ZFS

ZFS is an advanced filesystem created by **Sun Microsystems** (now owned by Oracle) and released for OpenSolaris in November 2005.

Related articles

File systems

ZFS/Virtual disks

Installing Arch Linux on ZFS

Features of ZFS include: pooled storage (integrated volume management – zpool), **Copy-on-write**, **snapshots**, data integrity verification and automatic repair (scrubbing), **RAID-Z**, a maximum **16 Exabyte** file size, and a maximum 256 Quadrillion **Zettabytes** storage with no limit on number of filesystems (datasets) or files**[1] (http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/index.html)**. ZFS is licensed under the **Common Development and Distribution License** (CDDL).

Described as **"The last word in filesystems" (http://web.archive.org/web/20060428092023/http://www.sun.com/2004-0914/feature/)** ZFS is stable, fast, secure, and future-proof. Being licensed under the CDDL, and thus incompatible with GPL, it is not possible for ZFS to be distributed along with the Linux Kernel. This requirement, however, does not prevent a native Linux kernel module from being developed and distributed by a third party, as is the case with **zfsonlinux.org (http://zfsonlinux.org/)** (ZOL).

ZOL is a project funded by the **Lawrence Livermore National Laboratory (https://www.llnl.gov/)** to develop a native Linux kernel module for its massive storage requirements and super computers.

Note: Due to potential legal incompatibilities between CDDL license of ZFS code and GPL of the Linux kernel (**[2] (https://sfconservancy.org/blog/2016/feb/25/zfs-and-linux/)**,**CDDL-GPL**,**ZFS in Linux**) - ZFS development is not supported by the kernel.

As a result:

- ZFSonLinux project must keep up with Linux kernel versions. After making stable ZFSonLinux release - Arch ZFS maintainers release them.
- This situation sometimes locks down the normal rolling update process by unsatisfied dependencies because the new kernel version, proposed by update, is unsupported by ZFSonLinux.

Contents

Installation

General

Warning: Unless you use the **dkms** versions of these packages, the ZFS and SPL kernel modules are tied to a specific kernel version. It would not be possible to apply any kernel updates until updated packages are uploaded to AUR or the **archzfs** repository.

Tip: You can **downgrade** your linux version to the one from **archzfs** repo if your current kernel is newer.

Install from the **Arch User Repository** or the **archzfs** repository:

- `zfs-linux (https://aur.archlinux.org/packages/zfs-linux/)`[AUR] for **stable (http://zfsonlinux.org/)** releases.
- `zfs-linux-git (https://aur.archlinux.org/packages/zfs-linux-git/)`[AUR] for **development (https://github.com/zfsonlinux/zfs/releases)** releases (with support of newer kernel versions).
- `zfs-linux-lts (https://aur.archlinux.org/packages/zfs-linux-lts/)`[AUR] for stable releases for LTS kernels.
- `zfs-linux-lts-git (https://aur.archlinux.org/packages/zfs-linux-lts-git/)`[AUR] for **development (https://github.com/zfsonlinux/zfs/releases)** releases for LTS kernels.
- `zfs-linux-hardened (https://aur.archlinux.org/packages/zfs-linux-hardened/)`[AUR] for stable releases for hardened kernels.
- `zfs-linux-hardened-git (https://aur.archlinux.org/packages/zfs-linux-hardened-git/)`[AUR] for **development (https://github.com/zfsonlinux/zfs/releases)** releases for hardened kernels.
- `zfs-linux-zen (https://aur.archlinux.org/packages/zfs-linux-zen/)`[AUR] for stable releases for zen kernels.
- `zfs-linux-zen-git (https://aur.archlinux.org/packages/zfs-linux-zen-git/)`[AUR] for **development (https://github.com/zfsonlinux/zfs/releases)** releases for zen kernels.
- `zfs-dkms (https://aur.archlinux.org/packages/zfs-dkms/)`[AUR] for versions with dynamic kernel module support.
- `zfs-dkms-git (https://aur.archlinux.org/packages/zfs-dkms-git/)`[AUR] for **development (https://github.com/zfsonlinux/zfs/releases)** releases for versions with dynamic kernel module support.

These branches have (according to them) dependencies on the `zfs-utils`, `spl`, `spl-utils` packages. SPL (Solaris Porting Layer) is a Linux Kernel module implementing Solaris APIs for ZFS compatibility.

Test the installation by issuing `zpool status` on the command line. If an "insmod" error is produced, try `depmod -a`.

Root on ZFS

When performing an Arch install on ZFS, **zfs-linux (https://aur.archlinux.org/packages/zfs-linux/)**[AUR] or **zfs-dkms (https://aur.archlinux.org/packages/zfs-dkms/)**[AUR] and its dependencies can be installed in the archiso environment as outlined in the previous section.

It may be useful to prepare a **customized archiso** with ZFS support builtin. For a much more detailed guide on installing Arch with ZFS as its root file system, see **Installing Arch Linux on ZFS**.

DKMS

Users can make use of DKMS **Dynamic Kernel Module Support** to rebuild the ZFS modules automatically with every kernel upgrade.

Install **zfs-dkms (https://aur.archlinux.org/packages/zfs-dkms/)**[AUR] or **zfs-dkms-git (https://aur.archlinux.org/packages/zfs-dkms-git/)**[AUR] and apply the post-install instructions given by these packages.

> **Tip:** Add an `IgnorePkg` entry to **pacman.conf** to prevent these packages from upgrading when doing a regular update.

> **Note:** Pacman does not take dependencies into consideration when rebuilding DKMS modules. This will result in build failures when pacman tries to rebuild DKMS modules after a kernel upgrade. See bug report **FS#52901 (https://bugs.archlinux.org/task/52901)** for details. The **dkms-sorted (https://aur.archlinux.org/packages/dkms-sorted/)**[AUR] package adds experimental support for such dependencies; technically, it is a drop-in replacement for the `dkms` package. The most convenient way to try out dkms-sorted is to install it *before* you install any DKMS modules.

Experimenting with ZFS

Users wishing to experiment with ZFS on *virtual block devices* (known in ZFS terms as VDEVs) which can be simple files like `~/zfs0.img ~/zfs1.img ~/zfs2.img` etc. with no possibility of real data loss are encouraged to see the **Experimenting with ZFS** article. Common tasks like building a RAIDZ array, purposefully corrupting data and recovering it, snapshotting datasets, etc. are covered.

Configuration

ZFS is considered a "zero administration" filesystem by its creators; therefore, configuring ZFS is very straight forward. Configuration is done primarily with two commands: `zfs` and `zpool`.

Automatic Start

For ZFS to live by its "zero administration" namesake, the zfs daemon must be loaded at startup. A benefit to this is that it is not necessary to mount the zpool in `/etc/fstab`; the zfs daemon can import and mount zfs pools automatically. The daemon mounts the zfs pools reading the file `/etc/zfs/zpool.cache`.

For each pool you want automatically mounted by the zfs daemon execute:

```
# zpool set cachefile=/etc/zfs/zpool.cache <pool>
```

Enable the service so it is automatically started at boot time:

```
# systemctl enable zfs.target
```

To manually start the daemon:

```
# systemctl start zfs.target
```

Note: Beginning with ZOL version 0.6.5.8 the ZFS service unit files have been changed so that you need to explicitly enable any ZFS services you want to run.

See **https://github.com/archzfs/archzfs/issues/72** for more information.

In order to mount zfs pools automatically on boot you need to enable the following services and targets:

```
# systemctl enable zfs-import-cache
# systemctl enable zfs-mount
# systemctl enable zfs-import.target
```

Creating a storage pool

Use `# parted --list` to see a list of all available drives. It is not necessary nor recommended to partition the drives before creating the zfs filesystem.

Note: If some or all device have been used in a software RAID set it is

paramount to erase any old RAID configuration information. (**Mdadm#Prepare the devices**)

Warning: For Advanced Format Disks with 4KB sector size, an ashift of 12 is recommended for best performance. Advanced Format disks emulate a sector size of 512 bytes for compatibility with legacy systems, this causes ZFS to sometimes use an ashift option number that is not ideal. Once the pool has been created, the only way to change the ashift option is to recreate the pool. Using an ashift of 12 would also decrease available capacity. See **1.10 What's going on with performance? (https://github.com/zfsonlinux/zfs/wiki/faq#performance-considerations)**, **1.15 How does ZFS on Linux handle Advanced Format disks? (https://github.com/zfsonlinux/zfs/wiki/faq#advanced-format-disks)**, and **ZFS and Advanced Format disks (http://wiki.illumos.org/display/illumos/ZFS+and+Advanced+Format+disks)**.

Having identified the list of drives, it is now time to get the IDs of the drives to add to the zpool. The **zfs on Linux developers recommend (https://github.com/zfsonlinux/zfs/wiki/faq#selecting-dev-names-when-creating-a-pool)** using device IDs when creating ZFS storage pools of less than 10 devices. To find the IDs, simply:

```
# ls -lh /dev/disk/by-id/
```

The IDs should look similar to the following:

```
lrwxrwxrwx 1 root root  9 Aug 12 16:26 ata-ST3000DM001-9YN166_S1F0JKRR -> ../../sdc
lrwxrwxrwx 1 root root  9 Aug 12 16:26 ata-ST3000DM001-9YN166_S1F0JTM1 -> ../../sde
lrwxrwxrwx 1 root root  9 Aug 12 16:26 ata-ST3000DM001-9YN166_S1F0KBP8 -> ../../sdd
lrwxrwxrwx 1 root root  9 Aug 12 16:26 ata-ST3000DM001-9YN166_S1F0KDGY -> ../../sdb
```

Warning: If you create zpools using device names (e.g. /dev/sda,/dev/sdb,...) ZFS might not be able to detect zpools intermittently on boot.

Disk labels and UUID can also be used for ZFS mounts by using **GPT** partitions. ZFS drives have labels but Linux is unable to read them at boot. Unlike **MBR** partitions, GPT partitions directly support both UUID and labels independent of the format inside the partition. Partitioning rather than using the whole disk for ZFS offers two additional advantages. The OS does not generate bogus partition numbers from whatever unpredictable data ZFS has written to the partition sector, and if desired, you can easily over provision SSD drives, and slightly over provision spindle drives to ensure that different models with slightly different sector counts can zpool replace into your mirrors. This is a lot of organization and control over ZFS using readily available tools and techniques at almost zero cost.

Use **gdisk** to partition the all or part of the drive as a single partition. gdisk does not automatically name partitions so if partition labels are desired use

gdisk command "c" to label the partitions. Some reasons you might prefer labels over UUID are: labels are easy to control, labels can be titled to make the purpose of each disk in your arrangement readily apparent, and labels are shorter and easier to type. These are all advantages when the server is down and the heat is on. GPT partition labels have plenty of space and can store most international characters **wikipedia:GUID_Partition_Table#Partition_entries** allowing large data pools to be labeled in an organized fashion.

Drives partitioned with GPT have labels and UUID that look like this.

```
# ls -l /dev/disk/by-partlabel
# lrwxrwxrwx 1 root root 10 Apr 30 01:44 zfsdata1 -> ../../sdd1
# lrwxrwxrwx 1 root root 10 Apr 30 01:44 zfsdata2 -> ../../sdc1
# lrwxrwxrwx 1 root root 10 Apr 30 01:59 zfsl2arc -> ../../sda1
```

```
# ls -l /dev/disk/by-partuuid
# lrwxrwxrwx 1 root root 10 Apr 30 01:44 148c462c-7819-431a-9aba-5bf42bb5a34e -> ../../sdd1
# lrwxrwxrwx 1 root root 10 Apr 30 01:59 4f95da30-b2fb-412b-9090-fc349993df56 -> ../../sda1
# lrwxrwxrwx 1 root root 10 Apr 30 01:44 e5ccef58-5adf-4094-81a7-3bac846a885f -> ../../sdc1
```

Now, finally, create the ZFS pool:

```
# zpool create -f -m <mount> <pool> raidz <ids>
```

- **create**: subcommand to create the pool.
- **-f**: Force creating the pool. This is to overcome the "EFI label error". See **#Does not contain an EFI label**.
- **-m**: The mount point of the pool. If this is not specified, then the pool will be mounted to `/<pool>`.
- **pool**: This is the name of the pool.
- **raidz**: This is the type of virtual device that will be created from the pool of devices. Raidz is a special implementation of raid5. See **Jeff Bonwick's Blog -- RAID-Z (https://blogs.oracle.com/bonwick/entry/raid_z)** for more information about raidz. The usage of **mirror** instead may be better when using RAID-1 **[3] (http://blog.programster.org/zfs-create-disk-pools)**.
- **ids**: The names of the drives or partitions that to include into the pool. Get it from `/dev/disk/by-id`.

Here is an example for the full command:

```
# zpool create -f -m /mnt/data bigdata \
               raidz \
                  ata-ST3000DM001-9YN166_S1F0KDGY \
                  ata-ST3000DM001-9YN166_S1F0JKRR \
```

```
                ata-ST3000DM001-9YN166_S1F0KBP8 \
                ata-ST3000DM001-9YN166_S1F0JTM1
```

Advanced Format disks

In case Advanced Format disks are used which have a native sector size of 4096 bytes instead of 512 bytes, the automated sector size detection algorithm of ZFS might detect 512 bytes because of backwards compatibility with legacy systems. This would result in degraded performance. To make sure a correct sector size is used, the `ashift=12` option should be used (See the **ZFS on Linux FAQ (https://github.com/zfsonlinux/zfs/wiki/faq#advanced-format-disks)**). The full command would in this case be:

```
# zpool create -f -o ashift=12 -m /mnt/data bigdata \
               raidz \
                  ata-ST3000DM001-9YN166_S1F0KDGY \
                  ata-ST3000DM001-9YN166_S1F0JKRR \
                  ata-ST3000DM001-9YN166_S1F0KBP8 \
                  ata-ST3000DM001-9YN166_S1F0JTM1
```

Verifying pool creation

If the command is successful, there will be no output. Using the `$ mount` command will show that the pool is mounted. Using `# zpool status` will show that the pool has been created.

```
# zpool status
```

```
  pool: bigdata
 state: ONLINE
 scan: none requested
config:

        NAME                                        STATE     READ WRITE CKSUM
        bigdata                                     ONLINE       0     0     0
          -0                                        ONLINE       0     0     0
            ata-ST3000DM001-9YN166_S1F0KDGY-part1  ONLINE       0     0     0
            ata-ST3000DM001-9YN166_S1F0JKRR-part1  ONLINE       0     0     0
            ata-ST3000DM001-9YN166_S1F0KBP8-part1  ONLINE       0     0     0
            ata-ST3000DM001-9YN166_S1F0JTM1-part1  ONLINE       0     0     0

errors: No known data errors
```

At this point it would be good to reboot the machine to ensure that the ZFS pool is mounted at boot. It is best to deal with all errors before transferring data.

GRUB-compatible pool creation

Note: This section frequently goes out of date with updates to GRUB and ZFS. Consult the manual pages for the most up-to-date information.

By default, *zpool create* enables all features on a pool. If `/boot` resides on ZFS when using **GRUB** you must only enable features supported by GRUB otherwise GRUB will not be able to read the pool. GRUB 2.02 supports the read-write features `lz4_compress`, `hole_birth`, `embedded_data`, `extensible_dataset`, and `large_blocks`; this is not suitable for all the features of ZFSonLinux 0.7.1, and must have unsupported features disabled.

You can create a pool with the incompatible features disabled:

```
# zpool create -o feature@multi_vdev_crash_dump=disabled \
               -o feature@large_dnode=disabled           \
               -o feature@sha512=disabled                \
               -o feature@skein=disabled                 \
               -o feature@edonr=disabled                 \
               $POOL_NAME $VDEVS
```

When running the git version of ZFS on Linux, make sure to also add `-o feature@encryption=disabled`.

Importing a pool created by id

Eventually a pool may fail to auto mount and you need to import to bring your pool back. Take care to avoid the most obvious solution.

```
# ###zpool import zfsdata # Do not do this! Always use -d
```

This will import your pools using `/dev/sd?` which will lead to problems the next time you rearrange your drives. This may be as simple as rebooting with a USB drive left in the machine, which harkens back to a time when PCs would not boot when a floppy disk was left in a machine. Adapt one of the following commands to import your pool so that pool imports retain the persistence they were created with.

```
# zpool import -d /dev/disk/by-id zfsdata
# zpool import -d /dev/disk/by-partlabel zfsdata
# zpool import -d /dev/disk/by-partuuid zfsdata
```

Tuning

General

Many parameters are available for zfs file systems, you can view a full list with `zfs get all <pool>`. Two common ones to adjust are **atime** and **compression**.

Atime is enabled by default but for most users, it represents superfluous writes to the zpool and it can be disabled using the zfs command:

```
# zfs set atime=off <pool>
```

As an alternative to turning off atime completely, **relatime** is available. This brings the default ext4/xfs atime semantics to ZFS, where access time is only updated if the modified time or changed time changes, or if the existing access time has not been updated within the past 24 hours. It is a compromise between atime=off and atime=on. This property *only* takes effect if **atime** is **on**:

```
# zfs set relatime=on <pool>
```

Compression is just that, transparent compression of data. ZFS supports a few different algorithms, presently lz4 is the default. **gzip** is also available for seldom-written yet highly-compressable data; consult the man page for more details. Enable compression using the zfs command:

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

Other options for zfs can be displayed again, using the zfs command:

```
# zfs get all <pool>
```

SSD Caching

You can also add SSD devices as a write intent log (external ZIL or SLOG) and also as a layer 2 adaptive replacement cache (l2arc). The process to add them is very similar to creating a new VDEV.

All of the below references to device-id are the IDs from `/dev/disk/by-id/*`.

To add a ZIL:

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

Or to add a mirrored ZIL:

```
# zpool add <pool> log mirror <device-id-1> <device-id-2>
```

To add an l2arc:

```
# zpool add <pool> cache <device-id>
```

Or to add a mirrored l2arc:

```
# zpool add <pool> cache mirror <device-id-1> <device-id-2>
```

Database

ZFS, unlike most other file systems, has a variable record size, or what is commonly referred to as a block size. By default, the recordsize on ZFS is 128KiB, which means it will dynamically allocate blocks of any size from 512B to 128KiB depending on the size of file being written. This can often help fragmentation and file access, at the cost that ZFS would have to allocate new 128KiB blocks each time only a few bytes are written to.

Most RDBMSes work in 8KiB-sized blocks by default. Although the block size is tunable for **MySQL/MariaDB**, **PostgreSQL**, and **Oracle**, all three of them use an 8KiB block size *by default*. For both performance concerns and keeping snapshot differences to a minimum (for backup purposes, this is helpful), it is usually desirable to tune ZFS instead to accommodate the databases, using a command such as:

```
# zfs set recordsize=8K <pool>/postgres
```

These RDBMSes also tend to implement their own caching algorithm, often similar to ZFS's own ARC. In the interest of saving memory, it is best to simply disable ZFS's caching of the database's file data and let the database do its own job:

```
# zfs set primarycache=metadata <pool>/postgres
```

If your pool has no configured log devices, ZFS reserves space on the pool's data disks for its intent log (the ZIL). ZFS uses this for crash recovery, but databases are often syncing their data files to the file system on their own transaction commits anyway. The end result of this is that ZFS will be committing data **twice** to the data disks, and it can severely impact performance. You can tell ZFS to prefer to not use the ZIL, and in which case, data is only committed to the file system once. Setting this for non-database file systems, or for pools with configured log devices, can actually *negatively* impact the performance, so beware:

```
# zfs set logbias=throughput <pool>/postgres
```

These can also be done at file system creation time, for example:

```
# zfs create -o recordsize=8K \
             -o primarycache=metadata \
             -o mountpoint=/var/lib/postgres \
             -o logbias=throughput \
              <pool>/postgres
```

Please note: these kinds of tuning parameters are ideal for specialized applications like RDBMSes. You can easily *hurt* ZFS's performance by setting these on a general-purpose file system such as your /home directory.

/tmp

If you would like to use ZFS to store your /tmp directory, which may be useful for storing arbitrarily-large sets of files or simply keeping your RAM free of idle data, you can generally improve performance of certain applications writing to /tmp by disabling file system sync. This causes ZFS to ignore an application's sync requests (eg, with `fsync` or `O_SYNC`) and return immediately. While this has severe application-side data consistency consequences (never disable sync for a database!), files in /tmp are less likely to be important and affected. Please note this does *not* affect the integrity of ZFS itself, only the possibility that data an application expects on-disk may not have actually been written out following a crash.

```
# zfs set sync=disabled <pool>/tmp
```

Additionally, for security purposes, you may want to disable **setuid** and **devices** on the /tmp file system, which prevents some kinds of privilege-escalation attacks or the use of device nodes:

```
# zfs set setuid=off <pool>/tmp
# zfs set devices=off <pool>/tmp
```

Combining all of these for a create command would be as follows:

```
# zfs create -o setuid=off -o devices=off -o sync=disabled -o mountpoint=/tmp <pool>/tmp
```

Please note, also, that if you want /tmp on ZFS, you will need to mask (disable) **systemd**'s automatic tmpfs-backed /tmp, else ZFS will be unable to mount your dataset at boot-time or import-time:

```
# systemctl mask tmp.mount
```

ZVOLs

ZFS volumes (ZVOLs) can suffer from the same block size-related issues as RDBMSes, but it is worth noting that the default recordsize for ZVOLs is 8KiB already. If possible, it is best to align any partitions contained in a ZVOL to your recordsize (current versions of fdisk and gdisk by default automatically align at 1MiB segments, which works), and file system block sizes to the same size. Other than this, you might tweak the **recordsize** to accommodate the data inside the ZVOL as necessary (though 8KiB tends to be a good value for most file systems, even when using 4KiB blocks on that level).

RAIDZ and Advanced Format physical disks

Each block of a ZVOL gets its own parity disks, and if you have physical media

with logical block sizes of 4096B, 8192B, or so on, the parity needs to be stored in whole physical blocks, and this can drastically increase the space requirements of a ZVOL, requiring 2× or more physical storage capacity than the ZVOL's logical capacity. Setting the **recordsize** to 16k or 32k can help reduce this footprint drastically.

See **ZFS on Linux issue #1807 for details (https://github.com/zfsonlinux/zfs/issues/1807)**

Usage

Users can optionally create a dataset under the zpool as opposed to manually creating directories under the zpool. Datasets allow for an increased level of control (quotas for example) in addition to snapshots. To be able to create and mount a dataset, a directory of the same name must not pre-exist in the zpool. To create a dataset, use:

```
# zfs create <nameofzpool>/<nameofdataset>
```

It is then possible to apply ZFS specific attributes to the dataset. For example, one could assign a quota limit to a specific directory within a dataset:

```
# zfs set quota=20G <nameofzpool>/<nameofdataset>/<directory>
```

To see all the commands available in ZFS, use :

```
$ man zfs
```

or:

```
$ man zpool
```

Native encryption

Native ZFS encryption has been made available in 0.7.0.r26 or newer provided by packages like **zfs-linux-git (https://aur.archlinux.org/packages/zfs-linux-git/)**[AUR], **zfs-dkms-git (https://aur.archlinux.org/packages/zfs-dkms-git/)**[AUR] or other development builds. Despite the fact that version 0.7 has been released, this feature is still not enabled in the stable version as of 0.7.3, so a development build still needs to be used. An easy of telling if encryption is available in the version of zfs, you have installed, is to check for the ZFS_PROP_ENCRYPTION definition in /usr/src/zfs-*/include/sys/fs/zfs.h.

- Supported encryption options: `aes-128-ccm`, `aes-192-ccm`, `aes-256-ccm`, `aes-128-gcm`, `aes-192-gcm` and `aes-256-gcm`. When encryption is set to `on`, `aes-256-ccm` will be used.

- Supported keyformats: `passphrase`, `raw`, `hex`

You can also specify iterations of PBKDF2 with `-o pbkdf2iters <n>` (it takes time to decrypt the key)

To create a dataset including native encryption with a passphrase, use:

```
# zfs create -o encryption=on -o keyformat=passphrase <nameofzpool>/<nameofdataset>
```

To use a key instead of using a passphrase:

```
# dd if=/dev/urandom of=/path/to/key bs=1 count=32
# zfs create -o encryption=on -o keyformat=raw -o keylocation=file:///path/to/key <nameofzpool>/<name
ofdataset>
```

You can also manually load the keys and then mount the encrypted dataset:

```
# zfs load-key <nameofzpool>/<nameofdataset> # load key for a specific dataset
# zfs load-key -a # load all keys
# zfs load-key -r zpool/dataset # load all keys in a dataset
```

When importing a pool that contains encrypted datasets: ZFS will by default not decrypt these datasets. To do this use `-l`

```
# zpool import -l pool
```

You can automate this at boot with a custom systemd unit. For example:

/etc/systemd/system/zfs-key@.service

```
[Unit]
Description=Load storage encryption keys
DefaultDependencies=no
Before=systemd-user-sessions.service
Before=zfs-mount.service

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/bin/bash -c 'systemd-ask-password "Encrypted storage password (%i): " | /usr/bin/zfs l
oad-key zpool/%i'

[Install]
WantedBy=zfs-mount.service
```

and enable a service instance for each encrypted volume:

```
# systemctl enable zfs-key@dataset.
```

The Before= reference to systemd-user-sessions.service ensures that systemd-ask-password is invoked before the local IO devices are handed over to the system UI

Scrub

ZFS pools should be scrubbed at least once a week. To scrub the pool:

```
# zpool scrub <pool>
```

To do automatic scrubbing once a week, set the following line in the root crontab:

```
# crontab -e
```

```
...
30 19 * * 5 zpool scrub <pool>
...
```

Replace `<pool>` with the name of the ZFS pool.

Check zfs pool status

To print a nice table with statistics about the ZFS pool, including and read/write errors, use

```
# zpool status -v
```

Destroy a storage pool

ZFS makes it easy to destroy a mounted storage pool, removing all metadata about the ZFS device. This command destroys any data contained in the pool:

```
# zpool destroy <pool>
```

And now when checking the status:

```
# zpool status
```

```
no pools available
```

To find the name of the pool, see **#Check zfs pool status**.

Exporting a storage pool

If a storage pool is to be used on another system, it will first need to be exported. It is also necessary to export a pool if it has been imported from the archiso as the hostid is different in the archiso as it is in the booted system. The zpool command will refuse to import any storage pools that have not been exported. It is possible to force the import with the `-f` argument, but this is

considered bad form.

Any attempts made to import an un-exported storage pool will result in an error stating the storage pool is in use by another system. This error can be produced at boot time abruptly abandoning the system in the busybox console and requiring an archiso to do an emergency repair by either exporting the pool, or adding the `zfs_force=1` to the kernel boot parameters (which is not ideal). See **#On boot the zfs pool does not mount stating: "pool may be in use from other system"**

To export a pool:

```
# zpool export <pool>
```

Renaming a zpool

Renaming a zpool that is already created is accomplished in 2 steps:

```
# zpool export oldname
# zpool import oldname newname
```

Setting a different mount point

The mount point for a given zpool can be moved at will with one command:

```
# zfs set mountpoint=/foo/bar poolname
```

Access Control Lists

To use **ACL** on a ZFS pool:

```
# zfs set acltype=posixacl <nameofzpool>/<nameofdataset>
# zfs set xattr=sa <nameofzpool>/<nameofdataset>
```

Setting `xattr` is recommended for performance reasons **[4] (https://github.com/zfsonlinux/zfs/issues/170#issuecomment-27348094)**.

Swap volume

ZFS does not allow to use swapfiles, but users can use a ZFS volume (ZVOL) as swap. It is importart to set the ZVOL block size to match the system page size, which can be obtained by the `getconf PAGESIZE` command (default on x86_64 is 4KiB). Another option useful for keeping the system running well in low-memory situations is not caching the ZVOL data.

Create a 8 GiB zfs volume:

```
# zfs create -V 8G -b $(getconf PAGESIZE) \
             -o logbias=throughput -o sync=always\
             -o primarycache=metadata \
             -o com.sun:auto-snapshot=false <pool>/swap
```

Prepare it as swap partition:

```
# mkswap -f /dev/zvol/<pool>/swap
# swapon /dev/zvol/<pool>/swap
```

To make it permanent, edit `/etc/fstab`. ZVOLs support discard, which can potentially help ZFS's block allocator and reduce fragmentation for all other datasets when/if swap is not full.

Add a line to `/etc/fstab`:

```
/dev/zvol/<pool>/swap none swap discard 0 0
```

Keep in mind the Hibernate hook must be loaded before filesystems, so using ZVOL as swap will not allow to use hibernate function. If you need hibernate, keep a partition for it.

Automatic snapshots

ZFS Automatic Snapshot Service for Linux

The **zfs-auto-snapshot-git (https://aur.archlinux.org/packages/zfs-auto-snapshot-git/)**[AUR] package from **AUR** provides a shell script to automate the management of snapshots, with each named by date and label (hourly, daily, etc), giving quick and convenient snapshotting of all ZFS datasets. The package also installs cron tasks for quarter-hourly, hourly, daily, weekly, and monthly snapshots. Optionally adjust the --keep parameter from the defaults depending on how far back the snapshots are to go (the monthly script by default keeps data for up to a year).

To prevent a dataset from being snapshotted at all, set `com.sun:auto-snapshot=false` on it. Likewise, set more fine-grained control as well by label, if, for example, no monthlies are to be kept on a snapshot, for example, set `com.sun:auto-snapshot:monthly=false`.

Note: zfs-auto-snapshot-git will not create snapshots during scrubbing (**scrub**). It is possible to override this by editing provided systemd unit (**Systemd#Editing provided units**) and removing `--skip-scrub` from `ExecStart` line. Consequences not known, someone please edit.

ZFS Snapshot Manager

The **zfs-snap-manager (https://aur.archlinux.org/packages/zfs-snap-manager/)**[AUR] package from **AUR** provides a python service that takes daily snapshots from a configurable set of ZFS datasets and cleans them out in a **"Grandfather-father-son"** scheme. It can be configured to e.g. keep 7 daily, 5 weekly, 3 monthly and 2 yearly snapshots.

The package also supports configurable replication to other machines running ZFS by means of `zfs send` and `zfs receive`. If the destination machine runs this package as well, it could be configured to keep these replicated snapshots for a longer time. This allows a setup where a source machine has only a few daily snapshots locally stored, while on a remote storage server a much longer retention is available.

Troubleshooting

Creating a zpool fails

If the following error occurs then it can be fixed.

```
# the kernel failed to rescan the partition table: 16
# cannot label 'sdc': try using parted(8) and then provide a specific slice: -1
```

One reason this can occur is because **ZFS expects pool creation to take less than 1 second (https://github.com/zfsonlinux/zfs/issues/2582)**. This is a reasonable assumption under ordinary conditions, but in many situations it may take longer. Each drive will need to be cleared again before another attempt can be made.

```
# parted /dev/sda rm 1
# parted /dev/sda rm 1
# dd if=/dev/zero of=/dev/sdb bs=512 count=1
# zpool labelclear /dev/sda
```

A brute force creation can be attempted over and over again, and with some luck the ZPool creation will take less than 1 second. Once cause for creation slowdown can be slow burst read writes on a drive. By reading from the disk in parallell to ZPool creation, it may be possible to increase burst speeds.

```
# dd if=/dev/sda of=/dev/null
```

This can be done with multiple drives by saving the above command for each drive to a file on separate lines and running

```
# cat $FILE | parallel
```

Then run ZPool creation at the same time.

ZFS is using too much RAM

By default, ZFS caches file operations (**ARC**) using up to two-thirds of available system memory on the host. To adjust the ARC size, add the following to the **Kernel parameters** list:

```
zfs.zfs_arc_max=536870912 # (for 512MB)
```

For a more detailed description, as well as other configuration options, see **gentoo-wiki:zfs#arc (http://wiki.gentoo.org/wiki/ZFS#ARC)**.

Does not contain an EFI label

The following error will occur when attempting to create a zfs filesystem,

```
/dev/disk/by-id/<id> does not contain an EFI label but it may contain partition
```

The way to overcome this is to use `-f` with the zfs create command.

No hostid found

An error that occurs at boot with the following lines appearing before initscript output:

```
ZFS: No hostid found on kernel command line or /etc/hostid.
```

This warning occurs because the ZFS module does not have access to the spl hosted. There are two solutions, for this. Either place the spl hostid in the **kernel parameters** in the boot loader. For example, adding `spl.spl_hostid=0x00bab10c`.

The other solution is to make sure that there is a hostid in `/etc/hostid`, and then regenerate the initramfs image. Which will copy the hostid into the initramfs image.

```
# mkinitcpio -p linux
```

Pool cannot be found while booting from SAS/SCSI devices

In case you are booting a SAS/SCSI based, you might occassionally get boot problems where the pool you are trying to boot from cannot be found. A likely reason for this is that your devices are initialized too late into the process. That means that zfs cannot find any devices at the time when it tries to assemble your pool.

In this case you should force the scsi driver to wait for devices to come online before continuing. You can do this by putting this into `/etc/modprobe.d/zfs.conf`:

```
/etc/modprobe.d/zfs.conf
```

```
options scsi_mod scan=sync
```

Afterwards, **regenerate the initramfs**.

This works because the zfs hook will copy the file at `/etc/modprobe.d/zfs.conf` into the initcpio which will then be used at build time.

On boot the zfs pool does not mount stating: "pool may be in use from other system"

Unexported pool

If the new installation does not boot because the zpool cannot be imported, chroot into the installation and properly export the zpool. See **#Emergency chroot repair with archzfs**.

Once inside the chroot environment, load the ZFS module and force import the zpool,

```
# zpool import -a -f
```

now export the pool:

```
# zpool export <pool>
```

To see the available pools, use,

```
# zpool status
```

It is necessary to export a pool because of the way ZFS uses the hostid to track the system the zpool was created on. The hostid is generated partly based on the network setup. During the installation in the archiso the network configuration could be different generating a different hostid than the one contained in the new installation. Once the zfs filesystem is exported and then re-imported in the new installation, the hostid is reset. See **Re: Howto zpool import/export automatically? - msg#00227 (http://osdir.com/ml/zfs-discuss/2011-06/msg00227.html)**.

If ZFS complains about "pool may be in use" after every reboot, properly export pool as described above, and then rebuild ramdisk in normally booted system:

```
# mkinitcpio -p linux
```

Incorrect hostid

Double check that the pool is properly exported. Exporting the zpool clears the hostid marking the ownership. So during the first boot the zpool should mount correctly. If it does not there is some other problem.

Reboot again, if the zfs pool refuses to mount it means the hostid is not yet correctly set in the early boot phase and it confuses zfs. Manually tell zfs the correct number, once the hostid is coherent across the reboots the zpool will mount correctly.

Boot using zfs_force and write down the hostid. This one is just an example.

```
% hostid
0a0af0f8
```

This number have to be added to the **kernel parameters** as `spl.spl_hostid=0x0a0af0f8` . Another solution is writing the hostid inside the initram image, see the **installation guide** explanation about this.

Users can always ignore the check adding `zfs_force=1` in the **kernel parameters**, but it is not advisable as a permanent solution.

Devices have different sector alignment

Once a drive has become faulted it should be replaced A.S.A.P. with an identical drive.

```
# zpool replace bigdata ata-ST3000DM001-9YN166_S1F0KDGY ata-ST3000DM001-1CH166_W1F478BD -f
```

but in this instance, the following error is produced:

```
cannot replace ata-ST3000DM001-9YN166_S1F0KDGY with ata-ST3000DM001-1CH166_W1F478BD: devices have dif
ferent sector alignment
```

ZFS uses the ashift option to adjust for physical block size. When replacing the faulted disk, ZFS is attempting to use `ashift=12` , but the faulted disk is using a different ashift (probably `ashift=9`) and this causes the resulting error.

For Advanced Format Disks with 4KB blocksize, an ashift of 12 is recommended for best performance. See **1.10 What's going on with performance? (https://github.com/zfsonlinux/zfs/wiki/faq#performance-considerations)** and **ZFS and Advanced Format disks (http://wiki.illumos.org/display/illumos/ZFS+and+Advanced+Format+disks)**.

Use zdb to find the ashift of the zpool: `zdb`, then use the `-o` argument to set the ashift of the replacement drive:

```
# zpool replace bigdata ata-ST3000DM001-9YN166_S1F0KDGY ata-ST3000DM001-1CH166_W1F478BD -o ashift=9 -
f
```

Check the zpool status for confirmation:

```
# zpool status -v
```

```
pool: bigdata
state: DEGRADED
status: One or more devices is currently being resilvered.  The pool will
        continue to function, possibly in a degraded state.
action: Wait for the resilver to complete.
scan: resilver in progress since Mon Jun 16 11:16:28 2014
    10.3G scanned out of 5.90T at 81.7M/s, 20h59m to go
    2.57G resilvered, 0.17% done
config:

        NAME                                    STATE     READ WRITE CKSUM
        bigdata                                 DEGRADED     0     0     0
        raidz1-0                                DEGRADED     0     0     0
            replacing-0                         OFFLINE      0     0     0
            ata-ST3000DM001-9YN166_S1F0KDGY     OFFLINE      0     0     0
            ata-ST3000DM001-1CH166_W1F478BD     ONLINE       0     0     0  (resilvering)
            ata-ST3000DM001-9YN166_S1F0JKRR     ONLINE       0     0     0
            ata-ST3000DM001-9YN166_S1F0KBP8     ONLINE       0     0     0
            ata-ST3000DM001-9YN166_S1F0JTM1     ONLINE       0     0     0

errors: No known data errors
```

Tips and tricks

Embed the archzfs packages into an archiso

Follow the **Archiso** steps for creating a fully functional Arch Linux live CD/DVD/USB image.

Enable the **archzfs** repository:

```
~/archlive/pacman.conf
```

```
...
[archzfs]
Server = http://archzfs.com/$repo/x86_64
```

Add the `archzfs-linux` group to the list of packages to be installed (the `archzfs` repository provides packages for the x86_64 architecture only).

```
~/archlive/packages.x86_64
```

```
...
archzfs-linux
```

Complete **Build the ISO** to finally build the iso.

> **Note:** If you later have problems running modprobe zfs, you should include the linux-headers in the packages.x86_64.

Encryption in ZFS using dm-crypt

The stable release version of ZFS on Linux does not support encryption directly, but zpools can be created in dm-crypt block devices. Since the zpool is created on the plain-text abstraction, it is possible to have the data encrypted while having all the advantages of ZFS like deduplication, compression, and data robustness.

dm-crypt, possibly via LUKS, creates devices in `/dev/mapper` and their name is fixed. So you just need to change `zpool create` commands to point to that names. The idea is configuring the system to create the `/dev/mapper` block devices and import the zpools from there. Since zpools can be created in multiple devices (raid, mirroring, striping, ...), it is important all the devices are encrypted otherwise the protection might be partially lost.

For example, an encrypted zpool can be created using plain dm-crypt (without LUKS) with:

```
# cryptsetup --hash=sha512 --cipher=twofish-xts-plain64 --offset=0 \
             --key-file=/dev/sdZ --key-size=512 open --type=plain /dev/sdX enc
# zpool create zroot /dev/mapper/enc
```

In the case of a root filesystem pool, the `mkinicpio.conf` HOOKS line will enable the keyboard for the password, create the devices, and load the pools. It will contain something like:

```
HOOKS="... keyboard encrypt zfs ..."
```

Since the `/dev/mapper/enc` name is fixed no import errors will occur.

Creating encrypted zpools works fine. But if you need encrypted directories, for example to protect your users' homes, ZFS loses some functionality.

ZFS will see the encrypted data, not the plain-text abstraction, so compression and deduplication will not work. The reason is that encrypted data has always high entropy making compression ineffective and even from the same input you get different output (thanks to salting) making deduplication impossible. To reduce the unnecessary overhead it is possible to create a sub-filesystem for each encrypted directory and use **eCryptfs** on it.

For example to have an encrypted home: (the two passwords, encryption and login, must be the same)

```
# zfs create -o compression=off -o dedup=off -o mountpoint=/home/<username> <zpool>/<username>
# useradd -m <username>
# passwd <username>
# ecryptfs-migrate-home -u <username>
<log in user and complete the procedure with ecryptfs-unwrap-passphrase>
```

Emergency chroot repair with archzfs

To get into the ZFS filesystem from live system for maintenance, there are two options:

1. Build custom archiso with ZFS as described in **#Embed the archzfs packages into an archiso**.
2. Boot the latest official archiso and bring up the network. Then enable **archzfs** repository inside the live system as usual, sync the pacman package database and install the *archzfs-archiso-linux* package.

To start the recovery, load the ZFS kernel modules:

```
# modprobe zfs
```

Import the pool:

```
# zpool import -a -R /mnt
```

Mount the boot partitions (if any):

```
# mount /dev/sda2 /mnt/boot
# mount /dev/sda1 /mnt/boot/efi
```

Chroot into the ZFS filesystem:

```
# arch-chroot /mnt /bin/bash
```

Check the kernel version:

```
# pacman -Qi linux
# uname -r
```

uname will show the kernel version of the archiso. If they are different, run depmod (in the chroot) with the correct kernel version of the chroot installation:

```
# depmod -a 3.6.9-1-ARCH (version gathered from pacman -Qi linux but using the matching kernel module
s directory name under the chroot's /lib/modules)
```

This will load the correct kernel modules for the kernel version installed in the

chroot installation.

Regenerate the ramdisk:

```
# mkinitcpio -p linux
```

There should be no errors.

Bind mount

Here a bind mount from /mnt/zfspool to /srv/nfs4/music is created. The configuration ensures that the zfs pool is ready before the bind mount is created.

fstab

See **systemd.mount (http://www.freedesktop.org/software/systemd/man/systemd.mount.html)** for more information on how systemd converts fstab into mount unit files with **systemd-fstab-generator (http://www.freedesktop.org/software/systemd/man/systemd-fstab-generator.html)**.

```
/etc/fstab

/mnt/zfspool            /srv/nfs4/music         none    bind,defaults,nofail,x-systemd.requires=zfs-m
ount.service    0 0
```

Monitoring / Mailing on Events

See **ZED: The ZFS Event Daemon (https://ramsdenj.com/2016/08/29/arch-linux-on-zfs-part-3-followup.html)** for more information.

At the minimum, an email forwarder, such as **msmtp**, is required to accomplish this. Make sure it is working correctly.

Uncomment the following in the configuration file:

```
/etc/zfs/zed.d/zed.rc

ZED_EMAIL_ADDR="root"
ZED_EMAIL_PROG="mail"
ZED_NOTIFY_VERBOSE=0
ZED_EMAIL_OPTS="-s '@SUBJECT@' @ADDRESS@"
```

Update 'root' in `ZED_EMAIL_ADDR="root"` to the email address you want to receive notifications at.

If you want to receive an email no matter the state of your pool, you will want to set `ZED_NOTIFY_VERBOSE=1`.

Start and enable `zfs-zed.service`.

If you set verbose to 1, you can test by running a scrub.

See also

- **Aaron Toponce's 17-part blog on ZFS (https://pthree.org/2012/12/04/zfs-administration-part-i-vdevs/)**
- **ZFS on Linux (http://zfsonlinux.org/)**
- **ZFS on Linux FAQ (https://github.com/zfsonlinux/zfs/wiki/faq)**
- **FreeBSD Handbook -- The Z File System (https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/zfs.html)**
- **Oracle Solaris ZFS Administration Guide (https://docs.oracle.com/cd/E19253-01/819-5461/index.html)**
- **Solaris Internals -- ZFS Troubleshooting Guide (http://www.solarisinternals.com/wiki/index.php/ZFS_Troubleshooting_Guide)**[**dead link** 2017-05-30]
- **How Pingdom uses ZFS to back up 5TB of MySQL data every day (http://royal.pingdom.com/2013/06/04/zfs-backup/)**
- **Tutorial on adding the modules to a custom kernel (https://www.linuxquestions.org/questions/linux-from-scratch-13/%5Bhow-to%5D-add-zfs-to-the-linux-kernel-4175514510/)**

Retrieved from "https://wiki.archlinux.org/index.php?title=ZFS&oldid=520032"
