Persistence: File System API

CS 537: Introduction to Operating Systems

Louis Oliphant & Tej Chajed

University of Wisconsin - Madison

Spring 2024

Administrivia

- Project 6 has been released
- Midterm 2 grades have been released
  - Average: 86%, standard deviation 11%
  - See Canvas Files for solution with explanations
- Project 4 grades out soon

Review: RAID

- RAID 0 (striping)
- RAID 1 (mirroring)
- RAID 4 and RAID 5 (parity)

Striping

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

$N \cdot S$ sequential read throughput

RAID 5 parity

Disk 0	Disk 1	Disk 2	Disk 3	Disk 3
0	1	2	3	**P0**
4	5	6	**P1**	7
8	9	**P2**	10	11
12	**P3**	13	14	15
P4	16	17	18	19

$N/4 \cdot R$ random write throughput

Review: RAID

https://tinyurl.com/cs537-sp24-q16

File Systems

Disks alone would be hard to use

A **file system** is an abstraction for persistent storage

Main concepts: files and directories

Why care about the file system?

Common to many, many systems: Window, macOS, Linux, Android, iOS

Essentially all storage goes through a file system

You will likely use this API

What's cool about file systems?

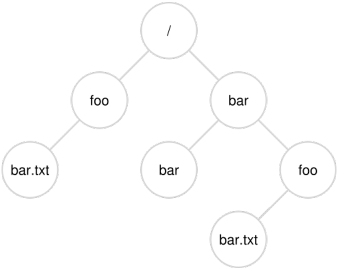
User management: you can interact with file system directly

Allocation: file system helps you dynamically allocate storage without thinking about it too much

Implementation: you'll be able to understand how the API is implemented

File System abstractions

- **File** – A linear array of bytes that you can read, write, and resize.
- **Directory** – Contains mappings from names to other directories and files. This creates a **directory tree.**
- **File system** – Refers to the whole collection. Also refers to the implementation (e.g., ext4, NTFS, APFS, btrfs).



Naming a file

API needs a way to refer to a file

Three types of names:

- inode number (unique number)
- path
- file descriptor

Why not just use paths?

```
read(char *path, void *buf, size_t nbyte)
write(char *path, void *buf, size_t nbyte)
```

Disadvantages: expensive traversal on every operation

File descriptors

Idea:

- do expensive traversal once (in `open` syscall)
- store inode in process memory as *file descriptor*
- do reads/writes/etc via descriptor

Note that we have a *per-process* file-descriptor table

File descriptors are just indexes into this table

Creating and opening Files

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

- `"foo"` – the relative or absolute pathname of the file to be opened
- `O_CREAT|O_WRONLY|O_TRUNC` – flags indicating creation, write-only, and truncate if file already exists
- `S_IRUSR|S_IWUSR` – permissions, readable and writable by the owner
- `fd` – file descriptor, an integer into array of opened files, managed by OS on per-process basis.

```
struct proc {
    ...
    struct file *ofile[NOFILE]; // Open files
    ...
}
```

Reading and Writing Files

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>

prompt> strace cat foo   -- prints system calls performed by program
...
open("foo", O_RDONLY|O_LARGEFILE)          = 3
read(3, "hello\n", 4096)                   = 6
write(1, "hello\n", 6)                     = 6
hello
read(3, "", 4096)                          = 0
close(3)                                   = 0
...
prompt>
```

Reading and Writing, But Not Sequentially

```
off_t lseek(int fildes, off_t offset, int whence);
```

- `fildes` – the file descriptor
- `offset` – position within the file
- `whence` – How offset is used
  - `SEEK_SET` – the offset is set to the offset in bytes
  - `SEEK_CUR` – the offset is set to its current location plus offset bytes
  - `SEEK_END` – the offset is set to the size of the file plus offset bytes

```
struct file {
    int ref;
    char readable;
    char writable;
    struct inode *ip;
    uint off;
}
```

Shared File Table Entries – fork() and dup()

File table entries are shared when calling fork() or dup():

```
int main(int argc, char *argv[]) {
   int fd = open("file.txt", O_RDONLY);
   int rc = fork();
   if (rc == 0) {
      rc = lseek(fd, 10, SEEK_SET);
      printf("child: offset %d\n", rc);
   } else if (rc > 0) {
      (void) wait(NULL);
      printf("parent: offset %d\n", (int) lseek(fd, 0, SEEK_CUR));
   }
}
```

```
prompt> ./fork-seek
child: offset 10
parent: offset 10
prompt>
```

When file table entry shared, reference count incremented; both processes close file before removed

Writing Immediately with `fsync()`

Typically, writes are buffered by the OS for some time (say 5 seconds, or 30 seconds)

`fsync(int fd)` – forces all dirty data to disk, Only returns after all writes are complete.

Renaming Files

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

An atomic instruction – file will either be oldpath name or newpath name.

Information About Files

The `inode` keeps **metadata** about a file or directory. You can see some of this information by using the command line tool `stat`:

```
prompt> echo hello > file
prompt> stat file
   File: 'file'
   Size: 6 Blocks: 8                  IO Block: 4096     regular file
Device:  811h/2065d Inode: 67158084    Links: 1
Access:  (0640/-rw-r-----) Uid: (30686/ remzi) Gid: (30686/ remzi)
Access:  2011-05-03 15:50:20.157594748 -500
Modify:  2011-05-03 15:50:20.157594748 -500
Change:  2011-05-03 15:50:20.157594748 -500
```

Removing Files

```
prompt> rm foo
```

```
unlink("foo");
```

Making Directories

```
prompt> mkdir foo
```

```
mkdir("foo",0777);
```

An "empty" directory has two entries: `"."` refers to itself, and `".."` refers to its parent. You can see these by passing the -a flag to `ls`:

```
prompt> ls -a
./      ../
```

Reading Directories

```
int main() {
   DIR *dp = opendir(".");
   struct dirent *d;
   while ((d = readdir(dp)) != NULL) {
      printf(%lu %s\n", (unsigned long) d->d_ino, d->d_name);
   }
   closedir(dp);
}
```

```
struct dirent {
   char d_name[256]; // filename
   ino_t d_ino; // inode number
   off_t d_off;  // offset to next dirent
   unsigned short d_reclen; // length of record
   unsigned char d_type; // type of file
}
```

Deleting Directories

```
prompt> rmdir directory
```

```
rmdir("directory");
```

Can only delete "empty" directories

Hard Links

Hard links create another name to the same inode number:

```
echo hello > file
ln file file2
echo bye > file
cat file2
```

That is why unlink is the same as removing a file (if no more references then inode is deleted)

Symbolic links

Symbolic (soft) links are special files containing linking information. If underlying file is deleted you can get **dangling references**.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

Permission Bits and Access Control Lists

Unix **permission bits** control who has access to a file. You can see these permissions with `ls`:

```
prompt> ls -l foo.txt
-rw-r--r-- 1 remzi wheel  0 Aug 24 16:29 foo.txt
```

First entry is file-type followed by 3 bits (rwx) of **owner**-permission, 3 bits (rwx) of **group** permissions, and 3 bits (rwx) of **other** permissions.

Access Control List in AFS

AFS permissions do not use the UNIX permission bits.

- More flexible in some ways (e.g., ACLs; separate delete, admin permissions)
- Less flexible in others (e.g., only per-directory permissions)

You can read about the CS department's AFS system https://csl.cs.wisc.edu/docs/csl/2012-08-16-file-storage/.

- `fs listacl <path>` – lists the access control list for the directory
- `fs setacl <path> <user> <acl>` – Set the access control list for the user to the path.

Making and Mounting File Systems

mkfs.<fs> <device> – creates an empty file system on the given device.

- e.g., mkfs.ext4 and mkfs.btrfs

sudo mount -t <type> <device> <mount point> – mounts the filesystem on the device to the given mount point. After running the command the contents under mount point will be the file system on the device.

Summary

- File-system abstractions: files, directories, directory trees
- API is based on per-process file descriptors
- Several categories of operations: links, directories, permissions
- Mounting a file system