FILE SYSTEMS

Main issues:

➜ performance,

➜ space usage.

Main considerations:

➜ file sizes,

➜ file usage patterns,

➜ memory vs. disk speeds,

➜ disk characteristics.

FILE SIZES AND USAGE PATTERNS:

Files tend to be [ODCH+85]:

➜ small (>50% <2kB, average size 22kB),

➜ short lived,

➜ open for very short times,

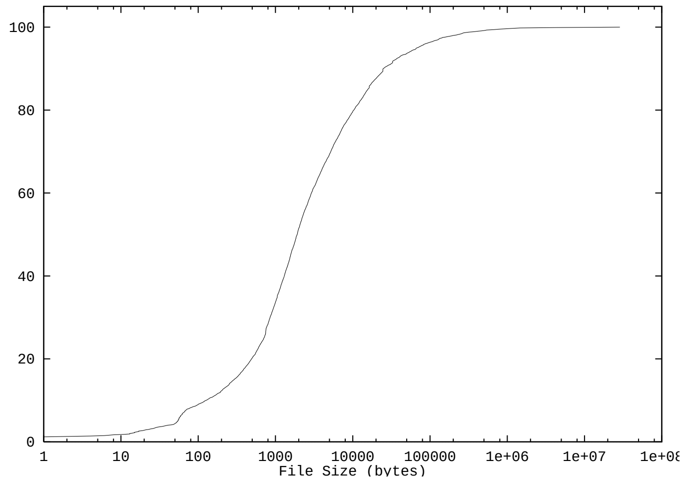
➜ mostly read/written entirely,

➜ mostly processed sequentially,

➜ much more reads than writes ("worm": *write once, read many*),

➜ caching can eliminate most reads.

EXAMPLE: File size distribution in CSE "VaST" network [Elp93].



DISK CHARACTERISTICS

➜ Unit of access is *sector*, order of kB.

➜ A number of sectors per *track*,

➜ *rotational delay* between non-consecutive sectors.

➜ Several patters, corresponding tracks form *cylinders*.

➜ Head movement required between cylinders, introduces *seek delay*.

➜ Seek times are order of ms.

➜ Maximum throughput is of the order of MB/s.

➜ Time to read/write single sector is dominated by seek time.

The Original Unix File System (UFS)

- ➜ Several *partitions* per physical disk.
- ➜ Each partition can contain a *file system*.
- ➜ This can be *mounted* in the hierarchical name space.
- ➜ First block of file system is *superblock*:
  - ➜ disk geometry, incl. total number of blocks,
  - ➜ pointer to block *free list*.
- ➜ Next is *inode* array, containing pointers to disk blocks.
- ➜ Finally data blocks.

UFS problems:

- Small block size (512b).
  - ➜ Easily fixed: Doubling doubled performance.
- No redundancy, hence easy to destroy.
  - ➜ Loss of superblock is disaster.
- No locality:
  - ➜ Inodes and data in different parts of disk.
  - ➜ Block allocation using free list leads to randomisation of file's blocks.
  - ➜ Generally no attempt to minimise rotational and seek latencies.

The Berkeley Fast File System (FFS)

FFS was designed to overcome the problems of UFS [MJLF84]

Main features:

- *Cylinder groups* to improve locality.
  - ➜ Copy of superblock in each cylinder group (on different platters) to improve fault tolerance.
  - ➜ *Free block bitmap* per cylinder group, supports allocation of consecutive blocks.
- Larger blocks (multiple of 4kB) for improved throughput.
- *Clustering* of I/O operations to further improve throughput [Pea88].
- *Fragments* (1/2 – 1/8 blocks) to reduce fragmentation.
- Pre-allocated inodes scattered throughout cylinder groups,
- *Bitmap free list* for fast allocation.
- Ability to *rebuild free lists* after crash (fsck).

Cylinder groups:

- ➜ If possible, file data and index blocks are allocated from same CG as inode,
- ➜ up to a maximum (25 % of CG capacity) to avoid squeezing out small files.
- ➜ Attempt to allocate consecutive logical blocks *rotationally optimal*.
  - ➜ Aided by bitmap free list.
- ➜ Attempt to allocate (plain) files' inodes in same CG as *directory*.
- ➜ Spread directories across CGs.

FRAGMENTS:

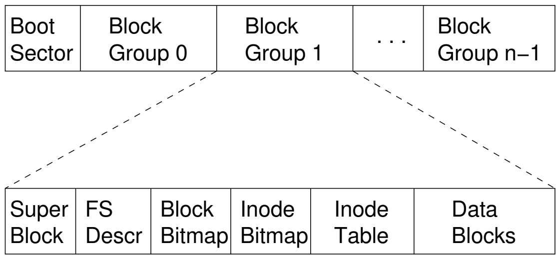
- Last block of small file is actually array of consecutive fragments.
- Fragments only allowed for files not using indirect blocks.
- Fragment array is unaligned but must not span blocks.
- When extending:
  - try allocating extra fragment from same block,
  - otherwise copy.
- Allocation supported by using fragment granularity in free block bitmap.

CLUSTERING:

- Try to allocate logically contiguous blocks contiguously (up to 64kB).
- Delay writes until ready to write whole cluster (if possible).
- Read ahead full cluster for sequentially accessed files.
- Reallocation of modified blocks to improve locality.
- Separate *cluster map* to help finding clusters.

VARIANT: LINUX EXTENDED-2 FILE SYSTEM (EXT2FS)

- (Logical) *block groups* rather then cylinder groups.
- Inode array in each block group.
- Store (short) symbolic links in inode.
- Preallocates contiguous blocks.



Ext2FS Disk layout

FFS PROBLEM: SYNCHRONOUS UPDATES

- Need to be able to recover *consistent state* of FS after crash.
- Perform *synchronous writes* of metadata to ensure consistency on disk:
  - On file creation:
    ① Write inode before updating directory.
    ② Write directory.
  - On file deletion:
    ① Write updated directory before deallocating inode.
    ② Write updated inode before freeing blocks in bitmap.
  - On file extension:
    ① Write data block *before* inode (for security).
    ② Write updated inode/indirect block before updating bitmap.
  - Similarly on file truncation.
- Running `fsck` after crash rebuilds consistent bitmaps.
- Synchronous I/O limits performance when creating/deleting many files.

Improvement: *Delayed writes/soft updates* [GP94]:

- Consistency does not *require* synchronous writes.
- Must ensure that *dependent writes* are *kept in sequence*.
  - ➜ E.g., ensure that on creation inode is written before bitmap.
- Unrelated reads or writes can still occur in arbitrary order.
- Need to:
  - ➜ enhance disk scheduler to honour write dependencies,
  - ➜ use copy-on-write on source blocks to avoid blocking further modifications.
- Cost:
  - ➜ Slightly more loss of user data during crash.
  - ➜ Much more complicated disk scheduling and interrupt processing.

Alternative: Logging to Improve Throughput

- Based on *write-ahead logging* used by databases.
- Maintain an *append-only redo-log* for metadata updates.
- Cedar [Hag87]:
  - ➜ Log written twice a second, grouping metadata updates.
  - ➜ Must ensure that log records are written *before* the logged metadata.
  - ➜ Other updates can be undone if crash before log write.
  - ➜ Use B-tree for directories.
  - ➜ Keep other metadata just before first data block.

SGI's XFS

- Scalable file system to support high-performance computing [SDH$^+$96]:
  - ➜ many files,
  - ➜ big directories,
  - ➜ huge files,
  - ➜ sparse files.
  - ➜ big file systems (TB),
  - ➜ fast crash recovery,
  - ➜ high bandwidth (uncomressed video), 500MB/s!
- Extensive use of B$^+$ trees:
  - ➜ directories
    - ➜ keyed by 4-byte hash of file name,
  - ➜ indirect block maps (using block ranges, *extents*),
  - ➜ free-block maps,
  - ➜ inode maps containing:
    - ➜ pointers to ranges of inodes, and
    - ➜ bitmap for range.

- *Allocation groups* (large sub-partitions) to localise metadata.
  - ➜ Limits size of free space and inode maps.
  - ➜ Allows use of 32-bit pointers.
  - ➜ Limits bottlenecks due to single-threaded updates.
  - ➜ Used similar to FFS CGs (cluster files, spread directories).
- Write-ahead logging for metadata updates:
  - ➜ inodes,
  - ➜ directory blocks,
  - ➜ free block maps,
  - ➜ inode maps,
  - ➜ file block maps,
  - ➜ AG headers,
  - ➜ superblock
- Batch and asynchronously write log.
- Buffer log to avoid blocking further metadata updates.
- Support separate log device.

- *Dynamically allocated inodes*.
- Block size up to 64kB.
- Block allocation as contiguous as possible:
  - ➜ *Extents*: large (up to 2^{21}) unaligned contiguous block ranges.
  - ➜ Physical pointers are extent descriptors:
    - ➜ file offset (for sparsity),
    - ➜ block address,
    - ➜ extent size.
  - ➜ Actual allocation of disk blocks is delayed as much as possible.
    - ➜ Allows maximising extents.
    - ➜ Prevents short-lived files from ever going to disk.
    - ➜ Requires extensive buffering in RAM.
- Aggressive read-ahead and write clustering.
- *Direct I/O*: DMA to user buffers.
- Multiple concurrent readers and writers supported to the same file.
- Log is only shared data structure on SMP system.

Limitations of Traditional Unix File Systems

Observations:

- Many write operations required for some operations, e.g., file creation:
  - ➜ 5 writes, 2 synchronous (inode (2), dir, dir's inode, data).
- Seeks are expensive:
  - ➜ Average seek costs as much as 100's of kB of data transfer.
- Big caches eliminate most reads:
  - ➜ Write costs become critical.

How can we eliminate

- synchronous writes, and
- seeks on writes?

Normal logging helps only partially.

Log-Structured File System (LFS)

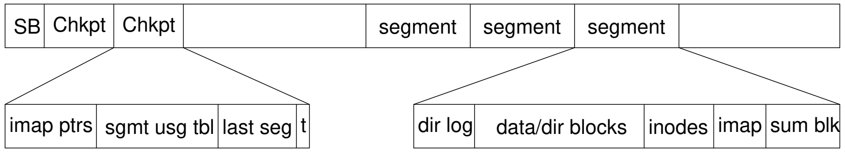
Idea: Whole disk is a log [RO92].

- Only write to end of log.
  - ➜ Eliminates most seeks.
- Write in large chunks, called *segments* (64kB–1MB).
  - ➜ Amortises costs of any seeks.
- Include in segments updated data block *as well as metadata*.
  - ➜ Can control write order by location within segment.
  - ➜ Ensures consistency of updates (like delayed writes).
- Keep track of life data in previously written segments.
  - ➜ Segments without life data can be reclaimed.
  - ➜ Partially life segments can be compacted.
  - ➜ Uses garbage collector process, called *cleaner*.
- Occasionally checkpoint metadata (using shadow paging).
  - ➜ Avoids processing whole log on restart.

LFS metadata:

- In segments:
  - ➜ inodes (as in FFS, but pointers change frequently),
  - ➜ directories (as in FFS),
  - ➜ inode map, giving locations of inodes (changes frequently),
  - ➜ log of directory changes (to avoid multiple updates),
  - ➜ segment summary:
    - ➜ number of next segment,
    - ➜ summary of contents,
    - ➜ checksums of contents (to verify successful write).
- In checkpoint:
  - ➜ timestamp,
  - ➜ address of last segment written,
  - ➜ segment usage table,
  - ➜ address of inode map.

LFS DISK LAYOUT



- LFS allows *fast restart* from crash:
  - ➜ Segment usage table allows locating unused segments to write.
  - ➜ Last segment pointer, and next pointer in segment summary allow recovery.
  - ➜ Imap pointers allow finding inode maps.
  - ➜ Time stamp in checkpoint allows finding last successful checkpoint.
- LFS provides *transaction semantics* on metadata updates.

LFS CLEANING

- Cleaning is critical:
  - ➜ LFS can only operate at full speed if sufficient free segments are available.
  - ➜ Segments utilisation drops over time as files get modified.
  - ➜ Most segments will not become totally empty.
  - ➜ Need garbage collection to compact partially-used segments.
- Cleaning is tricky:
  - ➜ Normal log use is dominated by writes and minimises seeks.
  - ➜ Cleaning is dominated by reads, and requires many seeks.
  - ➜ Cleaner has the potential to destroy LFS performance [SBMS93, SSB$^+$95].
- Basic approach:
  - ➜ Try cleaning at times of low disk activity;
  - ➜ Try keeping disk utilization low ($< 80\,\%$).
  - ➜ Preferably clean lowly utilised segments.
  - ➜ Group files of similar age.

MICROSOFT WINDOWS-NT FILE SYSTEM (NTFS)

Designed as a replacement of DOS FAT and OS/2 HPFS [Cus94].

- Requirements:
  - ➜ large files and devices ($>$ 2GB),
  - ➜ protection,
  - ➜ reliability and recoverability.
- Features:
  - ➜ transaction semantics on metadata updates,
  - ➜ support for mirroring and striping,
  - ➜ Unicode file names,
  - ➜ secondary indices (not implemented),
  - ➜ extensible attribute set, incl.
    - ➜ *multiple data streams*,
  - ➜ unified access to all attributes (incl. data),
  - ➜ POSIX features: case-sensitive names, creation time stamp, hard links.
  - ➜ Many others every non-PC user expects from a file system.

FILE SYSTEM META-METADATA: The *master file table* (MFT).

- "Relational database" of file metadata:
  - ➜ transaction semantics,
  - ➜ secondary indices for accessing files (only implemented on names),
  - ➜ indices are B$^+$ trees,
  - ➜ extensible on a per-record basis.
- One entry per file.
- Everything on disk is a file, including all metadata.
- Fixed length record (1–4kB) of variable-length attributes, incl.:
  - ➜ *all names* of each file (hard links, DOS name),
  - ➜ ACL
  - ➜ *data* (for small files ore directories).

- Entries are mostly one record, but can span several.
- Each attribute is represented by *header* and *value*.
- Value can be:
  - ➜ *resident*, i.e., in-line, or
  - ➜ *non-resident*, i.e., in separate file.
- Any value can be resident.
- Any value that can grow can be non-resident, incl.:
  - ➜ data,
  - ➜ file index (for directory),
  - ➜ ACL,
  - ➜ attribute list.
- Non-resident values
  - ➜ represented as array of: *(logical block #, physical block #, # blocks)*.
  - ➜ Supports sparse files (if *compression* attribute is on).
  - ➜ Also supports compression of data in extents.

INDICES

- Directories are indices of file names, containing:
  - ➜ file number (= MFT record number),
  - ➜ time stamps (duplicate of MFT value),
  - ➜ size (duplicate of MFT value).
- All file names are also contained in MFD.
- Indices are B^+ trees, represented as:
  - ➜ root node of names (key values),
  - ➜ corresponding child node (extent) pointers,
  - ➜ bitmap indicating which blocks in child extents are in use.

NTFS METADATA

- All metadata is in files, described by the first 16 records of the MFT:
  - ➜ MFT,
  - ➜ partial copy of MFT (for fault tolerance),
    - ➜ mirrors entries for metadata files,
    - ➜ is allocated in the middle of the disk;
  - ➜ log file,
  - ➜ root directory,
  - ➜ *boot file* (at known address, contains address of MFT),
  - ➜ free block bitmap,
  - ➜ bad block file,
  - ➜ *volume file* (volume label),
  - ➜ ...

NTFS RECOVERABILITY

- Uses write-ahead logging for metadata updates.
  - ➜ Log contains *redo* and *undo* info.
  - ➜ Log entries are *idempotent*.
  - ➜ Log records may contain data or operations.
- Logging operation:
  - ① Logs transaction sequentially in cache.
  - ② Flushes log file.
  - ③ Updates metadata in cache.
  - ④ Writes *commit* record to log file.
- Occasionally writes *checkpoint records* to log:
  - ➜ Location recorded in (shadow-paged) log header.
  - ➜ Indicates starting point for recovery.
- When log full:
  - ① stalls metadata update,
  - ② flushes buffers,
  - ③ restarts log.

MICRO$PEAK TRANSLATION TABLE

Cluster	Block
Volume	Partition
Run	Extent
Virtual cluster number	Logical block number
Logical cluster number	Physical block number

REFERENCES

[Cus94] Helen Custer. *Inside the Windows-NT File System*. Microsoft, 1994.

[Elp93] Kevin Elphinstone. Address space management issues in the Mungi operating system. Technical Report UNSW-CSE-TR-9312, School Comp. Science & Engin., University NSW, Sydney 2052, Australia, November 1993.

[GP94] Gregory Ganger and Yale N. Patt. Metadata update performance in file systems. In *Proc. 1st Symp. Operat. Syst. Design & Implem.*, pages 49–60, Monterey, CA, USA, November 1994. USENIX/ACM/IEEE.

[Hag87] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. 11th ACM Symp. OS Princip.*, pages 155–162, Austin, TX, USA, November 1987.

[MJLF84] Marshall K. McKusik, William N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Trans. Comp. Syst.*, 2:181–197, 1984.

[ODCH+85] J. Ousterhout, H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, and J. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proc. 10th ACM Symp. OS Princip.*, pages 15–24, 1985.

[Pea88] J. Peacock. The Counterpoint fast file system. In *Proc. 1988 Winter USENIX Techn. Conf.*, pages 243–249, January 1988.

[RO92] Mendel Rosenblum and John Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comp. Syst.*, 10:26–52, 1992.

[SBMS93] Margo Seltzer, Keith Bostic, M. Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. In *Proc. 1993 Winter USENIX Techn. Conf.*, pages

307–326, San Diego, CA, USA, January 1993.

[SDH+96] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proc. 1996 USENIX Techn. Conf.*, San Diego, CA, USA, January 1996.

[SSB+95] Margo Seltzer, K. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proc. 1995 USENIX Techn. Conf.*, pages 249–264, New Orleans, LA, USA, January 1995.