A primer to File Systems

(NetDiox R&D Team)

Authored By :

Peter Chacko, Founder & CTO

Netdiox Computing System Pvt. Ltd.
(Linux kernel research/training center)

About the Author:

Peter Chacko has been working on system/networking/storage systems development since 1994. His career spans Hard-real time/embedded systems, OS kernels like Unix and Linux, distributed storage and networking systems, virtualization, data security and related computer science disciplines. He had been working for companies like Bellcore (Bell labs spin-off), HP, IBM, US West communications and a couple of startups as a consultant in the United states for 6+ years. He holds a Masters degree in Computer science &applications(MCA), Bachelors in Physics. He is currently the founder & CTO of Bangalore-based cloudStorage startup(Sciendix data systems pvt.Ltd). He also runs NetDiox computing systems as a not- for-profitresearch center-cum-educational center on system and networking software alongside. Prior to this he was working for calsoftlabs heading all the R&D activities of storage networking business unit. His linked-in profile is at
https://in.linkedin.com/in/peterchacko

Introduction

The fundamental role of a computer is to run applications using compute and network resources, taking, processing, emitting digital data using storage resources or file systems and optionally transmitting various forms of data across data communication network using communication networks using networking stack. In this context, file systems define the fundamental character of a data processing machine or a computer as it relates to how data is created, stored, retrieved, protected and managed. File systems are the key software components of storage systems that abstract out the block level storage media into a hierarchy of file system directory structure that makes data available for access to users as files and directories. As operating systems makes each hardware-resource available to the user by various means, like a windows in Microsoft operating systems allows running a new applications making use of a new compute resources; like a web browser allows a user to send and receive HTML pages over HTTP protocol, the file explorer allows a user manage computer files in a way she can manipulate.

File systems is typically a layer of software that runs as part of the operating system kernel, that interface with storage hardware-media on a specific format. As we alluded earlier, main purpose of computers is to create, manipulate, store, and retrieve data and the file system delivers the needed machinery to support these tasks. At the highest level a file system is a way to organize, store, retrieve, and manage information on a permanent storage medium such as a disk. File systems manage permanent storage and form an integral part of all operating systems. It is imperative to remember the abstract goals of what a file system should be delivering in terms of storing, retrieving, locating, and manipulating information. Keeping the above goal stated in general terms met, we

can also think of alternative ways of building a file system for various other purposes and there are 100s of types of file systems in the market as there are many different approaches to the task of managing permanent storage.

Layout of user data and associated information to locate this data – a.k.a metadata- depends of the file system types in context. There are different ways a file system can be implemented mainly driven by the storage layouts in the physical media the capabilities which the file system is tuned for – like data reliability, random access versus sequential access performance, primary versus secondary storage use cases so on and so forth. File systems also differs by the targeted hardware environment for which it's optimized for. We will briefly cover the general characteristics of file systems and then will discuss the few leading file system technologies starting with key storage terms first.

Key Storage Terms

Disk: A permanent storage medium typically captive in your computer, of a certain size. A disk also has an attached sector or block size, which is the minimum unit that the disk can read or write at a given time, limited by its hardware implementation. Typically, the block size of most modern hard disks is 512 bytes.

Block: The smallest unit writable by the file system. Everything a file system does is composed of operations done at the granularity of blocks. A file system block is always the same size as or greater than the disk block size and is always a multiple of sector size.

Partition: A subset of all the blocks on a disk. A disk can have several partitions.

Volume: The software abstraction for a collection of blocks on some storage medium (i.e., a disk). Basically a volume may be all of the blocks on a single disk, some portion of the total number of blocks on a disk, or may span multiple disks and or any software-conceivable aggregation of the blocks. The term “volume” is used to refer to a disk or partition that has been initialized with a file system. Volume is a software-defined entity, not a hardware-centric term.

Superblock: The area of a volume where a file system stores its critical volume wide information. A superblock usually contains information such as how large a volume is, the name of a volume, file system resource information such as no of inodes and data blocks and so on.

Metadata: A general term referring to information that is about something but not directly part of it. For example, the size of a file is very important information about a file, but it is not part of the data in the file.

Journaling: A method of insuring the correctness of file system metadata and data consistency even in the presence of power failures or unexpected reboots. For example leading File systems in the industry such as Athinio cloud-integrated file systems employ the journalling for providing any point recovery.

inode: Also known as index node, historically in Unix.The fundamental storage unit in disk for housing the main metadata of file system stores about all the necessary metadata about a file. The inode connects the contents of the file and any other data associated with the file. An i-node is also known as a file control block.

Extent: A physically contiguous blocks with a starting block number and a length of successive blocks on a disk. For example an extent might start at block 4000 and continue for 5000 blocks.

Key File system concepts

Files: The primary capability that all file systems must provide is a way to store a named piece of data and to later access that data using the name given to it. We often refer to a named piece of data as a file. This abstraction provides only the most basic level of functionality in a file system. This identifier of a file is called its filename. A file is where a program stores data permanently in disk. If you removed this name entry from the systems, using the removal operations of the file system, data is never accessible unless there are other "references"(hard-links) to it. In its simplest form a file stores a single piece of information as a stream of bytes. A piece of information can be a bit of text , a video film, graphic image, a database, or any collection of bytes a user wishes to store permanently. The size of data stored may range from only a few bytes to the entire capacity of a volume. An industry-class file system should be able to hold a large number of files, where "large" ranges from tens of thousands to millions.

The Structure of a File Given the concept of a file, a file system may impose no structure on the file, or it may enforce a considerable amount of structure on the contents of the file. An unstructured or a raw file is often referred to as a stream of bytes, literally has no structure. The file system simply records the size of the file and allows programs to read the bytes in any order or fashion at any location that they desire. This is different from structured data where there is a fixed structure in the file and user operates at those structures for various structure-aware operations such as a data base record update.

Directory: Apart from a single file being stored as a stream of bytes, a file system must be capable of organizing multiple files. File systems employ the term directory or folder to describe a "bucket" that organizes files by name.

Inode: Inode uniquely represent the file object in the file system. The primary purpose of a directory is to manage a list of files and to connect the name in the directory with the associated inode. There are several ways to implement a directory, but the basic concept is the same for each. A directory contains a list of names. Associated with each name is a handle that refers to the contents of that name . Although all file systems differ on exactly what constitutes a file name, a directory needs to store both the name and the inode number of this file. The name is the key that the directory searches on when looking for a file, and the inode number is a reference that allows the file system to access the contents of the file and other metadata about the file. When a user wishes to open a particular file, the file system must search the directory to find the requested name. If the name is not present, the file system can return an error such as Name not found. If the file does exist, the file system uses the i-node number to locate the metadata about the file, load that information, and then allow access to the contents of the file.

Metadata: The name of a file is metadata because it is a piece of information about the file that is not in the stream of bytes that make up the file. There are many pieces of metadata about a file as well—for example, the owner, security access controls, date of various file operations and size. The file system needs a well-defined place to store this metadata in addition to storing the file contents. Generally the file system stores file metadata in an inode. But various file systems can choose to store inodes in various ways. For example, Athinio file storage systems store metadata and user data in separate storage systems, as that allows the integraton of various file systems, storage units into a cohesive , single virtual namespace. The types of information that a file system stores in an inode vary depend- ing on the file system. Examples of information stored in i-nodes are the last access time of the file, the type, the creator, a version number, and a reference to the directory that contains the file. The choice of what types of metadata information make it into the i-node depends on the needs of the rest of the system.

Mounting and un-mounting and Filesystem operations:

Mounting and un-mounting is the process of making the file system module "take the responsibility" of file system actions performed on a specific directory hierarchy. This typically involves setting up kernel structures for the virtual file system module of the operating systems access file system specific operation for the required access and execute the operation desired. Un-mounting the reverse process, where the mounted partition is detached and no-longer accessible, after syncing all dirty data and metadata to the persistent storage medium. During this mount operation, various file system specific operations for directories, inodes, file IO are registered to to kernel based VFS interface so that further file processing is smoothly directed to the right kernel functions which are "hooked" through this mount operation. Filesystem operations refers to the actual file system functions invoked when a certain operation on file system, by a user is performed. Such operations includes creation of a new file, updating a file, changing the metadata or accessing various attributes or data.

We will now discuss different features of XFS file system for a "sense of" a typical, enterprise class file system.

XFS - File system for the Big Iron

XFS was created for the environment where large numbers of CPUs and large disk arrays are the norm. It focuses on supporting large files and good streaming I/O performance. It also has some interesting administrative features not supported by other file systems supported by Linux kernel at the time it was built. Each XFS file system is partitioned into a fundamental abstraction called allocation groups (AGs). AGs are somewhat similar to the block groups in ext-n variants, but AGs are typically much larger than block groups and are used for scalability and parallelism rather than disk locality or IO throughput acceleration. Allocation groups are typically sized between 0.5 and 4 gigabytes and keep the size of the XFS data structures in a range where they can operate efficiently and in parallel. Many file systems, including ext3, use traditional bitmaps to manage free space, which is obviously inefficient especially for larger contiguous allocations. Instead, XFS make use of a pair of B+ trees in each allocation group. Each entry in the B+ tree nodes consists of a start block and length pair describing a free-space region. The first B+ tree is indexed by the starting block number of the free region while the other is indexed by the length of the AG. This double routes allows the allocator to consider two goals for new data placement: locality to existing file data and best fit into free space in terms of needed storage. Familiar extent concept is used for tracking the disk blocks allocated for each file. In addition to the start block on disk and the block length of the contiguous range, the extent descriptor also contains two additional fields. The first one is the logical offset into the file, which allows for efficient sparse file support by skipping ranges that do not have blocks allocated to them. The second one is a simple one-bit flag to mark an extent as unwritten. For most files, a simple linear array of extent descriptors is embedded into the inode, avoiding additional metadata blocks and management overhead. The XFS inode consists of two parts and an optional part; The inode core, the data fork, and the attribute fork. The inode core

contains, as expected traditional UNIX inode data structure such as ownerships, number of blocks, timestamps, and a few XFS-specific additions such as project ID. The data fork contains the previously mentioned extent descriptors or the root of the extent map. The optional attribute fork contains the extended attributes.

Extended attributes , as its is implemented in Linux, are simple name/value pairs assigned to a file that can be manipulated, one at a time. The attribute fork in XFS can either store extended attributes directly in the inode if the space required for the attributes is small enough, or use the same scheme of extent descriptors as described for the file data above to point to additional disk blocks. Extended attributes are used by Linux to implement various things such as access control lists (ACLs) and labels for SELinux.

Inodes in XFS are dynamically allocated, which means that, there is no need to statically allocate the expected number of inodes when creating the file system, with the possibility of under or over-provision. With such scheme every block in the file system can now possibly contain inodes, an additional data structure is needed to keep track of inode locations and allocations. For this, each allocation group contains another B+ Tree. XFS uses a pair of B+trees to manage free disk space. Because Inode structures are maintained in a B+Tree, inode lookup is not a constant time operation. XFS divides a disk up into large sized chunks called allocation groups (a term with a similar)Each allocation group maintains a pair of B+trees that record information about free space in the allocation group. One of the B+trees records free space sorted by starting block number. The other B+tree sorts the free blocks by their length. This scheme offers the ability for the file system to find free disk space based on either the proximity to already allocated space or based on the size needed. Clearly this organization offers significant advantages for efficiently finding the right block of disk space for a given file. The only potential drawback to such a scheme is that the B+trees both maintain the same information in different forms. This duplication can cause inconsistencies if, for

whatever reason, the two trees get out of sync. Because XFS is journaled, however, such accidental inconsistencies are minimized to near-zero and is not generally an issue.

XFS uses extent maps to manage the blocks allocated to a file. An extent map is a starting block address and a length .
Unlike other file systems, XFS maintain the mapping of extents to file offsets thats contained by the extents in the B+ Tree. Which allow XFS to use variable-sized extents, resulting in a more complex implementation, for performance benefits. (A small amount of data in an extent can map very large regions of a file. XFS can map up to two million blocks with one extent map.)
Since Directory entries are stored in the B+ Tree, name look ups are very fast.

To keep a RAID array busy, the file system should submit I/O requests that are large enough to span all disks in the array. In addition, I/O requests should be aligned to stripe boundaries where possible, to avoid read-modify-write cycles for common usage patterns. Because a single I/O can only be as large as a contiguous range of blocks, it is critical that files are allocated as contiguously as possible, to allow large I/O requests to be sent to the storage. The key to achieving large contiguous regions is a method known as “de-layed allocation.” In delayed allocation, specific disk locations are not chosen when a buffered write is submitted; only in-memory reservations take place. Actual disk blocks are not chosen by the allocator until the data is sent to disk due to memory pressure, periodic write-backs, or an explicit sync re- quest. With delayed allocation, there is a much better approximation of the actual size of the file when deciding about the block placement on disk. In the best case the whole file may be in memory and can be allocated in one contiguous region. For today’s large file systems, a full file system check on an unclean shutdown is not acceptable because it would take too long. To avoid the requirement for regular file system checks, XFS uses a write-ahead logging scheme that enables atomic updates of the file system. XFS only logs structural updates

to the file system metadata, but not the actual user data, for which the Posix file system interface does not provide useful atomic guarantees .XFS logs every update to the file system data structures and does not batch changes from multiple transactions into a single log write, as is done by ext3. This means that XFS must write significantly more data to the log in case a single metadata structure gets modified again and again in short sequence. To mitigate the impact of log writes to the system performance, an external log device can be used. With an external log the additional seeks on the main device are reduced, and the log can use the full sequential performance of the log device.

Disk Quota management is another strong area of XFS.
XFS provides an enhanced implementation of the BSD disk quotas. It supports the normal soft and hard limits for disk space usage and number of inodes as an integral part of the file system. Both the per-user and per-group quotas supported in BSD and other Linux file systems are supported. In addition to group quotas, XFS alternatively can support project quotas, where a project is an arbitrary integer identifier assigned by the system administrator. The project quota mechanism in XFS is used to implement directory tree quota, where a specified directory and all of the files and subdirectories below it are restricted to using a subset of the available space in the file system.

The final area that XFS excels in is its support for parallel I/O. Supporting fine-grained locking was essential for XFS. Although most file systems allow the same file to be opened multiple times, there is usually a lock around the i-node that prevents true simultaneous access to the file. XFS removes this limitation and allows single-writer/multireader access to files. For files residing in the buffer cache, this allows multiple CPUs to copy the data concurrently. For systems with large disk arrays, allowing multiple readers to access the file allows multiple requests to be queued up to the disk controllers. XFS can also support multiple-writer access to a file, but users can only achieve this using an access mode to the file that bypasses the cache.

XFS offers an interesting implementation of a traditional file system. It departs from the standard techniques, trading implementation complexity for performance gains.

The xfs fsr utility defragments existing XFS file systems. The xfs bmap utility can be used to interpret the metadata layouts for an XFS file system. The growfs utility allows XFS file systems to be enlarged on-line. The high level structure of XFS is similar to a conventional file system with the addition of a transaction manager. XFS supports all of the standard Unix file interfaces and is entirely POSIX- compliant. It sits below the VFS in the Linux kernel and takes full advantage of services provided by the kernel including the buffer/page cache, the directory name lookup cache.

XFS is modularized into several parts, each of which is responsible for a separate piece of the file system's functionality. The central and most important piece of the file system is the space manager. This module manages the file system free space, the allocation of inodes, and the allocation of space within individual files. The I/O manager is responsible for satisfying file I/O requests and depends on the space manager for allocating and keeping track of space for files. The directory manager implements the XFS file system name space. The buffer cache is used by all of these pieces to cache the contents of frequently accessed blocks from the underlying volume in memory. It is an integrated page and file cache shared by all file systems in the kernel. The transaction manager is used by the other pieces of the file system to make all updates to the metadata of the file system atomic. This enables the quick recovery of the file system after a crash.

XFS journals metadata updates by first writing them to an in-core log buffer, then asynchronously writing log buffers to the on-disk log. The on-disk log is a circular buffer: new log entries are written to the head of the log, and old log entries are removed from the tail once the in-place metadata updates occur. After a crash, the on-disk log is read by the recovery code which is called during a mount operation. XFS metadata

modifications use transactions: create, remove, link, unlink, allocate, truncate, and rename operations all require transactions. This means the operation from the standpoint of the file system on-disk metadata either never starts or always completes. These operations are never partially completed on-disk: they either happened or they didn't. Transactional semantics are required for databases, but until recently have not been considered necessary for file systems. This is likely to change, as huge disks and file systems require the fast recovery and good performance journaling can provide.

An important aspect of journaling is write-ahead logging: metadata objects are pinned in kernel memory while the transaction is being committed to the on-disk log. The metadata is unpinned once the in-core log has been written to the on-disk log. Note that multiple transactions may be in each in-core log buffer. Multiple in-core log buffers allow for transactions when another buffer is being written. Each transaction requires space reservation from the log system (i.e., the maximum number of blocks this transaction may need to write). All metadata objects modified by an operation, e.g., create, must be contained in one transaction.

With the above note on XFS, it's hoped that the reader of this article got a start on the exciting world of file systems!

About NetDiox Computing Systems:

NetDiox is a Bangalore (India) basedcenter of research and education focusing around OS technologies, networking, distributed storage systems, virtualization and cloud computing. It has been changing the way computer science is taught and applied to distributed data centers. NetDiox provides high quality training and R&D project opportunity on system software, Linux kernels, virtualization, and networking technologies to brilliant BE/M.Techs/MCA/MScs who are passionate about system/storage/networking software.

Contact Us:

NetDiox R&D Team

Netdiox Computing System Pvt. Ltd.

Address: #1514, First Floor,19th Main Road,

HSR Layout Sector-1, Bangalore 102,Karnataka, India

Contact No: 08197324604

E-mail:**info.netdiox@gmail.com**