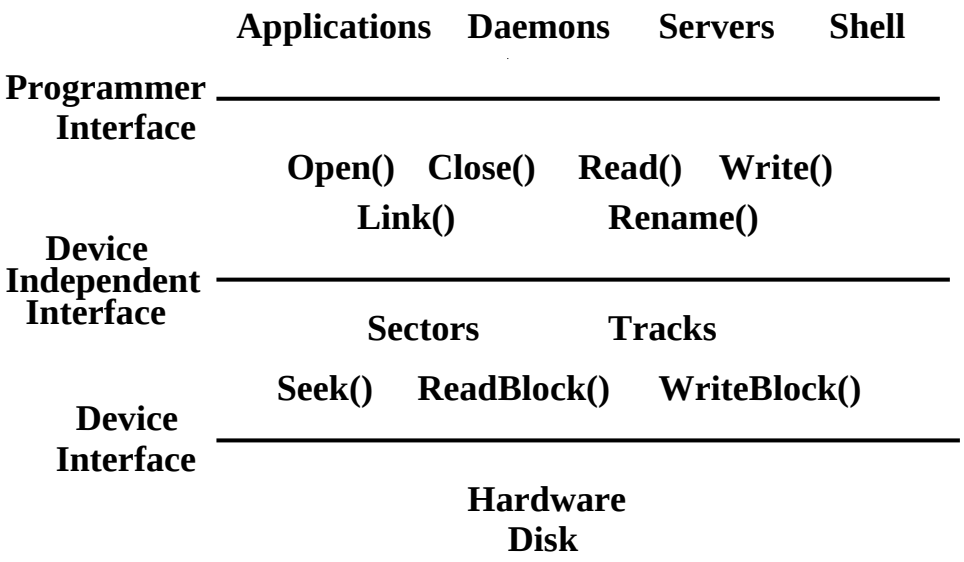
File System Functionality

Remember the high-level view of the OS as a translator from the user abstraction to the hardware reality.

User Abstraction		Hardware Resource
Processes/Threads		CPU
Address Space	$\Longleftarrow$ **OS** $\Longrightarrow$	Memory
Files		Disk

File System Abstraction



From the user's viewpoint, what's important?

Persistence - data stays around between jobs, power cycles, crashes

Speed - can get to data quickly

Size - can store lots of data

Sharing/Protection - users can share data where appropriate or keep them private when appropriate

Ease of Use - user can easily find, examine, modify, etc. data

Hardware provides:

Persistence - Disks provide non-volatile memory

Speed - Speed gained through random access

Size - Disks are big and getting bigger

OS provides:

Persistence - redundancy allows recovery from some additional failures

Sharing/Protection - Unix allows the owner of a file to control read, write, execute privileges for 3 classes of users: owner, group, world.

Ease of Use - provided through

- Associating names with chunks of data (files)
- Organize large collections of files into directories
- Maintaining the mapping from the user's concept of files and directories onto locations on disks

User Interface to the File System

File as an abstraction that contains information:

- Unix implements files as a series of bytes.
- Mainframes implement files as a series of records

Common file access patterns from the programmer's perspective:

Sequential: data processed in order, a byte or record at a time. Most programs use this pattern.

Example: compiler reading a source file.

Keyed: address a block based on a key value.

Example: database search, hash table, dictionary

Common file access patterns from the OS perspective:

Sequential: keep a pointer to the next byte in the file. Update the pointer on each read/write.

Random: address any block in the file directly given its offset within the file.

User Interface to the File System

Common file operations:

Data operations:

- Create()
- Delete()
- Open()
- Close()
- Read()
- Write()
- Seek()

Naming operations:

- HardLink()
- SoftLink()
- Rename()

Attributes (owner, protection,. . .):

- SetAttribute()
- GetAttribute()

OS File Data Structures

1. Open file table - shared by all processes with an open file.
    - open count
    - file attributes, including ownership, protection information, access times, ...
    - location(s) of file on disk
    - pointers to location(s) of file in memory
2. Per-process file table - for each file,
    - pointer to entry in the open file table
    - current position in file (offset)
    - mode in which the process will access the file (r, w, rw)

OS File Operations

1. **Create(name)** -
   - Allocate disk space (check disk quotas, permissions, etc.)
   - Create a file descriptor for the file including name, location on disk, and all file attributes.
   - Add the file descriptor to the directory that contains the file.
   - Optional file attribute: file type (Word file, executable, etc.)

     Advantages: better error detection, specialized default operations (double-clicking on a file knows what application to start), enables storage layout optimizations

     Disadvantages: makes the file system and OS more complicated, less flexible for user.

     In this case, Unix opts for simplicity. Macintosh opts for user-friendliness.

2. **Delete(name)** -
   - Find the directory containing the file.
   - Free the disk blocks used by the file.
   - Remove the file descriptor from the directory.

OS File Operations

3. **fileId = Open(name, mode)** -
   - Check if the file is already open by another process. If not,
     - Find the file.
     - Copy the file descriptor into the system-wide open file table.
   - Check the protection of the file against the requested mode. If not ok, abort the Open.
   - Increment the open count.
   - Create an entry in the process's file table pointing to the entry in the system-wide file table. Initialize the current file pointer to the start of the file.
   - Return the index into the process's file table. This index is used for read, write, seek, and close operations rather than the file name.
4. **Close(fileId)** -
   - Remove the entry for the file in the process's file table.
   - Decrement the open count in the system-wide file table.
   - If the open count == 0, remove the entry in the system-wide file table.

OS File Operations

5. **Read(fileID, from, size, bufAddress)** - random access

   - OS reads "size" bytes from file position "from" into "bufAddress"
   - basically:

```
for (i = from; i < from + size; i++)
     bufAddress[i − from] = file[i];
```

   Read(fileID, size, bufAddress) - sequential access

   - OS reads "size" bytes from current file position, fp, into "bufAddress" and increments current file position by size
   - basically:

```
for (i = 0; i < size; i++)
     bufAddress[i] = file[fp + i];
fp += size;
```

6. **Write** is similar, but copies from the buffer to the file.
7. **Seek** just updates fp.

Naming and Directories

Directories of named files:

- Need a method of getting back to files that are left on disk.
- OS wants to use numbers.
- Of course, users prefer textual names to refer to files.
- OS creates special disk structures called directories to tell what file descriptor indices correspond to what names.
- Naming strategies
  1. Simple & Stupid: One name space for the entire disk, where every name is unique.
     (a) Use a special area of disk to hold the directory.
     (b) Directory contains <name, index> pairs.
     (c) If one user uses a name, no one else can.
     (d) Some early computers used this strategy. Early personal computers also used this strategy because their disks were very small.

  2. Simple User Based Strategy - each user has a separate directory, but all of of each user's files must still have unique names (TOPS-10).

Naming Strategies (continued)

3. Multilevel Directories - tree structured name space (Unix, and all other modern operating systems).
   (a) Store directories on disk, just like files except the file descriptor for directories has a special flag bit.
   (b) User programs read directories just like any other file, but only special system calls can write directories.
   (c) Each directory contains <name, fileDesc> pairs in no particular order. The file referred to by a name may be another directory.
   (d) There is one special root directory.

   Example: How do we look up name: `/a/b/c`

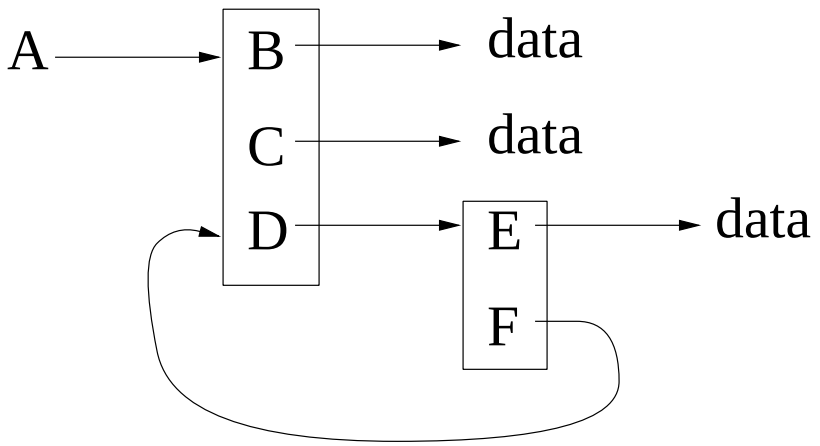
Naming

Referential naming:

1. Hard links (Unix: ln command)
   - A hard link adds a second connection to a file
   - *Example:* creating a hard link from B to A

Initially:	A $\rightarrow$ file #100
After "ln A B":	A $\rightarrow$ file #100
	B $\rightarrow$ file #100

   - OS maintains reference counts, so it will only delete a file after the last link to it has been deleted.
   - *Problem:* user can create circular links with directories and then the OS can never delete the disk space.



   - *Solution:* No hard links to directories.

2. Soft links (Unix: ln -s command)

   - A soft link only makes a symbolic pointer from one file to another.
   - *Example:* creating a soft link from B to A

Initially:	A $\rightarrow$ file #100
After "ln A B":	A $\rightarrow$ file #100
	B $\rightarrow$ A

   - removing B does not affect A
   - removing A leaves the name B in the directory, but its contents no longer exists
   - *Problem:* circular links can cause infinite loops in some algorithms, such as trying to list all the files in a directory and its subdirectories
   - *Solution:* limit number of links traversed.

Summary of File System Functionality

- Naming
- Protection
- Persistence
- Fast access