ZFS on Linux

Contents

ZFS is a combined file system and logical volume manager designed by Sun Microsystems. Starting with Proxmox VE 3.4, the native Linux kernel port of the ZFS file system is introduced as optional file system and also as an additional selection for the root file system. There is no need for manually compile ZFS modules - all packages are included.

By using ZFS, its possible to achieve maximum enterprise features with low budget hardware, but also high performance systems by leveraging SSD caching or even SSD only setups. ZFS can replace cost intense hardware raid cards by moderate CPU and memory load combined with easy management.

General ZFS advantages

- Easy configuration and management with Proxmox VE GUI and CLI.
- Reliable
- Protection against data corruption
- Data compression on file system level
- Snapshots

- Copy-on-write clone
- Various raid levels: RAID0, RAID1, RAID10, RAIDZ-1, RAIDZ-2, RAIDZ-3, dRAID, dRAID2, dRAID3
- Can use SSD for cache
- Self healing
- Continuous integrity checking
- Designed for high storage capacities
- Asynchronous replication over network
- Open Source
- Encryption
- ...

Hardware

ZFS depends heavily on memory, so you need at least 8GB to start. In practice, use as much as you can get for your hardware/budget. To prevent data corruption, we recommend the use of high quality ECC RAM.

If you use a dedicated cache and/or log disk, you should use an enterprise class SSD. This can increase the overall performance significantly.

Do not use ZFS on top of a hardware RAID controller which has its own cache management. ZFS needs to communicate directly with the disks. An HBA adapter or something like an LSI controller flashed in “IT” mode is more appropriate.

If you are experimenting with an installation of Proxmox VE inside a VM (Nested Virtualization), don’t use `virtio` for disks of that VM, as they are not supported by ZFS. Use IDE or SCSI instead (also works with the `virtio` SCSI controller type).

Installation as Root File System

When you install using the Proxmox VE installer, you can choose ZFS for the root file system. You need to select the RAID type at installation time:

RAID0 Also called “striping”. The capacity of such volume is the sum of the capacities of all disks. But RAID0 does not add any redundancy, so the failure of a single drive makes the volume unusable.

RAID1 Also called “mirroring”. Data is written identically to all disks. This mode requires at least 2 disks with the same size. The resulting capacity is that of a single disk.

RAID10 A combination of RAID0 and RAID1. Requires at least 4 disks.

RAIDZ-1 A variation on RAID-5, single parity. Requires at least 3 disks.

RAIDZ-2 A variation on RAID-5, double parity. Requires at least 4 disks.

RAIDZ-3 A variation on RAID-5, triple parity. Requires at least 5 disks.

The installer automatically partitions the disks, creates a ZFS pool called `rpool`, and installs the root file system on the ZFS subvolume `rpool/ROOT/pve-1`.

Another subvolume called `rpool/data` is created to store VM images. In order to use that with the Proxmox VE tools, the installer creates the following configuration entry in `/etc/pve/storage.cfg`:

```
zfspool: local-zfs
        pool rpool/data
        sparse
        content images,rootdir
```

After installation, you can view your ZFS pool status using the `zpool` command:

```
# zpool status
  pool: rpool
 state: ONLINE
  scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        rpool       ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            sda2    ONLINE       0     0     0
            sdb2    ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            sdc     ONLINE       0     0     0
            sdd     ONLINE       0     0     0

errors: No known data errors
```

The `zfs` command is used configure and manage your ZFS file systems. The following command lists all file systems after installation:

```
# zfs list
NAME               USED  AVAIL  REFER  MOUNTPOINT
rpool             4.94G  7.68T    96K  /rpool
rpool/ROOT         702M  7.68T    96K  /rpool/ROOT
rpool/ROOT/pve-1   702M  7.68T   702M  /
rpool/data          96K  7.68T    96K  /rpool/data
rpool/swap        4.25G  7.69T    64K  -
```

ZFS RAID Level Considerations

There are a few factors to take into consideration when choosing the layout of a ZFS pool. The basic building block of a ZFS pool is the virtual device, or `vdev`. All vdevs in a pool are used equally and the data is striped among them (RAID0). Check the `zpool(8)` manpage for more details on vdevs.

Performance

Each `vdev` type has different performance behaviors. The two parameters of interest are the IOPS (Input/Output Operations per Second) and the bandwidth with which data can be written or read.

A *mirror* vdev (RAID1) will approximately behave like a single disk in regard to both parameters when writing data. When reading data the performance will scale linearly with the number of disks in the mirror.

A common situation is to have 4 disks. When setting it up as 2 mirror vdevs (RAID10) the pool will have the write characteristics as two single disks in regard to IOPS and bandwidth. For read operations it will resemble 4 single disks.

A *RAIDZ* of any redundancy level will approximately behave like a single disk in regard to IOPS with a lot of bandwidth. How much bandwidth depends on the size of the RAIDZ vdev and the redundancy level.

For running VMs, IOPS is the more important metric in most situations.

Size, Space usage and Redundancy

While a pool made of *mirror* vdevs will have the best performance characteristics, the usable space will be 50% of the disks available. Less if a mirror vdev consists of more than 2 disks, for example in a 3-way mirror. At least one healthy disk per mirror is needed for the pool to stay functional.

The usable space of a *RAIDZ* type vdev of N disks is roughly N-P, with P being the RAIDZ-level. The RAIDZ-level indicates how many arbitrary disks can fail without losing data. A special case is a 4 disk pool with RAIDZ2. In this situation it is usually better to use 2 mirror vdevs for the better performance as the usable space will be the same.

Another important factor when using any RAIDZ level is how ZVOL datasets, which are used for VM disks, behave. For each data block the pool needs parity data which is at least the size of the minimum block size defined by the `ashift` value of the pool. With an ashift of 12 the block size of the pool is 4k. The default block size for a ZVOL is 8k. Therefore, in a RAIDZ2 each 8k block written will cause two additional 4k parity blocks to be written, 8k + 4k + 4k = 16k. This is of course a simplified approach and the real situation will be slightly different with metadata, compression and such not being accounted for in this example.

This behavior can be observed when checking the following properties of the ZVOL:

- `volsize`
- `refreservation` (if the pool is not thin provisioned)
- `used` (if the pool is thin provisioned and without snapshots present)

```
# zfs get volsize,refreservation,used <pool>/vm-<vmid>-disk-X
```

`volsize` is the size of the disk as it is presented to the VM, while `refreservation` shows the reserved space on the pool which includes the expected space needed for the parity data. If the pool is thin provisioned, the `refreservation` will be set to 0. Another way to observe the behavior is to compare the used disk space within the VM and the `used` property. Be aware that snapshots will skew the value.

There are a few options to counter the increased use of space:

- Increase the `volblocksize` to improve the data to parity ratio
- Use *mirror* vdevs instead of *RAIDZ*
- Use `ashift=9` (block size of 512 bytes)

The `volblocksize` property can only be set when creating a ZVOL. The default value can be changed in the storage configuration. When doing this, the guest needs to be tuned accordingly and depending on the use case, the problem of write amplification is just moved from the ZFS layer up to the guest.

Using `ashift=9` when creating the pool can lead to bad performance, depending on the disks underneath, and cannot be changed later on.

Mirror vdevs (RAID1, RAID10) have favorable behavior for VM workloads. Use them, unless your environment has specific needs and characteristics where RAIDZ performance characteristics are acceptable.

ZFS dRAID

In a ZFS dRAID (declustered RAID) the hot spare drive(s) participate in the RAID. Their spare capacity is reserved and used for rebuilding when one drive fails. This provides, depending on the configuration, faster rebuilding compared to a RAIDZ in case of drive failure. More information can be found in the official OpenZFS documentation. [1]


dRAID is intended for more than 10-15 disks in a dRAID. A RAIDZ setup should be better for a lower amount of disks in most use cases.


The GUI requires one more disk than the minimum (i.e. dRAID1 needs 3). It expects that a spare disk is added as well.

- `dRAID1` or `dRAID`: requires at least 2 disks, one can fail before data is lost
- `dRAID2`: requires at least 3 disks, two can fail before data is lost
- `dRAID3`: requires at least 4 disks, three can fail before data is lost

Additional information can be found on the manual page:

```
# man zpoolconcepts
```

Spares and Data

The number of `spares` tells the system how many disks it should keep ready in case of a disk failure. The default value is 0 `spares`. Without spares, rebuilding won't get any speed benefits.

`data` defines the number of devices in a redundancy group. The default value is 8. Except when `disks - parity - spares` equal something less than 8, the lower number is used. In general, a smaller number of `data` devices leads to higher IOPS, better compression ratios and faster resilvering, but defining fewer data devices reduces the available storage capacity of the pool.

Bootloader

Proxmox VE uses `proxmox-boot-tool` to manage the bootloader configuration. See the chapter on Proxmox VE host bootloaders for details.

ZFS Administration

This section gives you some usage examples for common tasks. ZFS itself is really powerful and provides many options. The main commands to manage ZFS are `zfs` and `zpool`. Both commands come with great manual pages, which can be read with:

```
# man zpool
# man zfs
```

Create a new zpool

To create a new pool, at least one disk is needed. The `ashift` should have the same sector-size (2 power of `ashift`) or larger as the underlying disk.

```
# zpool create -f -o ashift=12 <pool> <device>
```

Pool names must adhere to the following rules:

- begin with a letter (a-z or A-Z)
- contain only alphanumeric, `-`, `_`, `.`, `:` or ` ` (space) characters
- must **not begin** with one of `mirror`, `raidz`, `draid` or `spare`
- must not be `log`

To activate compression (see section Compression in ZFS):

```
# zfs set compression=lz4 <pool>
```

Create a new pool with RAID-0

Minimum 1 disk

```
# zpool create -f -o ashift=12 <pool> <device1> <device2>
```

Create a new pool with RAID-1

Minimum 2 disks

```
# zpool create -f -o ashift=12 <pool> mirror <device1> <device2>
```

Create a new pool with RAID-10

Minimum 4 disks

```
# zpool create -f -o ashift=12 <pool> mirror <device1> <device2> mirror <device3> <device4>
```

Create a new pool with RAIDZ-1

Minimum 3 disks

```
# zpool create -f -o ashift=12 <pool> raidz1 <device1> <device2> <device3>
```

Create a new pool with RAIDZ-2

Minimum 4 disks

```
# zpool create -f -o ashift=12 <pool> raidz2 <device1> <device2> <device3> <device4>
```

Please read the section for ZFS RAID Level Considerations to get a rough estimate on how IOPS and bandwidth expectations before setting up a pool, especially when wanting to use a RAID-Z mode.

Create a new pool with cache (L2ARC)

It is possible to use a dedicated device, or partition, as second-level cache to increase the performance. Such a cache device will especially help with random-read workloads of data that is mostly static. As it acts as additional caching layer between the actual storage, and the in-memory ARC, it can also help if the ARC must be reduced due to memory constraints.

Create ZFS pool with a on-disk cache

```
# zpool create -f -o ashift=12 <pool> <device> cache <cache-device>
```

Here only a single `<device>` and a single `<cache-device>` was used, but it is possible to use more devices, like it's shown in Create a new pool with RAID.

Note that for cache devices no mirror or raid modi exist, they are all simply accumulated.

If any cache device produces errors on read, ZFS will transparently divert that request to the underlying storage layer.

Create a new pool with log (ZIL)

It is possible to use a dedicated drive, or partition, for the ZFS Intent Log (ZIL), it is mainly used to provide safe synchronous transactions, so often in performance critical paths like databases, or other programs that issue `fsync` operations more frequently.

The pool is used as default ZIL location, diverting the ZIL IO load to a separate device can, help to reduce transaction latencies while relieving the main pool at the same time, increasing overall performance.

For disks to be used as log devices, directly or through a partition, it's recommend to:

- use fast SSDs with power-loss protection, as those have much smaller commit latencies.
- Use at least a few GB for the partition (or whole device), but using more than half of your installed memory won't provide you with any real advantage.

Create ZFS pool with separate log device

```
# zpool create -f -o ashift=12 <pool> <device> log <log-device>
```

In above example a single `<device>` and a single `<log-device>` is used, but you can also combine this with other RAID variants, as described in the Create a new pool with RAID section.

You can also mirror the log device to multiple devices, this is mainly useful to ensure that performance doesn't immediately degrades if a single log device fails.

If all log devices fail the ZFS main pool itself will be used again, until the log device(s) get replaced.

Add cache and log to an existing pool

If you have a pool without cache and log you can still add both, or just one of them, at any time.

For example, let's assume you got a good enterprise SSD with power-loss protection that you want to use for improving the overall performance of your pool.

As the maximum size of a log device should be about half the size of the installed physical memory, it means that the ZIL will mostly likely only take up a relatively small part of the SSD, the remaining space can be

used as cache.

First you have to create two GPT partitions on the SSD with `parted` or `gdisk`.

Then you're ready to add them to an pool:

Add both, a separate log device and a second-level cache, to an existing pool

```
# zpool add -f <pool> log <device-part1> cache <device-part2>
```

Just replay `<pool>`, `<device-part1>` and `<device-part2>` with the pool name and the two `/dev/disk/by-id/` paths to the partitions.

You can also add ZIL and cache separately.

Add a log device to an existing ZFS pool

```
# zpool add <pool> log <log-device>
```

Changing a failed device

```
# zpool replace -f <pool> <old-device> <new-device>
```

Changing a failed bootable device

Depending on how Proxmox VE was installed it is either using `systemd-boot` or `grub` through `proxmox-boot-tool` [2] or plain `grub` as bootloader (see Host Bootloader). You can check by running:

```
# proxmox-boot-tool status
```

The first steps of copying the partition table, reissuing GUIDs and replacing the ZFS partition are the same. To make the system bootable from the new disk, different steps are needed which depend on the bootloader in use.

```
# sgdisk <healthy bootable device> -R <new device>
# sgdisk -G <new device>
# zpool replace -f <pool> <old zfs partition> <new zfs partition>
```

Use the `zpool status -v` command to monitor how far the resilvering process of the new disk has progressed.

With `proxmox-boot-tool`:

```
# proxmox-boot-tool format <new disk's ESP>
# proxmox-boot-tool init <new disk's ESP> [grub]
```

`ESP` stands for EFI System Partition, which is setup as partition #2 on bootable disks setup by the Proxmox VE installer since version 5.4. For details, see Setting up a new partition for use as synced ESP.

make sure to pass *grub* as mode to `proxmox-boot-tool init` if `proxmox-boot-tool status` indicates your current disks are using Grub, especially if Secure Boot is enabled!

With plain `grub`:

```
# grub-install <new disk>
```

plain `grub` is only used on systems installed with Proxmox VE 6.3 or earlier, which have not been manually migrated to using `proxmox-boot-tool` yet.

Configure E-Mail Notification

ZFS comes with an event daemon `ZED`, which monitors events generated by the ZFS kernel module. The daemon can also send emails on ZFS events like pool errors. Newer ZFS packages ship the daemon in a separate `zfs-zed` package, which should already be installed by default in Proxmox VE.

You can configure the daemon via the file `/etc/zfs/zed.d/zed.rc` with your favorite editor. The required setting for email notification is `ZED_EMAIL_ADDR`, which is set to `root` by default.

```
ZED_EMAIL_ADDR="root"
```

Please note Proxmox VE forwards mails to `root` to the email address configured for the root user.

Limit ZFS Memory Usage

ZFS uses *50 %* of the host memory for the **A**daptive **R**eplacement **C**ache (ARC) by default. Allocating enough memory for the ARC is crucial for IO performance, so reduce it with caution. As a general rule of thumb, allocate at least `2 GiB Base + 1 GiB/TiB-Storage`. For example, if you have a pool with `8 TiB` of available storage space then you should use `10 GiB` of memory for the ARC.

You can change the ARC usage limit for the current boot (a reboot resets this change again) by writing to the `zfs_arc_max` module parameter directly:

```
 echo "$[10 * 1024*1024*1024]" >/sys/module/zfs/parameters/zfs_arc_max
```

To **permanently change** the ARC limits, add the following line to `/etc/modprobe.d/zfs.conf`:

```
options zfs zfs_arc_max=8589934592
```

This example setting limits the usage to 8 GiB ($8 * 2^{30}$).

In case your desired `zfs_arc_max` value is lower than or equal to `zfs_arc_min` (which defaults to 1/32 of the system memory), `zfs_arc_max` will be ignored unless you also set `zfs_arc_min` to at most `zfs_arc_max - 1`.

```
echo "$[8 * 1024*1024*1024 - 1]" >/sys/module/zfs/parameters/zfs_arc_min
echo "$[8 * 1024*1024*1024]" >/sys/module/zfs/parameters/zfs_arc_max
```

This example setting (temporarily) limits the usage to 8 GiB ($8 * 2^{30}$) on systems with more than 256 GiB of total memory, where simply setting `zfs_arc_max` alone would not work.

If your root file system is ZFS, you must update your initramfs every time this value changes:

```
# update-initramfs -u -k all
```

You **must reboot** to activate these changes.

SWAP on ZFS

Swap-space created on a zvol may generate some troubles, like blocking the server or generating a high IO load, often seen when starting a Backup to an external Storage.

We strongly recommend to use enough memory, so that you normally do not run into low memory situations. Should you need or want to add swap, it is preferred to create a partition on a physical disk and use it as a swap device. You can leave some space free for this purpose in the advanced options of the installer. Additionally, you can lower the "swappiness" value. A good value for servers is 10:

```
# sysctl -w vm.swappiness=10
```

To make the swappiness persistent, open `/etc/sysctl.conf` with an editor of your choice and add the following line:

```
vm.swappiness = 10
```

Table 1. Linux kernel swappiness parameter values

Value	Strategy
`vm.swappiness = 0`	The kernel will swap only to avoid an *out of memory* condition
`vm.swappiness = 1`	Minimum amount of swapping without disabling it entirely.
`vm.swappiness = 10`	This value is sometimes recommended to improve performance when sufficient memory exists in a system.
`vm.swappiness = 60`	The default value.
`vm.swappiness = 100`	The kernel will swap aggressively.

Encrypted ZFS Datasets

Native ZFS encryption in Proxmox VE is experimental. Known limitations and issues include Replication with encrypted datasets [3], as well as checksum errors when using Snapshots or ZVOLs. [4]

ZFS on Linux version 0.8.0 introduced support for native encryption of datasets. After an upgrade from previous ZFS on Linux versions, the encryption feature can be enabled per pool:

```
# zpool get feature@encryption tank
NAME  PROPERTY            VALUE            SOURCE
tank  feature@encryption  disabled         local

# zpool set feature@encryption=enabled

# zpool get feature@encryption tank
NAME  PROPERTY            VALUE            SOURCE
tank  feature@encryption  enabled         local
```

There is currently no support for booting from pools with encrypted datasets using Grub, and only limited support for automatically unlocking encrypted datasets on boot. Older versions of ZFS without encryption support will not be able to decrypt stored data.

It is recommended to either unlock storage datasets manually after booting, or to write a custom unit to pass the key material needed for unlocking on boot to `zfs load-key`.

Establish and test a backup procedure before enabling encryption of production data. If the associated key material/passphrase/keyfile has been lost, accessing the encrypted data is no longer possible.

Encryption needs to be setup when creating datasets/zvols, and is inherited by default to child datasets. For example, to create an encrypted dataset `tank/encrypted_data` and configure it as storage in Proxmox VE, run the following commands:

```
# zfs create -o encryption=on -o keyformat=passphrase tank/encrypted_data
Enter passphrase:
Re-enter passphrase:

# pvesm add zfspool encrypted_zfs -pool tank/encrypted_data
```

All guest volumes/disks create on this storage will be encrypted with the shared key material of the parent dataset.

To actually use the storage, the associated key material needs to be loaded and the dataset needs to be mounted. This can be done in one step with:

```
# zfs mount -l tank/encrypted_data
Enter passphrase for 'tank/encrypted_data':
```

It is also possible to use a (random) keyfile instead of prompting for a passphrase by setting the `keylocation` and `keyformat` properties, either at creation time or with `zfs change-key` on existing datasets:

```
# dd if=/dev/urandom of=/path/to/keyfile bs=32 count=1

# zfs change-key -o keyformat=raw -o keylocation=file:///path/to/keyfile tank/encrypted_data
```

When using a keyfile, special care needs to be taken to secure the keyfile against unauthorized access or accidental loss. Without the keyfile, it is not possible to access the plaintext data!

A guest volume created underneath an encrypted dataset will have its `encryptionroot` property set accordingly. The key material only needs to be loaded once per encryptionroot to be available to all encrypted datasets underneath it.

See the `encryptionroot`, `encryption`, `keylocation`, `keyformat` and `keystatus` properties, the `zfs load-key`, `zfs unload-key` and `zfs change-key` commands and the `Encryption` section from `man zfs` for more details and advanced usage.

Compression in ZFS

When compression is enabled on a dataset, ZFS tries to compress all **new** blocks before writing them and decompresses them on reading. Already existing data will not be compressed retroactively.

You can enable compression with:

```
# zfs set compression=<algorithm> <dataset>
```

We recommend using the `lz4` algorithm, because it adds very little CPU overhead. Other algorithms like `lzjb` and `gzip-N`, where `N` is an integer from `1` (fastest) to `9` (best compression ratio), are also available. Depending on the algorithm and how compressible the data is, having compression enabled can even increase I/O performance.

You can disable compression at any time with:

```
# zfs set compression=off <dataset>
```

Again, only new blocks will be affected by this change.

ZFS Special Device

Since version 0.8.0 ZFS supports `special` devices. A `special` device in a pool is used to store metadata, deduplication tables, and optionally small file blocks.

A `special` device can improve the speed of a pool consisting of slow spinning hard disks with a lot of metadata changes. For example workloads that involve creating, updating or deleting a large number of files will benefit from the presence of a `special` device. ZFS datasets can also be configured to store whole small files on the `special` device which can further improve the performance. Use fast SSDs for the `special` device.

The redundancy of the `special` device should match the one of the pool, since the `special` device is a point of failure for the whole pool.

Adding a `special` device to a pool cannot be undone!

Create a pool with `special` device and RAID-1:

```
# zpool create -f -o ashift=12 <pool> mirror <device1> <device2> special mirror <device3> <device4>
```

Add a `special` device to an existing pool with RAID-1:

```
# zpool add <pool> special mirror <device1> <device2>
```

ZFS datasets expose the `special_small_blocks=<size>` property. `size` can be `0` to disable storing small file blocks on the `special` device or a power of two in the range between `512B` to `1M`. After setting the property new file blocks smaller than `size` will be allocated on the `special` device.

If the value for `special_small_blocks` is greater than or equal to the `recordsize` (default `128K`) of the dataset, **all** data will be written to the `special` device, so be careful!

Setting the `special_small_blocks` property on a pool will change the default value of that property for all child ZFS datasets (for example all containers in the pool will opt in for small file blocks).

Opt in for all file smaller than 4K-blocks pool-wide:

```
# zfs set special_small_blocks=4K <pool>
```

Opt in for small file blocks for a single dataset:

```
# zfs set special_small_blocks=4K <pool>/<filesystem>
```

Opt out from small file blocks for a single dataset:

```
# zfs set special_small_blocks=0 <pool>/<filesystem>
```

ZFS Pool Features

Changes to the on-disk format in ZFS are only made between major version changes and are specified through **features**. All features, as well as the general mechanism are well documented in the `zpool-features(5)` manpage.

Since enabling new features can render a pool not importable by an older version of ZFS, this needs to be done actively by the administrator, by running `zpool upgrade` on the pool (see the `zpool-upgrade(8)` manpage).

Unless you need to use one of the new features, there is no upside to enabling them.

In fact, there are some downsides to enabling new features:

- A system with root on ZFS, that still boots using `grub` will become unbootable if a new feature is active on the rpool, due to the incompatible implementation of ZFS in grub.
- The system will not be able to import any upgraded pool when booted with an older kernel, which still ships with the old ZFS modules.
- Booting an older Proxmox VE ISO to repair a non-booting system will likewise not work.

Do **not** upgrade your rpool if your system is still booted with `grub`, as this will render your system unbootable. This includes systems installed before Proxmox VE 5.4, and systems booting with legacy BIOS boot (see how to determine the bootloader).

Enable new features for a ZFS pool:

```
# zpool upgrade <pool>
```

1. OpenZFS dRAID https://openzfs.github.io/openzfs-docs/Basic%20Concepts/dRAID%20Howto.html
2. Systems installed with Proxmox VE 6.4 or later, EFI systems installed with Proxmox VE 5.4 or later
3. https://bugzilla.proxmox.com/show_bug.cgi?id=2350
4. https://github.com/openzfs/zfs/issues/11688