CS 3204
Operating Systems

Lecture 26

Godmar Back

Announcements

- Office Hours moved to one of McB 133, 116, or 124. Last office hours today.
- Please see revised grading policy posted on website
  - Project 3 has been graded (and should have been posted)
  - Will provide standing grade by tomorrow
  - Accept project 4 until May 7, 23:59pm
- You must send email to your advisor by Wednesday 5pm if you want to switch to P/F!
- Reading assignment: Ch 10, 11, 12

Filesystems

Consistency & Logging

FFS's Consistency

- Berkeley FFS (Fast File System) formalized rules for filesystem consistency
- FFS acceptable failures:
  - May lose some data on crash
  - May see someone else's previously deleted data
    - Applications must zero data out if they wish to avoid this + fsync
  - May have to spend time to reconstruct free list
  - May find unattached inodes → lost+found
- Unacceptable failures:
  - After crash, get active access to someone else's data
    - Either by pointing at reused inode or reused blocks
- FFS uses 2 synchronous writes on each metadata operation that creates/destroy inodes or directory entries, e.g., creat(), unlink(), mkdir(), rmdir()
  - Updates proceed at disk speed rather than CPU/memory speed

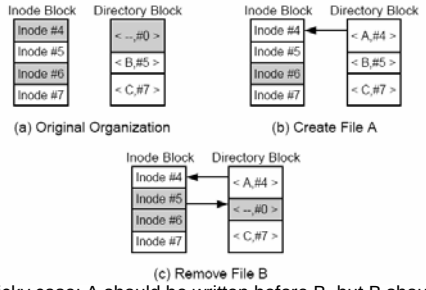
Write Ordering & Logging

- Problem: as disk sizes grew, fsck becomes infeasible
  - Complexity proportional to used portion of disk
  - takes several hours to check GB-sized modern disks
- In the early 90s, approaches were developed that
  - Avoided need for fsck after crash
  - Reduced the need for synchronous writes
- Two classes of approaches:
  - Write-ordering (aka Soft Updates)
    - BSD – the elegant approach
  - Journaling (aka Logging)
    - Used in VxFS, NTFS, JFS, HFS+, ext3, reiserfs

Write Ordering

- Instead of synchronously writing, record dependency in buffer cache
  - On eviction, write out dependent blocks before evicted block: disk will always have a consistent or repairable image
  - Repairs can be done in parallel – don't require delay on system reboot
- Example:
  - Must write block containing new inode before block containing changed directory pointing at inode
- Can completely eliminate need for synchronous writes
- Can do deletes in background after zeroing out directory entry & noting dependency
- Can provide additional consistency guarantees: e.g., make data blocks dependent on metadata blocks

Write Ordering: Cyclic Dependencies



- Tricky case: A should be written before B, but B should be written before A? … must unravel

Logging Filesystems

- Idea from databases: keep track of changes
  - "write-ahead log" or "journaling": modifications are first written to log before they are written to actually changed locations
  - reads bypass log
- After crash, trace through log and
  - redo completed metadata changes (e.g., created an inode & updated directory)
  - undo partially completed metadata changes (e.g., created an inode, but didn't update directory)
- Log must be written to persistent storage

Logging Issues

- How much does logging slow normal operation down?
- Log writes are sequential
  - Can be fast, especially if separate disk is used
  - Subtlety: log actually does not have to be written synchronously, just in-order & before the data to which it refers!
    - Can trade performance for consistency – write log synchronously if strong consistency is desired
- Need to recycle log
  - After "sync()", can restart log since disk is known to be consistent

Physical vs Logical Logging

- What & how should be logged?
- Physical logging:
  - Store physical state that's affected
    - before or after block (or both)
  - Choice: easier to redo (if after) or undo (if before)
- Logical logging:
  - Store operation itself as log entry (rename("a", "b"))
  - More space-efficient, but can be tricky to implement

Summary

- Filesystem consistency is important
- Any filesystem design implies metadata dependency rules
- Designer needs to reason about state of filesystem after crash & avoid unacceptable failures
  - Needs to take worst-case scenario into account – crash after every sector write
- Most current filesystems use logging
  - Various degrees of data/metadata consistency guarantees

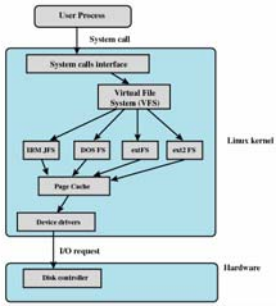
Filesystems

Volume Managers
Linux VFS

Virginia Tech

Example: Linux VFS

- Reality: system must support more than one filesystem at a time
  - Users should not notice a difference unless unavoidable
- Most systems, Linux included, use an object-oriented approach:
  - VFS-Virtual Filesystem



Virginia Tech

Example: Linux VFS Interface

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
        ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct *);
        int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct *);
        int (*open) (struct inode *, struct file *);
        int (*flush) (struct file *);
        int (*release) (struct inode *, struct file *);
        int (*fsync) (struct file *, struct dentry *, int datasync);
        int (*aio_fsync) (struct kiocb *, int datasync);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock *);
        ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
        ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
        ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
        unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
        int (*check_flags)(int);
        int (*dir_notify)(struct file *filp, unsigned long arg);
        int (*flock) (struct file *, int, struct file_lock *);
};
```

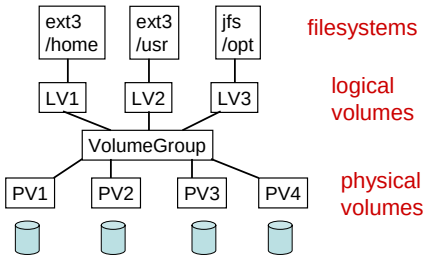
Virginia Tech

Volume Management

- Traditionally, disk is exposed as a block device (linear array of block abstraction)
  - Refinement: disk partitions = subarray within block array
- Filesystem sits on partition
- Problems:
  - Filesystem size limited by disk size
  - Partitions hard to grow & shrink
- Solution: Introduce another layer – the Volume Manager (aka “Logical Volume Manager”)

Virginia Tech

Volume Manager



- Volume Manager separates physical composition of storage devices from logical exposure

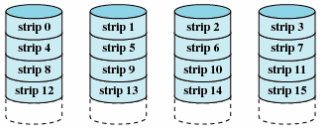
Virginia Tech

RAID – Redundant Arrays of Inexpensive Disks

- Idea born around 1988
- Original observation: it’s cheaper to buy multiple, small disks than single large expensive disk (SLED)
  - SLEDs don’t exist anymore, but multiple disks arranged as a single disk still useful
- Can reduce latency by writing/reading in parallel
- Can increase reliability by exploiting redundancy
  - I in RAID now stands for “independent” disks
- Several arrangements are known, 7 have “standard numbers”
- Can be implemented in hardware/software
- RAID array would appear as single physical volume to LVM

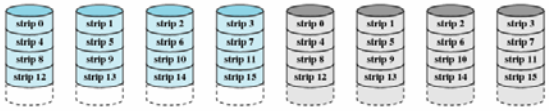
Virginia Tech

RAID 0



- RAID: Striping data across disk
- Advantage: If disk access go to different disk, can read/write in parallel, decrease in latency
- Disadvantage: Decreased reliability (MTTF(Array) = MTTF(Disk)/#disks

RAID 1



- RAID 1: Mirroring (all writes go to both disks)
- Advantages:
  - Redundancy, Reliability – have backup of data
  - Can have better read performance than single disk – why?
  - About same write performance as single disk
- Disadvantage:
  - Inefficient storage use

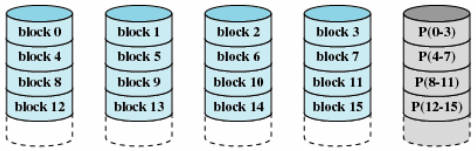
Using XOR for Parity

X Y Z W

XOR	0	1
0	0	1
1	1	0

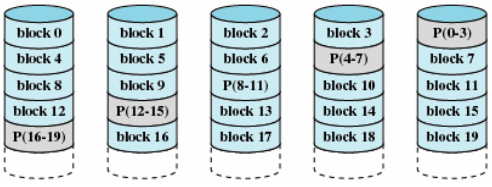
- Recall:
  - X^X = 0
  - X^1 = !X
  - X^0 = X
- Let's set: W=X^Y^Z
  - X^(W)=X^(X^Y^Z)=(X^X)^Y^Z=0^(Y^Z)=Y^Z
  - Y^(X^W)=Y^(Y^Z)=0^Z=Z
- Obtain: Z=X^Y^W (analogously for X, Y)

RAID 4



- RAID 4: Striping + Block-level parity
- Advantage: need only N+1 disks for N-disk capacity & 1 disk redundancy
- Disadvantage: small writes (less than one stripe) may require 2 reads & 2 writes
  - Read old data, read old parity, write new data, compute & write new parity
  - Parity disk can become bottleneck

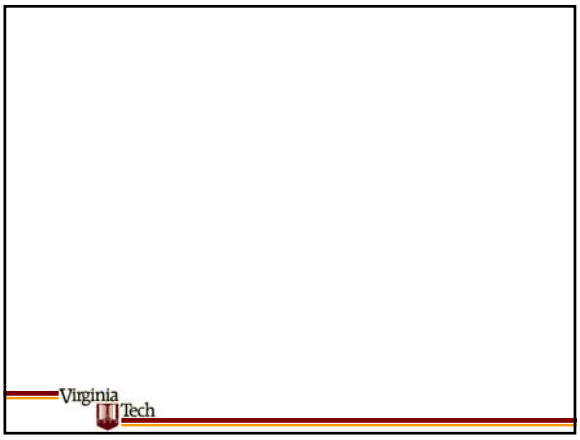
RAID 5



- RAID 5: Striping + Block-level Distributed Parity
- Like RAID 4, but avoids parity disk bottleneck
- Get read latency advantage like RAID 0
- Best large read & large write performance
- Only remaining disadvantage is small writes
  - "small write penalty"

Other RAID Issues

- RAID-6: dual parity, code-based, provides additional redundancy (2 disks may fail before data loss)
- RAID (0+1) and RAID (1+0):
  - Mirroring+striping
- Interaction with filesystem
  - WAFL (write anywhere filesystem layout) avoids in-place updates to avoid small write penalty
  - Based on LFS (log-structured filesystem) idea in which all writes go to new locations


Virginia
Tech