Contents of Lecture 7 on File Systems

- More about file systems

File System Implementations

- A traditional UNIX file system: s5fs
- Berkeley fast file system: FFS

The System V File System: s5fs

- A partition contains a complete file system and is an array of disk blocks.
- A disk block is either 512, 1024, or 2048 bytes in s5fs.
- The first parts of a partition contain a boot area and a superblock with meta information about the file system
- Then follows an array of inodes
- Then follows all data blocks of the files including directories
- A s5fs directory had 14 character file names and 2 byte inode numbers.

Inodes

- Two different types: on-disk inodes and in-core inodes.

`di_mode`	2	type and permissions
`di_nlinks`	2	ref count
`di_uid`	2	owner UID
`di_gid`	2	owner GID
`di_size`	4	bytes
`di_addr`	39	13 3-byte block numbers
`di_gen`	1	generation
`di_atime`	4	access time
`di_mtime`	4	modification time
`di_ctime`	4	inode change time except atime and mtime

Finding the data blocks from an inode

- There are different approaches including a linked list (cached in memory) in FAT in MS-DOS
- In UNIX an array of 13 blocks are used
- The 10 first elements directly refer to data block numbers
- The next refers to a block with data block numbers — called indirect block
- The next refers to a block with block numbers with data block numbers — double indirect.
- The next refers to a block with block numbers with block numbers with data block numbers — triple indirect.

File holes

- A UNIX file may contain holes due to the process issued an lseek
- No data blocks are allocated for holes
- Reading the hole returns zeroes.
- Backup programs which work at the file level (and not disk level) will not be aware of the hole and write zeroes.

The s5fs super block

The super block contains

- size in blocks of the file system
- size in blocks of the inode array
- number of free blocks and inodes
- partial list of free inodes
  - An inode with `di_mode == 0` is free.
  - When the partial list becomes empty the array of inodes is scanned to find more free inodes.
- partial list of free blocks: the first part of the list is in the superblock and the remaining in other blocks — it's not possible to inspect a block to see if it's in use or not.

Analysis of s5fs

- Poor reliability due to single copy of the super block
- Poor performance due to accesses of a file which need both the inode (may be cached in memory) and the data blocks requires long seeks since the inodes and their data are stored in different places.
- Poor performance due to related inodes (eg in the same directory) are not allocated near each other.
- Poor performance due to the data blocks of a file may be anywhere on the disk and can cause long seeks.
- Serious restrictions in file name size and only 65535 files per file system.

Disk Layout Optimization in Berkeley FFS

- A partition consists of a number of consecutive cylinders.
- FFS divides a partition into a number of **cylinder groups**.
- A cylinder group contains both the inode and the data blocks of a file which reduces the time waste on seeks.
- The super block is split in two parts with one readonly part describing file system parameters and one part describing the status of a cylinder group.
- The readonly part is duplicated in each cylinder group for improved reliability.

FFS Block Sizes and Fragments

- For improved I/O throughput larger disk blocks are possible, often up to 8196 bytes.
- Since each file in average wastes a half disk block and there are numerous small files on UNIX systems this can be a waste.
- The block size is therefore a parameter of the file system so disk can have different block sizes.
- Further, a disk block is divided into a number of **fragments**.
- It is the fragments which are addressed and allocated but the kernel tries to use all fragments in a block to one file.
- Applications can help the kernel by using large buffers and full buffering when writing to files (see `setvbuf` from first lecture).

FFS Allocation Optimizations

- Try to allocate both inode and data blocks of a file in one cylinder group.
- Try to allocate all files in a directory in one cylinder group.
- To do so, use a different cylinder group for a newly created directory.
- When a file grows too much in a cylinder group it is moved to a different cylinder group.
- New blocks of a file are allocated to reduce rotational wait.

FFS Functional Improvements

- Long file names: up to 255 characters.
- Symbolic links.
- New system call **rename** which was previously done with **link** and then **unlink**.
- Quota system.

Analysis of Berkeley FFS

- Read throughput increased from 29KB/s on s5fs to 221 KB/s on FFS.
- Write throughput increased from 48KB/s on s5fs to 142 KB/s on FFS.
- Modern disks eg SCSI and later have varying number of sectors per track which BSD is unaware of and thus some optimizations to reduce rotational delay are in vain.
- The grouping into cylinder groups is still useful though.
- Berkeley FFS quickly became more popular than s5fs.
- There are better file systems as we will see soon.

The Traditional UNIX Buffer Cache

- Instead of writing data directly to a hard disk, the data is copied to a kernel buffer and then the application can continue.
- Disk reads usually find the data in this cache and can avoid disk I/O.
- Most file accesses are reads.
- The buffer cache is copy back and modified blocks are marked as dirty.
- Dirty blocks are written periodically.

File System Consistency

- The problem with the buffer cache is lost writes if the kernel "crashes" eg due to a power failure.
- The system call `sync` schedules all dirty blocks to be written out —- but does not wait for completion.
- If writes of file system meta data is lost, the file system normally becomes corrupt and useless until it is repaired which takes time or is impossible.
- To limit the severity of corruption the kernel orders the writes — for example, to add a link to a file the operations are:
  - add an entry to a directory
  - increment the link count of the inode
- In which order should these writes be carried out?
- If the new directory entry is written and the system then crashes, then after reboot the file has more names than indicated in the inode, which is bad.

Slow write performance

- Even if the kernel wants the writes to be ordered so that the link count is written first, the device driver may find it more optimal to perform the writes in the opposite order to make the system faster.
- Therefore, the kernel can only give the device driver one write at a time for critical meta data, ie the writes must be synchronous.
- For this reason meta data writes in traditional file systems are slow.

Limited use of ordering meta data writes

- The purpose of ordering writes is **not** to eliminate the risk of a corrupt file system.
- The purpose is to make a corrupt file system recoverable after reboot, ie to repair it.
- The program file system check, **fsck** does the following:
  - Check that the data blocks of a file are used exactly once
  - Check that the link count is correct by inspecting directories
  - Move lost files to the **lost+found** directory.
- Such operations could take hours for file systems on modern large disks.

Design goals

- Improved performance over Berkeley FFS
- Crash recovery
- Support for larger files and file systems.

Improving performance beyond that of Bekreley FFS

- Most file accesses are reads.
- Most disk accesses are writes.
- Why?

Improving the speed of write accesses

- The main performance problem is that meta data writes must be synchronous.
- The meta data writes typically modify data at different disk locations which waste time waiting for seeks and rotations.

Improvements to FFS by Sun

- An **extent** is a large sequential disk area.
- Storing files in extents allows faster I/O but complicates file growth — the dynamic memory allocation problem.
- Sun made some minor modifications to FFS to improve performance using extents, called **clusters**.
- Basically file writes are delayed until a cluster is filled.
- The performance was doubled for sequential reads and writes.

The 4.4BSD Log-Structured File System LFS

- Most file reads are from kernel memory and writes must be improved.
- A partition is divided into **segments** and one segment is the current.
- All writes are done sequentially into the current segment until it is full.
- In BSD FFS each inode is in a fixed position.
- In LFS, an inode has no fixed position — it is written at the end of the log when it is modified.
- To find the correct version of an inode an **inode map** is used.
- The inode map is stored on disk but is cached in kernel memory.

Reading and Writing Files

- Writing is delayed until a full segment can be written, but sometimes partial segments must be written.
- For each modified file that will use the segment its logical data blocks are sorted and are assigned physical block numbers which are noted in the new inode version.
- A **cleaner** process reclaims storage of obsolete data blocks.
- The **segment usage table** contains information about the number of live (not yet overwritten) bytes each segment contains, and is used to decide which segments should be reclaimed (and live data moved).
- Reading a file is identical to BSD FFS with the exception of how the inode is located: in FFS its address is calculated (from the inode number) while in LFS it is looked up in the inode map.
- Most file reads can be serviced from the buffer cache.

Crash recovery

- The inode map and the segment usage table are in memory but is periodically written to disk as a readonly normal file called **ifile**.
- The ifile is a **checkpoint**: to recover a LFS file system the following is done:
  - the most recent checkpoint is located on the disk
  - the checkpoint is used to initialize the inode map and the segment usage table
  - the log is scanned forward and the two tables are updated to reflect what is in the log.
- The difference with FFS or s5fs or EXT2 are:
  - LFS recovery is simpler
  - LFS recovery is much faster
- EXT3 also uses a log but retains the structure of EXT2.

The Cleaner Process

- The log wraps around after the last segment has been written.
- The cleaner process makes sure there are available segments.
- A segment may be used directly if:
  - The blocks have been overwritten — and are in other segments
  - The files have been deleted
- If parts of some segments are still live, the cleaner may decide put them in new segments to free the old segments.

Awkward Problems in LFS

- If meta data writes are put in two different segments but the system crashes after only the first segment has been written.
- File blocks are allocated when the segment is written to disk and are not reserved when the file write is performed to kernel memory buffers.
- Therefore a write system call may return a positive value but fail later due to the disk is full.
- Large RAM memory is required but that should not be a big problem today.

Analysis of the Berkeley LFS

- Superior performance to Berkeley FFS in most situations.
- Much faster error recovery.
- Requires complete rewrite of some software: newfs and fsck.
- Approximately the same performance improvement as with Sun's FFS.
- The benefits of Berkeley LFS then is the fast recovery time.
- There are easier ways to achieve this as we will see next.

Meta Data Logging

- We want faster recovery and faster writes by avoiding the synchronous meta data modifications.
- File systems such as EXT3 add a meta data log to the normal file system.
- Meta data modifications are now done twice:
  - directly in the meta data log
  - at a later time in the normal file system data structures — called the **in-place** modifications
- Some in-place modifications can be merged or will never be needed.
- If the log becomes full the file system must wait until the log can be reused after performing the in-place modifications.
- To recover from a crash the last part of the log is redone.

Journals vs log-structured file systems

- As we saw previously two purposes of a log-structured file system are:
  - faster writes
  - faster recovery after a crash
- Switching to a log-structured file system from a traditional might be a good idea but there a practical aspects which sometimes make such a switch unattractive, for instance whether the file system is trusted
- An alternative is to extend an existing file system with a log
- This was done eg with **ext2** by Tweedie which resulted in **ext3**

The key motivation for designing ext3

- The key motivation for improving ext2 into ext3 was faster recovery
- Issues in reliably recovering data after a crash:
  - **Preservation** — the recovery should obviously not modify files already written to the disk before the crash
  - **Predictability** — it is easier to recover after a crash if there are guarantees about the order in which disk writes occured just before the crash
  - **Atomicity** — the file operations should either not happen at all or happen completely: suppose there is a crash while moving a file from one directory to another — then after recovery the moved file should be in exactly **one** of the directories
- Ext2 is neither predictable nor atomic

Achieving predictability

- Predictability becomes an issue when multiple disk blocks are modified due to some file operation
- When **fsck** tries to recover the file system it must try to figure out what was happening just before the crash
- How can the disk writes be performed in a predictable order?
- Although the kernel might issue the disk writes in a particular predictable order, the disk scheduler may optimize the order and change it
- The approach of the BSD FFS to this problem is to write all meta-data one block at a time, and this approach has inspired many UNIX file systems including ext2

Synchronous meta-data writes

- The disadvantage of writing meta-data synchronously of course is poor performance
- An alternative is to buffer the requests in memory and let the system call proceed and then writing the blocks sequentially
- This is the same idea for how relaxed consistency memory models work on multicore processors (by placing data in a write buffer which is later written to the cache)
- If there is a crash the in-memory buffers will get lost of course
- Recovering the file system still must scan the **entire** file system which takes time

Journaling file systems

- To avoid the time consuming file system recovery while still using the traditional file system disk layout, recall that **journaling** file systems do the following:
  - As in log-structured file systems, newly written data is written to a log
  - Once the data is safely in the log on the disk, it is copied to its normal disk blocks in the file system
- Ext3 works like this
- To do recovery, only the part of the log which has not yet been written to the normal file system blocks needs to be checked and written

Database transaction vs file systems

- A database transaction protects modifications and ends either with commit or abort
- A file system transaction must protect modifications eg of
  - data blocks
  - inode blocks
  - bitmap of free blocks
- Writing to a file system is similar but simpler than writing to a database
- Eg there is no **abort** for file systems, and database transactions can be much larger
- In a journaling file system, several file operations can be put into one transaction which can improve performance

More about Ext3

- An ext3 partition can be mounted by a kernel which only knows about ext2
- The ext2 file system has **reserved inodes** and one of these is used for the journal
- A new **compatibility bit** is used to mark a partition as ext3
- Ext3 is the most widely used file system on Linux
- An optional feature is a B-tree for storing directory entries

ext4: the next generation of the ext3 file system

- Based on an article by Avantika Mathur (IBM), Mingming Cao (IBM) and Andreas Dilger (Cluster Filesystems)

The Ext4 file system

- The goals with the ext4 file system include:
  - improved scalability (ext3 is limited by inode reference counter type, and data block index type (16 TB file system))
  - improved performance
  - smooth migration from ext3 to ext4
- Work on ext4 started June 2006 by Theodore Ts'o.
- Linux kernel 2.6.28 released in December 2008 contains ext4.
- Google switched from ext2 to ext4 in January 2010 for some disks — but they have their own file system as well.

Extents in Ext4

- An **extent** is a block of contiguous disk blocks
- An extent is up to 128 MB
- The ext4 inode can store four extent addresses and additional in an **extent tree**
- The performance improvement of extents compared with normal UNIX inode blocks for large files (eg DVDs) was found to be approximately 25%
- The buddy algorithm is used for allocating blocks to extents

The design goals of ZFS from Sun

- "Sufficiently" large file system (to fill it with data needs energy sufficient to boil the oceans — don't)
- Pooled storage which according to Sun is as important for disks as VM is for memory
- End-to-end data integrity
- Based on transactions
- Eliminate the RAID-5 "write hole" (a crash at certain point is problematic in RAID-5)

Security and integrity

- Based on transactions
- Data is never modified but rather copied:
  - When a data block is to be modified by a **write** system call, the data is written to new blocks and the inode then points to the new block
  - The inode is also not modified but copied
- Neither a log nor a journal is needed
- A side-effect of this copying is that the file system trivially can support versioning or backup similar to the **Time Machine** in MacOS X
- The Time Machine is useful for accidental deletes but does not replace backup or GIT.

Pooled storage

- Traditionally a UNIX file system is on a partition or multiple partitions and controlled by a volume manager
- In ZFS a number of devices (partitions) contribute with storage to a storage pool
- Between the file systems are a **data management unit** and a **storage pool allocator**
- This makes it more convenient to administrate the file systems