A Brief History of the BSD Fast Filesystem

Brought to you by

Dr. Marshall Kirk McKusick

EuroBSD Conference
Copenhagen, Denmark
15th September 2007

1979 – Early Filesystem Work

- Improved reliability
  - staged modifications to critical filesystem information
  - modifications could be either completed or repaired cleanly by **fsck** after a crash
- Increased the block size of the filesystem from 512 to 1K bytes
  - doubled performance because each disk transfer accessed twice as much data
  - eliminated the need for indirect blocks for many files
  - still utilized only about 4% of disk bandwidth

1982 – Birth of the Fast Filesystem

- Designed with a hybrid blocksize in which large blocks could be broken up into as many as eight fragments
- Large files used large blocks
- Small files could use as little as a single fragment
- First deployed with default blocksize 4K/512
- Still in use today on systems such as Solaris and Darwin

1986 – Dropping Disk-geometry Calculations

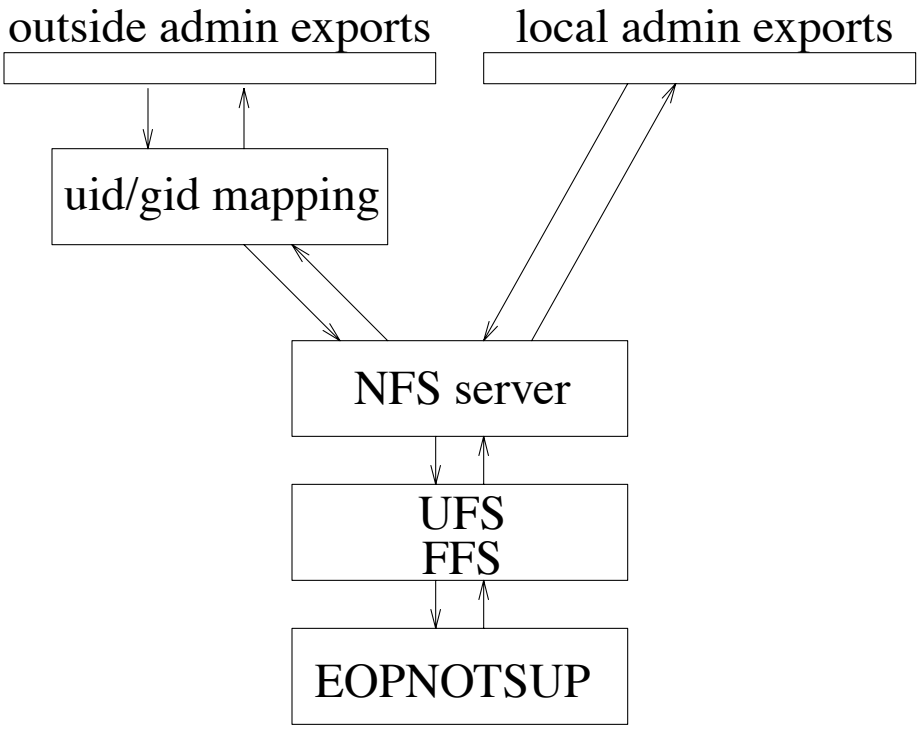
- Originally a cylinder group comprised one or more consecutive cylinders on a disk
- The filesystem could get an accurate view of the disk geometry and could compute the rotational location of every sector
- By 1986, disks were hiding this information and it was too complex to compute it
- All the rotational layout code was deprecated in favor of laying out files using numerically close block numbers (sequential being viewed as optimal)
- Cylinder group structure was retained only as a convenient way to manage logically-close groups of blocks

1987 – Filesystem Stacking

- From John Heidemann at The University of California at Los Angeles
- Based on Dave Rosenthal's original idea (formerly of Sun Microsystems)
- Filesystems easily widened:
  - Adding new VOP's, for example VOP_STARTTRANS and VOP_ENDTRANS to add transactions
  - Other filesystems need not know about or respond to new VOP's (kernel will automatically return EOPNOTSUPP)
- Filesystems easily stacked:
  - Umap filesystem for NFS
  - Loopback filesystem

Stacking Mounts

- Allows filesystem modules to be stacked
- When a request is not implemented by a layer it is passed down to the next lower layer.
- Requests that reach the bottom of the stack without being serviced return with EOPNOTSUPP
- Requests may be modified and then passed on to a lower layer



Loopback Mounts

Allow arbitrary directories in the filesystem to be mounted anywhere else

Implemented as a filesystem layer

- Original filesystem has a layer inserted above it
- This upper layer is mounted at new mount point
- Lookups through this mount point are redirected to the starting point in the original filesystem

Union Mounts

- Allows multiple mounted filesystems to be simultaneously accessible from the same mount point
- All filesystems except the topmost one are treated as if they were mounted read-only
- Descent into a directory that exists in a lower layer filesystem causes creation of the corresponding directory in the top layer filesystem

Union Mount Naming

- Directory listing shows the sum of all files in all directories involved in union mount
- If the same name appears in multiple union mounted directories, only the object from the topmost filesystem in which the name appears is accessible
- New files are created in topmost mounted filesystem
- Overwriting of existing files in a lower layer causes a new writable copy to be created in the topmost layer
- Last filesystem mounted is the first filesystem unmounted

Union Mount Examples

union mount /a on /mnt with files /a/x, /a/y, /a/z

union mount /b on /mnt with files /b/v, /b/w, /b/x

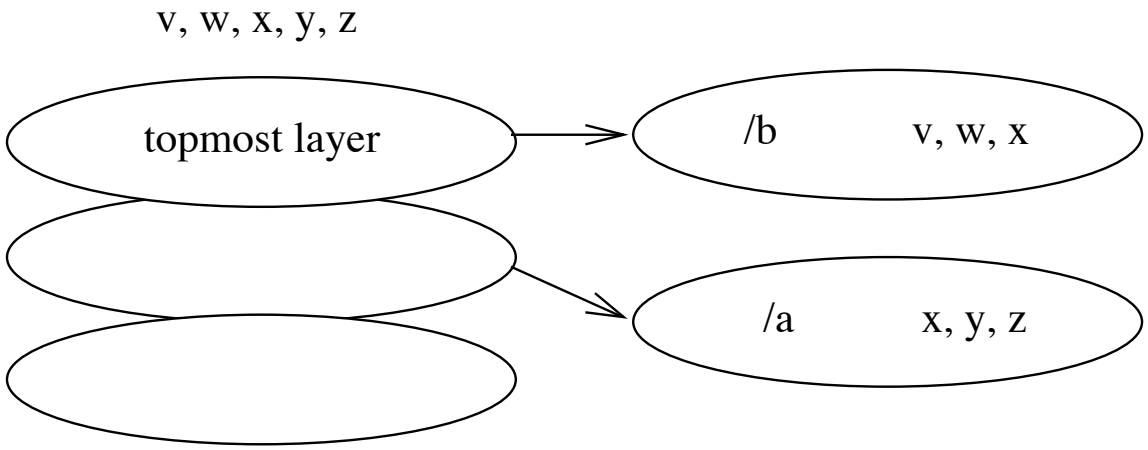
ls /mnt => v, w, x, y, z

File x is from /b

Creat t appears in /b/t

Open y for reading operates on /a/y

Open y for writing copies /a/y to /b/y,
then writes file /b/y

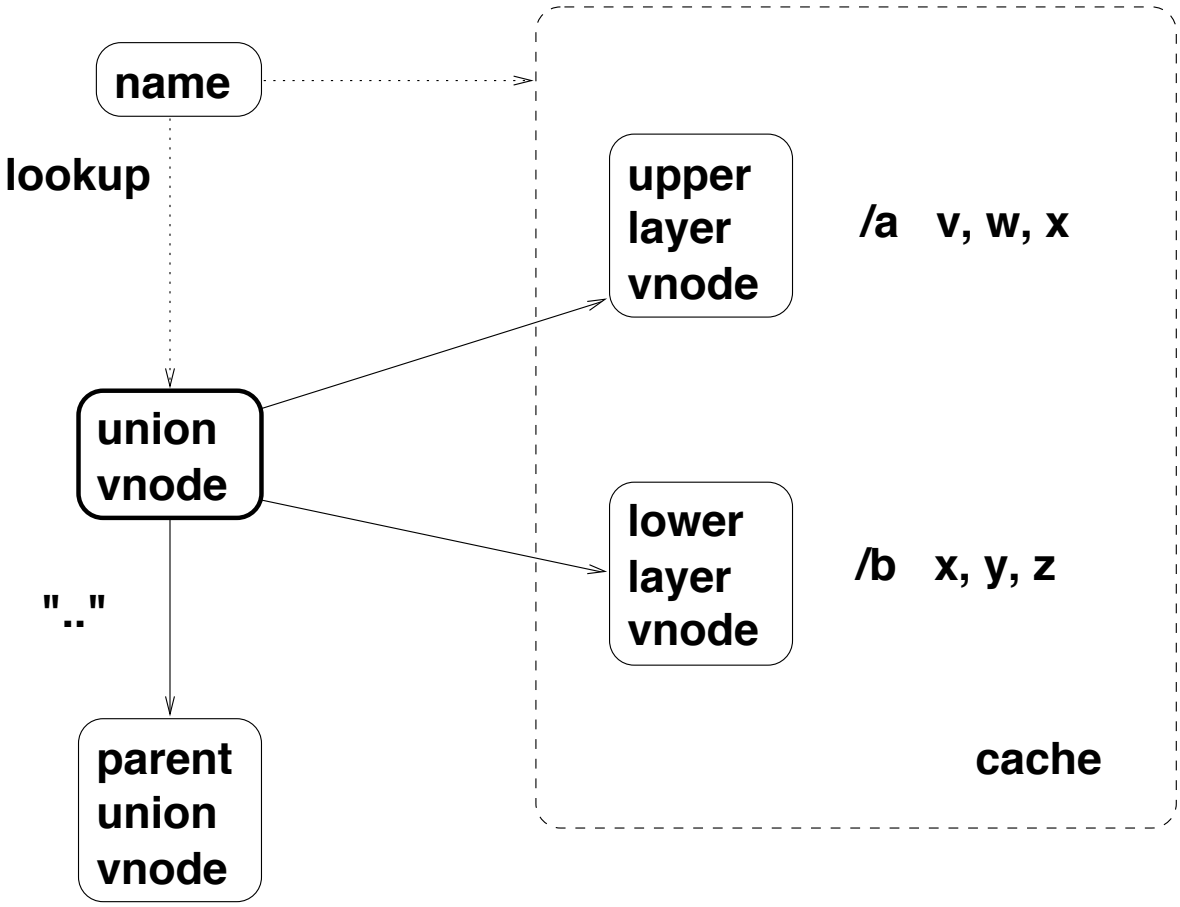


Union Mount Issues

- File removal in lower layer done using whiteout in top layer
- When creating a directory with the same name as one in a lower layer, it must be marked opaque
- Duplicate suppression is handled in the C-library
- Implication of allowing non-root users to do their own mounts

Implementation

- Built using stackable vnode framework
- Union layer handles namespace operations; all others are passed to the lower layers



1988 – Raising the Blocksize

- Default blocksize raised to 8K/1K
- Small files use a minimum of two disk sectors
- Nearly doubled throughput at a cost of only 1.4% additional wasted disk space

1990 – Dynamic Block Reallocation

- With the advent of disk caches and tag queueing it became desirable to begin laying files out contiguously

- Size of file unknown when first opened

  - If always assume big and place in biggest available space, then soon have only small areas of contiguous space available

  - If always assume small and place in areas of fragmented space, then beginning of large files will be poorly laid out

Implementation of Dynamic Block Reallocation

- Dynamic block reallocation places file in small areas of free space, then moves them to larger areas of free space if it grows
  - small files use the small chunks of free space
  - large files get laid out contiguously in the large areas of free space
- Little increase in I/O load as the buffer cache generally holds the file until its final location is known
- Free space remains largely unfragmented even after years of use (15% versus 40% degredation after three years)

1996 – Soft Updates

- Metadata that must be maintained
  - directories
  - inodes
  - bitmaps
- Rules
  1) Never point to a structure before it is initialized
  2) Never reuse a resource before nullifying all previous pointers to it
  3) Never reset an old pointer to a live resource before the new pointer has been set

Keeping Metadata Consistent 1

- Synchronous writes
  - Benefits: simple and effective
  - Drawbacks: create/delete intensive applications run slowly, slow recovery after a crash
- Non-Volatile RAM
  - Benefits: usually runs all operations at memory speed, quick recovery after a crash
  - Drawbacks: expensive hardware, somewhat complex recovery
- Atomic Updates (logging)
  - Benefits: create/remove do not slow down under under heavy load, quick recovery after a crash
  - Drawbacks: extra I/O generated, little speed-up for light loads

Keeping Metadata Consistent 2

- Partial ordering of buffer writes
  - Benefits: 25% reduction in synchronous writes
  - Drawbacks: still disk limited for create/delete intensive applications, slow recovery after a crash
- Soft updates
  - Benefits: most operations run at memory speed, reduced system I/O, instant recovery after a crash
  - Drawbacks: complex code and increased memory loading

Tracking File Removal Dependencies

Ordering constraints

1) Name in on-disk directory must be deleted
2) Deallocate (zero out) on-disk inode
3) Release file's blocks to free-space bitmap

How soft updates maintains this ordering

1) Zero out directory entry in kernel buffer and hang a dependency structure on buffer to be notified when buffer is written.
2) When notified that directory buffer is written, save list of inode's blocks, then zero out inode in kernel buffer and hang a dependency structure (containing the list of blocks) on buffer to be notified when buffer is written.
3) When notified that inode buffer is written, release list of saved blocks to free-space bitmap.

1999 – Snapshots

- Create a copy-on-write image of a filesystem partition

    1) Suspend processes initiating system calls that modify the filesystem

    2) Allow all modifications in progress to complete

    3) Write out all dirty buffers to disk

    4) Create an empty "snapshot" file the size of the filesystem partition

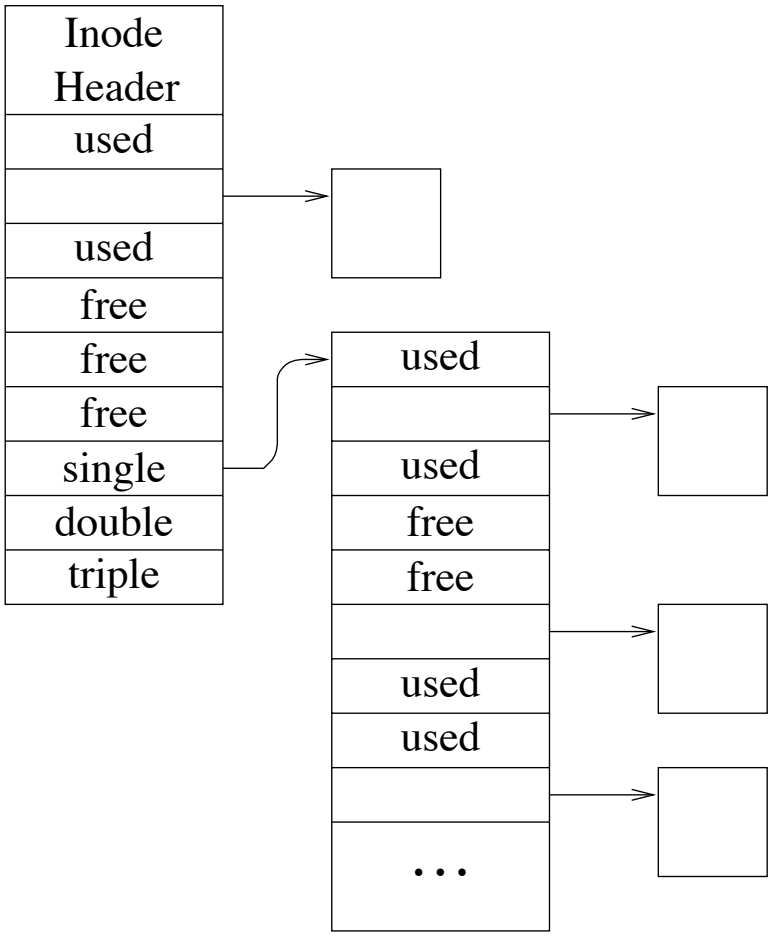
    5) Mark the blocks that are currently in use

    6) Resume write operations on the filesystem

    7) On each disk write, check to see if it has been copied making a copy if the write is for an in-use block that has not yet been copied

Snapshot Implementation

- Each Inode block pointer represents a disk block
- Copied blocks point to the location of the copied block
- Those marked “used” will read the underlying block, but cause a copy to be created if written
- Those marked “free” will read or write the underlying block



2001 – Raising the Blocksize, Again

- Default blocksize raised to 16K/2K
    - Small files use a minimum of four disk sectors
    - Nearly doubled throughput at a cost of only 2.9% additional wasted disk space

2002 – Background Fsck

- Disk state is always valid but behind in-memory state
- Only inconsistencies:
  - Blocks marked in use that are free
  - Inodes marked in use that are free
- It is safe to run immediately after a crash though eventually lost space must be reclaimed

Background Block Recovery

- Block recovery on an active system:

  1) Snapshot the filesystem

  2) Run standard filesystem check program on the snapshot

  3) Add a system call to add lost blocks and inodes to the filesystem map

Other Uses for Snapshots

- Live dumps

  1) Snapshot the filesystem

  2) Run standard dump on the snapshot

- Mid-day backups

  1) Snapshot the filesystem every two hours

  2) Mount each snapshot in a well known location

  3) Users can recover files from earlier in the day by copying them out of the snapshot

2003 – Multi-terabyte support

- Original fast filesystem used 32-bit pointers to reference a file's blocks
- The 32-bit block pointers of the original filesystem run out of space in the 1 to 4 terabyte range
- Considered other alternatives but chose to extend the original filesystem
  - Allowed reuse of most of existing code base which allowed quick development and deployment
  - Became stable and reliable rapidly
  - Same code base supported both 32-bit block and 64-bit block filesystem formats so bug fixes and feature or performance enhancements usually applied to both filesystem formats

Extended Attributes

- Extended attributes added at the same time as multi-terabyte support
- Extended attributes are a piece of auxiliary data storage associated with an inode that can be used to store auxiliary data that is separate from the contents of the file
- By integrating the extended attributes into the inode itself, **fsync**( ) can provide the same integrity guarantees as are made for the contents of the file itself

2004 – Access-control Lists

- Extended attributes were first used to support an access control list (ACL)
  - specific list of the users that are permitted to access the file
  - a list of the permissions that each user is granted

Implementation of Access-control Lists

- Replaced an earlier implementation using a single auxiliary file per filesystem indexed by inode number which had two problems:
  - fixed size of the space per inode meant only short user lists
  - difficult to atomically commit changes to the ACL
- Both problems fixed by using extended attributes:
  - extended attribute can be 32K, so long list of users possible
  - atomic update is easy since it can be updated with one write of inode

2005 – Mandatory-access Controls

- Extended attributes next used for mandatory access control (MAC)
- MAC framework permits dynamically introduced system-security modules to modify system security functionality
  - MAC framework provides control over kernel entry points affecting access control and object creation
  - When hit, MAC framework then calls out to security modules to offer them the opportunity to modify security behavior
- Filesystem does not codify how the labels are used or enforced; it just stores the labels associated and produces them when a security modules needs to do a permission check

2006 – Symmetric Multi-processing

- In the late 1990's, the FreeBSD Project began the long hard task of converting their kernel to support symmetric multi-processing
- Start with giant lock around kernel
- Piece-by-piece add multi-threaded locking and remove from giant lock

    2004 – Vnode interface

    2005 – Disk subsystem

    2006 – Fast filesystem

The End

May the Source Be With You!