CS 3204
Operating Systems

Lecture 24

Godmar Back

Announcements

- Project 4 due Dec 10 11:59pm
  - Don't postpone to after Thanksgiving
  - Should get done before break: buffer cache (so all regression tests pass), and have designed and partially implemented on-disk data structures (inode + index trees)
- Project 4 Help Session Slides online
- Check exam time and let me know of possible conflicts as per University policy
- Reading Chapter 10-12

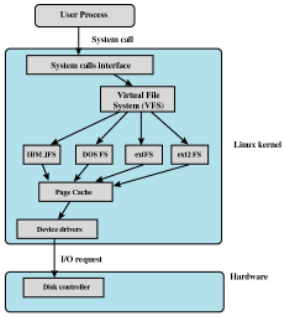
Filesystems

Linux VFS

Volume Managers

Example: Linux VFS

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



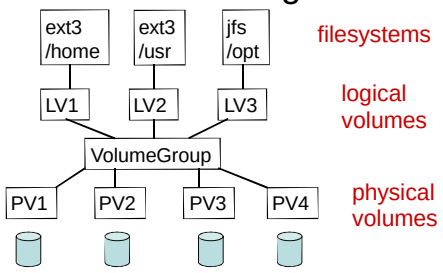
Example: Linux VFS Interface

```
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 *);
};
```


Volume Management

- Traditionally, disk is exposed as a block device (linear array of blocks 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")

Volume Manager

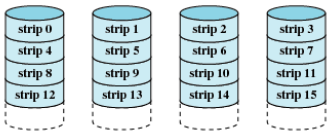


- Volume Manager separates physical composition of storage devices from logical exposure

RAID – Redundant Arrays of Inexpensive Disks

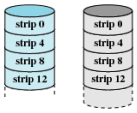
- Idea born around 1988
- Original observation: it's cheaper to buy multiple, small disks than single large expensive disk (SLED)
  - SLEDs don't exist anymore, but multiple disks arranged as a single disk still useful
- Can reduce latency by writing/reading in parallel
- Can increase reliability by 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

RAID 0



- RAID: Striping data across disk
- Advantage: If disk accesses go to different disks, 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
  - Potentially better read performance than single disk – why?
  - About same write performance as single disk
- Disadvantage:
  - Inefficient storage use

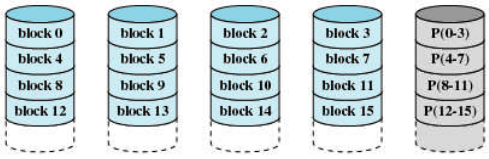
Using XOR for Parity

X Y Z W

XOR	0	1
0	0	1
1	1	0

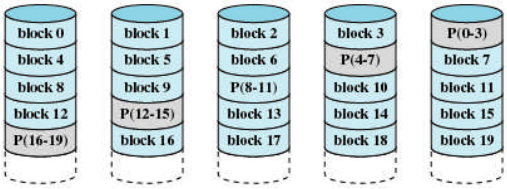
- Recall:
  - X^X = 0
  - X^1 = !X
  - X^0 = X
- Let's set: W=X^Y^Z
  - X^(W)=X^(X^Y^Z)=(X^X)^Y^Z=0^(Y^Z)=Y^Z
  - Y^(X^W)=Y^(Y^Z)=0^Z=Z
- Obtain: Z=X^Y^W (analogously for X, Y)

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

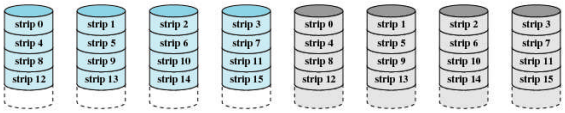
RAID 5



- 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 Combinations

- 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



Filesystems

Advanced Techniques

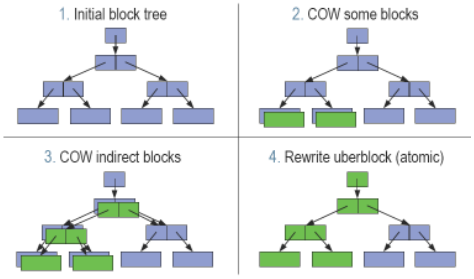
Delayed Block Allocation, Preallocation, and Defragmentation

- Idea: delay block allocation until write back (eviction time)
  - Combine with data structure that simplifies finding continuous sections of free blocks
  - Increases chances for contiguous physical layout of blocks that are likely to be accessed sequentially
- Online defragmentation
  - Some filesystem reallocate blocks to improve spatial locality
- Preallocation
  - Supports guarantee of contiguous space without actually writing

Avoiding in-place updates

- Most traditional designs allocate blocks once and for all (when files is created, grown, etc.)
- All subsequent updates go this location (whether it requires seeks or not – makes writes not sequential)
- Idea:
  - Write wherever there's a free block, write a new version of metadata that points to it – more to write, but sequential (thus faster)
  - What to do with old data
    - Can garbage collect and reclaim
    - Keep around and offer to user as snapshot of past (e.g., NetApp's .snapshot directory)
- Pioneered in LFS (log-structured filesystem), see [Rosenblum 1991]
  - For RAID, avoids small write problem

Example: COW transactions in ZFS



Source: ZFS – The Last Word in Filesystems

End-to-end Data Integrity

- Most current file systems assume no undetected bit errors in storage stack
  - No longer true in practice: disk capacity increases exponentially, error rate does not decrease (1 in 10^{14} to 1 in 10^{15} undetected and uncorrected errors)
- File systems can do end-2-end checksumming to detect corrupted data
  - Either only for metadata (ext4)
  - For all data (ZFS)

Increased Fault Tolerance

- Traditional approach:
  - File system does minimal state replication
    - Maybe superblock, but not file data or meta data
  - Relies on underlying layer: RAID mirroring
- Single bad block on disk may lead to loss of entire disk
  - (in RAID case: silent errors may occur, since first READ is believed)
- ZFS approach: have file system replicate data and metadata in storage pool
  - User decides how many copies

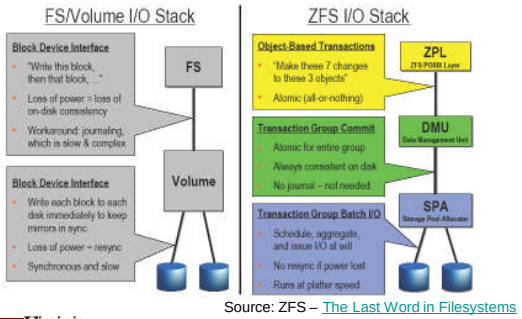
Variable Blocksizes

- Recall trade-off involving block size
  - Too small – low data rate, high metadata overhead
  - Too large – much space lost to internal fragmentation (since many files are small)
- Ideally, block size matches size of write requests done to file ("object size")
  - No internal fragmentation
  - No read/modify/write operations
- ZFS supports this

Metadata Consistency

- Traditional file systems separate designs for metadata (directories and index trees) from designs chosen for metadata consistency
  - Result: need synchronous writes, logging, or write ordering.
  - Consistency often retrofitted (e.g., ext2 to ext3)
  - Cannot make use of atomic updates (which would avoid need for either of these approaches!)
- Alternative: design entire filesystem so that atomic updates become possible

ZFS's Approach



Source: ZFS – The Last Word in Filesystems

Other Developments

- Built-in encryption and compression
- Built-in support for incremental backup
- Built-in support for indexing
- Explicit support for SSD (solid-state drives)
- Support for hybrid drives (or supporting solid state)
  - E.g. Vista Ready Boost – uses solid state to absorb random writes and reads