System Administration

From OpenZFS

This page is intended to document information that is useful for system administrators.

Contents

Overview

ZFS is a rethinking of the traditional storage stack. The basic unit of storage in ZFS is the pool and from it, we obtain datasets that can be either mountpoints (a mountable filesystem) or block devices. The ZFS pool is a full storage stack capable of replacing RAID, partitioning, volume management, fstab/exports files and traditional file-systems that span only 1 disk, such as UFS and XFS. This allows the same tasks to be accomplished with less code, greater reliability and simplified administration.

The creation of a usable filesystem with redundancy from a set of disks can be accomplished with 1 command and this will be persistent upon reboots. This is because a ZFS pool will always have a mountable filesystem called the root dataset, which is mounted at pool creation. At creation, a pool is imported into the system, such that an entry in the zpool.cache file is created. At time of import or creation, the pool stores the system's unique hostid and for the purposes of supporting multipath, import into other systems will fail unless forced.

ZFS itself is composed of three principle layers. The bottom layer is the Storage Pool Allocator, which handles organizing the physical disks into storage. The middle layer is the Data Management Unit, which uses the storage provided by the SPA by reading and writing to it transactionally in an atomic manner. The top layer is the dataset layer, which translates between operations on the filesystems and block devices (zvols) provided by the pool into operations in the DMU.

Low level storage

The organization of disks in a pool by the SPA is a tree of vdevs or virtual devices. At the top level of the tree is the root vdev. Its immediate children can be any vdev type other than itself. The main types of vdevs are:

- mirror (n-way mirrors supported)
- raidz
  - raidz1 (1-disk parity, similar to RAID 5)
  - raidz2 (2-disk parity, similar to RAID 6)
  - raidz3 (3-disk parity, no RAID analog)
- disk
- file (not recommended for production due to another filesystem adding unnecessary layering)

Any number of these can be children of the root vdev, which are called top-level vdevs. Furthermore, some of these may also have children, such as mirror vdevs and raidz vdevs. The commandline tools do not support making mirrors of raidz or raidz of mirrors, although such configurations are used in developer testing.

The use of multiple top level vdevs will affect IOPS in an additive manner where total IOPS will be the sum of the top level vdevs. Consequently, the loss of any main top level vdev will result in the loss of the entire pool, such that proper redundancy must be used on all top level vdevs.

The smallest supported file vdev or disk vdev size is 64MB (2^16 bytes) while the largest depends on the platform, but all platforms should support vdevs of at least 16EB (2^64 bytes).

There are also three special types:

- spare
- cache
- log

The spare devices are used for replacement when a drive fails, provided that the pool's autoreplace property is enabled and your platform supports that functionality. It will not replace a cache device or log device.

The cache devices are used for extending ZFS's in-memory data cache, which replaces the page cache with the exception of mmap(), which still uses the page cache on most platforms. The algorithm used by ZFS is the Adaptive Replacement Cache algorithm, which has a higher hit rate than the Last Recently Used algorithm used by the page cache. The cache devices are intended to be used with flash devices. The data stored in them is non-persistent, so cheap devices can be used.

The log devices allow ZFS Intent Log records to be written to different devices, such as flash devices, to increase performance of synchronous write operations, before they are written to main storage. These records are used unless the sync=disabled dataset property is set. In either case, the synchronous operations' changes are held in memory until they are written out on the next transaction group commit. ZFS can survive the loss of a log device as long as the next transaction group commit finishes successfully. If the system crashes at the time that the log default is lost, the pool will be faulted. While it can be recovered, whatever synchronous changes made in the current transaction group will be lost to the datasets stored on it.

Data Integrity

ZFS has multiple mechanisms by which it attempts to provide data integrity:

- Committed data is stored in a merkle tree that is updated atomically on each transaction group commit
- The merkle tree uses 256-bit checksums stored in the block pointers to protect against misdirected writes, including those that would be likely to collide for weaker checksums. sha256 is a supported checksum for cryptographically strong guarantees, although the default is fletcher4.
- Each disk/file vdev contains four disk labels (two on each end) so that the loss of data at either end from a head drop does not wipe the labels.
- The transaction group commit uses two stages to ensure that all data is written to storage before the transaction group is considered committed. This is why ZFS has two labels on each end of each disk. A full head sweep is required on mechanical storage to perform the transaction group commit and flushes are used to ensure that the latter half does not occur before anything else.
- ZIL records storing changes to be made for synchronous IO are self checksumming blocks that are read only on pool import if the system made changes before the last transaction group commit was made.

- All metadata is stored twice by default, with the object containing the pool's state at a given transaction group to which the labels point being written three times. An effort is made to store the metadata at least 1/8 of a disk apart so that head drops do not result in irrecoverable damage.
- The labels contain an uberblock history, which allows rollback of the entire pool to a point in the near past in the event of a worst case scenario. The use of this recovery mechanism requires special commands because it should not be needed.
- The uberblocks contain a sum of all vdev GUIDs. Uberblocks are only considered valid if the sum matches. This prevents uberblocks from destroyed old pools from being be mistaken as being valid uberblocks.
- N-way mirroring and up to 3 levels of parity on raidz are supported so that increasingly common 2-disk failures[1] (https://queue.acm.org/detail.cfm?id=1670144) that kill RAID 5 and double mirrors during recovery do not kill a ZFS pool when proper redundancy is used.

Misinformation has been circulated on the FreeNAS forums that ZFS data integrity features are somehow worse than those of other filesystems when ECC RAM is not used[2] (https://forums.freenas.org/index.php?threads/ecc-vs-non-ecc-ram-and-zfs.15449/). That has been thoroughly debunked[3] (http://jrs-s.net/2015/02/03/will-zfs-and-non-ecc-ram-kill-your-data/). All software needs ECC RAM[4] (http://open-zfs.org/wiki/Hardware#ECC_Memory) for reliable operation and ZFS is no different from any other filesystem in that regard.

Boot process

On a traditional POSIX system, the boot process is as follows:

1. The CPU starts executing the BIOS.

2. The BIOS will do basic hardware initialization and load the bootloader.
3. The bootloader will load the kernel and pass information about the drive that contains the rootfs.
4. The kernel will do additional initialization, mount the rootfs and start /sbin/init.
5. /sbin/init will run scripts that start everything else. This includes mounting filesystems from fstab and exporting NFS shares from exports.

There are some variations on this. For instance, the bootloaders will also load kernel modules on Illumos (via a boot_archive) and FreeBSD (individually). Some Linux systems will load an initramfs, which is an temporary rootfs that contains modules to load and moves some logic of the boot process into userland, mounts the real rootfs and switches into it via an operation called pivot_root. Also, EFI systems are capable of loading operating system kernels directly, which eliminates the need for the bootloader stage.

ZFS makes the following changes to the boot process:

1. When the rootfs is on ZFS, the pool must be imported before the kernel can mount it. The bootloader on Illumos and FreeBSD will pass the pool informaton to the kernel for it to import the root pool and mount the rootfs. On Linux, an initramfs must be used until bootloader support for creating the initramfs dynamically is written.
2. Regardless of whether there is a root pool, imported pools must appear. This is done by reading the list of imported pools from the zpool.cache file, which is at /etc/zfs/zpool.cache on most platforms. It is at /boot/zfs/zpool.cache on FreeBSD. This is stored as a XDR-encoded nvlist and is readable by executing the `zdb` command without arguments.
3. After the pool(s) are imported, the filesystems must be mounted and any filesystem exports or iSCSI LUNs must be made. If the mountpoint property is set to legacy on a dataset, fstab can be

used. Otherwise, the boot scripts will mount the datasets by running `zfs mount -a` after pool import. Similarly, any datasets being shared via NFS or SMB for filesystems and iSCSI for zvols will be exported or shared via `zfs share -a` after the mounts are done. Not all platforms support `zfs share -a` on all share types. Legacy methods may always be used and must be used on platforms that do not support automation via `zfs share -a`.

The following table shows which protocols are supported by `zfs share -a` on various platforms:

Platform	iSCSI	NFS	SMB
FreeBSD	No	Yes	No
Illumos	Yes	Yes	Yes
Linux	No	Yes	Yes
Mac OS X	No	No	No
OSv	No?	No?	No?

Pool creation

Administration of ZFS is performed through the zpool and zfs commands. To create a pool, you can use `zpool create poolname ...`. The part following the poolname is the vdev tree specification. The root dataset will be mounted at /poolname by default unless `zpool create -m none` is used. Other mountpoint locations can be specified by writing them in place of `none`.

Any file or disk vdevs before a top level vdev keyword is specified will be a top level vdev. Any after a top level vdev keyword will be a child of that vdev. Non-equal sized disks or files inside a top level vdev will restrict its storage to the smaller of the two. For production pools, it is important not to create top level vdevs that are not raidz, mirror, log, cache or spare vdevs.

Additional information can be found in the zpool man page on your platform. This is section 1m on Illumos and section 8 on Linux.

Important information

Sector Alignment

ZFS will attempt to ensure proper alignment by extracting the physical sector sizes from the disks. The largest sector size will be used per top-level vdev to avoid partial sector modification overhead is eliminated. This will not be correct when drives misreport their physical sector sizes. The Performance Tuning page explains more.

Whole disk tweaks

ZFS will also attempt minor tweaks on various platforms when whole disks are provided. On Illumos, ZFS will enable the disk cache for performance. It will not do this when given partitions to protect other filesystems sharing the disks that might not be tolerant of the disk cache, such as UFS. On Linux, the IO elevator will be set to noop to reduce CPU overhead. ZFS has its own internal IO elevator, which renders the Linux elevator redundant. The Performance Tuning page explains this behavior in more detail.

Drive error recovery control

One important tweak that ZFS does not do at this time, is adjust error recovery control on the drives. When a drive encounters a read error, it will retry reads in the hope that a read from a slightly different angle will succeed and allow it to rewrite the sector correctly. It wll continue this until a timeout is reached, which is often 7 seconds on many drives. During this time, the drive might not serve other IO requests, which can have a crippling effect on IOPS because the transaction group commit will wait for all IOs to finish before the next can start. Until ZFS is changed to set this on disks that it controls, system administrators should use tools like smartctl and the system local file to do it on each boot.

Drive and/or drive controller hangs

In a similar manner, if a drive or drive controller hangs, it will hang the pool and the deadman timer will trigger. On Illumos, this will trigger a system restart. On Linux, this will attempt to log the failure, but do nothing else. The default timeout for the deadman timer is 1000 seconds.

Hardware

ZFS is designed to use commodity hardware to store data reliably, but some hardware configurations are inherently unreliable to such an extent that no filesystem can compensate for it. Advice on hardware configurations can be found on the Hardware (http://open-zfs.org/wiki/Hardware) page.

Hardware RAID

Hardware RAID should not be used with ZFS. An explanation why can be found on the Hardware page.

Performance

ZFS is self tuning, but there are a few knobs that can benefit certain applications. Those are described on the Performance tuning page.

Redundancy

Two-disk failures are common enough[5] (https://queue.acm.org/detail.cfm?id=1670144) that raidz1 vdevs should not be used for data storage in production. 2-disk mirrors are also at risk, but not quite as much as raidz1 vdevs unless only two disks are used. This is because the failure of all disks in a 2-disk mirror is statistically less likely than the failure of only 2 disks in a parity configuration such as raidz1, unless only 2 disks are used, where the failure risk is the same. The same risks also apply to RAID 1 and RAID 5.

Single disk vdevs (excluding log and cache devices) should never be used in production because they are not redundant. However, pools containing single disk vdevs can be made redundant by attaching disks via `zpool attach` to convert the disks into mirrors.

General Administration

General administration is performed at the pool level with the zpool command and at the dataset level with the zfs command. Documentation can be found in the respective zpool and zfs man pages for your platform. They are in section 8 on Linux and Section 1m on Illumos.

See the Platform/Distribution documentation for links to some of the man pages. Note that the man pages for one platform are likely relevant on other platforms.

Additional Resources

Third party tools

- bdrewery/zfstools (https://github.com/bdrewery/zfstools) provides tools for snapshot management. It is written in ruby.
- fearedbliss/bliss-zfs-scripts (https://github.com/fearedbliss/bliss-zfs-scripts) Provides tools for "ZFS Snapshot & Backup Management" under the MPL 2.0. Designed for Gentoo Linux.
- jimsalterjrs/sanoid (https://github.com/jimsalterjrs/sanoid) Policy-driven snapshot management and replication tools under the GPL. Tested against Linux and FreeBSD.
- Rudd-O/zfs-tools (https://github.com/Rudd-O/zfs-tools) provides tools for snapshot management and replication. It is written in python.
- ZFS Watcher (http://zfswatcher.damicon.fi/) is a pool monitoring and notification daemon.

General documentation

- Features
- Publications and conference talks
- History
- Performance tuning
- Manual pages: zdb (http://illumos.org/man/1m/zdb) | zfs (http://illumos.org/man/1m/zfs) | zpool (http://illumos.org/man/1m/zpool) | zpool-features (http://illumos.org/man/5/zpool-features) | zstreamdump (http://illumos.org/man/1m/zstreamdump) – *from illumos; better-rendered pages would be appreciated*
- Oracle Solaris ZFS Administration Guide (http://docs.oracle.com/cd/E26505_01/html/E37384/index.html) – largely applicable
- ZFS – The Last Word in File Systems (http://www.snia.org/sites/default/files2/sdc_archives/2008_presentations/monday/JeffBonwick-BillMoore_ZFS.pdf) – an overview from SNIA 2008
- Series of blog posts on ZFS from Aaron Toponce (https://pthree.org/2012/04/17/install-zfs-on-debian-gnulinux/)
- Video tutorials from Kateley Co (http://kateleyco.com/?page_id=783)
- RAID-Z stripe width (http://blog.delphix.com/matt/2014/06/06/zfs-stripe-width/)
- RAID-Z calculation tool (http://wintelguy.com/raidcalc.pl) - includes RAID-Z1/Z2/Z3 calculations
- Applicable to most platforms, but maybe outdated:
  - ZFS Best Practices Guide (http://www.solarisinternals.com/wiki/index.php/ZFS_Best_Practices_Guide)
  - ZFS Configuration Guide (http://www.solarisinternals.com/wiki/index.php/ZFS_Configuration_Guide)

    - ZFS Evil Tuning Guide (http://www.solarisinternals.com/wiki/index.php/ZFS_Evil_Tuning_Guide)

Platform/Distribution documentation

FreeBSD

- Manual pages: zdb (http://www.freebsd.org/cgi/man.cgi?query=zdb&manpath=FreeBSD+8.4-RELEASE) | zfs (http://www.freebsd.org/cgi/man.cgi?query=zfs&manpath=FreeBSD+8.4-RELEASE) | zpool (http://www.freebsd.org/cgi/man.cgi?query=zpool&manpath=FreeBSD+8.4-RELEASE) | zpool-features (http://www.freebsd.org/cgi/man.cgi?query=zpool-features&manpath=FreeBSD+8.4-RELEASE) | zstreamdump (http://www.freebsd.org/cgi/man.cgi?query=zstreamdump&manpath=FreeBSD+8.4-RELEASE)
- ZFS Chapter (https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/zfs.html) of *The FreeBSD Handbook*
- FreeBSD ZFS Wiki (https://wiki.freebsd.org/ZFS)

Gentoo

- Wiki page (https://wiki.gentoo.org/wiki/ZFS)
- Richard Yao's Gentoo installation notes (https://github.com/ryao/zfs-overlay/blob/master/zfs-install)

illumos

- Manual pages: zdb (http://www.illumos.org/man/1m/zdb) | zfs (http://illumos.org/man/1m/zfs) | zpool (https://www.illumos.org/man/1M/zpool) | zpool-features (https://www.illumos.org/man/5/zpool-features) | zstreamdump (http://illumos.org/man/1m/zstreamdump)
- Wiki page (http://wiki.illumos.org/display/illumos/ZFS)
- OpenIndiana ZFS Administration Guide (http://wiki.openindiana.org/oi/ZFS)
