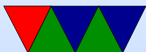
ECE 531 – Advanced Operating Systems
Lecture 19

Vince Weaver
https://web.eece.maine.edu/~vweaver
vincent.weaver@maine.edu

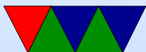
9 November 2023

Announcements

- Discuss HW6 lack of testing harness
- HW6 trouble attaching assignment: usually means there's executable code. make clean apparently wasn't getting rid of all the executables

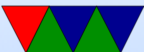


“Traditional” UNIX-style Filesystem



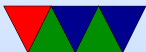
Ext2 FS – History

- Linux originally used "minixfs" but it had 16-bit offsets and max size limit of 64MB and filename of 14 chars
- Replacement: ext, but still limits
- Then ext2 and xiafs
- ext2 by Rémy Card
- later extended to ext3 (with journaling) and ext4
- Still issues with things like y2038 bug



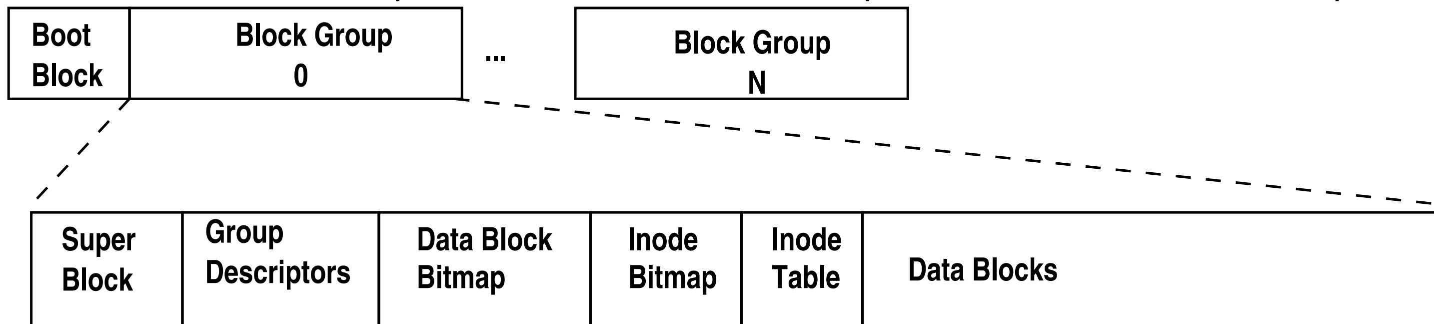
Ext2 FS

- Originally supports 4TB filesystem, 2GB file size
- All structures are little-endian
  Learned hard way not specifying, Atari disk images not work on x86
- Block size 1k-4k (for various reasons it's complicated on Linux to have a block size greater than the page size)
  (also, does blocksize have to be power of 2? Some CD-ROMs had blocksize of 2336 bytes)
- 5% of blocks reserved for root. Why? Still needed?

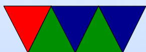


Overall Layout

- Low-level blocks, grouped together in block groups
- Boot sector, boot block 1, boot block 2, boot block 3

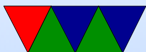


- Block group: superblock, fs descriptor, block bitmap, inode bitmap, inode table, data blocks



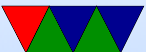
Block Group

- A bitmap for free/allocated blocks
- A bitmap of allocated inodes
- An inode table
- Possibly a backup of the superblock or block descriptor table
- Effort is made to make files be allocated in same block group as their dir entry.



Superblock

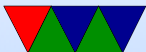
- located at 1k offset, 1k long
- Copies scattered throughout (fewer in later versions)
- Info on all the inode groups, block groups, etc.
- Copy in each block group, but typically only 1st one used



Superblock Layout

Offset	Size	Description
0	4	Number of inodes in fs
4	8	Number of blocks in fs
8	4	Blocks reserved for root
12	4	Unallocated blocks
16	4	Unallocated inodes
20	4	block num of superblock
24	4	block size shift
28	4	fragment size shift
32	4	blocks in each group
36	4	fragments in each group
40	4	inodes per group
44	4	last mount time
48	4	last write time
52	2	mounts since last fsck

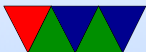
Offset	Size	Description
54	2	mounts between fsck
56	2	ext signature (0xef53)
58	2	fs status (dirty or clean)
60	2	what to do on error
62	2	minor version num
64	4	time of last fsck
68	4	interval between fsck
72	4	OS of creator
76	4	major version number
80	2	uid that can use reserved blocks
82	2	gid that can use reserved blocks
84	4	first non-reserved inode
88	2	size of each inode



Block Group Descriptor Table

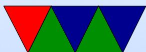
- Follows right after superblock

offset	size	Description
0	4	address of block usage bitmap
4	4	address of inode usage bitmap
8	4	address of inode table
12	2	number of unallocated blocks in group
14	2	number of unallocated inodes in group
16	2	number of directories in group



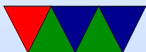
Block Tables

- Block bitmap
  - bitmap of blocks (1 used, 0 available)
  - block group size based on bits in a bitmap.
  - if 4kb, then 32k blocks = 128MB.



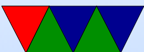
Inode (index-node) Tables

- Inode bitmap – bitmap of available inodes
- Inode table
  - all metadata (except filename) for file stored in inode
  - Second entry in inode table points to root directory
  - inode entries are 128 bytes.
- Can you run out of inodes before you run out of disk?

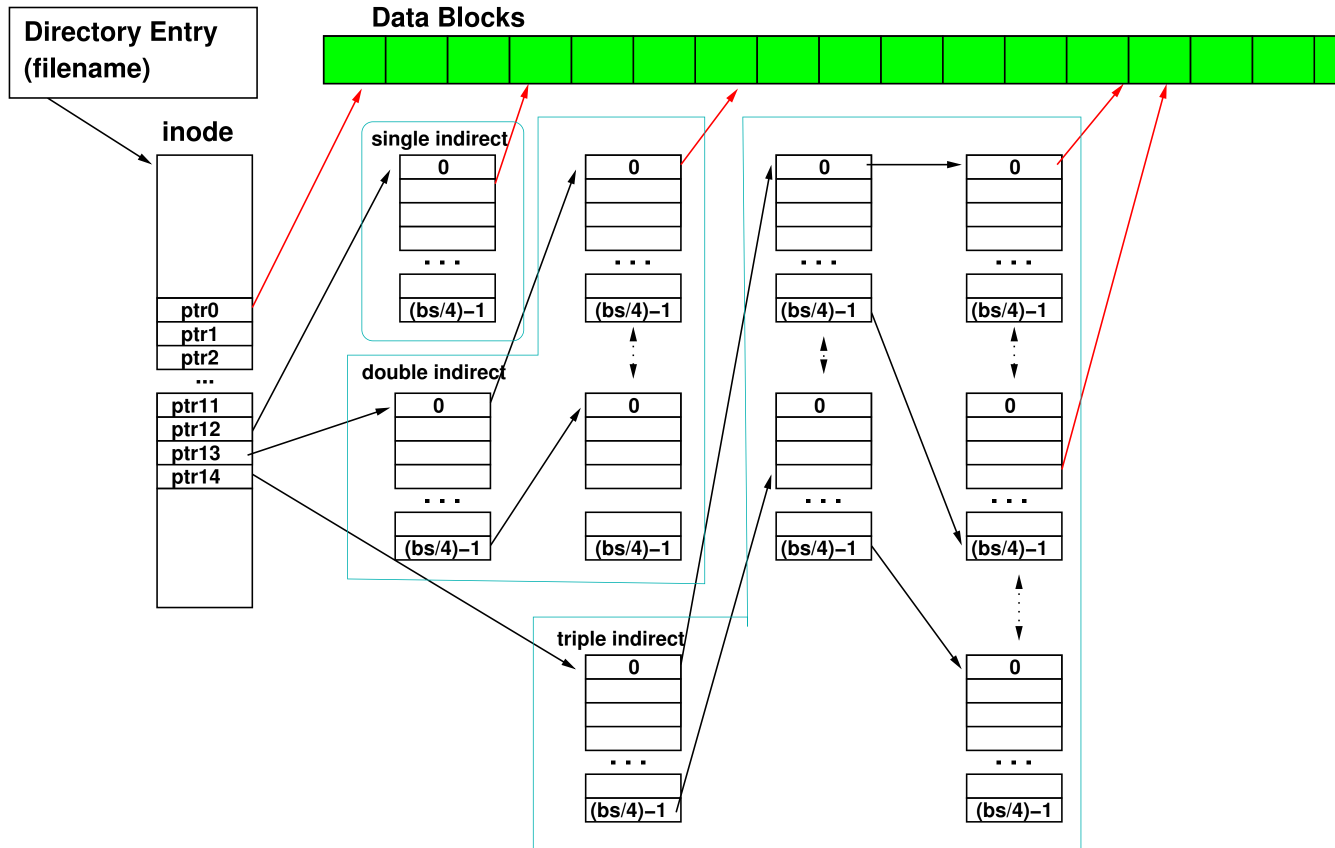


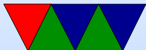
Inode Layout

offset	size	desc
0	2	type and permissions
2	2	userid
4	4	lower 32 bits of size
8	4	last access time (atime)
12	4	creation time (ctime)
16	4	modification time (mtime)
20	4	deletion time
24	2	group id
26	2	count of hard links
28	4	disk sectors used by file?
32	4	flags
36	4	os specific
40 - 84		direct pointers 0 - 11
88	4	single indirect pointer
92	4	double indirect pointer
96	4	triple indirect pointer
100	4	generation number (NFS)
104	4	extended ACL
108	4	ACL (directory) else top of filesize
112	4	address of fragment



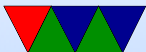
Inode Finding Blocks





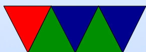
Directory Info

- Superblock links to root directory, (usually inode 2)
- Directory inode has info/permissions/etc just like a file
- The block pointers point to blocks with directory info.
- Initial implementation was single linked list. ext3 and newer use hash or tree.
- Holds inode, and name (up to 256 chars). inode 0 means unused.



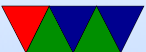
type	size
inode of file	4
size of entry	2
length of name	1
file type	1
file name	N

- Hard links – multiple directory entries can point to same inode
- `.` and `..` entries, point to inode of directory entry
- Subdirectory entries have name, and inode of directory



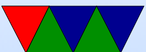
How to find a file

- Find root directory
- Iterate down subdirectories
- Get inode



How to read an inode

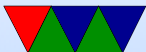
- Get blocksize, blocks per group, inodes per group, and starting address of first group from the superblock
- Determine which block group the inode belongs to
- Read the group descriptor for that block group
- Extract location of the inode table
- Determine index of inode in table
- Use the inode block pointers to read file



Ext3

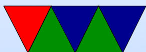
- Compatible with ext2
- Htree instead of linked list in directory search
- online fs growth
- Journal

  metadata and data written to journal before commit. Can be replayed in case of system crash.

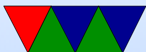


Ext4

- Filesize up to 1Exabyte, filesize 16TB
- Extents (Rather than blocks) , an extent can map up to 128MB of contiguous space in one entry
- Pre-allocate space, without having to fill with zeros (which is slow)
- Delayed allocation – only allocate space on flush, so data more likely to be contiguous
- Unlimited subdirectories (32k on ext3 and earlier)
- Checksums on journals

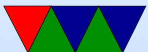


- Improved timestamps, nanosecond resolution, push beyond 2038 limit

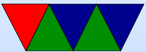


Why use FAT over ext2?

- FAT simpler, easy to code
- FAT supported on all major OSes
- ext2 faster, more robust filename and permissions

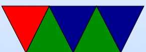


Advanced Filesystems



btrfs

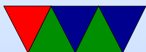
- Butter-fs? Butter-fuss? B-tree fs?
- Started in 2007 at Oracle (by Chris Mason, who had worked on Reiserfs)
- Address scaling
- Lack of pooling, snapshots, checksums in Linux
  - Pooling – preallocate resources so they can be quickly handed out when needed
  - Snapshots – instead of taking full backup (long) just take a snapshot of current state and then keep using



filesystem

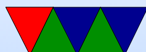
  - Checksums – mathematically check to make sure values in files are what they should be

- $2^{64} = 16$ Exabyte file size limit (Linux VFS limits you to 8EB)
- Space-efficient packing small files
- Dynamic inode allocation



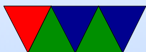
btrfs details

- Primary data structure is a copy-on-write B-tree
  - B-tree similar to a binary tree, but with pages full of leaves
    allow searches in logarithmic time
  - Btrees also used by ext4, NTFS, HFS+
  - Goal is to be able to quickly find disk block X
  - Copy-on-write when writing to file, rather than overwrite (which is what traditional filesystems do)
  - Copy on write. When write to a file, old data not

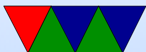


overwritten. Since old data not over-written, crash recovery better
Eventually old data garbage collected

- Data in extents
- Copy-on-write
- Forest of trees:
  - sub-volumes
  - extent-allocation
  - checksum tree
  - chunk device
  - reloc

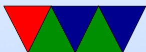


- On-line defragmentation
- On-line volume growth
- Built-in RAID
- Transparent compression
- Snapshots
- Checksums on data and meta-data, on-line data scrubbing
- De-duplication
- Cloning, reflinks
  - can make an exact snapshot of file, copy-on-write
  - different inodes, initially point to same blocks



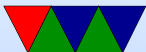
- different from hardlink (different dir entry, point to same inode)

- In-place conversion from ext3/ext4
- Superblock mirrors – at 64k, 64MB,256GB, and 1PB. All updated at same time. Has generation number. Newest one is used.

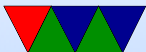


ZFS

- Advanced OS from Sun/Oracle
- 128-bit filesystem (opposed to btrfs which is 64-bit)
  Running out of space would require 10^{24} 3TB hard drives
- Not really included in Linux due to licensing issues (CDDL vs GPL2)
  Was originally proprietary, then open source, then proprietary again (with open fork)
- Vaguely similar in idea to btrfs
- indirect still, not extent based?

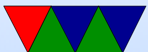


- Acts as both the filesystem *and* the volume manager (RAID array)
- Aim is to be super reliable, to know the state of underlying disks, make sure files stay valid, drives stay healthy
- Can take snapshots. Can roll back if something goes wrong.
- Checksums. Stored in parent. Other fs stores with file metadata so if that lost then checksum also lost
- Limitations: needs lots of RAM and lots of free disk space (due to copies and snapshots). If less than 80%



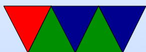
free then goes to space-conserve mode rather than high-performance

- Supports encryption (btrfs doesn't yet)



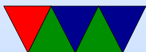
ReFS

- Resilient FS, codename “Protogon”
- Microsoft’s answer to btrfs and zfs
- Windows 8.1
- Initially removed features such as disk quotas, alt data streams, extended attributes (added later?)
- Uses B+ trees (not same as b-trees), similar to relational database
- All structures 64-bit
- Windows cannot be booted from ReFS



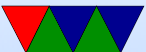
APFS

- New Apple OS for High Sierra and later, iOS 10.3 later
- Fix core problems of HFS+
- Optimized for solid-state drive, encryption
- 64-bit inode numbers
- checksums
- Crash protection: instead of overwriting metadata, creates new metadata, points to it, and only then removes old
- No hard-links to directories (most other OSes are like



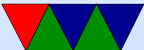
this) but this breaks “Time Machine” backup

- HighSierra auto-converts flash-based drives



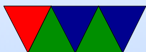
Networked File Systems

- Allow a centralized file server to export a filesystem to multiple clients.
- Provide file level access, not just raw blocks (NBD)
- Clustered filesystems also exist, where multiple servers work in conjunction.

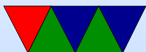


NFS – Network File System (NFS2/3/4)

- Developed by Sun in the 80s.
- Stateless. Means server and client can reboot without the other noticing.
- A server, nfsd, exports filesystems as described in `/etc/exports`. The server can be in userspace or in the kernel
- Needs some sort of "file handle" unique value to specify value. Often cheat and use inode value. Problem with older version of protocol with only 32-bit handles.

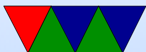


- UDP vs TDP
- Read-ahead can help performance
- Cache consistency a problem. One way is to just have timeouts that flush data regularly (3-30s)
- List of operations (sort of like syscalls) sent to server read sends a packet with file-handle, offset, and length No open syscall; server has no list of open files. This way there is no state needed, can handle reboots.
- nfsroot



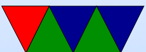
CIFS/SMB

- Windows file sharing.
- Poorly documented
- Samba reimplements it, originally reverse-engineered.

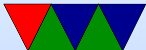


Virtual/Pseudo Filesystems

- Files do not exist on disk; they are virtual, fake files that the kernel creates dynamically in memory
- proc
- sys
- debugfs
- usbfs

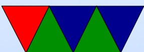


Distributed / Cluster Filesystems

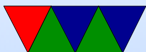


procfs

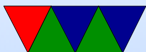
- Originally process filesystem. Each process gets a directory (named by the process id (pid)) under /proc Tools like `top` and `ps` use this info.
  - cmdline
  - cwd
  - environ
  - exe
  - fd
  - maps



- Eventually other arbitrary files were also included under proc, providing system information
  - cpuinfo
  - meminfo
  - interrupts
  - mounts
  - filesystems
  - uptime
- ABI issues – these files are part of the kernel, and even though the intention was that they could come and go at will, enough people write programs that depend

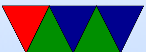


on them, the values cannot be easily changed without breaking the ABI

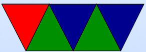


sysfs

- procfs was getting too cluttered, so sysfs was created
- intended to provide tree with information on devices
- one-item per file and strict documentation rule
- also hoped that it would replace sysctl() and ioctl() but that hasn't happened

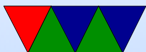


Other Filesystem Features



Sparse Files / Holes

- What if your file has lots of zeros?
- What if you seek way into a file (to write something at end)
- Do you need to allocate zeros on disk for these?
- Many filesystems support holes, where the inode list says a file has a zero, only allocates disk block if you write in this range
- Can save a lot of disk space



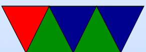
More Features

- Compression – transparently compress files. Does have some performance issues, write issues (do you have to decompress, write, then recompress?) and also files rarely compress to nice power-of-two sizes.

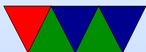
- Online fsck

- Defragmentation

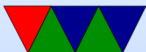
- Undelete



- Secure Delete
- Snapshots
- Journaling
- De-dup
- Quotas – especially an issue on multi-user machines, you want to keep any one user from filling up the disk.
- Encryption

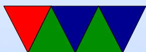


- Locking – may want to prevent more than one person writing a file at a time as it can get corrupted



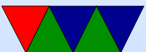
Linux VFS

- VFS interface - VFS / Virtual Filesystem / Virtual Filesystem Switch
- Makes all filesystems look like Linux filesystems. Might need hacks; i.e. for FAT have to fake a superblock, directory entries, and inodes (generate on the fly). Can be important having consistent inode numbers as filesystems like NFS use them even across reboots.
- Objects
  - superblock



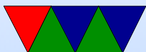
- ◦ inode object (corresponds to file on disk)
- ◦ file object – info on an open file (only exists in memory)
- ◦ dentry object – directory entry.

- Can use default versions, such as default_llseek
- dentries are cached. As they get older they are freed.
- dentry operations tale. hash. compare (how you handle case sensitive filesystems)



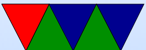
Linux Filesystem Interface

- linux/fs.h
- Module. Entry point init_romfs_fs(), exit_romfs_fs()
  - init_romfs_fs() – register_filesystem()
    name, romfs_mount, romfs_kill_sb
  - romfs_mount – mount_bdev(), romfs_fill_super
  - sb− >s_op=&romfs_super_ops();
  - romfs_iget() − > i_op struct, gets pointed to in each inode



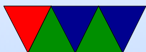
mounting

- Opens superblock
- Inserts into linked list of opened filesystems



pathname lookup

- If begins with /, starts with current->fs->root
- otherwise, relative path, starts with current->fs->path
- looks up inode for starting directory, then traverses until it gets to the one wanted
- the dentry cache caches directory entries so the above can happen without having to do any disk reads if the directory was used recently before

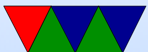


- the access rights of intervening directories must be checked (execute, etc)

- symbolic links can be involved

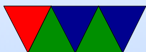
- you might enter a different filesystem

- Should you cache invalid file lookups?

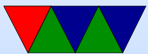


open syscall

- `getname()` – safely copies name we want to open from userspace process
- get_unused_fd() to get the file descriptor
- calls filp_open()
  - creates new file structure
  - open_namei() – checks dentry cache first, otherwise hits disk and looks up dentry
  - lookup_dentry()
- validates and sets up the file



- returns a fd



FUSE

- Allows creating filesystem drivers in userspace
- Works on various OSes