Persistence: Raid & File Systems

CS 537: Introduction to Operating Systems

Louis Oliphant

University of Wisconsin - Madison

Fall 2023

Administrivia

- Project 6 due Nov 22nd @ 11:59pm
  - Tests should be out now
  - Run `pip3 install grequests` for library for some tests
- Exam 2 Grades later this week

Review IO Devices, Disks, and Scheduling

- The cannonical IO Device has a hardware interface, OS communicates to device via layers, typically, from File-system layer, to block layer, to driver
- Disks physical properties and layout limit performance
  - Calculate time to handle IO request (seek + rotation + transfer)
  - Calculate rate = size / time
- Scheduling of IO requests can dramatically improve performance from random requests to more sequential requests

Quiz 15: Disks Transfer Rates

https://tinyurl.com/cs537-fa23-q15

RAID Agenda

- RAID Systems
  - Understand Levels 0 (striping), 1 (mirroring), 4 (parity), and 5 (rotating parity)
  - Measuring Capacity, Performance, and Reliability compared to a single disk
- File Systems
  - Creating, Reading, Writing, Deleting files and directories
  - Permissions, Access Control Lists
  - Make and Mounting File Systems

Redundant Arrays of Inexpensive Disks

- Externally, a **RAID** looks like a disk (it is transparent to the OS)
  - Works just like a single disk
- Internally, there are lots of configurationtypes to:
  - RAID Level **0**, **1**, 2, 3, **4**, **5**, 6
  - Be larger than a single disk (**Capacity**)
  - Work faster (**Performance**)
    - IO is often a bottleneck to performance
    - Highly dependent on workload type (**random** and **sequential**)
  - Provide **Reliability**
    - Functioning with failure of one or more disks

RAID Level 0 - Striping

- No redundancy, blocks are striped across the array of disks
- Blocks in the same row are called a stripe.
- Chunk size can vary between RAID arrays (1 block (4KB), 2 block, etc.)
  - Small chunk size means files will be striped across many disks, increasing parallelism
  - Reduces intra-file parallelism, relies on multiple concurrent requests

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

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

RAID 0 Analysis

RAID 0 of N disks

- Capacity: perfect – the same as N individual disks
- Reliability: perfectly horrible – any disk failure and data is lost
- Performance:
  - Single Read latency
  - Steady-state bandwidth
    - Sequential
    - Random
- Assume single disk performance:
  - Holds B blocks
  - S MB/s for sequential workload
  - R MB/s for random workload

RAID 0 Analysis (Performance)

Can use all disks at once (Maximize Parallelism):

- single read latency – nearly identical to that of a single disk
- Sequential Rate – $N \cdot S$ MB/s
- Random Rate – $N \cdot R$ MB/s

RAID Level 1 - Mirroring

- Make more than one copy of each block in the system; each copy should be placed on a separate disk
- When reading a block there is a choice (can read from either)
- When writing, need to write both copies (can be done in parallel)

RAID 1 + 0
Mirrored pairs and then stripes

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

RAID 0 + 1
Stripes and then mirrors

Disk 0	Disk 1	Disk 2	Disk 3
0	1	0	1
2	3	2	3
4	5	4	5
6	7	6	7

RAID 1 Analysis

- Capacity: $(N \cdot B)/2$ blocks
- Reliability: Tolerate 1 failure (if lucky up to N/2 failures)
- Performance:
    - Latency: Same as a single disk
    - Sequential Write: 2 physical writes for each logical write $(N/2) \cdot S$ MB/s
    - Sequential Read: Each disk skips every other block $(N/2) \cdot S$ MB/s
    - Random Read: $N \cdot R$ MB/s (can parallelize requests)
    - Random Write: $\frac{N}{2} \cdot R$ MB/s

RAID Level 4 - Saving Space with Parity

Disk 0	Disk 1	Disk 2	Disk 3	Disk 3
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Use **XOR Parity**, xor-ing the blocks

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1)=0
0	1	0	0	XOR(0,1,0,0)=0

RAID 4 Analysis

- Capacity: $(N-1) \cdot B$ blocks
- Reliability: Tolerate 1 disk failure
- Performance:
  - Latency: same as single disk for read, twice as long for write (why?)
  - Sequential Read: $(N-1) \cdot S$ MB/s
  - Sequential Write: $(N-1) \cdot S$ MB/s
    - Utilize **full-stripe** write
  - Random Read: $(N-1) \cdot R$ MB/s
  - **Random Write:** $(R/2)$ **MB/s**
    - **Parity Disk is a bottleneck**
    - **subtractive parity**: $P_{new} = (C_{old} \oplus C_{new}) \oplus P_{old}$

RAID 5 - Rotating Parity

- Rotate the parity block across drives
- Now the parity disk is not the bottleneck
  - Performance on Random Writes goes to $\frac{N}{4} \cdot R$

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

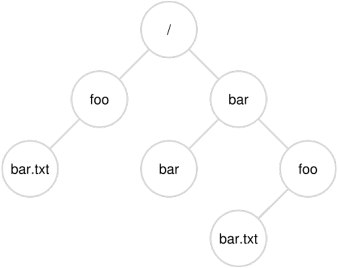
Comparing RAID Levels

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (maybe more)	1	1
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} \cdot R$
Latency Read	T	T	T	T
Latency Write	T	T	2T	2T

File Systems

A file system is an abstraction of a persistent device, containing data structures and access methods for interacting with this system. The two main abstractions are:

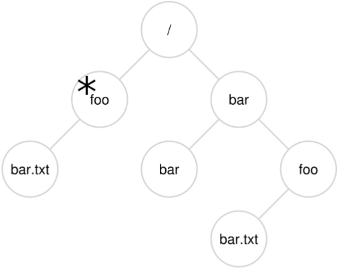
- **File** – A linear array of bytes that you can read or write (has a user-level name and low-level name(**inode number**))
- **Directory** – Contains list of mappings between (user-level name to low-level name) of files and other directories. This creates a **directory tree.**



User-level Names

A file or directory has an **absolute pathname**. They also have a **relative pathname** depending on the **current working directory**.

`/foo/bar.txt`
`/bar/foo/bar.txt`
`/bar/bar/`



Relative pathnames if current working directory is `/foo`:

`bar.txt`
`../bar/foo/bar.txt`
`../bar/bar/`

File extensions (e.g. .txt) are often used to indicate the content of the file.

Creating 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()`:

```c
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

```c
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);
}
```

```c
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
```

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

Can only delete “empty” directories.

Hard Links and Symbolic Links

- Hard links create another name to the same inode number:

```
echo hello > file
ln file file2
```

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

- 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

You can read about The CS departments 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 <device>` – creates an empty file system on the given device.

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