in the *Unix* environment is 1K, so allocating a 32K cylinder for each file, would waste 97 percent of the disk space.
- But if the file is broken into non-contiguous block, reading these blocks would result in compounding rotational and seek delays
- An example: A disk with *t* bytes per track, *r msec* rotation time, and *s msec* seek time. A block of *k* bytes then can be read in

$$s + r/2 + kr/t \text{ msec}$$

- Plotting this function against the disk space utilization, will show that good space utilization (block size less than 2K) gives low data rate (below 50 Kbytes/sec) and *vice versa*. An inherent conflict!
- A compromise: choose a block size of *512*, *1K*, or *2K* bytes. If a *1K* block is chosen on a disk with a *512*-byte sector size, the system will treat two sectors as an indivisible unit.

In-memory Caching

- Accessing the physical disk for frequently-used data can be avoided if the blocks in question are cached
- Blocks are read or written into the cache and then written out to disk when necessary or when the block must be removed from the cache to make way for a new block
- The tradeoff here is between the *performance* and *consistency* (more on this later)
- LRU is the algorithm of choice in most implementations



Managing Free Blocks

- Two methods are used
  1. Keep a linked list of (free) disk blocks
  2. A bit map (one bit for each block). If there is enough memory, this method is preferable: it makes it easy to allocate contiguous space

Managing Free Blocks (cont.)
Dragon Rules

- In general, disk space must be rationed on a mainframe
- The rationing mechanism enforces a *quota*. The quota record is kept in the file attributes. Whenever a user's file size increases, it is charged against the quota
- The *number of blocks (nob)* as well as the total *number of files (nof)* are typically kept track of. Two numbers, the *hard limit* and the *soft limit*, are defined for each *nob* and *nof*. The soft limit may be exceeded, at which point a warning is issued; if the hard limit is exceeded, the violating file operation causes an error
- A count of warnings for the soft limit violations is decreased with each warning (but is reset to its original default size when the warning is acted on). When the counter reaches zero, a user is denied a login.

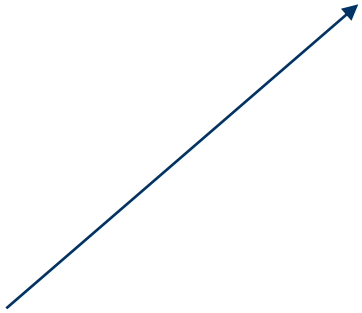
Quota Attributes (example)

Attributes:

User=*igorf*

Quota pointer

...



Soft block limit
Hard block limit
Current number of blocks
Number of block warnings left
Soft file limit
Hard file limit
Current number of files
Number of file warnings left

File System Reliability

- If a computer is lost or destroyed, it is a matter of money (possibly, for an insurance company)
- If a file system is lost, it could mean catastrophic loss of documents (including tax records, photographs, years of work...)
- The operating systems actions to prevent the catastrophe are:
  - Bad-block management (vs. *bad* block management)
  - Backup provision
  - Ensuring the file system consistency

Bad-block Management

- *Hardware solution*: dedicate a sector for the list of bad blocks and let the controller to use a spare track into which bad block addresses are mapped
- *Software solution:* create a file that consists of bad blocks—and ensure that it is never touched (even during back ups!)

Backups

- The image of the entire drive can be backed to a tape, but it is time-consuming. This job is performed by utilities (which may even not be a part of the operating system), which just dump (or restore) the disk track-by-track
- The file system can be stored, too, although that would take *even more* time than the complete-image dump (Why?)
- But then only the files that have changed after the full file system dump can be saved—this is called an *incremental backup*. A list of dump times for each file is stored on the disk, and the backup program checks each file against this list. The logistics: a few tapes to hold the whole file system dumped once a month and 31 tapes to hold daily incremental backups

File System Consistency

- If the system crashes before all the modified blocks are written out (especially, when it concerns i-nodes, free list blocks, and directory blocks), the file system will become inconsistent
- When the system is booted (particularly, after a crash), the operating system checks the consistency

The Consistency Check Algorithm

- Two kinds of checks can be made—for blocks, and for files
  1. *Block consistency check*: Build a table with two counters per block, both initialized to zero. The first counter keeps track of how many times the block has been used in a file; the second, how many times the block has been encountered in the free list. The algorithm reads all files, then all free blocks, and updates the counters.

The Consistency Check Algorithm (cont.)

1. *Block consistency check*: (cont.)

- *One entry is 0, and another is 1*: everything is perfect
- *Both entries are 0:* the block is *missing;* add it to the free list
- *Both entries are 1:* remove the block from (or add it to) the free list, depending on the allocation process
- *A free-list entry is greater than 1:* rebuild the free list
- *A file counter entry is greater than 1:* for *each* extra file, allocate a free block and copy the contents of the suspicious block into it; then insert the copy into the file. Report possible error.

The Consistency Check Algorithm (cont.)

File consistency check:

1) Starting from the root directory, build a table of counters for each file—while recursively descending the tree—for **each file**, in each directory, for each i-node, increment the counter for the i-node.
2) When this is done, there is a list indexed by i-node number, telling how many directories point to it. For each i-node, compare this number to the link counter stored in the i-node itself

The Consistency Check Algorithm (cont.)

We have:

Set of counters for directories pointing at i-node *vs. Set of counters* within the i-nodes

- If these sets of numbers and counters jibe, the file system is consistent
- If the i-node link count is too high, we are wasting space; the link count should be fixed (and the file should be deleted, if it is zero)
- If more directories point to an i-node than the i-node thinks they do—there could be a disaster (when the file is deleted); again the link count must be fixed to agree with the file

And finally, there may be special checks for protection inconsistencies (should a file give outsiders more rights than to the owner?), i-node size, etc. All peculiarities are reported.

File System Performance

- The disk access is about 100,000 times slower than memory access
- To make things faster, the disk blocks are *cached* into the main memory or the flash storage—if present
- This situation is similar to paging, and it is very easy to implement LRU (because cache references are relatively infrequent)
- Ironically, LRU is undesirable with i-nodes : certain blocks (e.g., i-nodes) need to be written to disk immediately after they have been modified. (*Unix* achieves that by using the *sync* system call, issued by a *daemon* every 30 sec.)
- Another important technique is reducing the amount of disk arm motion. The i-nodes, for example, should be put in the middle of the disk