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 FreeBSD and Solaris
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

• 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

outside admin exports

local admin exports

uid/gid mapping

NFS server

UFS
FFS

EOPNOTSUPP
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 file 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% degradation after three years)
1996 – Soft Updates

• Speed up file and directory creation, deletion, and renaming

• Keep filesystem consistent enough that `fsck` need not be run after a system crash

• Ensure that unwritten data blocks never show up in files

• Minimize need to do synchronous disk writes
1999 – Snapshots

- Create a read-only frozen-in-time copy of a filesystem
- Minimize time that the filesystem is unavailable while taking the snapshot
- Minimize amount of disk space overhead to hold the snapshot
- Allow multiple snapshots to be concurrently maintained
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
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
2003 – 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, \texttt{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 and groups that are permitted to access the file

- a list of the permissions that each user or group 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
Only need to journal operations that orphan resources

Journal needs only 16Mb independent of filesystem size

Filesystem operations that require journaling

- free operations in maps tracking blocks and inodes
- Link count changes
- Unlink while referenced
2011 – Raising the Blocksize, Yet Again

- Default blocksize raised to 32K/4K

- Driven by the change of disk technology to 4K sectors

- Small files once again use a minimum of one disk sector

- Nearly doubled throughput with no additional wasted disk space
Questions

mckusick@mckusick.com

http://www.mckusick.com

May the Source Be With You!