**file**: kind of storage facility (like a variable), with properties like:

**persistent and non-volatile**: Content and name is independent of processes, and of system reboots. Use of external media.

**sharable**: concurrent access.

**large size**: (generally).

(The file abstraction was different for generations of operating systems. Yesterday’s “indexed files” are today’s “databases”.)

(Different OSs have various **file** abstractions).

We begin: User View. Where we will go:

- File abstraction implementations.

- File systems=(standards for) **data structures on say a disk**. Tanenbaum’s Multimedia ch. covers specialized file systems for multimedia servers. (1) many files, each has a (2) name, to be (3) accessed efficiently.

- I/O architecture.
**file name:** a string passed to the OS in a system call (e.g. `open()`) to have a process access a file.

**Tree-structured name spaces:** Each **absolute pathname** has the form:

```
dirname1\dirname2... {filename}
```

where “\” delimits pathname **components**

Each file is an **object**: unique identity and contents.

(In current OS, one file might have several pathnames)

**Directory:** A file that contains names and other information about the files and directories directly underneath it in the name space. It’s a sequence of **directory entries**.

**Absolute pathname:** first char=\ 
Corresponds to the path from the root.

**Current Working Directory:** a process attribute that determines the meaning of **relative pathnames**

(which have 1st char ≠ \)

meaning=\CWD\relative pathname

In Unix shells, **cwd** is a **built-in shell command** that changes the current working dir. attribute of the process running the shell. (How? shell calls a syscall.)

```
$ cwd /tmp
$ /home/sdc/a.out
```
application is running...

system call int fd; fd=open("blooper",O_RDWR);
make kernel look up /tmp/blooper

**Hard link:** A directory entry that specifies a file (or
directory) directly.

**Symbolic or soft link, or shortcut:** a directory entry
that specifies just another *pathname*.

Symbolic links, (like **Universal Resource Locators**
(UR**Ls**), telephone numbers, etc.) might be “dangling
pointers”. Hard links are not (when the filesystem is
consistent.)

Files have different **types** that affect what system call
operations are legal on them and what these operations
mean. Unix file types: regular, directory, symbolic link,
block device, character device, named pipe, socket.

In Unix, each regular or directory file object is
implemented by an **inode**.

Inode attributes: file type, number of hard links
targeting me, file length, device ID (for device files),
inode number, owner user ID, group ID, time stamps,
access rights and modes, suid, sgid and “sticky” bits.

(The main data structure implementing one file on a
disk is the **on-disk-inode**)

3
When Unix process successfully calls `fd=open(...), the kernel
(1) constructs a new **open file object**;
(2) finds the first unused file descriptor array entry
`current->files[i]`;
(3) returns i as the new `fd` (file descriptor integer).

An open file object is different from a file: MAIN
FIELDS: **position**, **directory-entry**, **operations**

See Bovet Ch.12 p.341-2 for details.

Each open file object holds the data related to a
process-to-open-file interaction.

When a Unix process forks or execs\(^a\) the open file
objects are retained and the `files[]` array is copied.

\(^a\)by default. Any open file can be given the “close on exec”
attribute.
Concurrent process-to-open-file interaction semantics:

**UNIX Semantics:** Writes to an open file by a user are visible immediately to other users that have this file open concurrently. Under one mode of sharing, users share a current location (offset).

**Session Semantics:** Writes to an open file are not ... Once a file is closed, changes made to it are visible only in sessions starting later.
Some file-handling system calls (see them all in Bovet Ch.12, p.333):

\[
\text{fd} = \text{open} (\text{path}, \text{flags}, \text{mode});
\]

\[
\text{newoffset} = \text{lseek} (\text{fd}, \text{offset}, \text{whence}); \text{ for regular files only.}
\]

\[
\text{nread} = \text{read} (\text{fd}, \text{pbuff}, \text{count});
\]

\[
\text{res} = \text{close} (\text{fd}); \text{ closing a file means releasing one reference to the open file object. When all references are released, the open file object is “destroyed”,}
\]

\[
\text{rename} (\text{oldpath}, \text{newpath});
\]

\[
\text{unlink} (\text{pathname}); \text{ What a shell’s \text{rm} command does. The inode \text{remains} until all referring fd.s are closed and all referring pathnames are unlinked.}
\]
Introduction to **virtual functions**

Here’s an example of a common situation in programming:

```c
fd=open("somefile",O_RDONLY);
fun(fd);  //read from a regular file.
fd=socket(...);
connect(fd,..);
fun(fd);  //read from the internet.

fun(int fd) {
    char buf[SIZE]; int nr;
    nr=read(fd, buf, SIZE);
    //The GENERIC POSIX read()!!
    ...
    close(fd);
}
```

*The SAME CODE* (body of `fun()`) calls for reading a disk file or reading from the network **DEPENDING ON THE KIND OF OBJECT** the file descriptor `fd` describes.
A **class** is a data type (programmer defined structure) that has bundled into it some named operations called **methods** or **function members**.

An **object** or **instance** is a (structured) variable whose type is a given class.

We write programs that refer to an object and call methods belonging to that object’s class. We sometimes want the methods to belong to the object itself. So, when the *same named operation* is done on *different kinds of objects*, different implementations of that operation are called!

How to make this happen:

**New Way**: Use an object oriented language (Java or C++), define a class with the named methods, and define one **subclass** for each different kind of object. Give the same named method in each different subclass different implementations. (In C++, those methods must be declared **virtual** in the common class. In Java, all methods are virtual.)

**Old Way**: Include **function pointer fields** in the structure declaration. Initialize them with the addresses of the correct functions when the structure is instantiated.