File Systems: Implementation

- Now that we have seen *what* an operating system presents to its users and developers (and sometimes to itself) in terms of logical storage units (files), we can take a look at *how* the OS implements this abstraction
- We focus on disks, since those are by far the most common secondary storage device in use today; some notes on disk hardware:
  - Disks typically transfer data in *blocks* of bytes, not one at a time
  - A disk's two primary activities are *seek*, which positions the *drive head* at the location where I/O is to take place, and *transfer*, which moves the data at the current location to the requesting system; seek generally weighs more heavily when evaluating performance

File System Levels

Beneath the logical file system are layers that ultimately lead to the devices themselves:

- A *file-organization module* knows how to map a file's logical blocks (always 1 to *n*) to its physical blocks
- A *basic file system* handles read and write requests for physical blocks (typically 512 bytes per block)
- A *device driver* takes care of transferring information to and from the actual device (commands, results)
- At this point, we reach the hardware

File System Data Structures

- Ultimately, a file system is a large, expansive, persistent data structure, requiring both on-disk and in-memory substructures and information
- On disk (also called a volume), we have items such as:
  - A *boot control block* that tells a computer system how to start up from that particular disk
  - A *volume control block* that tracks the states and counts of blocks on a particular volume
  - A *directory structure* that maps a volume's blocks to its logical files
  - A *file control block* that stores information about an individual file
- As with many other operating system concepts, different OSes may use different terms, but their roles and functions are generally quite similar

- In memory (or sometimes also on-disk, but used primarily for runtime/dynamic activities), we have:
  - A *mount table* that lists the volumes whose file systems are currently visible
  - A directory-structure *cache* (among other caches) that help in speeding up frequently- or recently-accessed information
  - *Open-file tables* list the files that are currently being accessed by processes; typically a cache of the FCB for the open file — these come in two flavors, *system-wide* or *per-process*
- These data structures combine to implement a file system's operations:
  - *File creation* typically involves allocating a new FCB and updating the directory structure under which the created file is to reside
  - *File opening* involves mapping a file's logical name to its FCB (i.e., directory traversal and search), then copying that FCB into the open-file table
  - *Reads* and *writes* use a reference to the opened file, typically called a *file descriptor* or *file handle*; at this point the filename or path is no longer necessary
  - *File closing* reverses the activities performed upon file opening — the corresponding entry in the open-file table(s) is removed and directory entries may be updated with the latest file metadata (access timestamps, etc.)

Partitions and Mounting

- Though sometimes interchangeable, "disk" and "volume" don't always mean exactly the same thing — a disk refers mainly to the actual physical device, while a volume refers to a self-contained file system
- Disks may contain more than one volume via *partitions*; special-function partitions include a *raw* partition — for which the logical file system is bypassed and blocks are manipulated directly, typically for performance reasons — and a *root* partition, which refers to the partition that holds the operating system
- Conversely, multiple partitions across multiple disks may also be made to appear as a single volume, usually via RAID (a.k.a. "redundant array of inexpensive (or independent) disks")
- Volumes form units that may be *mounted* onto a computer system; the volume on the root partition is typically mounted first, with other volumes mounted later either automatically or manually, whether by command or upon detection of removable media
- Upon mounting, the OS checks for a valid file system; successful mounts add entries to the mount table
- *Mount points* vary across operating systems — Windows traditionally uses drive letters, while Unix mounts to any directory

Virtual File Systems

- Different devices and technologies may have different specific file system implementations with common operations but differing specific details
  - CDs have their own file system, based on specifications such as ISO 9660 and Redbook
  - Mac OS X supports both the traditional Unix file system, UFS, and its own HFS+
  - Remotely accessible file systems such as NFS and CIFS don't use disk commands at all, but are based on network protocols
- In all cases, we don't want to worry about underlying differences; what we care about is that we see a single, uniform abstraction, regardless of the implementation specifics — this is the *virtual file system* layer

- A virtual file system works a lot like an object-oriented system: a common set of "objects" and "methods" is available, and each underlying file system plugs in its own implementation
- The operating system handles the presence of different file system implementations, and positions them underneath the VFS interface
- Linux's VFS, for example, has four primary objects: an *inode* for a file, a *file* for an open file, a *superblock* for an overall file system, and a *dentry* for a directory entry
- Each object has operations; for example, *file* includes *open()*, *read()*, *write()*, and *mmap*()
- Linux's supported file systems "plug into" these objects

Directory Implementation

- A *directory* serves as a collection data structure for files; conceptually, it behaves as precisely that — a place for listing and locating files
- Internally, directories themselves reside in disk blocks; some OSes treat them as a special file type (Unix), while others treat them as distinct from files, with a completely different API (Windows)
- Two predominant implementations of a directory: first as a linear list, with each entry referring to a file in that directory; second as a hash table, for faster file search

Allocation Methods

- At the file-organization level, the "file" storage unit must be mapped into the sequence of disk blocks on which the file's data are actually stored; this mapping is bound to how these blocks are allocated
- One approach is *contiguous allocation* — similar to main memory, contiguous allocation maps a file to a linear sequence of blocks on the disk
- Contiguous allocation supports sequential and direct access well, but is very prone to external fragmentation and requires *a priori* file size knowledge

Linked Allocation

- *Linked allocation* treats disk blocks as nodes on a linked list; a file's directory entry serves as the list head, and each disk block stores a pointer to the next disk block
- No external fragmentation, no need for compaction, no issues with knowing the file size in advance or with files that grow over time — start with an empty directory entry, then allocate more blocks as needed
- However, very difficult to implement direct access, and the space used to store the pointers per disk block amounts to storage overhead

- Variations on linked allocation seek to address the disadvantages of its "default" version:
  - To address the space overhead, disk blocks can be *clustered* for a lower pointer-to-data ratio, at the cost of increased internal fragmentation
  - The relative fragility of a linked list (i.e., one bad pointer screws everything up) can be addressed by adding more data to the blocks/clusters: doubly-linked lists, filenames, etc.; the cost is increased overhead
  - Direct/random access can be improved by moving the linked list out of the blocks and into a *file allocation table* (FAT) — a FAT includes one entry per disk block, where each entry points to the next block in the file to which it belongs

Indexed Allocation

- *Indexed allocation* uses an *index block*, which resembles the page tables used for main memory management
- Each file has an index block, and this index block contains all of the pointers to the file's data blocks; the file's directory entry points to its index block
- Like linked allocation, indexed allocation addresses external fragmentation and file size issues, but may actually have more overhead — the index block itself — particularly for small files (because even a zero-length file uses up at least one disk block)

- But direct access is no longer a problem, and the external fragmentation and file size issues remain; index blocks are also a little more robust than a FAT
- A new issue that arises is maximum file size: after all, an index block is finite
  - *Linked index blocks* turn the index blocks themselves into a linked list; we allocate more index blocks as a file's size grows
  - A *multilevel index* is like a hierarchical page table; index blocks can point to other index blocks for *n* levels
  - A *combined scheme* mixes direct and multilevel blocks: the first *k* entries point directly to data blocks, while the remaining entries point to further index blocks
- The UFS *inode* is an indexed allocation scheme using combined pointers for file size; the last 3 pointers in an inode are single-, double-, and triple-indirect index blocks, respectively

Allocation Method Performance

- This is where the distinction between disk *seek* and *transfer* times have an impact on performance
- Contiguous allocation is fast: we need a single seek to locate the first disk block that we want to use (including direct access, since we can calculate the block where byte *b* resides), and that's it
- Linked allocation requires multiple seeks, plus direct access multiple seeks if we don't use the FAT variant
  - The speed benefits of contiguous allocation, particularly for direct access, may be compelling enough to have both methods available in an operating system — if a program can state a file's maximum length on creation, contiguous allocation may be used
- Indexed allocation performs differently based on whether the index block is cached or if we are using a multilevel scheme; without a cache, we need one seek per index block and have a final seek for the data
  - This is still better than linked allocation, where direct access to logical blocks near the end of the file would take more and more seeks — with indexed allocation, once we've read in the index block, we only need a single seek to reach any of a file's data blocks
  - Again for performance reasons, an OS may combine contiguous and indexed allocation — the usual approach is to start with contiguous allocation, then switch to indexed allocation when the file gets too large
- Note that the disparity of disk and CPU/main memory performance is such that the time savings for even a single seek would far exceed the execution of thousands of instructions in order to minimize the number of disk head movements — in other words, we can afford to get fancy, in a way

Free-Space Management

Hand-in-hand with *how* we allocate blocks for new files is knowing *what* blocks are available for allocation; as always, there are a number of options:

- A *bit vector* tracks free space as a single bit field, one bit per disk block — simple and supported by some hardware, but unwieldy for today's G-size disks
- A *linked list* tracks the free blocks in the same way that linked allocation tracks files — grabs the next free block easily, but not so good for making list-wide calculations unless the linked list is done "FAT-style"

- The *grouping* method is equivalent to indexed allocation — maintain a linked list of "free-block blocks" that point to (index) the free disk blocks
- Orthogonal to these data structures is a *counting* technique, which doesn't track individual free blocks but *contiguous sequences* of free blocks:
  - Instead of a single disk block reference, the item being stored is a "starting" block and a "block count" for the number of free blocks that are contiguously available after the starting block
  - Main advantage is an overall shortening of the free "cluster" list, assuming that most free areas consist of more than one contiguous disk block

Overall Efficiency

- Because disk activity is extremely costly when compared to CPU and main memory time, the slightest file system implementation detail can have a huge impact on overall efficiency and performance
- The rule of thumb is that, since disk seeks are significantly more costly than transfers, we strive for contiguity when possible: accessing *n* contiguous blocks is a single seek, vs. *n* blocks scattered all over
- Another factor is accommodating the future: what sizes should pointers be, what structures are dynamic

Here is a sampling of issues that real-world file systems have encountered:

- In UFS, inodes are preallocated and scattered across a volume, increasing the likelihood that a file's blocks are adjacent to its inode and thus decreasing the number of seeks required to read that file
- BSD Unix accommodates variable-size clusters to reduce internal fragmentation
- Certain metadata, such as "last access date," *always* requires a read and write of a file's directory entry — must weigh this cost against its benefit (one can say that the ubiquity of this date in current file systems indicates a consensus that the benefit of recording this date does outweigh the cost of having to update it for every opened file)
- MS-DOS FAT had to shift from 12- to 16- to 32-bit pointers as hard disks grew in size; each FAT transition caused inconvenience for the installed base
- Mac OS encountered a similar transition from its original HFS (16-bit pointers, or 65,536 total blocks) to the current HFS+ (32-bit pointers, or 4+ billion total blocks)
- Sun's Solaris originally had fixed-length data structures (such as for its open-file table), requiring a kernel recompile to resize; these data structures were all transitioned to dynamic allocation, causing a speed hit but resulting in better functionality

Overall Performance

- *Caching* is a major performance optimization for file systems, since main memory is faster than disk by multiple orders of magnitude
- In a way, virtual memory is like an inverted cache — we use disk storage to store memory content that exceeds physical memory capacity
- It turns out that the mechanisms for implementing virtual memory can also be used for caching file I/O; when an OS unifies the two functionalities, we have *unified virtual memory*

- A *unified buffer cache* is a variation on unified virtual memory that pipes memory-mapped I/O and *read()*-*write()*-based I/O through the same cache
- Synchronous vs. asynchronous I/O also contributes to better perceived performance: if I/O can be asynchronous, a process doesn't have to wait for the operation to complete before moving on — excellent for metadata writes (such as the "last accessed" date!)
- Finally, the predictable behavior of sequential file access results in additional speedup techniques:
  - *Free-behind* cleans up sequential pages as soon as the next page is accessed, since sequential access is likely to go backward within the file
  - *Read-ahead* is a converse activity, where we buffer succeeding contiguous pages while the current page is still being processed

Recovery

- As they say, stuff happens, so we must accommodate the possibility that our precious directory entries, linked lists, FATs, and index blocks might get corrupted
- The job of most OS-provided disk repair utilities (*chkdsk*, *fsck*, *Disk Utility*) is to perform consistency checks on a particular file system; each data structure is subject to different consistency rules
- Of course, not every corruption or inconsistency issue is reparable, so straight-up backup and recovery utilities are also needed

Log-Structured (a.k.a. Journaled) File Systems

- In a neat case of transfer from another area of computer science, transaction-log techniques from databases have turned out to be useful for file systems
- A *journaled file system* writes all activities as *transactions* to a *log*; processes view a file operation as completed after the log entry is written, but the system may perform the actual disk update a little later
- Recovery involves "replaying" log entries upon system startup; uncommitted transactions at crash time still lose data, but at least the file system stays consistent

Network File Systems

- A significant functionality gained by having a VFS layer is transparent file system access over a network
- Sun's *network file system* (NFS) is the great granddaddy of network-transparent file systems, but today we also have the *common Internet file system* (CIFS, originated by Windows as SMB ["server message block"] with an available open-source implementation called Samba), the *Apple filing protocol* (AFP, originated by Apple), and *WebDAV* ("Web-based distributed authoring and versioning," an *http* extension), among others

- General operation of network file systems follows a similar flow, though specific details may differ:
  - A *mount protocol* determines how a remote file system can be made visible on the local file system; following the same metaphor as local device mounts works well
  - The *file system protocol* consists of a *remote procedure call* (RPC) interface, specifying what file operations the client can request of a server
  - The protocol itself is then subsumed underneath the VFS layer, so that applications can open, close, read, and write files using the same interface
  - Some protocols are *stateless*, meaning that each RPC call is a standalone command