

Topic:File Systems

OSCJ Ch.11 (system interface: how to use files)

OSCJ Ch.12 (File implementation)

ULK Ch.12 (Virtual File System)

ULK Ch.17 (The Ext2 Filesystem)

Topic:I/O

OSCJ Ch. 13 (I/O Systems)

OSCJ Ch. 14 (Mass Storage **DISKS** incl. RAID)

ULK Ch.4 (Interrupts and Exceptions)

ULK Ch.13 (Device Drivers)

We begin: User View. Where we will go:

- File abstraction implementations.
- File systems=(standards for) **data structures on say a disk.**

THE PROBLEM: (1) many files, each has a (2) name, to be (3) accessed efficiently when open, (4) stored safely when system is not running, and (5) supporting flexible, extendable implementation on different kinds of hardware, new and old.

(Tanenbaum's *Modern Operating Systems* textbook's Multimedia ch. covers specialized file systems for multimedia servers.)

- I/O architecture.

System Interfaces

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).

directory structure: "organizes, provides information ..."

(Today, more generally, it provides a **name space** for names of several classes of objects with system support and significance: For example, sockets, shared memory areas, locks, device interfaces, kernel variables, ...)

Other examples of name spaces: (1) URLs such as

`http://domain-name/rather-arbitrary-stuff`

"RFC-2616" documenting **hypertext transfer**

protocol is published by `http://www.ietf.org`

(2) the *domain-name* (which typically contains dots)

itself belongs to a namespace implemented by a

worldwide distributed database called the **Domain**

Name System (DNS).

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:

$\backslash \textit{dirname1} \backslash \textit{dirname2} \dots \{ \backslash \textit{filename} \}$

where “\” delimits pathname **components**

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

(In current OS, one file might have several pathnames. The directory structure is generally a **directed acyclic graph** rather than just a directed tree.)

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 $\neq \backslash$)

meaning= $CWD \backslash \textit{relative\ pathname}$

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.)

```
$ pwd /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 (URLs)**, 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.

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.

File IMPLEMENTATION now!

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**)

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 ULK(ed.2) Ch.12 p.385-8 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.

^aby default. Any open file can be given the “close on exec” attribute.

Some file-handling system calls (see them all in ULK, Table 12-1):

```
fd=open(path, flags, mode);
```

```
newoffset=lseek(fd, offset, whence);
```

 for regular files only.

```
nread=read(fd, pbuff, count);
```

```
res=close(fd);
```

 closing a file means releasing one reference to the open file object. When all references are released, the open file object is “destroyed”,

```
rename(oldpath, newpath);
```

```
unlink(pathname);
```

 What a shell’s `rm` command does. The inode *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:

```
fd=open("somefile",O_RDWR);

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.

(See OSCJ Chapter 5 on Java threads for an example: Different `run()` methods are defined in different implementations of `Runnable` to give different programs to different threads.)

(example of) How to declare a function pointer:

```
int (*pfun)(int arg1, void * arg2, int arg3);
```

(example of) How to define a function pointer:

```
int fun1(int a, void * arg2, int X)
{ //body..
}
```

```
pfun = fun1; //assignment statement
```

How to call the function pointed to by a function pointer:

```
returnval = pfun(3, array, 17);
returnval = (*pfun)(3, array, 17);
//two alternative C/C++ syntaxes
```

Real examples: Consult definitions of `struct file` and `struct file_operations` in `/usr/src/linux/include/linux/fs.h`

Note: In the Linux definition of the virtual functions defined in `struct file_operations`, a pointer to a `struct file` is usually **THE FIRST OPERAND** to each virtual function.

In OOP, every call of an instance method is done **WITH RESPECT TO** or **FOR** an instance of the class. The pointer to this object is passed implicitly—programmers can access it with the reserved Java/C++ word **this**.

Some data structures involved for each “open file” in Linux under the “virtual file system”.

(`struct file` from `/usr/src/linux/include/fs.h`
fd array linked from `task_struct`

ULK fig. 12-2,

then page 375 under the discussion of the “Common File Model”)

File Types (OS/360) Unix: NO OS support, file contents=array of bytes; OS must understand certain file formats (executable and DLL,)

IBM OS/360: different interfaces and implementation for sequential, direct, and indexed-sequential access files.

Unix's answers: (1) Separate files for a database's index and data records (2) Raw access to disk partitions for use by data base software.

Windows 95/Macintosh: Registry of file types, with "icons" and "applications" to "open them".

Windows XP: file=structured object consisting of typed attributes

Important to distinguish:

1. “Access Methods and Directory Structures” provided by application programming interface API.
2. Hardware on which a file is ultimately stored.
3. “Partitioning” of a hardware unit (such as a disk), done by one level of software.
4. A data structure that implements directories, file names, attributes and file contents residing on one partition is called a **file system**.
5. Internal kernel data structures and methods to make use of say a disk file system efficient.
6. Device driver software.

Allocation Methods, Free Space Management,
and **Directory Implementation** are key issues in the
architecture of **file systems** (disk-based data
structures)

Summary of Magnetic Disk Hardware:

1. Interface presented to software provides the disk as an array of fixed size “sectors”, usually 512 bytes each, which can be read or written via direct memory access in contiguous groups of 1 or more.
2. On older disk drives, each sector had a particular position on the magnetic platters. Each magnetic surface was used as a set of concentric tracks. Each track had the same number of sectors. All the tracks accessible with the arm holding the disk heads in a fixed position are called one “cylinder”. Therefore, one form of sector address used three numbers: Cylinder, Track, and Sector. On newer drives, inner tracks have fewer sectors than outer tracks. The drive controller (an embedded CPU in the disk drive unit) together with local memory implemented a mapping from linear or 3-dim sector address to physical sector location.
3. Data was stored encoding it into **error detecting and correcting code** words, and then the bits of code words are stored as different directions of magnetization of regions on the magnetic surface. The drive controller does calculations to (1) translate data to be written into code words and (2) check a code word read from the disk to detect

if it has a few errors. If the number of errors is small enough, the original data can still be calculated and the decoder will “know” a sector is going bad.

4. Modern disk drives detect bad sectors automatically, and will then automatically change the mappings so the corresponding logical sector is mapped to a different physical place. Therefore, sectors with contiguous logical addresses might not be contiguous on the disk surface.
5. Disk drives have buffer and cache memory, typically 8-16MB. Therefore a read or write operation will not always require the time to physically move the disk arms and wait for the addressed sectors to move under them.
6. The caching and automatic bad sector mappings are “transparent” to the software (ie, device drivers in the OS). Indeed their details are proprietary to the drive manufacturers.

(Right after class, a member told me Western Digital just announced (Nov. 2004) that its new disk drives will have built-in request scheduling.)

The “16-25” error correcting code (16 data bits d_{ij} and 9 parity check bits c_{ij}):

c_{00}	c_{01}	c_{02}	c_{03}	c_{04}
c_{10}	d_{11}	d_{12}	d_{13}	d_{14}
c_{20}	d_{21}	d_{22}	d_{23}	d_{24}
c_{30}	d_{31}	d_{32}	d_{33}	d_{34}
c_{40}	d_{41}	d_{42}	d_{43}	d_{44}

10 parity check conditions: For each of the 5 rows, and for each of the 5 columns, the number of 1 bits is EVEN.

If one bit is in error, parity check will fail in one row and one column—the disk controller can observe which row and column they are and **correct the error**.

All cases of up to 3 errors can be detected, and sometimes more than 3 errors can be detected.