

## File System Abstraction

### Levels of abstraction:

	applications	daemons	servers
User Interface	create( ) delete( )	open( ) close( ) rename( ) link( )	read( ) write( )
Device-Independent Interface	tracks sectors blocks		
Device Interface	seek( )	readblock( )	writeblock( )
		disk	other hardware

### The hardware underneath:

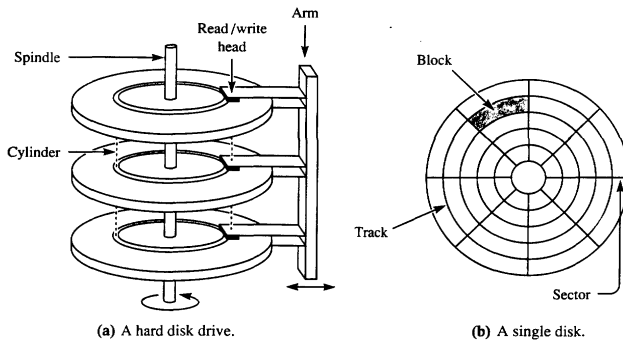


Diagram from *Computer Science*, Volume 2, J. Stanley Warford, Heath, 1991.

1

Spring 2001, Lecture 30

## File System Issues

### Important to the user:

- Persistence — data stays around between power cycles and crashes
- Ease of use — can easily find, examine, modify, etc. data
- Efficiency — uses disk space well
- Speed — can get to data quickly
- Protection — others can't corrupt (or sometimes even see) my data

### OS provides:

- File system with directories and naming — allows user to specify directories and names instead of location on disk
- Disk management — keeps track of where files are located on the disk, accesses those files quickly
- Protection — no unauthorized access

2

Spring 2001, Lecture 30

## User Interface to the File System

### A *file* is a logical unit of storage:

- A series of records (IBM mainframes)
- A series of bytes (UNIX, most PCs)
- A resource fork and data fork (Macintosh)
  - Resource fork — labels, messages, etc.
  - Data fork — code and data

### What is stored in a file?

- C++ source code, object files, executable files, shell scripts, PostScript...
- Macintosh OS explicitly supports file types — TEXT, PICT, etc.
- Windows uses file naming conventions — “.exe” and “.com” for executables, etc.
- UNIX looks at contents to determine type:
  - Shell scripts — start with “#”
  - PostScript — starts with “%!PS-Adobe...”
  - Executables — starts with *magic number*

3

Spring 2001, Lecture 30

## File Operations

### Create(*name*)

- Constructs a *file descriptor* on disk to represent the newly created file
  - Adds an entry to the *directory* to associate *name* with that file descriptor
- Allocates disk space for the file
  - Adds disk location to file descriptor

### *fileId* = Open(*name*, *mode*)

- Allocates a unique identifier called the *file ID* (identifier) (returned to the user)
- Sets the mode (r, w, rw) to control concurrent access to the file

### Close(*fileId*)

### Delete(*fileId*)

- Deletes the file's file descriptor from the disk, and removes it from the directory

4

Spring 2001, Lecture 30

## Common File Access Patterns

- Sequential access
  - Data is processed in order, one byte at a time, always going forward
  - Most accesses are of this form
  - Example: compiler reading a source file
- Direct / random access
  - Can access any byte in the file directly, without accessing any of its predecessors
  - Example: accessing database record 12
- Keyed access
  - Can access a byte based on a *key* value
  - Example: database search, dictionary
  - OS does not support keyed access
    - User program must determine the address from the key, then use random access (provided by the OS) into the file

5

Spring 2001, Lecture 30

## File Operations (cont.)

- **Read(*fileId*, *from*, *size*, *bufAddress*)**
  - Random access read
  - Reads *size* bytes from file *fileId*, starting at position *from*, into the buffer specified by *bufAddress*  

```
for (pos=from, i=0 ; i < size ; i++)  
    *bufAddress[i] = file[pos++];
```
- **Read(*fileId*, *size*, *bufAddress*)**
  - Sequential access read
  - Reads *size* bytes from file *fileId*, starting at the current file position *fp*, into the buffer specified by *bufAddress*, and then increments *fp* by *size*  

```
for (pos=fp, i=0 ; i < size ; i++)  
    *bufAddress[i] = file[pos++];  
fp += size;
```
- **Write** — similar to **Read**

6

Spring 2001, Lecture 30

## Directories and Naming

- Directories of named files
  - User and OS must have some way to refer to files stored on the disk
  - OS wants to use numbers (index into an array of file descriptors) (efficient, etc.)
  - User wants to use textual names (readable, mnemonic, etc.)
  - OS uses a *directory* to keep track of names and corresponding file indices
- Simple naming
  - One name space for the entire disk
    - Every name must be unique
  - Implementation:
    - Store directory on disk
    - Directory contains <name, index> pairs
  - Used by early mainframes, early Macintosh OS, and MS DOS

7

Spring 2001, Lecture 30

## Directories and Naming (cont.)

- User-based naming
  - One name space for each user
    - Every name in that user's directory must be unique, but two different users can use the same name for a file in their directory
  - Used by TOPS-10 (DEC mainframe from the early 1980s)
- Multilevel naming
  - Tree-structured name space
  - Implementation:
    - Store directories on disk, just like files
    - Each directory contains <name, index> pairs in no particular order
      - The file pointed to by a directory can be another directory
        - » Names have “/” separating levels
      - Resulting structure is a tree of directories
  - Used by UNIX
    - More on UNIX disk structures next time...

8

Spring 2001, Lecture 30