SysadminGuide

From btrfs Wiki

This page is intended to give a slightly deeper insight into what the various Btrfs features are doing behind the scenes.

Contents

- 1 Btrfs introduction in a talk
- 2 Data usage and allocation
  - 2.1 RAID and data replication
  - 2.2 Balancing
- 3 Copy on Write (CoW)
- 4 Subvolumes
  - 4.1 Description
  - 4.2 Layout
    - 4.2.1 Flat
    - 4.2.2 Nested
    - 4.2.3 Mixed

    - 4.3 When To Make Subvolumes
    - 4.4 Snapshots
        - 4.4.1 Managing Snapshots
    - 4.5 Special Cases
- 5 Btrfs on top of dmcrypt

Btrfs introduction in a talk

If you'd like an overview with pointers to the more useful features and cookbooks, you can try Marc MERLIN's Btrfs talk at Linuxcon JP 2014 (http://marc.merlins.org/perso/btrfs/post_2014-05-21_My-Btrfs-Talk-at-Linuxcon-JP-2014.html) .

Data usage and allocation

Btrfs, at its lowest level, manages a pool of raw storage, from which it allocates space for different internal purposes. This pool is made up of all of the block devices that the volume (and any subvolumes) lives on, and the size of the pool is the "total" value reported by the ordinary `df` command. As the volume needs storage to hold file data, or volume metadata, it allocates chunks of this raw storage, typically in 1GiB lumps, for use by the higher levels of the filesystem. The allocation of these chunks is what you see as the output from `btrfs filesystem usage`. The extents belonging to many files may be placed within a chunk, and files may span across more than one chunk. A chunk is simply a piece of storage that Btrfs can use to put data on. The most important function of a chunk is that it has a profile that is how it is replicated within or across a member device.

Terminology: space is allocated to chunks, and space is used by blocks. A chunk with no used blocks in it is unallocated; a chunk with 1 or more used blocks in it is allocated. All chunks can be allocated even if not all space is used.

RAID and data replication

Btrfs's "RAID" implementation bears only passing resemblance to traditional RAID implementations. Instead, Btrfs replicates data on a per-chunk basis. If the filesystem is configured to use "RAID-1", for example, chunks are allocated in pairs, with each chunk of the pair being taken from a different block device. Data written to such a chunk pair will be duplicated across both chunks.

Stripe-based "RAID" levels (RAID-0, RAID-10) work in a similar way, allocating as many chunks as can fit across the drives with free space, and then perform striping of data at a level smaller than a chunk. So, for a RAID-10 filesystem on 4 disks, data may be stored like this:

```
Storing file: 01234567...89
```

```
Block devices:  /dev/sda   /dev/sdb   /dev/sdc   /dev/sdd

Chunks:          +-A1-+  +-A2-+  +-A3-+  +-A4-+
                 | 0  |  | 1  |  | 0  |  | 1  | }
                 | 2  |  | 3  |  | 2  |  | 3  | } stripes within a chunk
                 | 4  |  | 5  |  | 4  |  | 5  | }
                 | 6  |  | 7  |  | 6  |  | 7  | }
                 |... |  |... |  |... |  |... | }
                 +----+  +----+  +----+  +----+

                 +-B3-+  +-B2-+  +-B4-+  +-B1-+   a second set of chunks may be needed for large files.
                 | 8  |  | 9  |  | 9  |  | 8  |
                 +----+  +----+  +----+  +----+
```

Note that chunks within a RAID grouping are not necessarily always allocated to the same devices (B1-B4 are reordered in the example above). This allows Btrfs to do data duplication on block devices with varying sizes, and still use as much of the raw space as possible. With RAID-1 and RAID-10, only two copies of each byte of data are written, regardless of how many block devices are actually in use on the filesystem.

Balancing

A Btrfs balance operation rewrites things at the level of chunks. It runs through all chunks on the filesystem, and writes them out again, discarding and freeing up the original chunks when the new copies have been written. This has the effect that if data is missing a replication copy (e.g. from a missing/dead disk), new replicas are created. The balance operation also has the effect of re-running the allocation process for all of the data on the filesystem, thus more effectively "balancing" the data allocation over the underlying disk storage.

All filesystems for best performance should have some minimum percentage of free space, usally 5-10%. Btrfs, using a two-level space manager (chunks and blocks within chunks) plus being based on (nearly) copy-on-write, benefits from more space to be kept free, around 10-15%, and in particular from having at least one or a few chunks fully unallocated.

It is *quite useful* to `balance` periodically any Btrfs volume subject to updates, to prevent the allocation of every chunk in the volume. Usually it is enough to balance periodically only the chunks that are just 50% used or 70% used and this is quick, for example as:

```
btrfs balance start -musage=50 -dusage=50 ...
```

Q - what kernel threads do this operation?

Copy on Write (CoW)

- The CoW operation is used on <u>all</u> writes to the filesystem (unless turned off, see below).
- This makes it much easier to implement lazy copies, where the copy is initially just a reference to the original, but as the copy (or the original) is changed, the two versions diverge from each other in the expected way.
- If you just write a file that didn’t exist before, then the data is written to empty space, and some of the metadata blocks that make up the filesystem are CoWed. In a "normal" filesystem, if you then go back and overwrite a piece of that file, then the piece you’re writing is put directly over the data it is replacing. In a CoW filesystem, the new data is

written to a piece of free space on the disk, and only then is the file’s metadata changed to refer to the new data. At that point, the old data that was replaced can be freed up because nothing points to it any more.

- If you make a snapshot (or a cp --reflink=always) of a piece of data, you end up with two files that both reference the same data. If you modify one of those files, the CoW operation I described above still happens: the new data is written elsewhere, and the file’s metadata is updated to point at it, but the original data is kept, because it’s still referenced by the other file.
  - This leads to fragmentation in heavily updated-in-place files like VM images and database stores.
  - Note that this happens even if the data is not shared, because data is stored in segments, and only the newly updated part of a segment is subject to CoW.
- If you mount the filesystem with `nodatacow`, or use `chattr +C` on the file, then it only does the CoW operation for data if there’s more than one copy referenced.
- Some people insist that Btrfs does "Redirect-on-write" rather than "Copy-on-write" because Btrfs is based on a scheme for redirect-based updates of B-trees by Ohad Rodeh, and because understanding the code is easier with that mindset.

Subvolumes

Description

A Btrfs subvolume is an independently mountable POSIX filetree and **not a block device** (and cannot be treated as one). Most other POSIX filesystems have a single mountable root, Btrfs has an independent mountable root for the volume (top level subvolume) and for each subvolume; a Btrfs volume can contain more than a single filetree, it can contain a forest of filetrees. A Btrfs subvolume can be thought of as a POSIX file namespace.

A subvolume in Btrfs is **not** similar to a LVM logical volume or a ZFS subvolume. With LVM, a logical volume is a block device in its own right (which could for example contain any other filesystem or container like dm-crypt, MD RAID, etc.), this is not the case with Btrfs.

A Btrfs subvolume root directory differs from a directory in that each subvolume defines a distinct inode number space (distinct inodes in different subvolumes can have the same inumber) and each inode under a subvolume has a distinct device number (as reported by `stat`(2)). Each subvolume root can be accessed as implicitly mounted via the volume (top level subvolume) root, if that is mounted, or it can be mounted in its own right.

So, given a filesystem structure like this:

```
toplevel            (volume root directory)
+-- dir_1           (normal directory)
|   +-- file_2      (normal file)
|   \-- file_3      (normal file)
\-- subvol_a        (subvolume root directory)
    +-- subvol_b    (subvolume root directory, nested below subvol_a)
    |   \-- file_4  (normal file)
    \-- file_5      (normal file)
```

The top-level subvolume (with Btrfs id 5) (which one can think of as the root of the volume) can be mounted, and the full filesystem structure will be seen at the mount point; alternatively any other subvolume can be mounted (with the mount options `subvol` or `subvolid`, for example `subvol=subvol_a`) and only anything below that subvolume (in the above example the subvolume `subvol_b`, its contents, and file `file_4`) will be visible at the mount point.

Subvolumes can be nested and each subvolume (except the top-level subvolume) has a parent subvolume. Mounting a subvolume also makes any of its nested child subvolumes available at their respective location relative to the mount-point.

A Btrfs filesystem has a default subvolume, which is initially set to be the top-level subvolume and which is mounted if no `subvol` or `subvolid` option is specified.

Changing the default subvolume with `btrfs subvolume default` will make the top level of the filesystem inaccessible, except by use of the `subvol=/` or `subvolid=5` mount options.

Subvolumes can be moved around in the filesystem.

Layout

There are several *basic* schemas to layout subvolumes (including snapshots) as well as mixtures thereof.

Flat

Subvolumes are children of the top level subvolume (ID 5), typically directly below in the hierarchy or below some directories belonging to the top level subvolume, but especially not nested below other subvolumes, for example:

```
toplevel         (volume root directory, not to be mounted by default)
  +-- root       (subvolume root directory, to be mounted at /)
  +-- home       (subvolume root directory, to be mounted at /home)
  +-- var        (directory)
  |   \-- www    (subvolume root directory, to be mounted at /var/www)
  \-- postgres   (subvolume root directory, to be mounted at /var/lib/postgresql)
```

Here, the toplevel mountable root directory should not normally be visible to the system. Of course, the `www` subvolume could have also been placed directly below the top level subvolume (without an intermediate directory) and the `postgres` subvolume could have also been placed at a directory `var/lib/postgresql`; these are just examples and the actual layout depends largely on personal taste. All subvolumes are however direct children of the top level subvolume in this scheme.

This has several implications (advantages and disadvantages):

- Management of snapshots (especially rolling them) may be considered easier as the effective layout is more directly visible.
- All subvolumes need to be mounted manually (e.g. via fstab) to their desired locations, e.g. in the above example this would look like:

```
LABEL=the-btrfs-fs-device   /                    subvol=/root,defaults,noatime  0  0
LABEL=the-btrfs-fs-device   /home                subvol=/home,defaults,noatime  0  0
LABEL=the-btrfs-fs-device   /var/www             subvol=/var/www,noatime        0  0
LABEL=the-btrfs-fs-device   /var/lib/postgresql  subvol=/postgres,noatime       0  0
```

- Each of these subvolumes/mountpoints can be mounted with some options being different.

  This means however as well, that any typically useful mount options (for example `noatime`) need to be specified again for each mountpoint.
- Everything in the volume that's not beneath a subvolume that has been mounted, is not accessible or even visible. This may be beneficial for security, especially when being used with snapshots, see below.

Nested

Subvolumes are located anywhere in the file hierarchy, typically at their desired locations (that is where one would manually mount them at in the flat schema), especially below other subvolumes that are not the top-level subvolume, for example:

```
toplevel                  (volume root directory, to be mounted at /)
+-- home                  (subvolume root directory)
+-- var                   (subvolume root directory)
    +-- www               (subvolume root directory)
    +-- lib               (directory)
         \-- postgresql   (subvolume root directory)
```

This has several implications (advantages and disadvantages):

- Management of snapshots (especially rolling them) may be considered more difficult as the effective layout isn't directly visible.
- Subvolumes don't need to be mounted manually (or via fstab) to their desired locations, they "appear automatically" at their respective locations.
- For each of these subvolumes the mount options of their mountpoint applies.
- Everything is visible. Of course one could only mount a subvolume, but then important parts of the filesystem (in this example much of the system's "main" filesystem) would be missing as well. This may have disadvantages for security, especially when being used with snapshots, see below.

The above layout, which obviously serves as the system's "main" filesystem, places data directly within the top-level subvolume (namely everything for example /usr, that's not in a child subvolume) This makes changing the structure (for example to something more flat) more difficult, which is why it's generally suggested to place the actual data in a subvolume (that is not the top-level subvolume), in the above example, a better layout would be the following:

```
toplevel                        (volume root directory, not mounted)
  \-- root                      (subvolume root directory, to be mounted at /)
      +-- home                  (subvolume root directory)
      +-- var                   (subvolume root directory)
          +-- www               (subvolume root directory)
          +-- lib               (directory)
               \-- postgresql   (subvolume root directory)
```

Mixed

Of course the above two basic schemas can be mixed at personal convenience, e.g. the base structure may follow a flat layout, with certain parts of the filesystem, being placed in nested subvolumes. But care must be taken when snapshots should be made (nested subvolumes are not part of a snapshot, and as of now, there are no recursive snapshots) as well when those are rotated, where the nested subvolumes would then need to be manually moved (either the "current" versions or a snapshot thereof).

When To Make Subvolumes

This is a not exhaustive list of typical guidelines where/when to make subvolumes, rather than keeping things in the same subvolume.

- Nested subvolumes are not going to be part of snapshots created from their parent subvolume. So one typical reason is to exclude certain parts of the filesystem from being snapshot.

  A typical example could be a directory which contains just builds of software (in other words, nothing precious) that use up much space. Having those being part of the snapshot may be useless, as they can be easily regenerated from source.

- "Split" of areas which are "complete" and/or "consistent" in themselves.

Examples would be "/home", "/var/www", or "/var/lib/postgresql/", which are usually more or less "independent" of the other other parts of system, at least with respect to a certain state.
In contrast, splitting parts of the system which "belong together" or depend on a certain state of each other into different subvolumes, may be a bad idea, at least when snapshots are being made.
For example, "/etc", many parts of "/var/" (especially "/var/lib"), "/usr", "/lib", "/bin/", etc. are typically closely related, at least by being managed from the package management system, which assumes them to be all in the same state. If these were not, because being snapshot at different times, the package manager may assume packages to be installed at a certain version, which would apply for e.g. "/usr" but not for "/bin" or "/etc". It's obvious that this means tickling the dragon.
One reason for "splitting" of areas into subvolumes may be the usage Btrfs' send/receive feature (for example for snapshots and backups or simply for moving/copying data):
When making a snapshot on the local system, having more in the subvolume than just the actually desired data (e.g. the whole system's "main" filesystem instead of just "/var/lib/postgresql") doesn't matter too much. This is at least more or less the case when CoW is used, since then all data is just ref-linked, which means it's fast and doesn't cost much more space (there may be however other implications, for example with respect to fragmentation).
But when that data shall be sent/received to other Btrfs filesystems this "undesired" data makes a big difference and not just only that it needs to be transferred there at least once (Btrfs' send/receive feature works on subvolumes), which may already be bad, for security and privacy reasons.

If send/receive of the desired data shall happen more often, typically for backups, then even when btrfs-send’s -p or -c features would be used for incremental transfers, the undesired data would need to be transferred at least once (assuming it would never change again on the source system, which is unlikely).

- Split of areas which need special properties / mount options.

Snapshots

A snapshot is simply a subvolume that shares its data (and metadata) with some other subvolume, using Btrfs's COW capabilities.

Once a [writable] snapshot is made, there is no difference in status between the original subvolume, and the new snapshot subvolume. To roll back to a snapshot, unmount the modified original subvolume, use mv to rename the old subvolume to a temporary location, and then again to rename the snapshot to the original name. You can then remount the subvolume.

At this point, the original subvolume may be deleted if wished. Since a snapshot is a subvolume, snapshots of snapshots are also possible.

Beware: Care must be taken when snapshots are created that are then visible to any user (e.g. when they're created in a nested layout) as this may have security implications. Of course, the snapshot will have the same permissions as the subvolume from which it was created at the time it was, but these permissions may be tightened later on, while those of the snapshot wouldn't change, possibly allowing access to files that shouldn't be accessible anymore. Similarly, especially on the system's "main" filesystem, the snapshot would contain any files (for example setuid programs) of the state when it was created. In the meantime however, security updates may have been rolled out on the original subvolume, but when the snapshot is accessible (and for example the vulnerable setuid has been accessible before) a user could still invoke it.

Managing Snapshots

One typical structure for managing snapshots (particularly on a system's "main" filesystem) is based on the to flat schema from above. The top-level subvolume is mostly left empty except for subvolumes. It can be mounted temporarily while subvolume operations are being done and unmounted again afterwards. Alternatively an empty "snapshot" subvolume can be created (which in turn contains the actual snapshots) and permanently mounted at a convenient directory path.

The actual structure could look like this:

```
toplevel                   (volume root directory, not mounted)
  +-- root                 (subvolume root directory, to be mounted at /)
  +-- home                 (subvolume root directory, to be mounted at /home)
  \-- snapshots            (directory)
      +-- root             (directory)
          +-- 2015-01-01   (root directory of snapshot of subvolume "root")
          +-- 2015-06-01   (root directory of snapshot of subvolume "root")
      \-- home             (directory)
          \-- 2015-01-01   (root directory of snapshot of subvolume "home")
          \-- 2015-12-01   (root directory of snapshot of subvolume "home")
```

or even flatter, like this:

```
toplevel                         (volume root directory)
 +-- root                        (subvolume root directory, mounted at /)
 |   +-- bin                     (directory)
 |   +-- usr                     (directory)
 +-- root_snapshot_2015-01-01   (root directory of snapshot of subvolume "root")
 |   +-- bin                     (directory)
 |   +-- usr                     (directory)
 +-- root_snapshot_2015-06-01   (root directory of snapshot of subvolume "root")
 |   +-- bin                     (directory)
 |   +-- usr                     (directory)
 +-- home                        (subvolume root directory, mounted at /home)
 +-- home_snapshot_2015-01-01   (root directory of snapshot of subvolume "home")
 +-- home_snapshot_2015-12-01   (root directory of snapshot of subvolume "home")
```

A corresponding fstab could look like this:

```
LABEL=the-btrfs-fs-device   /                     subvol=/root,defaults,noatime      0  0
LABEL=the-btrfs-fs-device   /home                 subvol=/home,defaults,noatime      0  0
LABEL=the-btrfs-fs-device   /root/btrfs-top-lvl  subvol=/,defaults,noauto,noatime   0  0
```

Creating a ro-snapshot would work for example like this:

```
# mount /root/btrfs-top-lvl
# btrfs subvolume snapshot -r /root/btrfs-top-lvl/home /root/btrfs-top-lvl/snapshots/home/2015-12-01
# umount /root/btrfs-top-lvl
```

Rolling back a snapshot would work for example like this:

```
# mount /root/btrfs-top-lvl
# umount /home
```

now either the snapshot could be directly mounted at /home (which would be temporary, unless fstab is adatped as well):

```
# mount -o subvol=/snapshots/home/2015-12-01,defaults,noatime LABEL=the-btrfs-fs-device /home
```

or the subvolume itself could be moved and then mounted again

```
# mv /root/btrfs-top-lvl/home /root/btrfs-top-lvl/home.tmp     #or it could  have been deleted see below
# mv /root/btrfs-top-lvl/snapshots/home/2015-12-01 /root/btrfs-top-lvl/home
# mount /home
```

finally, the top-level subvolume could be unmounted:

```
# umount /root/btrfs-top-lvl
```

The careful reader may have noticed, that for the rollback, a rw-snapshot was used, which is in the above example of course necessary, as a read-only /home wouldn't be desired. If a ro-snapshot would have been created (with -r), then one would have simply made another snapshot of that, this time of course rw, for example:

```
…
# btrfs subvolume delete /root/btrfs-top-lvl/home
# btrfs subvolume snapshot /root/btrfs-top-lvl/snapshots/home/2015-01-01 /root/btrfs-top-lvl/home
…
```

Special Cases

- Read-only subvolumes cannot be moved.

  As mentioned above, subvolumes can be moved, though there are exceptions, namely read-only subvolumes, which cannot be moved from their parent directory (which may be a subvolume itself), because the snapshot's ".." file is pointed to by that, but cannot be changed because of being read-only.
  Consider the following example:

```
…
+-- foo               (directory, optionally a subvolume)
+-- bar               (directory, optionally a subvolume)
|   \-- ro-snapshot   (ro-snapshot)
…
```

  It wouldn't be possible to move ro-snapshot to foo, as this would need to modify ro-snapshot`s ".." file.
  Moving bar (including anything below it) to foo, works however.

Btrfs on top of dmcrypt

As of Linux kernel 3.2, it is now considered safe to have Btrfs on top of dmcrypt (before that, there are risks of corruption during unclean shutdowns).

With multi-device setups, decrypt_keyctl (https://github.com/gebi/keyctl_keyscript) may be used to unlock all disks at once. You can also look at this start-btrfs-dmcrypt from Marc MERLIN (http://marc.merlins.org/perso/btrfs/post_2014-04-27_Btrfs-Multi-Device-Dmcrypt.html) that shows you how to manually bring up a Btrfs dmcrypted array.

The following page shows benchmarks of Btrfs vs ext4 on top of dmcrypt or with ecryptfs: http://www.mayrhofer.eu.org/ssd-linux-benchmark . The summary is that Btrfs with lzo compression enabled on top top of dmcrypt is either slightly faster or slightly slower than ext4 and encryption only creates a slowdown in the 5% range on an SSD (likely even less on a spinning drive).

```
# cryptsetup -c aes-xts-plain -s 256 luksFormat /dev/sda3
# cryptsetup luksOpen /dev/sda3 luksroot
# mkfs.btrfs /dev/mapper/luksroot
# mount -o noatime,ssd,compress=lzo /dev/mapper/luksroot /mnt
```
