CS 423 Operating System Design: COW FS & Remote Access Apr 28

Ram Kesavan

LOGISTICS

Midterm papers: pickup from my office during OH

MP4 due by May 6, 11:59 CT

Last lecture: 5/5 will be a review

Will publish some practice questions next week

Finals: 5/13 at 7-10pm, DCL 1320 (next door)

AGENDA / LEARNING OUTCOMES

Enterprise-grade COW File Systems

- Eg. WAFL, ZFS, BtrFS

Remote Access of File Systems

RECAP

ENTERPRISE-GRADE 2-NODE HA SYSTEM

Clients

Ethernet, FCP, NVMe

NFS, SMB, SCSI, NVMe-oF, etc.	NFS, SMB, SCSI, NVMe-oF, etc.
File System / NV Log	NV Log / File System
RAID	RAID
Storage Drivers	Storage Drivers

NVMe, FCP, Ethernet

Storage Media

Many RAID groups: DP or TP

FILE SYSTEM LAYOUT

WAFL file system is a tree of blocks

- **everything is a file**, including all metadata
- metadata files have well-known inums
- each dirty buffer written to new (free) block in storage, except superblock

Modify op:

- dirties in-memory state
- creates log entry in NVRAM, is mirrored to partner's NVRAM
- then ack'ed
- **COW file system**: so everything is written to a new spot in persistent storage (except superblock)

Large checkpoints: ~GB worth of dirty buffers

- ~2s to 5s long
- all dirty buffers can be persisted in any order; superblock written last
- COW: so persistent file system is always self-consistent

PICTURE OF FILE SYSTEM TREE

Original WAFL paper: https://rcs.uwaterloo.ca/papers/wafl.pdf

Everything is a file

All indirects have the same fanout (#ptrs)

File trees are symmetric & balanced

4 GiB of storage
= 4GiB/4KiB blocks
= 1M/32k bits-per-blk
= 32 bitmap blocks

4 TiB of storage
= 32k bitmap blocks

END RECAP

COW: Atomic Checkpoint

CP_i superblk A B C D E

CP_{i+1} superblk B' E'

Only ordering required in a CP:

All blocks of a CP must be written out before new superblock is written.

So, B' & E' before superblk

CHECKPOINTS, JOURNAL, & DOUBLE BUFFERING

In-memory FS

clients sending modify ops

dirty inodes & buffers due to op_1 thru' op_n

in-progress: CP_{i+1}

yet to start: CP_{i+2}

dirty inodes & buffers due to op_{n+1} onwards

overlapped state: copy of inode+bufs created for CP_{i+2}

Persistent FS, has CP_i

NVRAM Journal

CP_{i+1}: op_1, op_2, . . . op_n

CP_{i+2}: op_{n+1}, op_{n+2}, . . .

Block Bitmap Updates

CP_i CP_{i+1}

superblk → A, B
superblk → A, B'
A → C
B → D, E
B' → D, E'

4 GB/s writes need

- 1 M/s allocations of new blocks
- 1 M/s frees of unused blocks

Allocations: [ICPP paper]
Frees: [ACM paper]

Block Bitmap: used(1)⟶free(0)

Long CPs problem:

- with larger bitmaps
- random frees → more dirty bitmap blocks
- worsened by long bitmap "chains"

Long CPs hurt file system write throughput

- too many dirty bufs in memory
- journal gets filled up

4 bitmap blocks of CP_i

A
B
C
D
Block A
Block B
Block C
Block D

bit in A is cleared in CP_{i+1}

A
B
C
D
Block A'
Block B'
Block C'
Block D'

Free Block
Used Block
Used Block in Chain

Solved: ACM paper

SNAPSHOTS

volume = current FS + all snapshots

Bitmap for S_1

A	B	C	D	E	. . .	B’	. . .	E’	. . .

Bitmap for S_0 (current FS)

A	B	C	D	E	. . .	B’	. . .	E’	. . .

Free Block

Used Block

SUMMARY

Modern COW file systems (WAFL, ZFS, BtrFS)

Are log-structured like LFS

But keep a separate journal to guarantee zero data loss

Enterprise-grade storage systems

Can be built using COW file systems

Large checkpoints→efficient sequential writes to RAID4-style storage

Enterprise-grade features:

High performance, reliability, snapshots, clones, compression, deduplication, worm, encryption, tiering, remote replication, tiering-to-cloud, etc.

NEXT

Remote File Access

Client-Server architecture
NFS/SMB

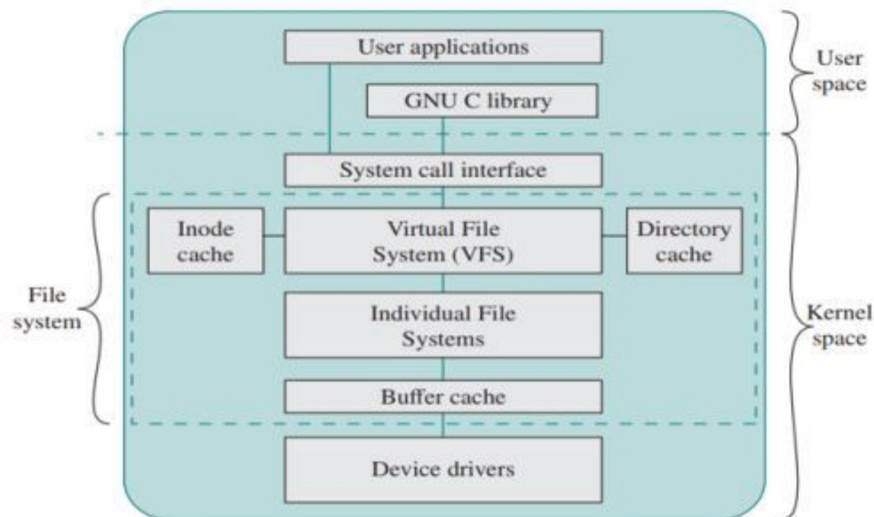
Distributed File Systems

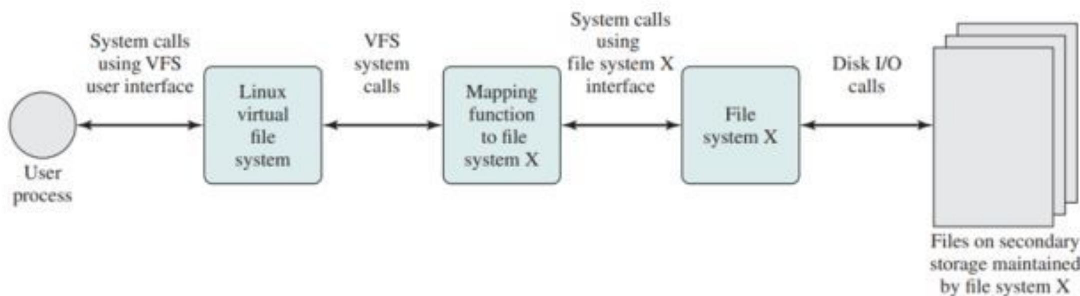
GFS + Colossus

Conflation of these 2 terms:

“remote file access”: many clients → a single server exports a file system
“distributed file systems”: multiple servers collaborate to export a single file system

VIRTUAL FILE SYSTEM LAYER

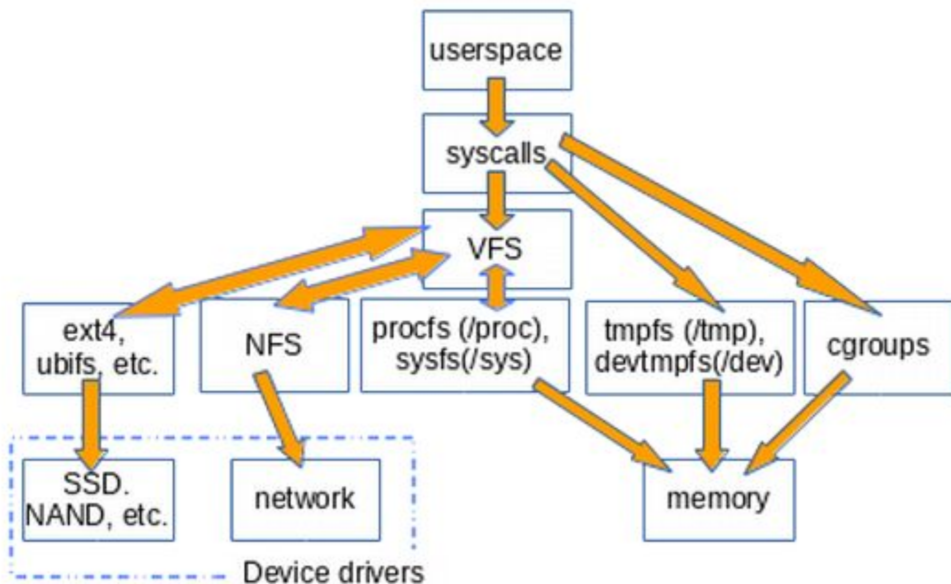




VFS

- used by many OSes
- abstraction layer to different FS types
- VFS translates each system call into the underlying FS's API
- maintains info from all FSs in the caches– inode, dirent, buffer.
- Allows a single machine to mount many file systems (of different types)

VIRTUAL FILE SYSTEM LAYER



Local

- File systems on locally attached HDDs & SSDs
- File systems on local DRAM: do not survive crashes
- pseudo file systems

Remote

- Protocol-specific client that talks with remote file service
- Protocols:
  - NFS: Sun
  - SMB: Microsoft
  - AFS: CMU (legacy)

EVERYTHING IS A FILE

no, there's dirs, pipes, links, files, sockets, and blocks

everything is a file

everything is a file

34% 34%

68%

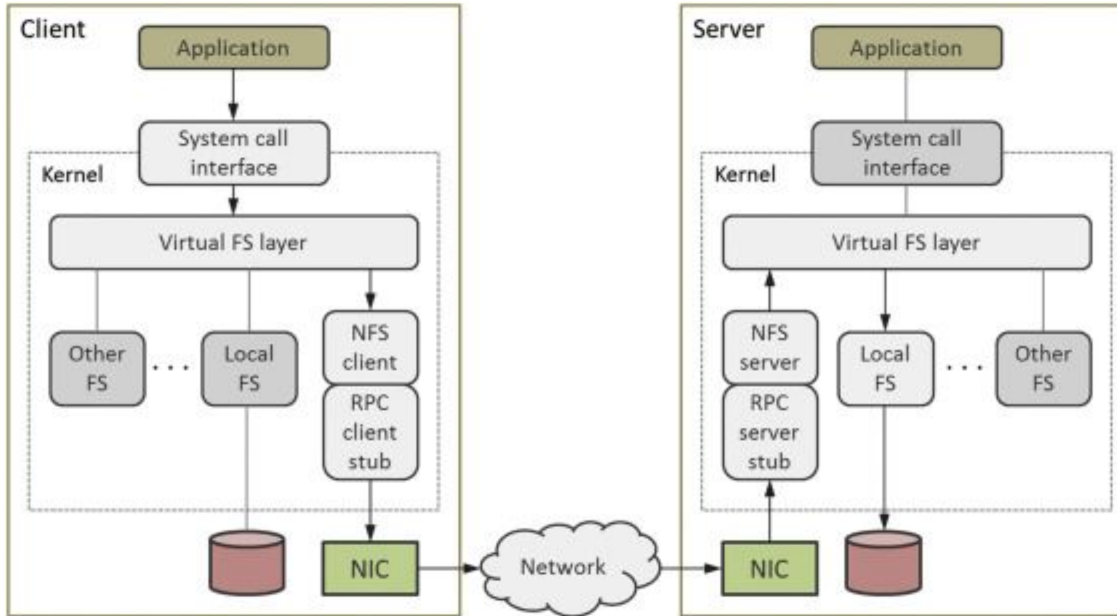
14% 14%

0.1% 2% 2% 0.1%

95%

IQ score 55 70 85 100 115 130 145

NETWORK FILE SYSTEM (NFS)



OSTEP Chapter 48

- Client-server model
- Client allocates a socket and establishes communication over TCP or UDP
- Uses remote procedure call (RPC) abstraction to send & receive messages
- NFS protocol messages

NFS V2 GOAL: SIMPLE CRASH RECOVERY

- Many clients to a single server
- Client or server could crash
- A protocol can be *stateless* or *stateful*
- Stateless: server has no info about clients' states
  - Crash recovery becomes easy*
  - Server simply reboots/restarts
    - Each NFSv2 message must include all requisite info
  - Discuss: `open()` system call semantics
    - fd returned must work until `close()`
    - offset must be tracked

**ignore atomicity of file system updates: journaling handles that*

OPEN SYSCALL: STATELESSNESS

- Client tracks much of it
  - offset, file descriptor mapping, etc.
- open( ): does client need to send NFS message to server?
  - if no, why?
  - if yes, what should server respond with?

OPEN SYSCALL: STATELESSNESS

- Client tracks everything
  - offset, file descriptor mapping, etc.
- open(): does client have to send a NFS message to server?
  - Yes. To confirm file or dirpath actually exists
  - What should server respond with?

```
int fd = open("/foo/bar", O_READONLY);
read(fd, buf, 1024);
                              ← server crash & restart
read(fd, buf, 1024);
```

OPEN SYSCALL: SERVER RESPONSE

- A file identifier that survives server restarts
  - <volume ID, inode#>
  - volume ID: unique file system ID
    - typically persisted in superblock

Client A
1. fdA = open(“foo/bar”, O_READONLY);
// sends NFS messages

Server
2. // inum of “bar” = 501
respond with <v, 501>
3. server crashes & restarts

Client A
4. read(fdA, buf, 4096);
// sends NFS read for <v,501> at offset 0

Server
5. ack // returns 4K of data

Client A
6. write(fdA, buf, 1024);
// sends NFS write for <v,501> at offset 4K

Server
7. ack // writes 1K at offset 4K

OPEN SYSCALL: SERVER RESPONSE

- A file identifier that survives server restarts
- But…

Client A	**Client B**	**Server**
1. fdA = open(“foo/bar”, O_READONLY);		
		2. // inum of “bar” = 501 respond with <v, 501>
	3. unlink(“foo/bar”);	
		4. // inum 501 is now free
	5. fdB = open(“foo/baz”, O_CREAT);	
		6. // 501 is reused for baz respond with <v, 501>
	7. write(fdB, buf, 4096);	
		8. ack
9. read(fdA, buf, 4096);		
		10. returns baz’s data!

INODE GENERATION#

- A file identifier that survives everything
  - <volume ID, inode#, generation#>
  - volume ID: unique file system ID
    - typically persisted in superblock
  - generation#: incremented on each new use of that inode#
- Generation# is a standard systems technique
  - Used to avoid race conditions protocols
  - Ensures that the version of an object has remained unchanged since last viewed.

OPEN SYSCALL: WITH GENERATION#

- A file identifier that survives server restarts
- But…

Client A	**Client B**	**Server**
1. fdA = open(“foo/bar”, O_READONLY);		
		2. // inum of “bar” = 501 respond with <v, 501, g>
	3. unlink(“foo/bar”);	
		4. // inum 501 is now free
	5. fdB = open(“foo/baz”, O_CREAT);	
		6. // 501 is reused for baz respond with <v, 501, g+1>
	7. write(fdB, buf, 4096);	
		8. ack
9. read(fdA, buf, 4096); // NFS read with <v, 501, g>		10. responds with ESTALE